第10章 并发控制

金培权（jpq@ustc.edu.cn）

再论丢失更新问题

Step	T1	T2
1	read(A, t), t = t+1	
2		read(A, t), t = t + 1
3	write(A, t)	
3		write(A, t)
4		Commit
6	Commit	

Lost update

两次提交写导致的写覆盖

Step	T1	T2
1	read(A, t), t = t+1	
2		read(A, t), t = t + 1
3	write(A, t)	
4		write(A, t)
5		read(B, u), u = u + 1
6		write(B, u)
7		Commit
8	Abort	

Dirty write

由于Rollback导致的提交事务的写失效
破坏了T2的原子性

DBMS中不允许出现Dirty write
在任何情况下都要求X锁保留到事务结束

Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, Patrick E. O'Neil: **A Critique of ANSI SQL Isolation Levels.** SIGMOD 1995: 1-10

主要内容

- 并发操作与并发问题
- 并发事务调度与可串性 (Scheduling and Serializability)
- 锁与可串性实现 (Locks)
- 事务的隔离级别 
- 死锁

四、事务的隔离级别

- 并发控制机制可以解决并发问题。这使所有事务得以在彼此完全隔离的环境中运行
- 然而许多事务并不总是要求完全的隔离。如果允许降低隔离级别，则可以提高并发性

四、事务的隔离级别

- SQL92标准定义了四种事务隔离级别
  - Note 1：隔离级别是针对连接（会话）而设置的，不是针对一个事务。MySQL允许设置Global隔离级别
  - Note 2：不同隔离级别影响读操作。X锁必须保持到事务结束后释放（因此不会出现Dirty Write）

Oracle, MS SQL Server 默认

MySQL默认

隔离级别	脏读	不可重复读取	幻像
未提交读	是	是	是
提交读	否	是	是
可重复读	否	否	是
可串行读	否	否	否

Oracle只支持提交读和可串行读，MySQL和MS SQL Server都支持四种隔离级别

四、事务的隔离级别

- **MySQL**
  - 查看当前的隔离级别
  - 修改隔离级别
    - set [global | session] transaction isolation level Read uncommitted | Read committed | Repeatable | Serializable;

```
mysql> select @@tx_isolation;
+-----------------+
| @@tx_isolation  |
+-----------------+
| REPEATABLE-READ |
+-----------------+
1 row in set (0.00 sec)
```

```
mysql> set session transaction isolation level read committed;
Query OK, 0 rows affected (0.00 sec)

mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from account;
+------+--------+---------+
| id   | name   | balance |
+------+--------+---------+
|    1 | lilei  |     450 |
|    2 | hanmei |   16000 |
|    3 | lucy   |    2400 |
+------+--------+---------+
3 rows in set (0.00 sec)
```

四、事务的隔离级别

- 未提交读（脏读） **Read Uncommitted**
  - 允许读取当前数据页上的任何数据，不管数据是否已提交
  - 事务不必等待任何锁，也不对读取的数据加锁
  - 实际DBMS中一般不使用

隔离级别	脏读	不可重复读取	幻像
未提交读	是	是	是
提交读	否	是	是
可重复读	否	否	是
可串行读	否	否	否

四、事务的隔离级别

- 提交读 **Read Committed**
  - 保证事务不会读取到其他未提交事务所修改的数据（可防止脏读）
  - 事务必须在所访问数据上加S锁，数据一旦读出，就马上释放持有的S锁

隔离级别	脏读	不可重复读取	幻像
未提交读	是	是	是
提交读	否	是	是
可重复读	否	否	是
可串行读	否	否	否

四、事务的隔离级别

时间	连接1	连接2
1		Set transaction isolation level READ COMMITTED
2		Start transaction
3		Select SNAME from S Where SNAME=‘王红’
4	Start transaction	
5	Update s set SNAME=‘王红’	
6		Select SNAME from S Where SNAME=‘王红’
7	Commit	
8		Commit

等待

四、事务的隔离级别

- 可重复读 **Repeatable Read**
  - 保证事务在事务内部如果重复访问同一数据（记录集），数据不会发生改变。即，事务在访问数据时，其他事务不能修改正在访问的那部分数据
  - 可重复读可以防止脏读和不可重复读取，但不能防止幻像
  - 事务必须在所访问数据上加S锁，防止其他事务修改数据，而且S锁必须保持到事务结束

隔离级别	脏读	不可重复读取	幻像
未提交读	是	是	是
提交读	否	是	是
可重复读	否	否	是
可串行读	否	否	否

四、事务的隔离级别

时间	连接1	连接2
1		Set transaction isolation level REPEATABLE READ
2		Start transaction
3	Start transaction	Select SNAME from S Where SNAME=‘王红’
4	Insert into s values(s08,’王红’,23)	
5	Update s set age＝22 where sname=‘张三’	
6	Update s set age＝22 where sname=‘王红’	
7		Select SNAME from S Where SNAME=‘王红’
8		Commit

此两步执行

此步须等待

出现幻象

四、事务的隔离级别

- 可串行读 **Serializable**
  - 保证事务调度是可串化的
  - 事务在访问数据时，其他事务不能修改数据，也不能插入新元组
  - 事务必须在所访问数据上加S锁，防止其他事务修改数据，而且S锁必须保持到事务结束
  - 事务还必须锁住访问的整个表

隔离级别	脏读	不可重复读取	幻像
未提交读	是	是	是
提交读	否	是	是
可重复读	否	否	是
可串行读	否	否	否

四、事务的隔离级别

时间	连接1	连接2
1		Set transaction isolation level SERIALIZABLE
2		Start transaction
3	Start transaction	Select SNAME from S Where SNAME=‘王红’
4	Insert into s values(s08,’王红’ ,23)	
5	此步须等待	Select SNAME from S Where SNAME=‘王红’
6		Commit

四、事务的隔离级别

- 不同隔离级别下DBMS加锁的动作有很大的差别

五、死锁(deadlock)

- Two transactions each acquire a lock on a DB element the other needs
- Two transactions try to upgrade locks on elements the other is reading

1、锁导致死锁

t	T1	T2
1	sL1(A)	
2		sL2(B)
3	Read(A)	Read(B)
4	xL1(B)	xL2(A)
5	Wait	Wait
6	Wait	Wait
7	wait	Wait
8	Wait	Wait
9	Wait	Wait
10	……	……

1、锁导致死锁

t	T1	T2
1	sL1(A)	
2		sL2(A)
3	Read(A)	Read(A)
4	A=A+B	
5	Upgrade(A)	A=A+100
6	Wait	Upgrade(A)
7	Wait	Wait
8	Wait	Wait
9	Wait	Wait
10	……	……

使用Update Lock

2、死锁的两种处理策略

- 死锁检测 Deadlock Detecting
  - 检测到死锁，再解锁
- 死锁预防 Deadlock Prevention
  - 提前采取措施防止出现死锁

3、Deadlock Detecting

- **Timeout 超时**
  - **Simple idea:** If a transaction hasn't completed in x minutes, abort it
- **Waiting graph 等待图**

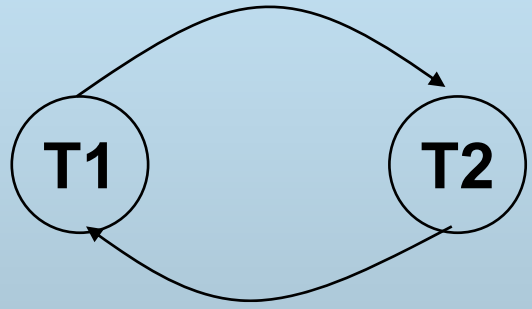
3、Deadlock Detecting

- **Waiting graph**
  - **Node：Transactions**
  - **Arcs：Ti → Tj, Ti必须等待Tj释放所持有的某个锁才能继续执行**

如果等待图中存在环路，
说明产生了死锁

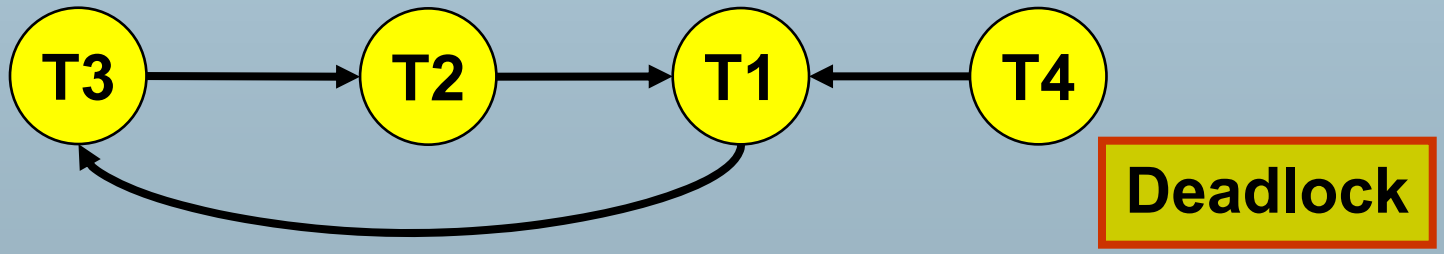
3、Deadlock Detecting

t	T1	T2
1	sL1(A)	
2		sL2(A)
3	Read(A)	Read(A)
4	sL1(B)	A=A+100
5	Read(B)	Upgrade(A)
6	A=A+B	Wait
7	Upgrade(A)	Wait
8	Wait	Wait
9	Wait	Wait
10	……	……



3、Deadlock Detecting

t	T1	T2	T3	T4
1	L1(A); r1(A)			
2		L2(C);r2(C)		
3			L3(B);r3(B)	
4				L4(D);r4(D)
5		L2(A); wait		
6			L3(C);wait	
7				L4(A);wait
8	L1(B);wait			



4、Deadlock Prevention

- 方法1： Priority Order
  - (按封锁对象的某种优先顺序加锁)
- 方法2：Timestamp
  - (使用时间戳)

4、Deadlock Prevention

- 方法1： Priority Order
  - 把要加锁的数据库元素按某种顺序排序
  - 事务只能按照元素顺序申请锁

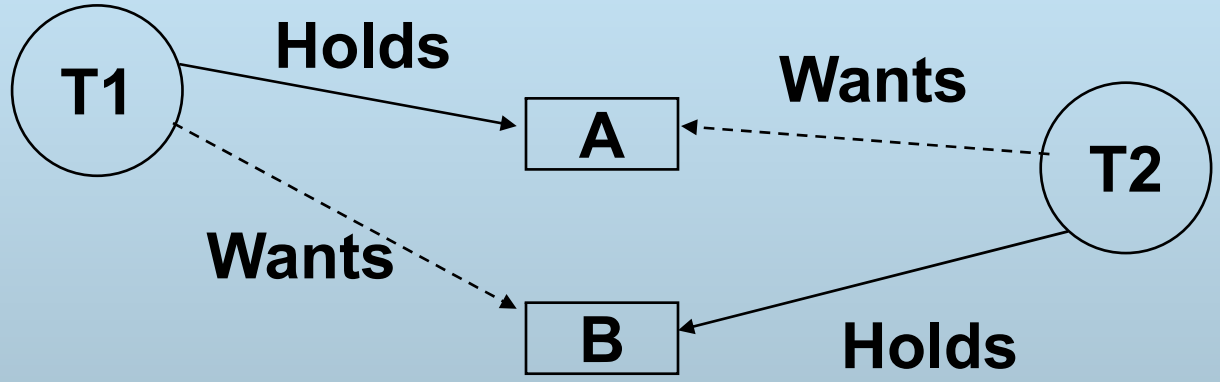
4、Deadlock Prevention

t	T1	T2	T3	T4
1	L1(A); r1(A)			
2		L2(A);wait		
3			L3(B);r3(B)	
4				L4(A);wait
5			L3(C);w3(C)	
6			U3(B);U3(C)	
7	L1(B);w1(B)			
8	U1(A);U1(B)			
9		L2(A);L2(C)		
10		r2(C);w2(A)		
11		u2(A);u2(C)		
12				L4(A);L4(D)
13				r4(D);w4(A)
14				U4(A);U4(D)

T1: A,B
T2: A,C
T3: B,C
T4: A,D

4、Deadlock Prevention

- 按序加锁可以预防死锁



Impossible!
T2获得**B**上的锁之前，必须先要获得**A**上的锁

4、Deadlock Prevention

- 方法2：Timestamp
  - 每个事务开始时赋予一个时间戳
  - 如果事务T被Rollback然后再Restart，T的时间戳不变
  - Ti请求被Tj持有的锁，根据Ti和Tj的timestamp决定锁的授予

4、Deadlock Prevention

- **Wait-Die Scheme 等待－死亡**
  - **T请求一个被U持有的锁**
    - If T is earlier than U then T WAITS for the lock
    - If T is later than U then T DIES 【 *rollback*】
      - We later restart T with its original timestamp

Assumption：
timestamp(T) < timestamp(U) means T is earlier than U

4、Deadlock Prevention

t	T1	T2	T3	T4
1	L1(A); r1(A)			
2		L2(A);DIE		
3			L3(B);r3(B)	
4				L4(A);DIE
5			L3(C);w3(C)	
6			U3(B);U3(C)	
7	L1(B);w1(B)			
8	U1(A);U1(B)			
9				L4(A);L4(D)
10		L2(A);WAIT		
11				r4(D);w4(A)
12				U4(A);U4(D)
13		L2(A);L2(C)		
14		r2(C);w2(A)		
15		u2(A);u2(C)		

4、Deadlock Prevention

- **Wound-Wait Scheme 伤害－等待**
  - **T请求一个被U持有的锁**
    - If T is earlier than U then T WOUNDS U
      - U must release its locks then rollback and restart and the lock is given to T
    - If T is later than U then T WAITS for the lock

4、Deadlock Prevention

t	T1	T2	T3	T4
1	L1(A); r1(A)			
2		L2(A);WAIT		
3			L3(B);r3(B)	
4				L4(A);WAIT
5	L1(B);w1(B)		WOUNDED	
6	U1(A);U1(B)			
7		L2(A);L2(C)		
8		r2(C);w2(A)		
9		u2(A);u2(C)		
10				L4(A);L4(D)
11				r4(D);w4(A)
12				U4(A);U4(D)
13			L3(B);r3(B)	
14			L3(C);w3(C)	
15			U3(B);U3(C)	

4、Deadlock Prevention

- **Comparison**
  - **Wait-Die:**
    - **Rollback总是发生在请求锁阶段，因此要Rollback的事务操作比较少，但Rollback的事务数会比较多**
  - **Wound-Wait:**
    - **发生Rollback时，要Rollback的事务已经获得了锁，有可能已经执行了较长时间，因此Rollback的事务操作会较多，但Rollback的事务数预期较少，因为可以假设事务开始时总是先请求锁**
    - **请求锁时WAIT要比WOUND要更普遍，因为一般情况下一个新事务要请求的锁总是被一个较早的事务所持有**

4、Deadlock Prevention

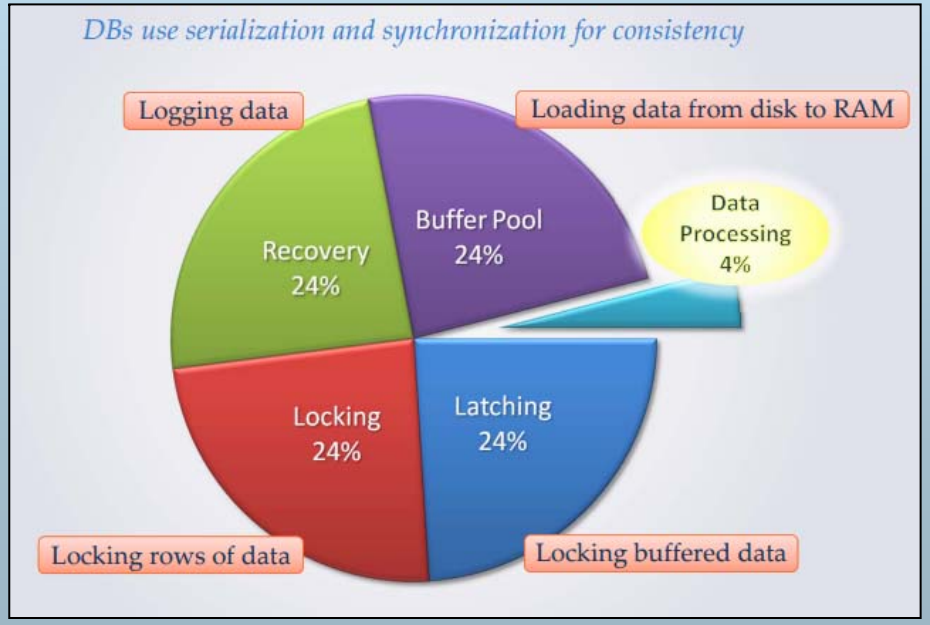
- **Why wait-die and wound-wait work?**
  - 假设 T1→T2 →…… →Tk →T1 【*deadlock*】
  - 在**wait-die scheme**中，只有当**Ti < Tj**时才会有**Ti→Tj**，因此有
    - **T1<T2<…<Tk<T1　-- Impossible!**
  - 在**wound-wait scheme**中，只有当 **Ti > Tj** 时才会有**Ti→Tj**，因此有
    - **T1>T2>…>Tk>T1　-- Still impossible!**

再论“锁机制”

- 两种并发控制思路
  - 悲观并发控制（Pessimistic） --- “悲观锁”
    - 立足于事先预防事务冲突
    - 采用锁机制实现，事务访问数据前都要申请锁
    - 锁机制影响性能，容易带来死锁、活锁等副作用
  - 乐观并发控制（Optimistic） --- “乐观锁”
    - 乐观并发控制假定不太可能（但不是不可能）在多个用户间发生资源冲突，允许不锁定任何资源而执行事务。只有试图更改数据时才检查资源以确定是否发生冲突。如果发生冲突，应用程序必须读取数据并再次尝试进行更改。

乐观并发控制

- 动机
  - **Lock代价太高！如果大部分事务都是只读事务，则并发冲突的概率比较低；即使不加锁，也不会破坏数据库的一致性；加锁反而会带来事务延迟。——Lock-free?**



Source: The White paper of VoltDB (M. Stonebreaker)

乐观并发控制

- 基于事后协调冲突的思想，用户访问数据时不加锁；如果发生冲突，则通过回滚某个冲突事务加以解决
- 由于不需要加锁，因此开销较小，并发度高
- 但需要确定哪些事务发生了冲突
  - 使用“有效性确认(Validation)”

乐观并发控制

- 有效性确认协议
  - 每个更新事务Ti在其生命周期中按以下三个阶段顺序执行
    - 读阶段：数据被读入到事务Ti的局部变量中。此时所有write操作都针对局部变量，并不对数据库更新
    - 有效性确认阶段：Ti进行有效性检查，判定是否可以将write操作所更新的局部变量值写回数据库而不违反可串性
    - 写阶段：若Ti通过有效性检查，则进行实际的写数据库操作，否则回滚Ti

乐观并发控制

- 有效性检查方法（第二阶段）
  - 基于时间戳（行版本）的方式
    - **Timestamp --- MS SQL Server, Oracle**
    - **Version --- MySQL，需要自己通过代码控制**
      - 为表增加一个 **version** 字段（**int**）；
      - 当读取数据时，连同这个 **version** 字段一起读出；
      - 数据每更新一次就将此值加一；
      - 当提交更新时，判断数据库表中对应记录的当前版本号是否与之前取出来的版本号一致，如果一致则可以直接更新，如果不一致则**rollback**

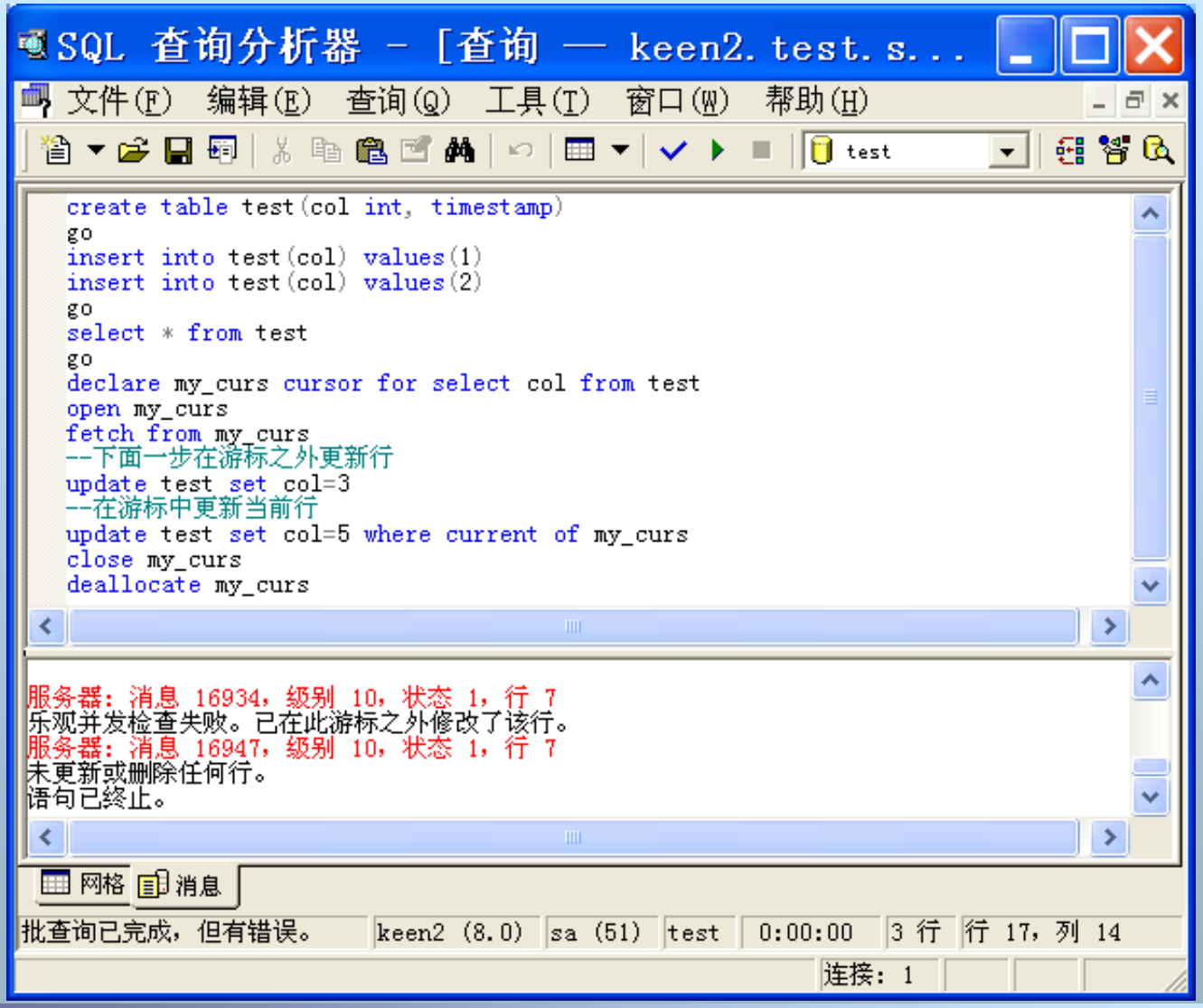
乐观并发控制

- 基于行版本的乐观并发控制（MS SQL Server）
  - MS SQL Server使用特殊数据类型timestamp（数据库范围内唯一的8字节二进制数）
  - 全局变量@@DBTS返回当前数据库最后所使用的时间戳值
  - 如果一个表包含 timestamp 列，则每次由 INSERT、UPDATE 或DELETE 语句修改一行时，此行的 timestamp 值就被置为当前的 @@DBTS 值，然后 @@DBTS 加1
  - 服务器可以比较某行的当前timestamp和游标提取时的timestamp值，确定是否更新

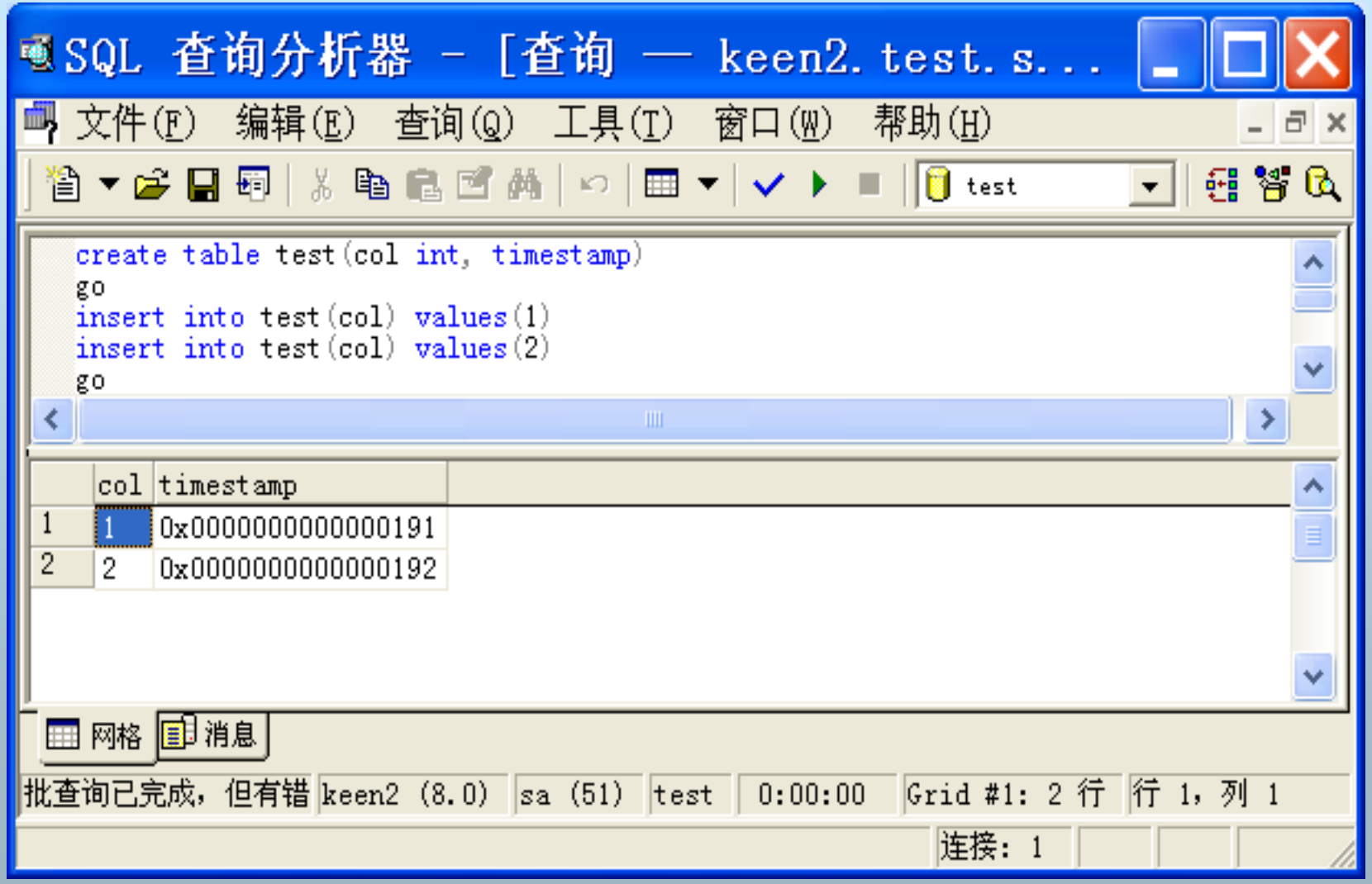
乐观并发控制

- 基于行版本的乐观并发控制
  - 如果表中有timestamp字段，MS SQL Server在游标中会自动采用乐观并发控制
    - 当用户打开游标时，SQL Server保存行的当前timestamp；当在游标中想更新一行时，SQL Server为更新数据自动添加一条Where子句
      - WHERE timestamp列 <= <old timestamp>
    - 如果不相等，则报错并回滚事务

乐观并发控制


SQL 查询分析器 - [查询 — keen2.test.s...
文件(F) 编辑(E) 查询(Q) 工具(T) 窗口(W) 帮助(H)
test
create table test(col int, timestamp)
go
insert into test(col) values(1)
insert into test(col) values(2)
go
select * from test
go
declare my_curs cursor for select col from test
open my_curs
fetch from my_curs
--下面一步在游标之外更新行
update test set col=3
--在游标中更新当前行
update test set col=5 where current of my_curs
close my_curs
deallocate my_curs
服务器：消息 16934，级别 10，状态 1，行 7
乐观并发检查失败。已在此游标之外修改了该行。
服务器：消息 16947，级别 10，状态 1，行 7
未更新或删除任何行。
语句已终止。
网格
消息
批查询已完成，但有错误。
keen2 (8.0)
sa (51)
test
0:00:00
3 行
行 17，列 14
连接：1

乐观并发控制


SQL 查询分析器 - [查询 — keen2.test.s...
文件(F) 编辑(E) 查询(Q) 工具(T) 窗口(W) 帮助(H)
test
create table test(col int, timestamp)
go
insert into test(col) values(1)
insert into test(col) values(2)
go
col timestamp
1 1 0x0000000000000191
2 2 0x0000000000000192
网格 消息
批查询已完成，但有错 keen2 (8.0) sa (51) test 0:00:00 Grid #1: 2 行 行 1，列 1
连接: 1

乐观并发控制 vs 悲观并发控制

- 悲观并发控制：
  - 操作之前先加锁，操作完释放锁，无法避免死锁。适合写频繁的应用场景
- 乐观并发控制：
  - 操作之前不加锁，只有写提交时才检查是否发生了写冲突。适合读频繁的应用场景
  - 在写频繁场景下，冲突频繁，导致大量的写回滚，此时还不如直接加锁

*多版本并发控制MVCC

- 另一种Lock-free的并发控制技术，基于时间戳
- 可以实现无阻塞的读，避免读-写冲突（悲观锁需要各自加锁）
  - 优点：无锁，读永远不用等待；有助于Undo恢复
  - 缺点：空间代价，维护代价，需要定期垃圾回收；写频繁时会出现频繁的事务回滚

Time	Sno	Sname	Age
1	s1	John	23
2	s1	Rose	23
3	s1	Rose	24

T1: Read(最近time), no wait

T2: Update (time++), append a new version
原子更新time

本章小结

- 并发操作问题
- 调度与可串性
  - 可串化调度
  - 冲突可串性及判断
- 锁与可串性实现
  - 2PL
  - 多种锁模式：X、S、U
  - 多粒度锁与意向锁
- 事务的隔离级别
- 死锁