数据库原理

Chp. 17
Transactions

王朝坤
清华大学软件学院
2025年/春

Query Processing

StarfishDB: a Query Execution Engine for Relational Probabilistic Programming

OUAEL BEN AMARA*, University of Michigan-Dearborn, U.S.A.
SAMI HADOUAJ*, University of Michigan-Dearborn, U.S.A.
NICCOLÒ MENEGHETTI, University of Michigan-Dearborn, U.S.A.

We introduce StarfishDB, a query execution engine optimized for relational probabilistic programming. Our engine adopts the model of Gamma Probabilistic Databases, representing probabilistic programs as a collection of relational constraints, imposed against a generative stochastic process. We extend the model with the support for recursion, factorization and the ability to leverage just-in-time compilation techniques to speed up inference. We test our engine against a state-of-the-art sampler for Latent Dirichlet Allocation.

CCS Concepts: • **Information systems** → **Uncertainty**; • **Computing methodologies** → **Statistical relational learning**; **Probabilistic reasoning**; • **Mathematics of computing** → *Gibbs sampling*.

Additional Key Words and Phrases: Probabilistic Programming, Probabilistic Databases, Bayesian Inference

ACM Reference Format:
Ouael Ben Amara*, Sami Hadouaj*, and Niccolò Meneghetti. 2024. StarfishDB: a Query Execution Engine for Relational Probabilistic Programming. *Proc. ACM Manag. Data* 2, 3 (SIGMOD), Article 185 (June 2024), 31 pages. https://doi.org/10.1145/3654988

Query Processing

ABSTRACT

Autonomous database management systems (DBMSs) aim to optimize themselves automatically without human guidance. They rely on machine learning (ML) models that predict their run-time behavior to evaluate whether a candidate configuration is beneficial without the expensive execution of queries. However, the high cost of collecting the training data to build these models makes them impractical for real-world deployments. Furthermore, these models are instance-specific and thus require retraining whenever the DBMS's environment changes. State-of-the-art methods spend over 93% of their time running queries for training versus tuning.

To mitigate this problem, we present the Boot framework for automatically accelerating training data collection in DBMSs. Boot utilizes macro- and micro-acceleration (MMA) techniques that modify query execution semantics with approximate run-time telemetry and skip repetitive parts of the training process. To evaluate Boot, we integrated it into a database gym for PostgreSQL. Our experimental evaluation shows that Boot reduces training collection times by up to 268× with modest degradation in model accuracy. These results also indicate that our MMA-based approach scales with dataset size and workload complexity.

Hit the Gym: Accelerating Query Execution to Efficiently Bootstrap Behavior Models for Self-Driving Database Management Systems

Wan Shen Lim
wanshenl@cs.cmu.edu
Carnegie Mellon University

Lin Ma
linmacse@umich.edu
University of Michigan

William Zhang
wz2@cs.cmu.edu
Carnegie Mellon University

Matthew Butrovich
mbutrovi@cs.cmu.edu
Carnegie Mellon University

Samuel Arch
sarch@cs.cmu.edu
Carnegie Mellon University

Andrew Pavlo
pavlo@cs.cmu.edu
Carnegie Mellon University

PVLDB Reference Format:
Wan Shen Lim, Lin Ma, William Zhang, Matthew Butrovich, Samuel Arch, and Andrew Pavlo. Hit the Gym: Accelerating Query Execution to Efficiently Bootstrap Behavior Models for Self-Driving Database Management Systems. PVLDB, 17(11): 3680 - 3693, 2024.
doi:10.14778/3681954.3682030

Query Optimization

An Elephant Under The Microscope: Analyzing The Interaction Of Optimizer Components In PostgreSQL

RICO BERGMANN, Technische Universität Dresden, Germany
CLAUDIO HARTMANN, Technische Universität Dresden, Germany
DIRK HABICH, Technische Universität Dresden, Germany
WOLFGANG LEHNER, Technische Universität Dresden, Germany

Despite an ever-growing corpus of novel query optimization strategies, the interaction of the core components of query optimizers is still not well understood. This situation can be problematic for two main reasons: On the one hand, this may cause surprising results when two components influence each other in an unexpected way. On the other hand, this can lead to wasted effort in regard to both engineering and research, e.g., when an improvement for one component is dwarfed or entirely canceled out by problems of another component. Therefore, we argue that making improvements to a single optimization component requires a thorough understanding of how these changes might affect the other components. To achieve this understanding, we present results of a comprehensive experimental analysis of the interplay in the traditional optimizer architecture using the widely-used PostgreSQL system as prime representative. Our evaluation and analysis revisit the core building blocks of such an optimizer, i.e. per-column statistics, cardinality estimation, cost model, and plan generation. In particular, we analyze how these building blocks influence each other and how they react when faced with faulty input, such as imprecise cardinality estimates. Based on our results, we draw novel conclusions and make recommendations on how these should be taken into account.

CCS Concepts: • **Information systems** → **Query optimization**.

Additional Key Words and Phrases: Query Optimization; Data Analysis; Cardinality Estimation; Join Enumeration; Cost Model

ACM Reference Format:
Rico Bergmann, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2025. An Elephant Under The Microscope: Analyzing The Interaction Of Optimizer Components In PostgreSQL. *Proc. ACM Manag. Data* 3, 1 (SIGMOD), Article 9 (February 2025), 28 pages. https://doi.org/10.1145/3709659

Query Optimization

ABSTRACT

Query optimizers are a performance-critical component in every database system. Due to their complexity, optimizers take experts months to write and years to refine. In this work, we demonstrate for the first time that learning to optimize queries without learning from an expert optimizer is both possible and efficient. We present Balsa, a query optimizer built by deep reinforcement learning. Balsa first learns basic knowledge from a simple, environment-agnostic simulator, followed by safe learning in real execution. On the Join Order Benchmark, Balsa matches the performance of two expert query optimizers, both open-source and commercial, with two hours of learning, and outperforms them by up to 2.8× in workload runtime after a few more hours. Balsa thus opens the possibility of automatically learning to optimize in future compute environments where expert-designed optimizers do not exist.

Balsa: Learning a Query Optimizer Without Expert Demonstrations

Zongheng Yang
UC Berkeley
zongheng@berkeley.edu

Wei-Lin Chiang*
UC Berkeley
weichiang@berkeley.edu

Sifei Luan*
UC Berkeley
lsf@berkeley.edu

Gautam Mittal
UC Berkeley
gbm@berkeley.edu

Michael Luo
UC Berkeley
michael.luo@berkeley.edu

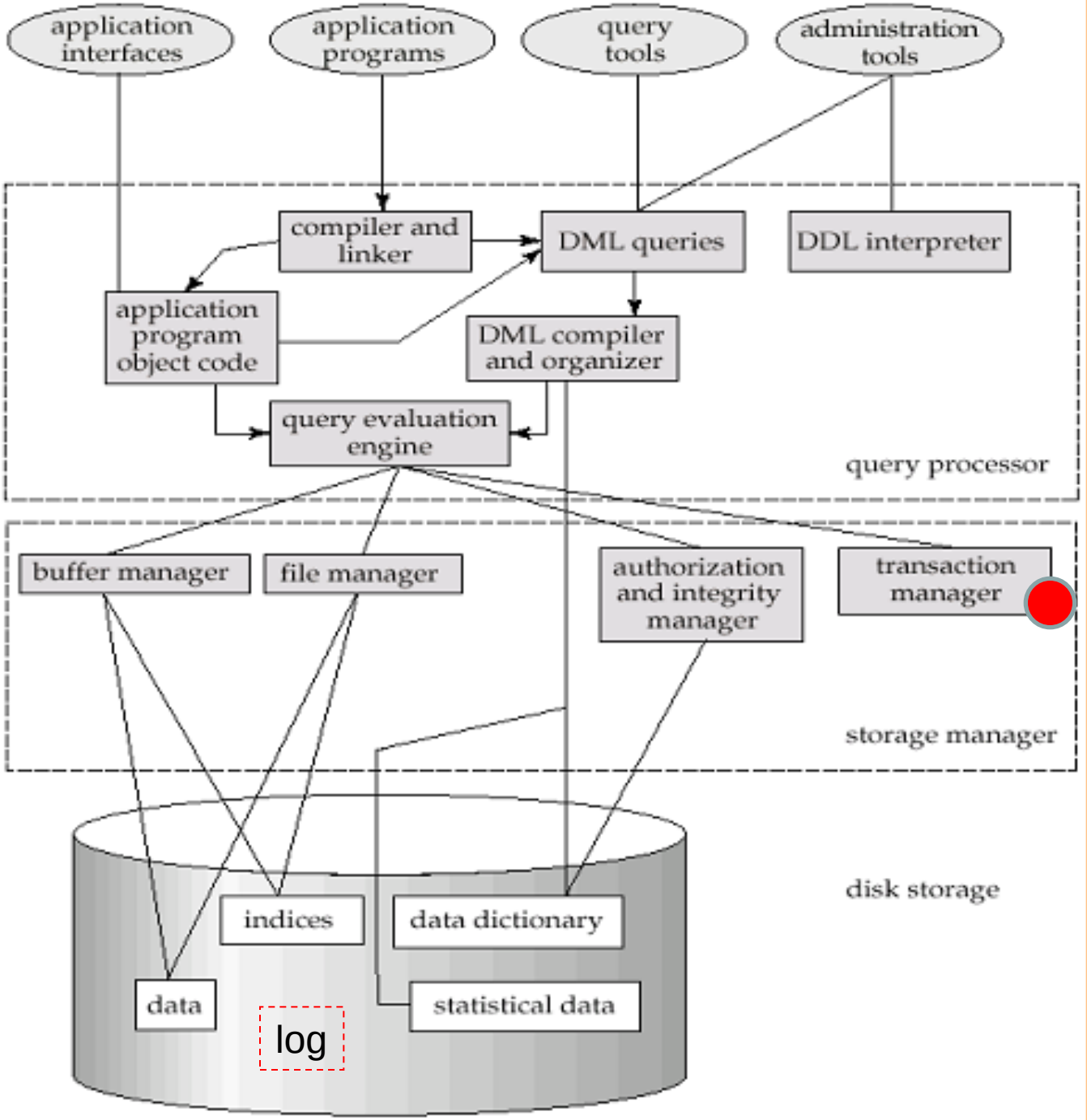
Ion Stoica
UC Berkeley
istoica@berkeley.edu

ACM Reference Format:
Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3517885

本节课的内容

内容	核心知识点	对应章节
DBMS理论	关系数据模型（以及关系演算)、SQL语言	CHP. 1、2、3
DBMS设计	存贮、索引、查询、优化、**事务**、并发、恢复	CHP. 12、13、14、15、16、**17**、18、19
DBMS实现	大作业	小班辅导
DBMS应用	实体-联系图、关系范式、DDL、JDBC	CHP. 4、5、6、7

DBMS Architecture



Outline

1. **Transaction Concept（17.1）**
2. Storage Structure（17.3）
3. Transaction State（17.2、17.4）
4. Concurrent Executions（17.5）
5. Serializability（17.6）
6. Recoverability （17.7）
7. Isolation Levels（17.8、17.9）
8. Transaction Definition in SQL（17.10）

隔离性 - **Isolation**

- **Isolation requirement** — if between steps 3 and 6 (of the fund transfer transaction) , another transaction **T2** is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

T1	**T2**
1. **read**(*A*)	
2. *A* := *A* – 50	
3. **write**(*A*)	
	read(A), read(B), print(A+B)
4. **read**(*B*)	
5. *B* := *B* + 50	
6. **write**(*B*)	

- Isolation can be ensured trivially by running transactions **serially /* 顺序执行**
  - That is, one after the other.

原子性-**Atomicity**

- Consider a transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read(*B*)**
  5. ***B* := *B* + 50**
  6. **write**(*B*)
- **Atomicity requirement**
  - If the transaction fails after Step 3 and before Step 6, money will be “lost” leading to an inconsistent database state
    - Failure could be due to software or hardware
  - The system should ensure that updates of a partially executed transaction are not reflected in the database

持久性- **Durability**

- Consider a transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)

- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

一致性- Consistency

- Consider a transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)
- **Consistency requirement** in above example:
  - Implicit integrity constraints
    - **The sum of A and B is unchanged by the execution of the transaction**
  - Explicitly specified integrity constraints such as primary keys and foreign keys

事务的ACID属性- “酸性”

- **Atomicity/原子性.**
  - 要么都做，要么都不做.
- **Consistency/一致性.**
  - 每个事务的执行不影响数据库的一致性.
- **Isolation/隔离性.**
  - 并发执行的事务互不影响.
- **Durability/持久性.**
  - 事务成功执行后，无论什么情况数据都会存在.

Outline

1. Transaction Concept
2. **Storage Structure**
3. Transaction State
4. Concurrent Executions
5. Serializability
6. Recoverability
7. Level of Isolation
8. Transaction Definition in SQL

可靠存储 – 存放日志

- **易失存储：volatile storage**
  - loses contents when power is switched off
- **非易失存储：non-volatile storage**
  - Contents persist even when power is switched off.
  - Includes secondary and tertiary storage, as well as battery-backed up main-memory.
- **可靠存储：Stable storage**
  - Information residing in stable storage is *never* lost
  - To implement stable storage, we replicate the information in several nonvolatile storage media (usually disk) with independent failure modes
  - **Store the log file to keep track of the old values of any data on which a transaction performs a write**

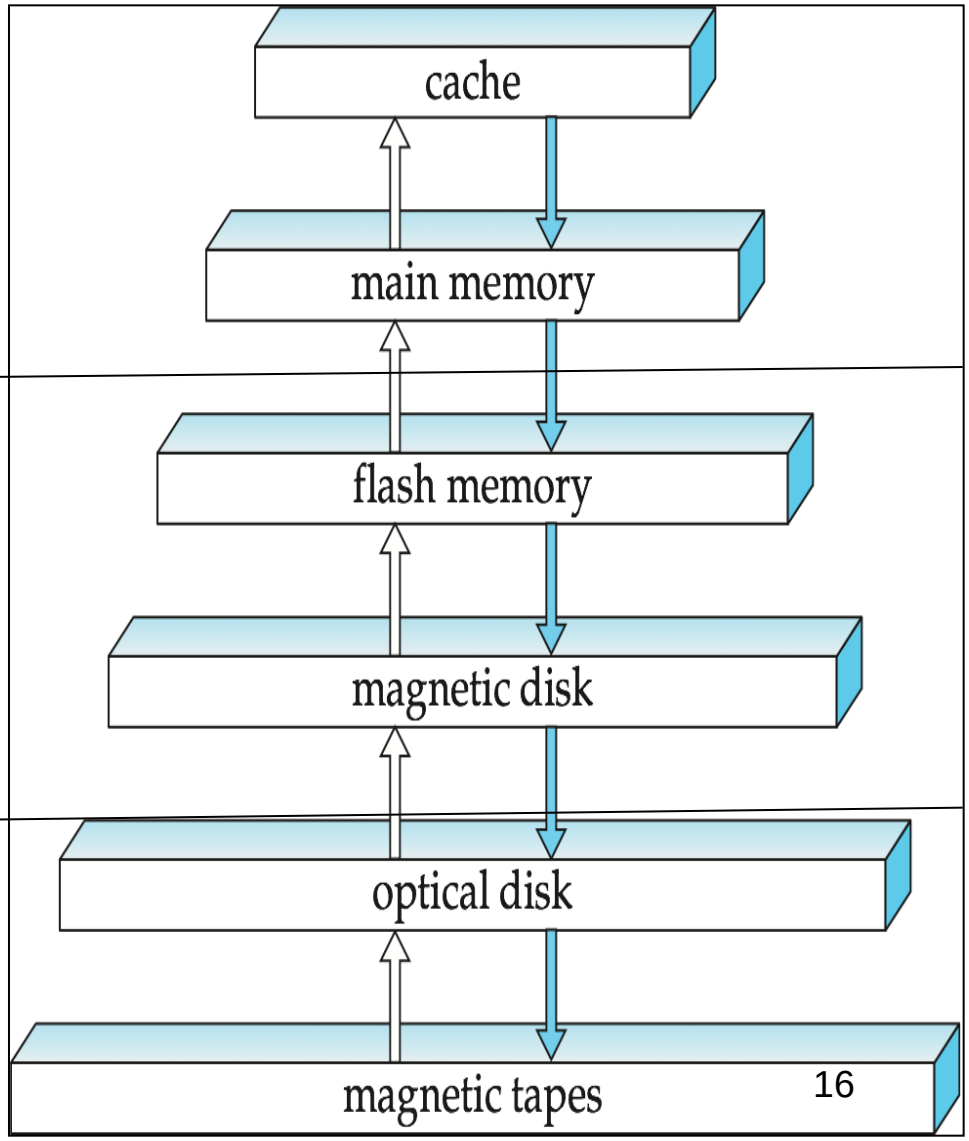
存储的层次结构

主存

在线存储

可靠存储（多副本）

离线存储



Outline

1. Transaction Concept
2. Storage Structure
3. **Transaction State**
4. Concurrent Executions
5. Serializability
6. Recoverability
7. Isolation Levels
8. Transaction Definition in SQL

简化的事务模型

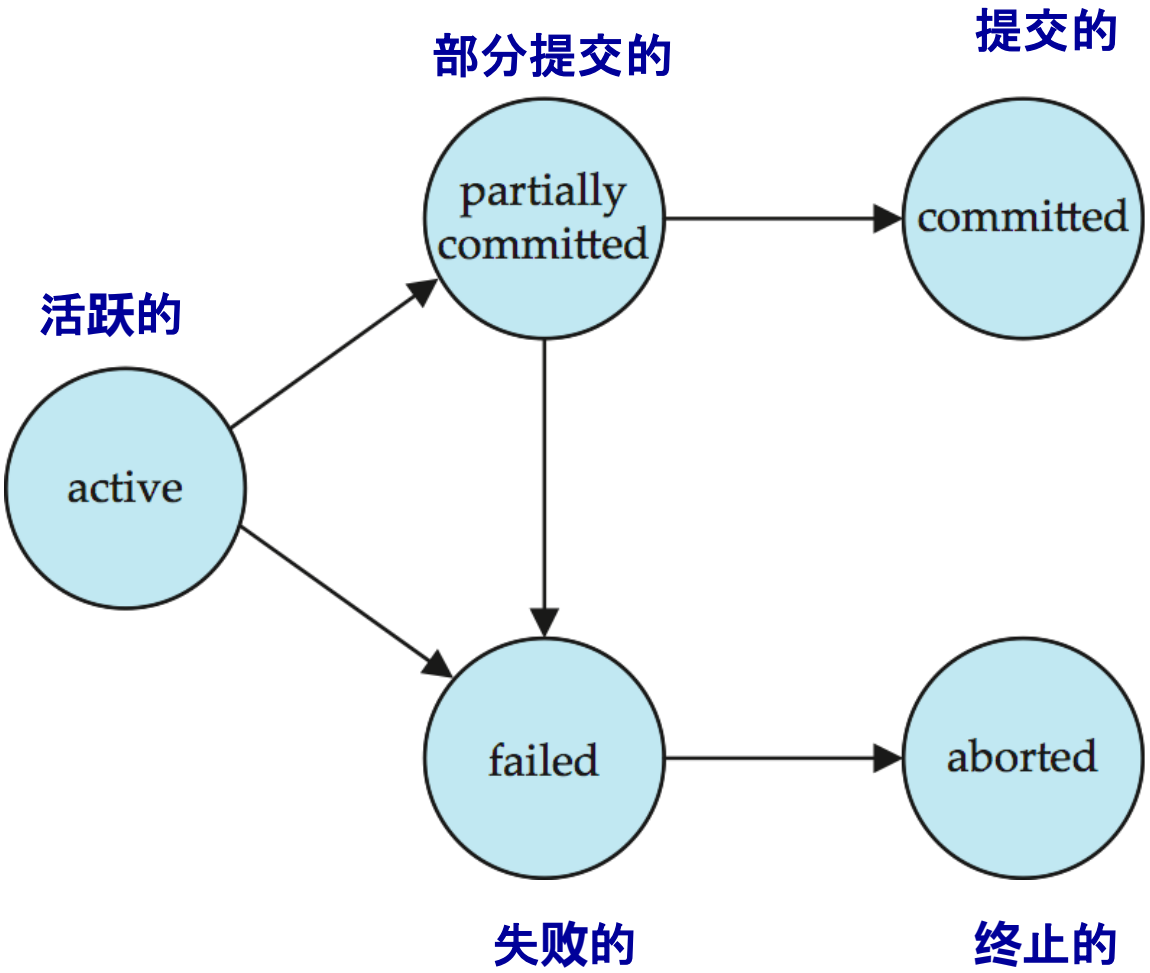
- We begin our study of transactions with **a simple SQL language, where SQL insert and delete operations are ignored.**
- The data items in our simplified model contain a single data value, which are identified by a name ,such as A, B, C, etc.
- Transactions access data using two operations:
  - **read(X),** which transfers the data item X from the database to a variable, also called X
  - **write(X),** which transfers the value in the variable X of the transaction to the data item X in the database

事务的状态

- **Active/活跃的** – the initial state; the transaction stays in this state while it is executing
- **Partially committed/部分提交的** – after the final statement has been executed.
- **Committed/提交的** – after successful completion.
- **Failed/失败的** – after the discovery that normal execution can no longer proceed.
- **Aborted/终止的** – after the transaction has been ***rolled back*** and the database restored to its state prior to the start of the transaction. **Two options after it has been aborted**:
  - Restart the transaction
    - can be done only if no internal logical error
  - Kill the transaction

事务的状态转移图

数据库中事务一般是线程实现的



已经提交的事务如何补偿

- Once a transaction has committed, we **cannot undo** its effects by aborting it.
- The only way to undo the effects of a committed transaction is to execute a **compensating transaction**.
  - Eg. if a transaction added $20 to an account, the compensating transaction would subtract $20 from the account.
- The responsibility of **writing and executing a compensating transaction** is left to the user and is not handled by the database system.
- 补偿事务怎样执行？其责任在用户，不在数据库系统

Outline

1. Transaction Concept
2. Storage Structure
3. Transaction State
4. **Concurrent Executions**
5. Serializability
6. Recoverability
7. Level of Isolation
8. Transaction Definition in SQL

并发执行

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **Increased processor and disk utilization**, leading to better transaction *throughput*
    - E.g. one transaction can be using the CPU while another is reading from or writing to the disk
  - **Reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrent（并发）vs Parallel（并行）**

并发控制机制

- **Concurrency control schemes** – mechanisms to achieve isolation - ACID
  - That is, **to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database**
    - Will study in Chapter 18, after studying notion of correctness of concurrent executions.

调度

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - A schedule for a set of transactions must consist of all instructions of those transactions /* 每个事务内任一指令都不能少
  - Must preserve the order in which the instructions appear in each individual transaction. /* 每个事务内各指令的顺序不变

调度 1 – 第一种串行调度

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.
- An example of a **serial** schedule in which T_1 is followed by T_2 :

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

调度 2 – 第二种串行调度

- A **serial** schedule in which T_2 is followed by T_1 :

T_1	T_2
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B) $B := B + temp$ write (B) commit
read (A) $A := A - 50$ write (A) read (B) $B := B + 50$ write (B) commit	

调度 3 – 与串行调度等价的调度

Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is **equivalent** to Schedule 1.

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

Note -- In schedules 1, 2 and 3, the sum “A + B” is preserved.

调度 4 – 有问题的调度

- The following concurrent schedule does not preserve the sum of "*A* + *B*"

T_1	T_2
read (*A*)	
A := *A* – 50	
	read (*A*)
	temp := *A* * 0.1
	A := *A* - *temp*
	write (*A*)
	read (*B*)
write (*A*)	
read (*B*)	
B := *B* + 50	
write (*B*)	
commit	
	B := *B* + *temp*
	write (*B*)
	commit

如何避免有问题的调度呢?

Outline

1. Transaction Concept
2. Storage Structure
3. Transaction State
4. Concurrent Executions
5. **Serializability**
6. Recoverability
7. Isolation Levels
8. Transaction Definition in SQL

串行调度与串行化

- **Basic Assumption** – Each transaction preserves database consistency. Thus, serial execution of a set of transactions, i.e. **serial schedule**, preserves database consistency.
- A (possibly concurrent) schedule is **serializable** if it is equivalent to a **serial schedule**.
- Different forms of schedule equivalence give rise to the notions of:
  1. **conflict serializability**
  2. **view serializability**

如何避免有问题的调度呢?

哪些是没问题的调度呢?

对事务指令的简化

- We ignore operations other than **read** and **write** instructions
  - **read(*X*),** which transfers the data item *X* from the database to a variable, also called *X*, in a buffer in main memory belonging to the transaction that executed the read operation.
  - **write(X),** which transfers the value in the variable *X* in the main-memory buffer of the transaction that executed the write to the data item *X* in the database.
- Our simplified schedules consist of only **read** and **write** instructions.

指令冲突

- Let I_i and I_j be two Instructions of transactions T_i and T_j respectively. Instructions I_i and I_j are **conflict** if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.

  1. I_i = **read**(Q), I_j = **read**(Q). I_i and I_j don't conflict.
  2. I_i = **read**(Q), I_j = **write**(Q). They conflict.
  3. I_i = **write**(Q), I_j = **read**(Q). They conflict
  4. I_i = **write**(Q), I_j = **write**(Q). They conflict

指令冲突的含义

- Intuitively, **a conflict between I_i and I_j forces a (logical) temporal order between them**.
  - If I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

冲突等价与冲突串行化

- If a schedule *S* can be transformed into a schedule *S´* by a series of swaps of non-conflicting instructions, we say that *S* and *S´* are **conflict equivalent**.

- We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule

调度3和5是冲突等价的

- Schedule 3 can be transformed into Schedule 5 -- a serial schedule where T_2 follows T_1, by a series of swaps of **non-conflicting instructions**. Therefore, Schedule 3 is conflict serializable.

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 3

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 5

调度6-不可冲突串行化

- Example of a schedule that is not conflict serializable:

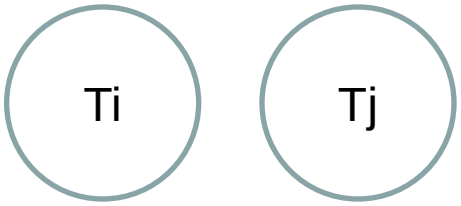
T_3	T_4
read (Q)	
	write (Q)
write (Q)	

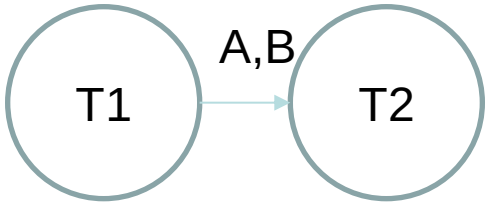
Schedule 6

- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

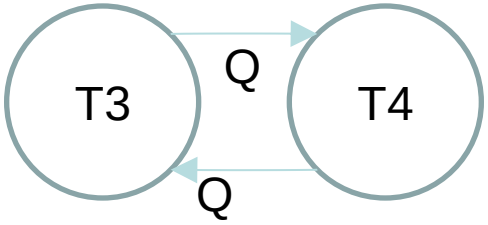
事务的先序图

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from T_i to T_j if the two transactions conflict, and T_i accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.





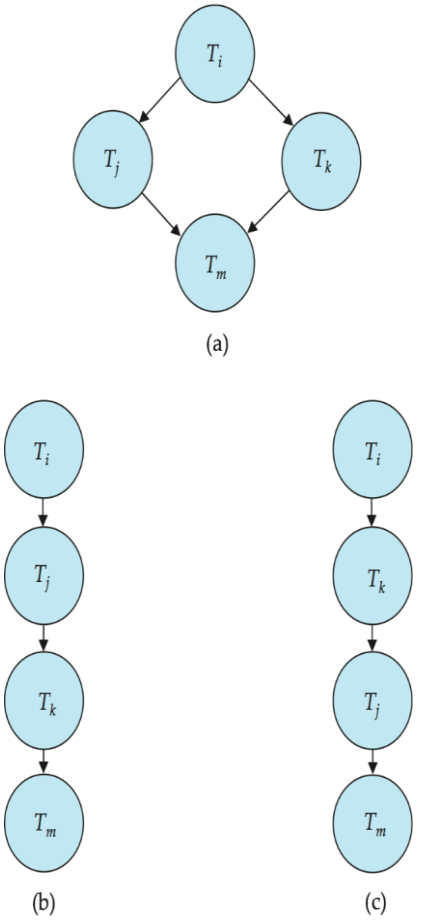
Schedule 3



Schedule 6

利用先序图判断可串行化

- **A schedule is conflict serializable if and only if its precedence graph is acyclic.**
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph.
  - (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a ***topological sorting*** of the graph.
  - That is, a linear order consistent with the partial order of the graph.



视图可串行化

- Let *S* and *S´* be two schedules with the same set of transactions.
- *S* and *S´* are **view equivalent** if the following three conditions are met, for each data item *Q,*
  1. If in schedule S, transaction T_i reads the initial value of *Q*, then in schedule *S'* also transaction T_i must read the initial value of *Q.*
  2. If in schedule S transaction T_i executes **read**(*Q*), and that value was produced by transaction T_j (if any), then in schedule *S'* also transaction T_i must read the value of *Q* that was produced by the same **write**(Q) operation of transaction T_j .
  3. The transaction (if any) that performs the final **write**(*Q*) operation in schedule *S* must also perform the final **write**(*Q*) operation in schedule *S'.*

盲写与视图可串行化

- A schedule *S* is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

Schedule 7

- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has **blind writes.**

视图可串行化判定

- The precedence graph test for conflict serializability cannot be used directly to test for view serializability.
  - Extension to test for view serializability has cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems.
  - Thus, existence of an efficient algorithm is *extremely* unlikely.

调度8—冲突与视图以外的等价调度

- The schedule below produces the same outcome as the serial schedule < T_1, T_5 >, yet is not conflict equivalent or view equivalent to it.

T_1	T_5
read (A)	
$A := A - 50$	
write (A)	
	read (B)
	$B := B - 10$
	write (B)
read (B)	
$B := B + 50$	
write (B)	
	read (A)
	$A := A + 10$
	write (A)

Schedule 8

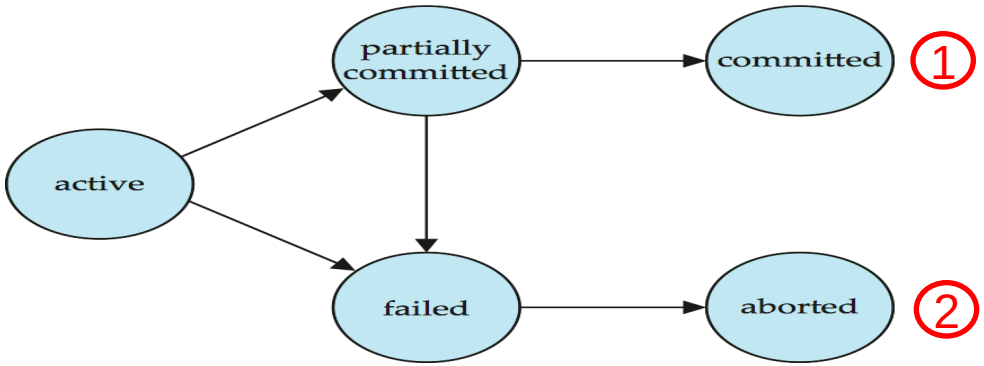
- If we start with A = 1000 and B = 2000, the final result is 960 and 2040
- Determining such equivalence requires analysis of operations other than read and write.

Outline

1. Transaction Concept
2. Storage Structure
3. Transaction State
4. Concurrent Executions
5. Serializability
6. **Recoverability**
7. Isolation Levels
8. Transaction Definition in SQL

提交态与终止态

1. A transaction that successfully completes its execution will have a **commit** instructions as the last statement
   - By default transaction assumed to execute commit instruction as its last step
2. A transaction that fails to successfully complete its execution will have an **abort** instruction as the last statement
   - Once the changes caused by the aborted transaction have been undone, we say the transaction has been **rolled back**



可恢复的调度

- **Recoverable schedule** — if a transaction T_j reads a data item previously written by a transaction T_i, then the commit operation of T_i **must** appear before the commit operation of T_j.

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

Schedule 9

- If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state. Hence, **database must ensure that schedules are recoverable.**

级联回滚的调度

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks.
  - Consider the following schedule where none of the transactions has yet committed (**so the schedule is recoverable**)

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

Schedule 10

If T_{10} fails, T_{11} and T_{12} must also be rolled back.

- **Can lead to the undoing of a significant amount of work**

无级联回滚的调度

- **Cascadeless schedules** — for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, **the commit operation** of T_i appears before the read operation of T_j.
- **Every cascadeless schedule is also recoverable**
- It is desirable to restrict the schedules to those that are cascadeless

Outline

1. Transaction Concept
2. Storage Structure
3. Transaction State
4. Concurrent Executions
5. Serializability
6. Recoverability
7. **Isolation Levels**
8. Transaction Definition in SQL

SQL-92定义的隔离级别

1. **Read uncommitted** — even uncommitted records may be read.
2. **Read committed** — only committed records can be read, but successive reads of record may return different (but committed) values.
3. **Repeatable read** — only committed records to be read, repeated reads of same record must return same value. However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
4. **Serializable** — default

SQL-92定义的隔离级别

隔离级别	脏读	不可重复读	幻读
Read uncommitted-读未提交	V	V	V
Read committed-读提交	X	V	V
Repeatable read-可重复读	X	X	V
Serializable-可串行化	X	X	X

Read uncommitted-读未提交

事务1	事务2
消费50元	消费50元
1.read(*A:100*)	
2.A := *A* – 50	
3.write(*A:50*)	
	1.read(*A:50*)
	2.A := *A* – 50
4.Rollback	

Nonrepeatable read-不可重复读

用户甲	用户乙
消费50元	消费50元
1.read(*A:100*)	
	1.Read(*A:100*)
	2.A:=A-50
	3.write(*A:50*)
	4.commit
2.read(*A:50*)	

Phantom read -幻影读

事务1	事务2
查询账户	增加账户
1.select(*B<100*)	
-(01,’BJ’, 90)	
	1.Insert
	-(*03,’BJ’,50*)
2.select(*B<100*)	
-(01,’BJ’, 90)	
-(*03,’BJ’,50*)	

隔离级别如何实现

1. Locking
   - Lock on whole database vs lock on items
   - How long to hold lock?
   - Shared vs exclusive locks
2. Timestamps
   - Transaction timestamp assigned e.g. when a transaction begins
   - Data items store two timestamps
     - Read timestamp
     - Write timestamp
   - Timestamps are used to detect out of order accesses
3. Multiple versions of each data item
   - Allow transactions to read from a “snapshot” of the database

Outline

1. Transaction Concept
2. Storage Structure
3. Transaction State
4. Concurrent Executions
5. Serializability
6. Recoverability
7. Isolation Levels
8. **Transaction Definition in SQL**

SQL事务语句

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
  - **Begin/start transaction**
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off
    - E.g. in JDBC, connection.setAutoCommit(false);

Chp. 18
Concurrency Control

王朝坤
清华大学软件学院
2025年/春

Main Contents

1.Lock-Based Protocols（18.1）

2.Deadlock Handling（18.2）

3.Multiple Granularity（18.3）

4.Insert and Delete Operations（18.4）

5.Timestamp-Based Protocols（18.5）

6.Validation-Based Protocols（18.6）

7.Multiversion Schemes（18.7）

基于锁的机制

A lock is a mechanism to control concurrent access to a data item

Data items(*tuple, page, table etc*.) can be locked in two modes

1. *exclusive (X) mode*. /* 排他锁

   Data item can be read as well as written.

   X-lock is requested using **lock-X** instruction.

2. *shared (S) mode*. /* 共享锁

   Data item can only be read.

   S-lock is requested using **lock-S** instruction.

Lock requests are made to lock manager.

锁相容性矩阵

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.

  -If a lock cannot be granted, ***the requesting transaction*** **is made to wait** till all incompatible locks held by other transactions have been released. The lock is then granted.

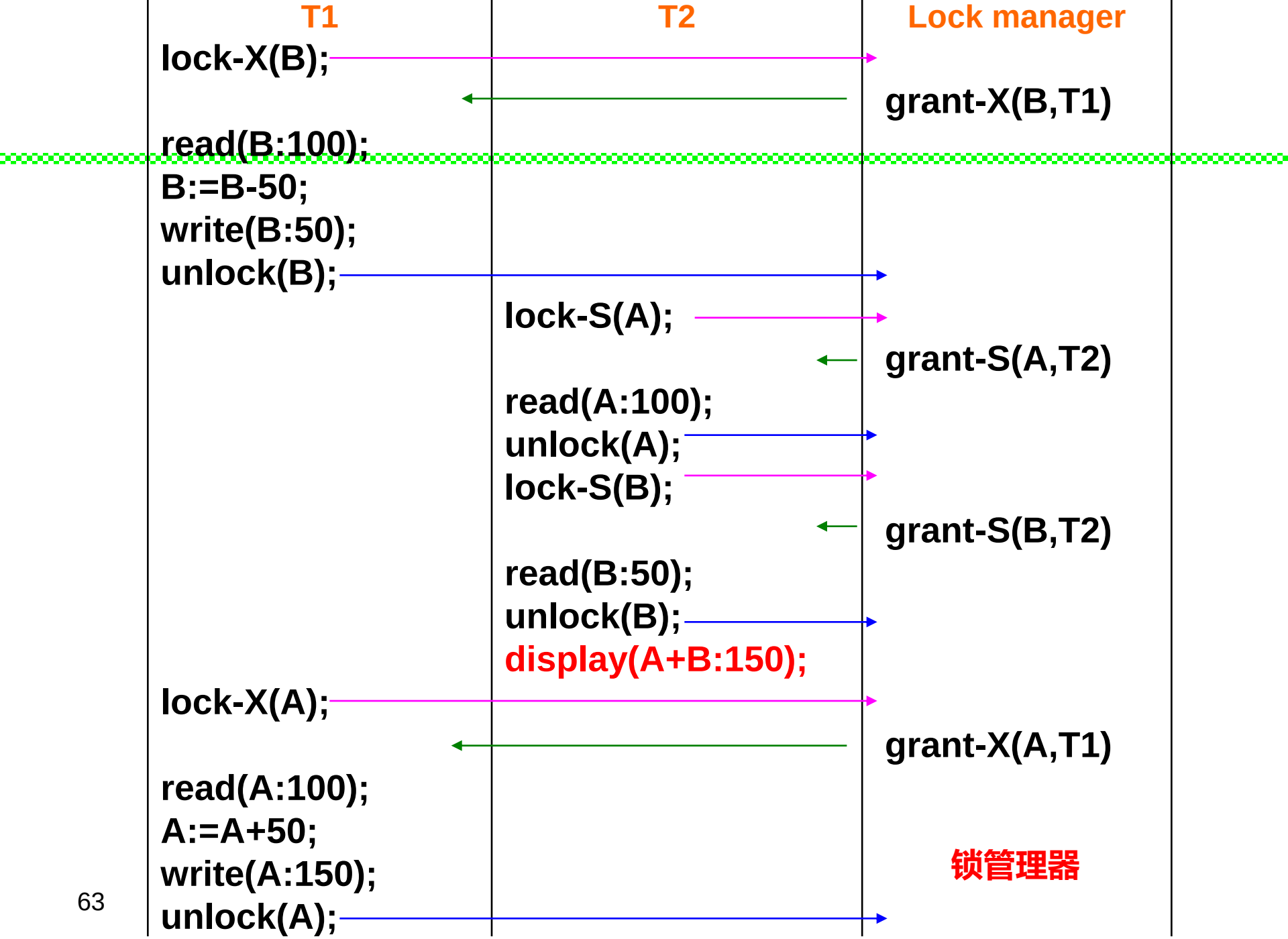
简单的加锁不可行

T1 - 转账

```
lock-X(B);
read(B);
B:=B-50;
write(B);
unlock(B);
lock-X(A);
read(A);
A:=A+50;
write(A);
unlock(A);
```

T2 - 求和

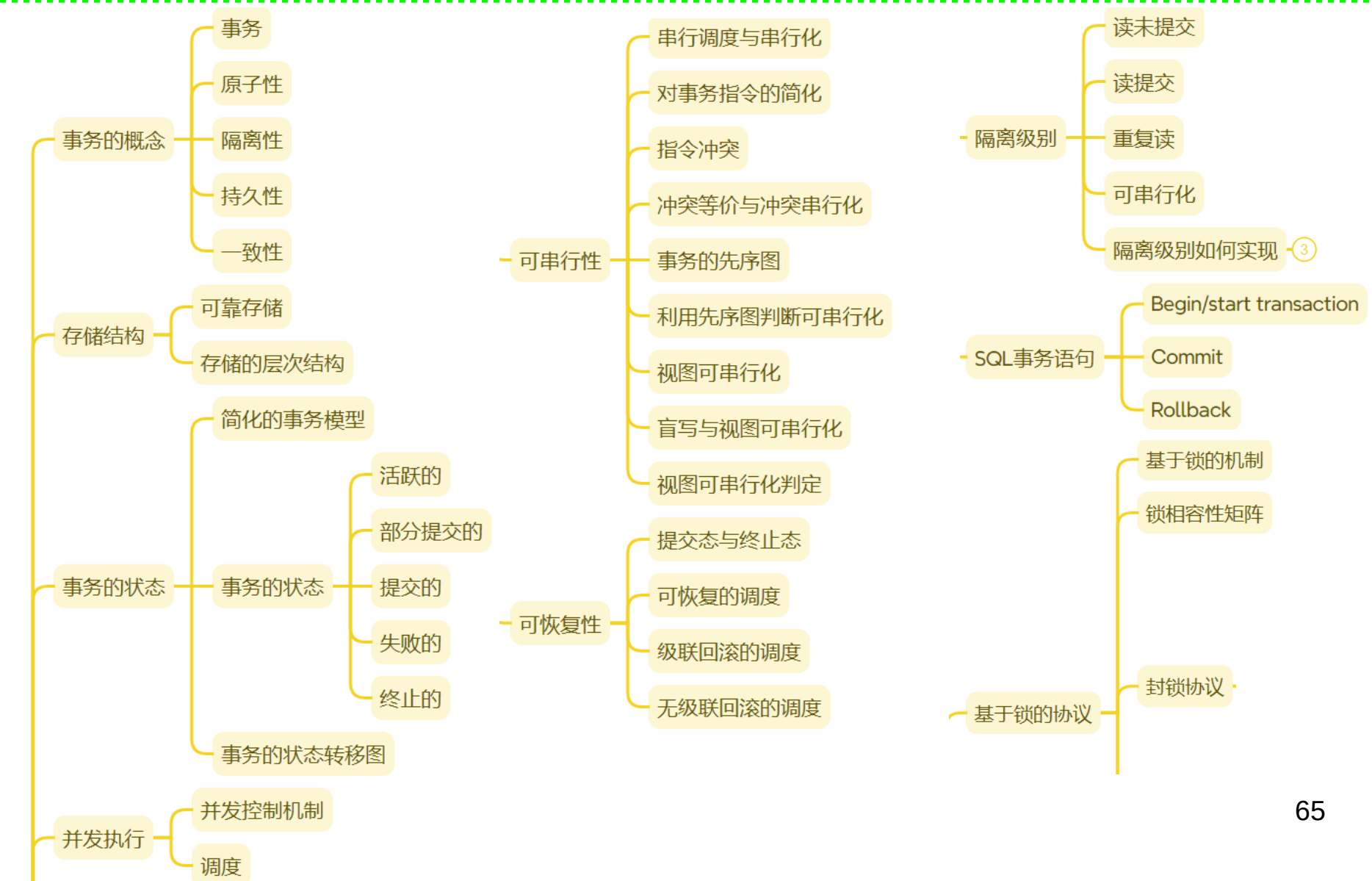
```
lock-S(A);
read(A);
unlock(A);
lock-S(B);
read(B);
unlock(B);
display(A+B);
```


T1
T2
Lock manager
lock-X(B);
grant-X(B,T1)
read(B:100);
B:=B-50;
write(B:50);
unlock(B);
lock-S(A);
grant-S(A,T2)
read(A:100);
unlock(A);
lock-S(B);
grant-S(B,T2)
read(B:50);
unlock(B);
display(A+B:150);
lock-X(A);
grant-X(A,T1)
read(A:100);
A:=A+50;
write(A:150);
unlock(A);
锁管理器

封锁协议

- A locking protocol is a set of rules followed by all transactions while requesting and releasing locks. /* 加解锁遵循的规则
- Locking protocols restrict the set of possible schedules. /* 得到了相关类别的调度

Conclusions



Thanks