2021 年春季

分布式数据库课程报告

课程教师：卢 暾

目 录

姓名：陈 霁 璇

学号：20210240341

导师：谈子敬

2021 年 8 月 10 日

论文报告与总结

论文 PPT

2021-8-13

TARDiS:A Branch-and-Merge Approach To Weak Consistency

事务异步复制的不同存储 弱一致性的分支与合并方法

-陈霁霞

背景与挑战

一个应用可以分为两部分A 和 B， A 是业务逻辑处理系统 B 是底层的存储系统 本文提出了一个针对弱一致性系统设计的事务型键值对存储系统TARDIS，TARDIS是B类系统，为具有 ALPS特性的 A 类系统提供底层支持。

关于弱一致性系统的推理很难 因为因果一致性和每个对象的最终收敛都不允许应用程序满意地处理冲突。

根据CAP理论，很多大范围服务和应用选择了放弃强一致性来换取PS（可用性，低延迟，网络延迟容忍 高扩展性）。

很多系统通过因果一致性(针对读写冲突) 和单对象最终收敛(针对写写冲突) 技术的组合来将应用程序与冲突隔离开

当冲突出现时的解决方法有两种 在存储层（通过确定的解决机制强制单对象收敛，或交给应用程序通过冲突更新解决对象的状态

但是这些技术不能解决同时更新时顺序存储的抽象概念，并且因果顺序和单对象收敛不能解决针对多对象的同时顺序更新。 而且，这些技术推毁了应用程序可以用来解决错误的信息

介绍

TARDIS的设计基于一个简单的概念 帮助开发人员解决这些应用程序中出现的异常，每个副本应该记录完整的上下文 以便了解异常是如何发生的 但是只在需要的时候才向应用程序暴露异常

TARDIS是异步复制的、多master节点，事务型键值存储系统 是为构建在弱一致性和非事务型系统的应用设计的

创新与贡献

TARDIS将弱一致性系统中出现的冲突->分支集作为其基本抽象公开 提出了一种新的并发控制机制 冲突->分支，它的存储好于扩展分支的任何执行线程来说都是顺序的保证了应用程序逻辑简单 还保证了应用程序随时按需自动合并分支

TARDIS在冲突时新建分支比较快 弱一致性系统可以采用这种方式来处理不同客户端的操作以及本地冲突操作。TARDIS会降低了这类系统的代码复杂度 可以增加客户端吞吐量-8倍。

其他的分支系统也有的暴露了分支如git、Olive文件系统，但是他们暴露的分支太明确了 信息太多，增加了合并的复杂度 这给基ALPS应用程序希望避免的 TARDIS提供的分支概念明比较简单 容易进行合并

还有一类因果一致系统 将写冲突直接暴露给应用 其实是将单个对象的多个值暴露给应用这类系统没有提供跟踪冲突或分支的功能 将多个值暴露给应用增加了复杂度 在TARDIS中，开发人员只在合并模式需要时才处理一个对象的多个值 便于掌控。

特性

TARDIS有四个特性：

（1）系统保留历史信息 每个节点存储了一个记录所有操作产生的分支的有向无环图并且使用了新的DAG压缩算法。这些保存的上下文环境会在编程时带来好处

（2）合并分支，不是对象。TARDIS提出了一种跟踪冲突的概念 通过将分支总结为一个分歧点和合并点的集合 这减少了使用因果一致性系统遭遇的元数据负载

（3）良好的表现性 TARDIS支持很多隔离层次(可串行性，快照隔离，确定的读）和一致性保证（读我所写，因果一致性）。

（4）提高了本地节点的性能 TARDIS为ALPS应用程序提供了将弱一致性准则应用在端到端，通过跟踪不同节点和本地冲突操作产生的冲突分支当这样配置时 TARDIS处理本地冲突并不通过终止回滚和锁，而是逻辑上将本地数据库分支 这一特性并不对所有应用有利 但是对TARDIS针对的那种ALPS应用程序（弱一致性和合并时第一考虑因素），可以利用这一特性显著增加吞吐率

核心技术点

TARDIS的设计理念：将ALPS应用程序与各自独立的写冲突以及应用需要的解决过程隔离开

当冲突产生时 TARDIS提供了两个新功能来缓解

（1）将结果独立的分支和这个分支产生的状态集合(fork点和merge点）暴露给应用程序

（2）提供原子级合并分支 并且允许应用程序选择when和how来合并。

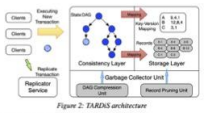

Figure 2: TARDiS architecture

每个副本包含四个部分

（1）存储层：将记录存储在基于磁盘的B-Tree。每个节点存储了一个完整的数据库的副本（TARDIS也可扩展支持数据分区），每个更新型族都会创建一个新的记录版本版本和key的mapping存储在内存cache中，便于快速遍历

（2）一致性层：通过一个DAG跟踪分支，顶点是一个数据库的逻辑状态 每个更新记录的事务产生一个新的状态

（3）垃圾回收单元 包含一个DAG压缩子模块 和一个记录剪枝子模块 用来控制空间复杂度。

评估

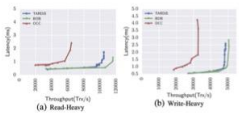

Figure 9: TARDiS-BDB vs BerkeleyDB vs OCC

对比负载，比较了全读和全写性能，基本与BerkeleyDB一致，比OCC好，因为OCC要确认只读事务，而TARDIS的确认阶段（提交状态检查）消耗小，只需要检查事务的写集合是否是选择的读状态的子集。

接着对比不同吞吐率时的负载 在集群上模拟提交请求分别对TARDiS、BerkeleyDB和OCC三种系统在读密集写密集、符合zipf分布的写密集、标准的忙写情况下不同吞吐率的负载情况 在数据库中对象访问符合zipf分布时，由于竞争激烈 TARDiS性能较差，TARDiS的lock free实现的写跳转链表保证写不会阻塞即使3个实时）

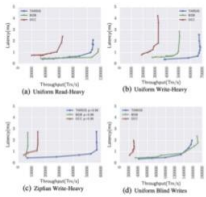

Figure 10: Benefit of branching as a function of workload

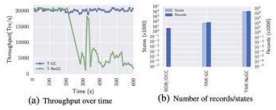

Figure 13: Impact of garbage collection

垃圾回收
垃圾回收使得吞吐率维持在平稳水平 而不进行垃圾回收则会因为java的垃圾回收机制使得系统停顿，而状态和记录个数上 回收是不回收的3%，但是是BDB的100倍，尽管状态在变成垃圾可回收状态时被清除掉 但是状态提升表首先要被刷磁盘 在这之前要运行一遍记录版本提升过程），这些记录一般都不会在cache中，所以记录提升线程滞后 这个是缺点

个人总结

通过显式跟踪并发分支并在需要时将它们公开给应用程序TARDiS简化了冲突解决

通过为应用程序提供端到端应用弱一致性原则的选项TARDiS可以显着提高本地站点的性能。

TARDiS提供了两类约束 begin约束和end约束，合理的选择不同的约束搭配可以模拟出传统数据库的各种隔离级别 但是如何根据需求选择搭配不清楚

但是TARDiS的副本不支持分布式事务 是它的缺点

论文总结

TARDiS:A Branch-and-Merge Approach To Weak Consistency

前提：我们把一个应用分为两部分 A 和 B， A 是业务逻辑处理系统， B 是底层的存储系统。TARDIS 是 B 类系统，为具有 ALPS （availability，low latency，partition tolerance， scalability）特性的 A 类系统提供底层支持。

1. 论文解决了什么问题，从 business problem 到 technical problem 。

Business problem： 很多 A 系统的数据存储都是依赖 BerkeleyDB 或其他现有的数据库系统的，这些数据库系统将 A 系统与数据冲突隔离开，B 系统需要自己实现全局弱一致性的保证措施和解决冲突的机制，使得 B 系统的实现比较复杂，而且由于 B 系统要解决冲突，吞吐率也会受到限制。反应到应用层面就是不能有效的支持竞争增多的情况。

Technical problem： B 类系统中有两个比较难解决的问题：（1）写写冲突，很多系统采用一套固定的解决策略，如保存最后写成功的值。但是不能完全符合 A 系统的真实想法，而且没有给 A 系统解决冲突的充分信息。而 A 系统在不同的应用场景下可能有不同的需要。（2）缺乏跨对象语义，无法解决两个对象间的语义冲突问题（A （人）和 B （人）对同一个对象 a 的不同副本进行了修改，这两个事务完成后， C （人）和 D （人）分别根据 a 的这两个不同副本的两种语义对其他对象 c 和 d 进行了修改，这时 c 和 d 就有了语义冲突，但是正常合并时只能检查出 a 的冲突并进行冲突解决，不能检查出 c 和 d 有了冲突，这样就会出现对象间的语义冲突)

2. 论文的问题是不是成立的？有哪些地方是 solid 的的，有哪些地方其实应用场景是存在疑问的。

论文的问题是成立的

solid 的地方：

（1）现有的大多数 B 系统没有保留用以解决冲突的上下文（在解决冲突后就扔掉了）。而且 B 系统无法根据实际需求来处理冲突，只能根据系统逻辑判断。

（2） B 类系统在遇到事务之间冲突时，需要先解决冲突，使数据保持一致状态，再进行下一批事务的执行。这样就会等待。吞吐率会受到影响，如果不解决冲突，直接在不同的分支写数据可以大大提升吞吐率。

存在疑问的地方：

(1) TARDIS 提供了两类约束，begin 约束和 end 约束，合理的选择不同的约束搭配可以
模拟出传统数据库的各种隔离级别。但是如何根据需求选择搭配不清楚。

(2) TARDIS 解决的是为 A 类系统提供解决冲突所需信息的问题。如果这个冲突指的是多节点间副本的一致性，那么这个冲突由谁感知？状态的 DAG 存在哪个节点？如果这个冲突指的单节点的并发操作时产生的冲突，那么 TARDIS 解决就是单节点的并发控制问题。这样的话如何解决多节点间的数据不一致问题？

(3) TARDIS 针对的是数据在各个节点间全备份的情况，针对不同节点间保存数据的不同部分（如 hdfs ）如何检测冲突并创建分支？

3. 如果我沿着这个大方向研究下去，我会对这个问题做一些什么样的改变，或者衍生出一些什么样的新问题，问题和原有问题差异在哪。

我理解 TARDIS 是解决单个节点高并发情况下如何解决写数据冲突的问题，那么如何解决不同节点间数据冲突的问题？比如 A 节点写 a 为 1，B 节点写 a 为 2。这样如何感知冲突并在不同节点间维护状态 DAG 图？

4. 现有的技术是什么，为什么不能很好地解决这个问题。

其他的分支系统也有的暴露了分支，如 git，Olive 文件系统。但是他们暴露的分支太明确了，信息太多，增加了合并的复杂度，这恰是 ALPS 应用程序希望避免的。TARDIS 提供的分支概念则比较简单，容易进行合并。

还有一类因果一致系统，将写冲突直接暴露给应用。其实是将单个对象的多个值暴露给应用。这类系统没有提供跟踪冲突或分支的功能，将多个值暴露给应用增加了复杂度。在 TARDIS 中，开发人员只在合并模式需要是才处理一个对象的多个值，便于掌控。

在拜占庭问题上，SUNDR 和 FAUST 开发了分支的一致性和线性来避免客户端看到错误客户端的不同的值。Depot 系统扩展了 SUNDR 的模型来支持加入分支，但是 Depot 没有暴露分支的抽象概念，也不支持跨对象原子合并。Sporc 系统支持原子合并分支，但是只针对有限的情况（协同文本应用），并且使用操作变换来解决冲突。

5. 作者采取的方法是什么。和 baseline 或 pre-art 相比，理论上为什么会好？实验是怎么做的？实验结果是否成立（比如实验集群的规模过小…）？

作者认为每一个副本应该完整地记录描述一个冲突是如何产生的所有信息，但是只在上层应用需要地时候再暴露出去。而且将上层应用与底层冲突隔离不是一个好的方式。

TARDIS 是一个为弱一致性系统提供支持的存储系统 B，为 A 系统提供了很好的解决端到端冲突的方法。

每个节点都保存了全部数据副本。每个副本包含四个部分：

（1）存储层次：将记录存储在基于磁盘的 B-Tree。每个节点存储了一个完整的数据库的副本。每个更新操作都会创建一个新的记录版本，版本和 key（记录）的 mapping 存储在内存 cache 中，便于快速遍历。

（2）一致性层次：通过一个 DAG 跟踪分支，顶点是一个数据库的逻辑状态。每个更新记录的事务产生一个新的状态。

（3）垃圾回收单元，包含一个 DAG 压缩子模块，和一个记录剪枝子模块。用来控制空间复杂度。

（4）复制服务器：传播提交的事务，将远程的事务正确应用。

TARDIS 在内存中维护了一个状态 DAG。在遇到写冲突时会建立新的状态分支，每个写操作都会产生一个新的状态，并且用一些压缩算法来保证一定的空间复杂度。同时将具体数据按 key-version set 存储为 map 结构，每个记录都对应一个按版本新旧顺序连接的链表，存储多个版本的数据，这样可以快速找到一个分支上的所有的数据的值。

为 A 系统提供合并分支需要的所有信息（如冲突时的分支点，某一条分支路径上的所有冲突对象等）。使得 A 系统可以自己定义分支合并规则。简化了 A 系统的复杂度。

在冲突时建立分支而不是等待冲突解决使得在写数据竞争激烈时本地吞吐率比其他系统要大很多，理论上是正确的。

实验做法：

（1）首先对比了实现 ALPS 系统的复杂度。分别基于 TARDIS 和 BerkeleyDB 实现了几个 CRDTs（一致可复制数据类型）和 Retwis（Twitter 的简化版），对比

了他们的代码行数和吞吐率。TARDIS 的代码是 BerkelyDB 的一半到三分之一。复杂度大大降低。吞吐率提高了 2 到 8 倍。

(2) 对比负载：比较了全读和全写性能，基本与 BerkeleyDB 一致，比 OCC 好，因为 OCC 要确认只读事务，而 TARDIS 的确认阶段（提交状态检查）消耗小，只需要检查事务的写集合是否是选择的读状态的子集。

(3) 接着对比不同吞吐率时的负载，在集群上模拟提交请求。分别对 TARDIS、BerkeleyDB 和 OCC 三种系统在读密集、写密集、符合 zipf 分布的写密集、标准的忙写情况下不同吞吐率的负载情况。在数据库中对象访问符合 zipf 分布时，由于竞争激烈，TARDIS 性能较好。TARDIS 的 lock free 实现的写跳转链表保证写不会阻塞（即使冲突时）。

在 zipf 分布时，由于竞争激励（加锁），BDB 的读写比读密集或写密集时慢十倍，TARDIS 只慢了一点。OCC 表现最好，因为 OCC 保证了 (1) 至少一个事务总会被提交 (2) 读不会阻塞写。但是 OCC 的高召回率和复杂的提交过程使其吞吐率是 TARDIS 的 1/5。

在一致的写数据时，由于锁发生的几率小，TARDIS 的垃圾回收速度不能跟上状态的产生。增加的内存使 TARDIS 的性能比 BDB 差 10%。

实验结果是成立的是对比实验，对比的内容比较有代表性而且比较大众。

6. 作者为什么会想出这样的方法?

因为作者在做存储系统尝试解决并发产生的冲突时总觉得少了一些信息，而这些信息只有上层应用才能获取到，这些信息就是实际需求。所以为什么不把冲突保存下来留给上层应用去解决呢？就像 git 一样，人工处理冲突。

7. 作者在解决这个问题中是不是尝试建立了一些假设或者约束去简化问题，或者尝试把读者从一个问题引导到另一个问题。

在开头举的例子是两个人，对一个网页的不同副本进行修改，之后又来了两个人，在前面的改动的基础上继续改动，由此引发的单对象多值冲突和多对象间语义冲突。但是后面解决的问题都是单节点的冲突。并没涉及多节点间冲突如何检测和解决。

8. 作者的方法有什么缺陷或者可改进的空间。

在数据存储类型上有可能进行优化，在多节点冲突时的分支状态产生。

9. 如果我来仅仅从方法层面做接下来的工作，我会选择什么方向，为什么。

本文方法是记录系统演进状态并为上层应用提供简单的获得所需信息的接口，思想是存储系统不解决冲突，交给上层应用利用专业知识合并冲突分支，类似 git。

方法上的改进主要针对数据分区存储时跨节点的冲突检测和状态分支。以及更优秀的垃圾回收机制来减少因保存状态 DAG 图和多个数据库版本带来的空间消耗。

思想上没有什么问题。这个思想挺好，甩锅。

项目技术报告

项目由我一个人完成

事务管理器 Transaction Manager及相关资源类的实现

- 事务管理器是分布式事务的核心管理者。事务管理器与每个资源管理器（resource manager）进行通信，协调并完成事务的处理。事务的各个分支由唯一命名进行标识
  - 使用了2个map来事务id与事务状态、资源管理器的对应关系

```
//xid -> trasactional status
Map<Integer, String> xidStatusMap;
```

```
//xid -> relevant ResourceManagers
Map<Integer, Set<ResourceManager>> enlistRMs;
```

  - ResourceManager 调用enlist()函数，用来通知TransctionManager哪个ResourceManager参与了哪个事务

```
  public void enlist(int xid, ResourceManager rm) throws RemoteException,
InvalidTransactionException {
    if (!this.xidStatusMap.containsKey(xid)) {
      rm.abort(xid);
      return;
    }

    synchronized(this.xidStatusMap) {
      String status = this.xidStatusMap.get(xid);
      if (status.equals("ABORTED")) {
        rm.abort(xid);

        synchronized(this.enlistRMs) {
          Set<ResourceManager> rms = enlistRMs.get(xid);
          ResourceManager tmp = rms.iterator().next();
          rms.remove(tmp);
          if (rms.size() > 0) {
            enlistRMs.put(xid, rms);
            this.storeToFile(TM_TRANSACTION_RMs_LOG_FILENAME, enlistRMs);
          }
          else {
            enlistRMs.remove(xid);
            this.storeToFile(TM_TRANSACTION_RMs_LOG_FILENAME, enlistRMs);

            this.xidStatusMap.remove(xid);
            this.storeToFile(TM_TRANSACTION_LOG_FILENAME, xidStatusMap);
          }
        }
        return;
      }
      else if (status.equals("COMMITED")) {
```

```
        rm.commit(xid);

        synchronized(this.enlistRMs) {
          Set<ResourceManager> rms = enlistRMs.get(xid);
          ResourceManager tmp = rms.iterator().next();
          rms.remove(tmp);
          if (rms.size() > 0) {
            enlistRMs.put(xid, rms);
            this.storeToFile(TM_TRANSACTION_RMs_LOG_FILENAME, enlistRMs);
          }
          else {
            enlistRMs.remove(xid);
            this.storeToFile(TM_TRANSACTION_RMs_LOG_FILENAME, enlistRMs);

            this.xidStatusMap.remove(xid);
            this.storeToFile(TM_TRANSACTION_LOG_FILENAME, xidStatusMap);
          }
        }
        return;
      }

      synchronized(this.enlistRMs) {
        Set<ResourceManager> temp = this.enlistRMs.get(xid);
                ResourceManager findSameRMId = null;
                boolean abort = false;
                for (ResourceManager r : temp) {
                    try {
                        if (r.getID().equals(rm.getID())) {
                            findSameRMId = r;
                        }
                    } catch (Exception e) {
                        // if some RM die, then r.getID() will cause an
exception
                        // dieRM, dieRMAfterEnlist,
                        abort = true;
                        break;
                    }
                }

                if (abort) {
                    rm.abort(xid);

                    ResourceManager randomRemove = temp.iterator().next();
                    temp.remove(randomRemove);
                    if (temp.size() > 0) {
                        this.enlistRMs.put(xid, temp);
                        this.storeToFile(TM_TRANSACTION_RMs_LOG_FILENAME,
this.enlistRMs);
```

```
                        // for dieRM, dieRMAfterEnlist
                        this.xidStatusMap.put(xid, "ABORTED");
                        this.storeToFile(TM_TRANSACTION_LOG_FILENAME,
this.enlistRMs);
                    } else {
                        this.enlistRMs.remove(xid);
                        this.storeToFile(TM_TRANSACTION_RMs_LOG_FILENAME,
this.enlistRMs);

                        this.xidStatusMap.remove(xid);
                        this.storeToFile(TM_TRANSACTION_LOG_FILENAME,
this.xidStatusMap);
                    }

          return;
        }

        // 新的 enlist
        if (findSameRMId == null) {
                    temp.add(rm);
                    this.enlistRMs.put(xid, temp);
                    this.storeToFile(TM_TRANSACTION_RMs_LOG_FILENAME,
this.enlistRMs);
                    return;
                }
      }
    }
  }
```

- 生命周期相关
  - `start()`：事务的开始
  - `commit()`： 事务的提交
  - `abort()`：事务的终止
  - `dieNow()`： 事务的消亡

流程控制器 Workflow Control及相关资源类的实现

- 流程控制器主要是用来处理客户端对事务管理器和资源管理器的请求工作，并将相关操作分配给相关的部件
  - 相关类的实现
    - `Car`：price, location, numCars, numAvail以及相关操作，可以对车辆的数量进行预定和取消预定的加减
    - `Customer`： custName
    - `Flight`： flightNum，price，numSeats，numAvail及相关操作，可以对航班的数量进行预定和取消预定的加减
    - `Hotel` location，price，numRooms，numAvail及相关操作，可以对房间的数量进行预定和取消预定的加减
  - 对事物管理器的任务分配

- TransactionManager的接口start(),commit(),abort()都在WC中被调用，然后将相关的操作分配给TM

```
 public int start() throws RemoteException {
    // do not need synchronized, because tm.start() is synchronized, so xid is
unique
    int xid = tm.start();
    this.xids.add(xid);

    this.storeTransactionLogs(this.xids);

    return xid

 public boolean commit(int xid)
      throws RemoteException, TransactionAbortedException,
InvalidTransactionException {
    if (!this.xids.contains(xid)) {
      throw new InvalidTransactionException(xid, "commit");
    }

    boolean ret = this.tm.commit(xid);
    this.xids.remove(xid);
    this.storeTransactionLogs(this.xids);

    return ret;
  }

 public void abort(int xid) throws RemoteException, InvalidTransactionException
{
    if (!this.xids.contains(xid)) {
      throw new InvalidTransactionException(xid, "abort");
    }

    this.tm.abort(xid);
    this.xids.remove(xid);
    this.storeTransactionLogs(this.xids);
  }
```

- 对资源管理器的访问
  - 管理员操作，需要权限，对数据库进行增删行为
    - `addFlight`：添加航班
    - `deleteFlight`：删除航班
    - `addRooms`：添加房间
    - `deleteRooms`：删除房间
    - `addCars`：添加车辆
    - `deleteCars`：删除车辆
    - `newCustomer`：添加新的消费者

        - `deleteCustomer`：删除消费者
    - 不需要权限的操作，对数据库进行查询
        - `queryFlight`：查询航班
        - `queryFlightPrice`：查询航班价格
        - `queryRooms`：查询房间
        - `queryRoomsPrice`：查询房间价格
        - `queryCars`：查询车辆
        - `queryCarsPrice`：查询车辆价格
        - `queryCustomerBill`：查询消费者账单
    - 为消费者提供预定服务
        - `reserveFlight`：预定航班
        - `reserveCar`：预定车辆
        - `reserveRoom`：预订房间
        - `reserveItinerary`：预定行程

相关测试用例

- 基本业务逻辑 增删改查极其组合业务，包含数据 Flight, Room, Car, Customer。可以添加修改删除 Flight, Room, Car, Customer。用户可以预订或者取消预订 Flight, Room, Car。8-16 个测试用例。
    - 增加 Flight, Room, Car, Customer。
    - 删除 Flight, Room, Car, Customer。
    - 查看 Flight, Room, Car, Customer。
    - 修改 Flight, Room, Car。
    - 用户预订 Flight, Room, Car。
    - 用户取消预订 Flight, Room, Car。
    - 输入异常
    - 异常 key，数据
- 并发测试 当并发执行以上事务逻辑，能够正确执行。基于两阶段锁测试。随机选择基本事务。5-10 个 测试用例
    - 读读共享: T_L_RR
    - 读写等待: T_L_RW
    - 写读等待: T_L_WR
        - 该测试用例主要用于测试并发执行的过程中的读写等待情况：对于一个数据 d, 事务 A 首先 获取 WRITE 锁，然后事务 B 在申请 d 的 READ 锁的时候就要等待事务 A 提交或者放弃， 然后释放锁。事务 B 拿到 d 的 READ 锁之后，从数据库读出现有的 d 值。其核心过程如 下

```
1 call addCars xid "SFO" 300 40
1 return true
2 call queryCarsPrice xid "SFO"
1 call commit xid
1 return true
2 return 40
```

        - 首先线程 1 的事务获取了 car 的写锁，将其价格修改为 40. 然后线程 2 的事务要读取 car 的价格，这时候就需要等待线程 1 提交。等到线程 1 提交后，线程 2 获取锁，读到修改后的价格 40. 结合 1.2.1 资源管理的实现，我们一定到在从数据库读取数据前先申请锁。在 UCI 的测试用例中没有包含对读取内容的检验，这也恰恰是一个忽略点。
    - 写写等待: T_L_WW
    - 死锁，后者放弃事务
    - 读读写写死锁: T_L_RRWW
    - 2 数据，读写读写死锁: T_L_RWRW
    - 2 数据，写读写读死锁: T_L_WRWR
    - 2 数据，写写写写死锁: T_L_WWW

课程思考体会

分布式系统是由一组通过网络进行通信、为了完成共同的任务而协调工作的计算机节点组成的系统。分布式系统的出现是为了用廉价的、普通的机器完成单个计算机无法完成的计算、存储任务。其目的是利用更多的机器，处理更多的数据。

分布式数据库系统是在冀中是数据库系统的基础上发展来的，比较分布式数据库系统与集中式数据库系统，可以发现分布是数据库系统具有下列优点：

(1) 更适合分布式的管理与控制。分布式数据库系统的结构更适合具有地理分布特性的组织或机构使用，允许分布在不同区域、不同级别的各个部门对其自身的数据实行局部控制。例如：实现全局数据在本地录入、查询、维护，这时由于计算机资源靠近用户，可以降低通信代价，提高响应速度，而涉及其他场地数据库中的数据只是少量的，从而可以大大减少网络上的信息传输量；同时，局部数据的安全性也可以做得更好。

(2) 具有灵活的体系结构。集中式数据库系统强调的是集中式控制，物理数据库是存放在一个场地上的，由一个 DBMS 集中管理。多个用户只可以通过近程或远程终端在多用户操作系统支持下运行该 DBMS 来共享集中是数据库中的数据。而分布式数据库系统的场地局部 DBMS 的自治性，使得大部分的局部事务管理和控制都能就地解决，只有在涉及其他场地的数据时才需要通过网络作为全局事务来管理。分布式 DBMS 可以设计成具有不同程度的自治性，从具有充分的场地自治到几乎是完全集中式的控制。

(3) 系统经济，可靠性高，可用性好。与一个大型计算机支持一个大型的冀中是数据库在加一些进程和远程终端相比，由超级微型计算机或超级小型计算机支持的分布式数据库系统往往具有更高的性价比和实施灵活性。分布式系统比集中式系统具有更高的可靠性和更好的可用性。如由于数据分布在多个场地并有许多复制数据，在个别场地或个别通信链路发生故障时，不致于导致整个系统的崩溃，而且系统的局部故障不会引起全局失控。

(4) 在一定条件下响应速度加快。如果存取的数据在本地数据库中，那末就可以由用户所在的计算机来执行，速度就快。

(5) 可扩展性好，易于集成现有系统，也易于扩充。

对于一个企业或组织，可以采用分布式数据库技术在以建立的若干数据库的基础上开发全局应用，对原有的局部数据库系统作某些改动，形成一个分布式系统。这比重建一个大型数据库系统要简单，既省时间，又省财力、物力。也可以通过增加场地数的办法，迅速扩充已有的分布式数据库系统。

分布式数据库系统的缺点

(1) 通信开销较大，故障率高。例如，在网络通信传输速度不高时，系统的响应速度慢，与通信县官的因素往往导致系统故障，同时系统本身的复杂性也容易导致较高的故障率。当故障发生后系统恢复也比较复杂，可靠性有待提高。

(2) 数据的存取结构复杂。一般来说，在分布时数据库中存取数据，比在集中时数据库中存取数据更复杂，开销更大。

(3) 数据的安全性和保密性较难控制。在具有高度场地自治的分布时数据库中，不同场地的局部数据库管理员可以采用不同的安全措施，但是无法保证全局数据都是安全的。安全性问题式分布式系统固有的问题。因为分布式系统式通过通信网络来实现分布控制的，而通信网络本身却在保护数据的安全性和保密性方面存在弱点，数据很容易被窃取。

分布式数据库系统的优点与缺点：分布式数据库的设计、场地划分及数据在不同场地的分配比较复杂。数据的划分及分配对系统的性能、响应速度及可用性等具有极大的影响。不同场地的通信速度与局部数据库系统的存取部件的存取速度相比，是非常慢的。通信系统有较高的延迟，在 CPU 上处理通信信息的代价很高。分布式数据库系统中要注意解决分布式数据库的设计、查询处理和优化、事务管理及并发控制和目录管理等问题。