腾讯事务处理技术验证系统

3TS 腾讯事务处理技术验证系统（下）

作者：李海翔

Tencent Transaction Processing Testbed System（简称 3TS），是腾讯公司 CynosDB（TDSQL）团队与中国人民大学数据工程与知识工程教育部重点实验室，联合研制的面向**数据库事务处理的验证系统**。该系统旨在通过设计和构建事务（包括分布式事务）处理统一框架，并通过框架提供的访问接口，方便使用者快速构建新的并发控制算法；通过验证系统提供的测试床，可以方便用户根据应用场景的需要，对目前主流的并发控制算法在相同的测试环境下进行公平的性能比较，选择一种最佳的并发控制算法。目前，验证系统已集成 13 种主流的并发控制算法，提供了 TPC-C、Sysbench、YCSB 等常见基准测试。3TS 还进一步提供了一致性级别的测试基准，针对现阶段分布式数据库系统的井喷式发展而造成的系统选择难问题，提供一致性级别判别与性能测试比较。

3TS 系统旨在深度探索数据库事务处理相关理论与实现技术，其核心理念是：**开放、深度、进化**。开放，秉承开源之心，共享知识、共享技术；深度，践行系统化钻研之精神，对于事务处理技术的本质问题进行研究，不破楼兰终不还；进化，路漫漫其修远兮，吾将上下而求索，不断前行，不断推进。

5、3TS 提供的并发访问控制算法

前文介绍了 3TS 的框架和基础内容，本节继续深入介绍多种并发访问控制算法。

5.4 乐观并发控制协议（OCC、FOCC、BOCC）

在乐观并发控制协议下，事务的执行流程被分成三个阶段：读取、验证、写入阶段[5]，如图 5 所示。

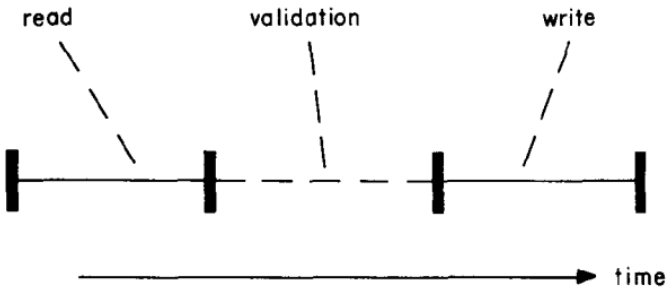

Fig. 1. The three phases of a transaction.

图 5 OCC 算法的三个阶段图

这三个阶段的划分，带来的优势非常明显：

1. **事务处理性能高**：事务的效率的提升主要是依靠第一阶段通过读写互不阻塞来保证的，这极大提高了读写、写读这两种情况的并发度，多核的硬件资源能够得到充分利

用；并且对于只读事务因不被阻塞而倍显友好。

2. **可避免死锁问题**：OCC 可以在第一阶段通过对读写对象排序、第二阶段按序加锁可避免死锁。这在与封锁并发访问控制算法对死锁问题解决的对比下获胜。这两点优势使得 OCC 在处理分布式事务、高数据热点、高通信延时等场景下依然能够支持高事务吞吐率，在高并发场景下没有明显的系统性能抖动现象（文献[169]通过实验表明高竞争下 OCC 算法性能不高）。
3. **数据一致性的正确性得到保证**：正确性是在第二阶段验证阶段保证的，其原理是通过事务冲突关系构造有向图检测是否存在有环的情况而通过回滚某个事务破除环的存在，达到解决事务冲突的目的；而写写冲突是在验证阶段通常通过封锁机制保证。但工程实现，有不同的方式，如文献[9]改进了 OCC 算法，在验证阶段检查本事务的读集如果被其他并发事务写过即触发回滚以避免数据不一致，从而不用构造有向图检测是否存在有环存在。

3TS 中，根据验证机制的不同，实现了三个不同的乐观并发控制协议：（1）OCC：对[5]中的 Parallel Validation 算法的实现；（2）BOCC：对[6]中的 Backward Validation 算法的实现；（3）FOCC：对[6]中的 Forward Validation 算法的实现。需要注意的是，3TS 中，由于没有全局时间戳机制（后续计划会增加全局时钟），验证阶段需要对比的读写集大小可能由于时钟不同步产生偏差，因此可能会对不同算法的效率产生不同程度的影响。

1. 三个协议对读取阶段的处理是相同的，主要为：
   a) 读操作时，先把读操作存入读集，并按要求读到所需要的数据；
   b) 写操作时，把写操作写入写集
2. 验证阶段，三个协议的主要思想都是保证事务按照进入验证的顺序进行排序，通过检查读写集保证事务的操作结果满足进入验证的先后顺序。不同协议的检查读写集的方法存在不同。

5.4.1 OCC

验证操作的主要流程为（当前待验证事务的开始时间戳记为 start_ts，进入验证的时间戳记为 finish_ts）：

a) 获取在(start_ts,finish_ts]这一时间段内的提交的事务集合，记为 History，遍历 History 中事务的写集，如果与当前事务的读集存在交集，则当前事务验证失败；
b) 获取处在验证阶段的事务集合，记为 Active，检查集合中事务的写集和当前事务的读集是否存在交集，如果存在，则验证失败；

5.4.2 BOCC

要求验证阶段和写入阶段在同一个临界区中执行,流程为获取在(start_ts,finish_ts]这一时间段内的提交的事务集合，记为 History，遍历 History 中事务的写集，如果与当前事务的读集存在交集，则当前事务验证失败。BOCC 具有比较明显的缺点，包括只读事务也需要进行验证、待验证事务的读集较大时对验证效率影响较大，对于长事务需要保留大量期间已提交事务的写集等。

5.4.3 FOCC

要求验证阶段和写入阶段在同一个临界区中执行，检查待验证事务的写集是否与当前活跃（正在读写阶段）事务的读集如果存在有交集，则当前事务验证阶段。FOCC 较 BOCC 具有只读事务可以跳过验证阶段，与活跃事务进行验证开销较小的优点。

3. 三个协议对写入阶段的处理是相同的，主要为：获取提交时间戳，将写集中数据写入数据库，并设置数据的提交时间戳为获取到的提交时间戳

5.5 优化的乐观并发控制协议（MaaT、Sundial、Silo）

传统的乐观并发控制协议依照进入验证的顺序来确定事务是否可以提交，与传统的 OCC 相比，一些优化的乐观并发控制协议通过放宽这一要求，减少了不必要回滚。现在，基于 OCC 的改进版本有很多，如 ROCC[20]、自适应 OCC[21]等。

在 3TS 中，我们集成了三种较新的乐观并发控制算法，包括 MaaT、Sunidal 和 Silo。期待有更多的并发算法集成到 3TS 中。

5.5.1 MaaT

MaaT[6]采用了**动态时间戳**范围调整的方式来降低事务回滚率。其主要思想是通过事务间的读写操作之间形成的关系，确定事务的先后顺序，从而确定可串行化要求的等价串行序列中事务的先后顺序。例如，Ti 事务读 x 之后，Tj 事务需要更新 x，则在等价串行序列中，Ti 需要排在 Tj 的前面。

MaaT 需要在每个数据项上额外维护元数据，包括：（1）记录读了该数据项但仍未提交的事务 ID，称为读事务列表 readers；（2）记录要写该数据项但仍未提交的事务 ID，称为写事务列表 writers；（3）读过该数据项的事务中最大的提交时间戳，记为 Rts；（4）写过该数据项的事务中最大的提交时间戳，记为 wts。每个事务会有一个时间戳范围[lower,upper)，并初始化为[0,+)。事务中的各个操作的流程主要包括：

1. 读操作
   a) 将数据项的写事务列表存入事务的 uncommitted_writes；
   b) 更新当前事务的 greatest_write_timestamp=Max{greatest_write_timestamp,wts}；
   c) 将当前事务 ID 写入所读数据项的读事务列表；
   d) 读对应数据项，并将读到的数据存入读集。
2. 写操作
   a) 将数据项的写事务列表存入事务的 uncommitted_writes_y。
   b) 将数据项的读事务列表存入事务的 uncommitted_reads。
   c) 更新当前事务的 greatest_write_timestamp=Max{greatest_write_timestamp,wts}，greatest_read_timestamp=Max{greatest_red_timestamp,rts}；
   d) 将当前事务 ID 写入要写数据项的写事务列表；
   e) 将要写的数据项新值存入写集；
3. 验证阶段（事务协调者根据所有参与者返回的 lower 和 upper 取交集确定 lower 和 upper，如下操作均在参与者上执行）
   a) 更新 lower=Max{greatest_write_timestamp+1,lower}；
   b) 保证 uncommitted_writes（未提交写事务列表）中事务的 lower 大于当前事务的 upper；
      i. 如果 uncommitted_writes 中的事务已经验证通过，修改当前事务的 upper；

ii. 否则将 uncommitted_writes 中的事务放进当前事务的 after 队列（队列中的事务需要在当前事务之后提交）；

c) 更新 lower=Max{greatest_read_timestamp+1,lower}；

d) 保证 uncommitted_reads（未提交读事务列表）中事务的 upper 小于当前事务的 lower；

i. 如果 uncommitted_reads 中的事务已经验证通过，修改当前事务的 lower；

ii. 否则将列表中的事务放进当前事务的 before 队列（队列中的事务需要在当前事务之前提交）；

e) 调整 uncommitted_writes_y（存在写写冲突的未提交事务列表）和当前事务的先后关系；

i. 如果 uncommitted_writes_y 中的事务已经验证通过，修改当前事务的 lower 大于列表中已验证通过事务的 upper；

ii. 否则将列表中的事务事务放进当前事务 after 队列；

f) 检查 lower<upper 是否成立，不成立则回滚当前事务；

g) 协调调整当前事务的 lower 和 before 队列中事务的 upper，保证当前事务的 lower 大于 before 队列事务的 upper；

h) 协调调整当前事务的 upper 和 after 队列中事务的 lower，保证当前事务的 upper 小于 after 队列中事务的 lower；

4. 写入阶段（首先在协调者上确定提交时间戳（commit_ts）为最终时间戳区间的 lower，然后在参与者上执行如下操作）

a) 对于读集中的每个元素，将当前事务从对应数据项的读事务列表中清除，并进行如下操作：

i. 保证写事务列表中事务的 lower 大于当前事务的 commit_ts；

ii. 更新 Rts=Max{commit_ts,Rts}；

b) 对于写集中的每个元素，将当前事务从对应数据项的写事务列表中清除，并进行如下操作：

i. 保证写事务列表中事务的 upper 小于当前事务的 commit_ts。

ii. 保证读事务列表中事务的 upper 小于当前事务的 commit_ts。

iii. 更新 Wts=Max{commit_ts,Wts}。

5.5.2 Sundial

Sundial[8]通过动态计算提交时间戳以减少回滚率。同时在数据项上维护租约（即数据项的可以被访问到的逻辑时间范围），便于在发生冲突时快速确定事务的先后顺序。此外 Sundial 在乐观并发控制的基础上，结合了悲观并发控制的思路，读写/写读冲突用 OCC、写写冲突用 2PL 锁的方式来减少分布式事务协调调度的开销。

Sundial 在数据项上维护租约(wts,rts)，分别代表了数据项最后被写入的时间和数据项可以被读到的最晚时间。在事务上维护 commit_ts，代表事务的提交时间戳。在读写集中额外维护 orig.rts 和 orig.wts，代表访问数据项当时的 rts 和 wts。我们对 Sundial 的主要操作的执行流程介绍如下：

1. 读操作

a) 首先从读写集中读取所需要的数据项，如果读写集中不存在所需数据项

i. 则需要访问数据存储，找到对应数据项并读取，并记录此时数据项的 wts 和 rts，记为 orig.wts 和 orig.rts；

ii. 更新当前事务的 commit_ts=max{orig.wts,commit_ts}；

b) 如果读写集中存在所需数据，直接返回对应数据；

2. 写操作
   a) 首先从写集中找到所要修改的数据项，如果写集中不存在所需数据项
      i. 对元组加锁，若加锁失败，存入等待队列 waiting_set；
      ii. 否则，直接返回数据项，以及对应的 wts 和 rts，记为 orig.wts 和 orig.rts；
   b) 如果读写集中存在当前待更新的数据项对应元素，则在写集中对其进行更新；
   c) 更新当前事务的 commit_ts=max{orig.rts,commit_ts}；
3. 验证阶段
   a) 首先计算出提交时间戳 commit_ts，主要通过如下两步（该步骤为 3TS 中实现新增，由于读写操作在参与者上进行，协调者在进入验证前需要汇总所有参与者的信息得到 commit_ts）：
      i. 遍历写集，更新 commit_ts 大于等于写集中所有元素的 orig.rts；
      ii. 遍历读集，更新 commit_ts 大于等于读集的 orig.wts；
   b) 验证读集中的每一个元素：
      i. 如果提交时间戳 commit_ts 小于 rts，跳过当前元素；
      ii. 尝试更新元组租约：（1）如果 orig.wts！=wts，即当时读取的 wts 和元组现在的 wts 不同，当前事务需要回滚；（2）如果当前元组被加了锁，当前事务回滚；（3）否则更新元组的 rts=Max{rts,commit_ts}；
4. 写入阶段
   a) 提交操作，对写集中元素对应的数据项更新并解锁；
   b) 回滚操作，对写集中元素对应的数据项解锁。

5.5.3 Silo

Silo[9]与传统乐观并发控制协议的主要区别在验证阶段。其主要思想是验证自己读到的数据是否被其他事务修改。因此，事务的验证流程为：

1. 为所有写集中的元素对应的数据项加锁；
2. 验证读集中的数据：（1）被别的事务修改或（2）由别的事务加锁。如果存在两种情况中的一种，则当前事务回滚；
3. 获得提交时间戳并进入写入阶段。

5.6 确定性并发控制协议（Calvin）

Calvin[10]的主要思想是提前确定好事务的顺序,之后事务则会严格按照确定的顺序进行执行。避免了其他并发控制协议所需的分布式协调开销。

Calvin 算法需要增加两个模块：定序器（Sequencer）和调度器（Scheduler）。其中定序器用于拦截事务并且为这些事务规定顺序（顺序就是事务进入定序器的顺序），调度器负责按照定序器给定的顺序执行事务。

Calvin 事务执行流程主要包括（假设事务需要使用到 Server1 和 Server2 的数据）：

1. Client 将事务发送给 Server1 节点；
2. Server1 的定序器接受到事务，将事务放入一个 batch 中；
3. 经过 batch 规定的时间后，定序器将包含事务的 batch 发送给事务对应的两个参与者节点 Server1 和 Server2 的调度器上；
4. Server1 和 Server2 的调度器接收到 batch，根据 batch 事先规定的顺序进行加锁。之后将 batch 中的事务放入 WorkThread(工作线程)中执行；
5. Server1 和 Server2 执行完毕 batch 中的所有事务后，将返回消息发送给 Server1；

6. Server1 向 Client 返回事务执行完毕。

事务执行时的加锁机制依然遵循 2PL 的逻辑，主要包括：

1. 读操作：
   a) 检查数据项上是否存在排它锁，检查数据项的 waiters 列表是否为空。若不存在排他锁且 waiters 列表为空，则读取对应数据项，并将当前事务放入 owner；
   b) 否则，加锁存在冲突，将当前事务存入 waiters 等待事务列表；
2. 写操作：
   a) 检查数据项上是否存在锁，检查数据项的 waiters 事务是否存在。若不存在排他锁且 waiters 列表为空，将当前事务放入 owner；
   b) 否则，加锁存在冲突，将当前事务存入 waiters 等待事务列表。

5.7 基于快照隔离的并发控制协议（SSI、WSI）

快照隔离（Snapshot Isolation, SI）[11]主要对同一数据项上的写写冲突和写读冲突进行了约束。对于写写冲突，其规定，数据项不能同时被两个事务并发修改，另外遵循“先提交者获胜策略”，先提交的写事务将会成功，另一个事务将会回滚。对于写读冲突，其规定事务只能读取最新已提交的数据项版本，即读取事务开始时符合一致性状态的数据。因此，其具有 读写互不阻塞的事务处理特性。SI 机制本身不能做到可串行化，因此在 SI 基础上实现可串行化，需要引入额外的操作。在 3TS 中，实现了两种主流的可串行化快照隔离机制：（1）SSI：Serializable Snapshot Isolation；（2）WSI：Write Snapshot Isolation。

5.7.1 SSI

如果事务 Ti 读了 x，事务 Tj 写入了一个 x 的新版本，那么我们称 Ti 读写依赖于 Tj。SSI[12,13]通过理论证明，发现要在 SI 基础上做到可串行化，只需要禁止 Ti 读写依赖于 Tj，且 Tk 读写依赖于 Ti 这种情况即可。算法的核心就在于动态检测出这种情况，因此会在每个事务记录 inConflict 和 outConflict 两个字段：inConflict 记录了读写依赖于当前事务的事务，outConflict 记录了当前事务读写依赖的事务。当发现当前事务这两个字段都不为空时，则立刻回滚当前事务，从而保证了可串行化。

5.7.2 WSI

WSI[14]通过将写写冲突的检测转化为读写冲突的检测，并避免读写冲突来做到可串行化。

对于每个事务，WSI 需要维护它的读集和写集。为了避免幻象，对于范围查询读集里放的是查询谓词。对于每一个记录需要维护 last commit 时间戳，每当事务提交会更新所有修改过的行的 last commit 时间戳为事务的提交时间戳。事务提交前的检查如下：检查所有读集中元素对应的数据项，如果它的 last commit 时间戳大于当前事务的 start timestamp（消除了读写冲突），就回滚当前事务。

5.8 基于动态时间戳的并发访问控制算法

第五点，我们介绍了一些 OCC 的改进算法，其中提及的 MaaT、Sundial，是在利用了 OCC 的框架，结合 TO 算法进行改进的一种方式。但是，他们又不只是基于 TO，传统的 TO 算法是一种静态的算法，时间戳是确定的、刚性的。而 MaaT、Sundial 以及 Tictoc[22]等，采用的是动态时间戳分配算法。这样把 OCC 框架（策略）的优势、动态时间戳的优势结合起来。

动态时间戳分配（dynamic timestamp allocation，简称 DTA），最先在文献[23]中提出，之后被多篇文献引用和应用。此算法的核心思想：是不依赖中心化的时间戳机制、根据数据项上的并发事务冲突关系、通过动态调整数据项上的事务的执行时间段，来实现全局事务的

可串行化。该算法避免一些在非动态时间戳分配算法下被认为是存在冲突而被回滚的情况。

文献[22]介绍了一种名为"Time Traveling Optimistic Concurrency Control(TicToc)"的算法，该算法基于 OCC 算法，提出"data-driven timestamp management"的思路，即不给每个事务分配独立的(全局)时间戳，而是在访问数据项时嵌入必要的(本地)时间戳信息，用于为每个事务在提交之前计算出有效的提交时间戳，而经计算（不是预先分配）而得的提交时间戳用于解决并发冲突从而保证事务是可串行化的。因不用在分布式事务开始和提交阶段依赖全局的协调器为事务分配时间戳，所以在这个阶段，可实现去中心化的目的。因结合使用 OCC 机制，所以可缩小事务冲突重叠的执行时间段，提高了并发度。

文献[7]基于 OCC 框架，实现了 DTA 算法，即前述的 MaaT 算法（第 5.5 节），这里不再展开。

6、3TS 待改进功能

3TS 系统提供了统一的技术研制平台，可以对多种并发访问控制算法进行统一对比、分析。目前仍然存在如下待改进的地方，会对不同并发控制协议带来不同程度的影响，影响实验结果的准确性。我们主要总结了如下待改进的功能点：

1. 消息通信机制，可以考虑通过 RPC 等方式替代现有的消息通信机制，从而减少消息队列中的等待对事务性能的影响。
2. 线程调度模型（一个线程绑定一个核），可以考虑引入更多的调度模型，帮助分析线程调度方法对并发控制协议性能的影响。
3. 不支持 SQL 语句，需要引入 SQL 解析等操作，来更好的模拟真实数据库场景。
4. 不支持全部的 TPCC 事务，需要进一步引入 Delivery 等事务类型，从而支持全部的 TPCC 测试。
5. 全局时间,没有全局时间戳生成模块,使用机器本地的时间戳可能会存在时钟偏差，对 OCC 等协议造成影响。
6. 死锁检测算法，可以考虑引入死锁检测算法，来更好的分析其他的 2PL 协议。
7. Deneva 中各个算法不可以动态切换，每种算法使用宏（C 语言的宏）来进行切换，这要求系统切换算法执行时，每次都要动态编译后再运行，这是一个待改进点。

致谢

感谢腾讯 CynosDB（TDSQL）团队与中国人民大学数据工程与知识工程教育部重点实验室对本项工作的支持，感谢赵展浩，刘畅，赵泓尧等同学为本文做出贡献。

Reference

[1] Rachael Harding, Dana Van Aken, Andrew Pavlo, Michael Stonebraker: An Evaluation of Distributed Concurrency Control. Proc. VLDB Endow. 10(5): 553-564 (2017)

[2] Philip A. Bernstein, Nathan Goodman: Concurrency Control in Distributed Database Systems. ACM Comput. Surv. 13(2): 185-221 (1981)

[3] Daniel J. Rosenkrantz, Richard Edwin Stearns, Philip M. Lewis II: System Level Concurrency Control for Distributed Database Systems. ACM Trans. Database Syst. 3(2): 178-198 (1978)

[4] D. P. Reed. Naming and synchronization in a decentralized computer system. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.
[5] H. T. Kung, John T. Robinson: On Optimistic Methods for Concurrency Control. ACM Trans. Database Syst. 6(2): 213-226 (1981)
[6] Theo Härder: Observations on optimistic concurrency control schemes. Inf. Syst. 9(2): 111-120 (1984)
[7] Hatem A. Mahmoud, Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, Amr El Abbadi: MaaT: Effective and scalable coordination of distributed transactions in the cloud. Proc. VLDB Endow. 7(5): 329-340 (2014)
[8] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sánchez, Larry Rudolph, Srinivas Devadas:
Sundial: Harmonizing Concurrency Control and Caching in a Distributed OLTP Database Management System. Proc. VLDB Endow. 11(10): 1289-1302 (2018)
[9] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, Samuel Madden: Speedy transactions in multicore in-memory databases. SOSP 2013: 18-32
[10] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, Daniel J. Abadi: Calvin: fast distributed transactions for partitioned database systems. SIGMOD Conference 2012: 1-12
[11] 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 Conference 1995: 1-10
[12] Alan D. Fekete, Dimitrios Liarokapis, Elizabeth J. O'Neil, Patrick E. O'Neil, Dennis E. Shasha: Making snapshot isolation serializable. ACM Trans. Database Syst. 30(2): 492-528 (2005)
[13] Michael J. Cahill, Uwe Röhm, Alan D. Fekete: Serializable isolation for snapshot databases. SIGMOD Conference 2008: 729-738
[14] Maysam Yabandeh, Daniel Gómez Ferro: A critique of snapshot isolation. EuroSys 2012: 155-168
[15] https://en.wikipedia.org/wiki/Distributed_transaction
[16] P. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency Control and Recovery in Database Systems. Addison－Wesley, 1987.
[17] D. R. Ports and K. Grittner, "Serializable snapshot isolation in postgresql," PVLDB, vol. 5, no. 12, pp. 1850－1861, 2012.
[18] J. Böttcher, et al., Scalable Garbage Collection for In-Memory MVCC Systems, in *VLDB*, 2019
[19] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, Andrew Pavlo:An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. Proc. VLDB Endow. 10(7): 781-792 (2017)
[20] D. Lomet and M. F. Mokbel, "Locking key ranges with unbundled transaction services," VLDB, pp. 265－276, 2009.
[21] Jinwei Guo, Peng Cai, Jiahao Wang, Weining Qian, Aoying Zhou: Adaptive Optimistic Concurrency Control for Heterogeneous Workloads. PVLDB 12(5): 584-596 (2019)
[22] X. Yu, A. avlo, D. Sanchez, and S. Devadas, "Tictoc: Time traveling

optimistic concurrency control," in Proceedings of SIGMOD, vol. 8, 2016, pp. 209–220.
[23] Rudolf Bayer, Klaus Elhardt, Johannes Heigert, Angelika Reiser:Dynamic Timestamp Allocation for Transactions in Database Systems. DDB 1982: 9-20