



**本科毕业设计（论文）**

外文参考文献译文及原文

**学 院** **应用数学学院**

**专 业 信息与计算科学**

**（信息安全方向）**

**年级班级 2014级（2）班**

**学 号 3114008254**

**学生姓名 方彦凯**

**指导老师 蔡前凤**

**2018年 5 月**

目录

[1 一个关于MongoDB事务处理方法的建议（译文） 1](#_Toc514359021)

[1.1介绍 1](#_Toc514359022)

[1.2 MongoDB中事务处理的问题 2](#_Toc514359023)

[1.2.1RDBMS中的事务处理 2](#_Toc514359024)

[1.2.2 MongoDB中数据更新操作的问题 3](#_Toc514359025)

[1.3 MongoDB事务处理方法的建议 4](#_Toc514359026)

[1.3.1数据结构和提案方法的操作 4](#_Toc514359027)

[1.3.2通过锁定方法进行独家控制的过程 6](#_Toc514359028)

[1.4 实施和评估 7](#_Toc514359029)

[1.4.1程序结构 7](#_Toc514359030)

[1.4.2事务处理的实验 8](#_Toc514359031)

[1.5 注意事项 10](#_Toc514359032)

[1.6 结论和未来的工作 10](#_Toc514359033)

[2 A proposal of transaction processing method for MongoDB（原文） 12](#_Toc514359034)

[2.1 Introduction 12](#_Toc514359035)

[2.2 Problem of transaction processing in MongoDB 14](#_Toc514359036)

[2.2.1 Transaction processing in RDBMS 14](#_Toc514359037)

[2.2.2 Problem of data update manipulations in MongoDB 15](#_Toc514359038)

[2.3 Proposal of transaction processing method for MongoDB 17](#_Toc514359039)

[2.3.1 Data structure and manipulations of the proposal method 18](#_Toc514359040)

[2.3.2 Procedure of exclusive control by the lock method 21](#_Toc514359041)

[2.4 Implementations and evaluations 22](#_Toc514359042)

[2.4.1 Program structures 22](#_Toc514359043)

[2.4.2 Experiments of transaction processing 24](#_Toc514359044)

[2.5 Considerations 28](#_Toc514359045)

[2.6 Conclusion and future work 29](#_Toc514359046)

**1 一个关于MongoDB事务处理方法的建议（译文）**

目前，为了处理数据库中的大量数据，各种NoSQL数据库已经被提出并投入实际使用。然而，由于它们中的大多数仅支持单个数据的事务处理，所以存在这样的问题：复数数据不能一次更新并保持ACID属性。为解决这个问题，本文提出了一种将多个数据作为单个事务处理的方法，MongoDB是一种面向文档的NoSQL数据库。具体来说，每个数据都具有更新前后的字段，并且管理事务的状态。然后，在提交之前，查询前一个数据; 提交后，后面的数据被查询。通过这种方法，我们显示复数数据可以更新为具有指定隔离级别的事务。

**1.1介绍**

随着大数据的发展，现实世界中的各种信息被发布在数据库中。这里，关系数据库管理系统（以下称为RDBMS）针对特定的公司，主要处理由方案预定义的值和字符数据以保持高一致性。 但是，现在，数据库被全球用户访问，其数据也传播到各种范围，如文档和视频。 所以，已经有必要适应称为3V的特征，即音量（巨量），速度（速度），变化（广泛的多样性）。 因此，与传统RDBMS不同的被称为NoSQL数据库的各种数据库管理系统已经被提出并投入实际使用。

这里，对于RDBMS，执行并发控制。因此，即使在多个事务同时访问数据库的情况下，它们也会像一个接一个地执行一样执行。因此，交易的ACID属性，即原子性，一致性，隔离性和耐久性得以保持。另一方面，为了处理3V特性，许多NoSQL数据库仅维护BASE属性，即基本可用，软状态和最终一致。对于此属性，上述ACID属性是 仅在更新单个数据的情况下保持。因此，在更新多个数据的情况下，数据一个接一个地更新，最后它们变得更新。也就是说，在更新过程中会出现异常情况，例如一个数据已更新，另一个尚未更新。

BASE属性对于不需要严格并发控制的事务有效。例如，有补货的网店。在此，即使在多个交易同时购买同一商品的情况下，也不是首先执行任一交易的问题。另一方面，在酒店预订等库存有限的情况下，成为问题。因此，数据库所需的并发控制级别取决于业务流程。

BASE属性对于不需要严格并发控制的事务有效。例如，有补货的网店。 在此，即使在多个交易同时购买同一商品的情况下，也不是首先执行任一交易的问题。 另一方面，在酒店预订等库存有限的情况下，成为问题。 因此，数据库所需的并发控制级别取决于业务流程。

对于这个问题，对于RDBMS，在SQL标准中定义了多个隔离级别，并且为每个事务分别选择了级别。同样，对于NoSQL数据库，也知道这种事务处理对于某些业务流程是必需的。因此，两阶段提交方法用于MongoDB，这是一种面向文档的NoSQL数据库。该方法管理多个数据更新。并且在中止的情况下，它执行补偿事务以在更新之前恢复数据。但是，由于它是基于BASE的特性，所以上述异常仍然在更新和恢复数据中。

本文的目标是提出一个具有MongoDB指定隔离级别的事务处理方法。具体来说，每个数据都具有更新前后的字段，并且管理事务的状态。然后，在提交之前，查询前一个数据; 提交后，后面的数据被查询。此外，我们将此方法作为使用Java的实验程序实施，并确认数据库访问可以作为具有指定隔离级别的事务同时执行。

本文的其余部分安排如下。第2节展示了MongoDB事务处理的问题，我们在第3节提出了这个问题的方法。第4节展示了这个方法的实现和实验结果。第5节给出了考虑因素，第6节结束了本文。

**1.2 MongoDB中事务处理的问题**

1.2.1RDBMS中的事务处理

为了阐明MongoDB中事务处理的问题，我们给出了RDBMS中事务处理的概述。至于RDBMS中的事务处理，ACID属性定义为以下4个属性：原子性意味着事务完全更新或根本不更新; 一致性意味着数据库更新后维持一致性; 隔离意味着每个事务的执行都不会影响其他同时执行的事务; 耐久性意味着更新结果在失败中幸存下来。

此外，为了在保持ACID特性的同时执行多个事务，执行并发控制。因此，即使许多事务同时执行，它们的更新结果就好像它们一个接一个地执行一样。这种并发控制通常由锁定方法执行。也就是说，通过在访问数据之前执行锁定，可以保护从其他事务对该数据的冲突访问。至于事务处理，已经表明需要两个锁相协议（以下称为2PL），其中所有锁都在解锁之前执行; 为了保护级联中止，需要严格的2PL，它保持所有的锁直到提交或中止。

但是，在锁定方法中的事务之间发生冲突的情况下，一个事务必须等待另一个事务的解锁。因此，长时间延迟可能发生在以下情况：与长时间事务的冲突;访问集中到一个特定的数据。因此，事务的多个隔离级别已经定义如表1所示。这里，事务服从2PL来更新每个隔离级别的数据。读取未提交（1）允许事务在提交之前查询更新数据，并且锁定不会像查询那样执行。读提交（2）允许事务在提交后查询更新的数据，并且共享锁仅在查询期间执行。可重复读取（3）允许事务重复查询相同的数据而不会被其他事务改变，并且事务服从2PL也查询数据。顺便说一下，对于隔离级别，还有可序列化以防止幻像读取：未由前一个查询查询的数据不会被下一个查询查询。但是，由于没有要锁定的目标数据，因此该级别不能通过简单的锁定协议来实现。因此，本文不包括可序列化。

此外，显示至少在所有事务的隔离级别为未读的情况下，每个事务的隔离级别保持其自身的状态。也就是说，如果每个事务服从表1中所示的锁定协议之一，则可以使用指定的隔离级别执行任何事务。因此，即使在MongoDB中，如果其他事务是在隔离级别Read未提交的情况下执行的，则可以执行具有指定隔离级别的任何事务。

1.2.2 MongoDB中数据更新操作的问题

另外，类似于RDBMS中的SQL，它具有CRUD（创建，读取，更新，删除）方法：插入，查找（对应于选择），更新和删除（对应于删除）。但是，这些方法并不打算将复数文档作为类似于RDBMS的事务处理。他们的交易功能仅在单个文档中提供。换句话说，在一次更新多个文档的情况下，存在更新的中间状态可以被其他事务查询的问题，即不能维持ACID属性。顺便说一句，在MongoDB中通过单一方法更新多个文档的情况下，可以指定“$ isolated”选项。利用这个选项，可以防止目标数据被其他交易更新。但是，这个选项并不能确保原子性。另外，它不能用于多种方法的更新。此外，HBase提出了基于时间戳的乐观并发处理的事务处理方法，它是一种NoSQL数据库，通过时间戳管理每个数据的版本。但是，要将此方法应用于MongoDB，有必要对HBase实施类似的版本管理功能。

对于这个问题，使用两阶段提交方法，通过这种方法可以在维护原子性的情况下更新多个文档。在这种方法中，例如，在执行从账户A到账户B的银行账户转账的情况下，其交易编号被存储到管理集合中。在更新账户A和B的文件之前，该号码被存储到两个文件中以通过该交易来管理更新文件。在成功完成的情况下，该号码被移除并且处理结束;在失败的情况下，执行补偿事务以在更新之前恢复数据。

这种方法类似于RDBMS中的“传奇”：在这种方法中，许多数据的总和更新被分成小粒子更新过程，并且它们被一个接一个地执行;在出现故障的情况下，它将按顺序执行补偿事务以恢复更新前的数据。但是，显示在与其他事务同时执行的情况下，隔离不能由此方法维护。因此，在上述情况下，例如，即使账户A和账户B的总量没有变化，也会出现问题。

实际上，暂时更改的总金额是由其他交易查询的。另外，我们已经表明存在另一个问题：就彼此相关的数据而言，即使更新成功完成，数据的一致性也可能不会保持。也就是说，由于MongoDB的事务处理不能一次更新多个文档并保持ACID属性，所以存在不能应用于需要这种数据操作的业务系统的问题。

**1.3 MongoDB事务处理方法的建议**

1.3.1数据结构和提案方法的操作

要将多个文档作为MongoDB中的事务更新，必须同时在所有目标文档上执行提交或回滚。因此，在提案方法中，每个文档都具有更新前后的字段，以隐藏更新结果直到提交。另外，它具有锁信息字段以执行排他控制。另外，TP（事务处理管理）集合被添加来管理正在运行的事务。图2显示了这两个集合的文档结构。（a）显示数据收集。有各种各样的数据收集，并存储相应的业务数据。它们由这些字段组成：“data0”存储更新前的数据; “ctl”存储该文件的锁定状态; “data1”存储更新后的数据。此外，“ctl”字段由以下字段组成：“rn”存储通过共享锁锁定此数据的事务数量，哪些事务正在查询此文档; “r id”将其对象ID的数组存储在TP集合中;同样，“w id”存储交易的对象ID，该交易专门锁定该文件。顺便提一下，图2中带下划线的字段在数据出现时被添加，当数据消失时被删除。

（b）显示TP集合，其存储正在运行的事务信息，并且当相应的事务连接到MongoDB时插入其文档;它在事务关闭此连接时被删除。其字段“tno”存储交易标识号; “st”存储下面列出的交易状态; “level”存储表1中所示的事务隔离级别。图3显示了事务的状态转换：p（prepare）显示连接到数据库的事务; d（执行）显示第一个数据操作已完成; c（提交）表示由于成功完成而开始提交; r（回滚）显示由于失败而开始回滚。

图4示出了由于这种方法的数据操纵的例子，其中从账户转移到另一账户的银行账户被作为交易执行。从图4的（a）到（e）中，上面两个数据示出了账户收集的文件，它是图2（a）的一种数据收集。在“数据0”和“数据1”字段，“ac”是帐号的字段; “bal”是它的平衡。其余文件显示TP集合。这里，下划线显示了更改的字段，附带的字段显示它们是查询的数据。此外，尽管图4显示了为了清楚起见而仅更新两个文档的简单情况，但可以更新任意数量的以相同的方式将账户收款的文件作为交易。

（a）示出更新前的状态，并且更新事务信息作为具有对象ID“Id30”和状态“p”的文档被存储在TP集合中。 （b）显示排他锁完成后的状态。这里，顺序地对每个文档执行排他锁定，因此当执行第一次锁定时，TP集合的文档的状态字段转换为“d”。 （c）示出了在更新数据被添加之后的状态，其依次执行。 （d）显示状态提交，并且状态字段转换为“c”。 （e）表示更新后的状态，其中在每个文档中同时执行以下处理：“data1”的数据被反映为“data0”，并且“data1”被移除;由“w id”管理的排他锁被解锁。这个过程也是按顺序在每个文档上执行的。顺便提及，在回滚的情况下，执行以下处理：在（d）处，状态字段转变为“r”;在（e）处，“data1”字段被移除，并且“data0”字段保持原样;如上所述，独占锁被解锁。另外，在插入的情况下，在（c）处插入没有“data0”字段的文档;在删除的情况下，（c）和（d）中没有“data1”，相应的文档在（e）删除。

在查询的情况下，数据操作与表1所示的隔离级别不同。首先，在读取已提交（2）的情况下，Account集合中两个文档的查询结果需要为以下任一项：两者都没有更新;两者都已更新。因此，查询按以下过程执行。由于事务没有提交，所以在（a）和（c）之间查询“data0”字段。尽管在这些过程中文档是按顺序更新的，但“数据0”不会更新。所以，它们不会影响查询结果。接下来，在（d），由于事务已经提交，查询“data1”字段。如上所述，由于“data1”的数据被顺序地反映在每个文档中的“data0”中，所以可以如下查询更新结果：同时查询“data0”和“data1”两者;并且在有“data1”的情况下被查询;否则，“data0被查询。

要执行这样的查询，有必要确认事务状态是：它没有提交;它已承诺。因此，使用存储在“w id”字段中的对象ID将此事务的状态查询到TP集合。在“d”的情况下，即事务没有提交，查询过程服从（a）到（c）。因此，查询“data0”。在“c”的情况下，服从（d）。这里，（e）包含在其中。因此，查询“data1”。

其次，在未提交的情况下（1），即使事务未提交，也会查询更新的数据。因此，更新的数据总是通过（d）中所示的过程在未提交的读取中查询。顺便提一下，由于数据收集的文档是按顺序更新的，因此可能会出现以下结果：某些数据在更新之前;其他数据在更新之后。最后，在可重复读（3）的情况下，其查询过程与读取已提交相同，但它们在锁定协议上不同，如下一节所示。

1.3.2通过锁定方法进行独家控制的过程

如表1中的“独占锁定”一栏所示，对于所有隔离级别的更新操作，执行基于2PL的锁定。顺便说一下，在这种方法中，为防止级联回滚，严格的2PL作为2PL的替代协议执行。也就是说，在事务执行提交或回滚之前，不会执行解锁。因此，在更新事务之间存在冲突的情况下，等待另一个事务提交。

同样，如“共享锁定”一栏所示，排他控制取决于查询操作的隔离级别。首先，在未提交Read的情况下，目标数据未被锁定用于查询操作。因此，在事务仅执行查询操作的情况下，可以在没有由冲突引起的延迟的情况下执行它。顺便提一句，在Read未提交更新的事务并逐一提交每个文档的情况下，它与MongoDB的事务处理类似，尽管不是严格的。 Read提交的事务仅在查询期间执行共享锁。共享锁取决于如图2所示的“rn”字段。并且，如图4所示，在该方法中，更新的提交完成取决于以下两者：“w id”字段和状态（“st”字段）在TP集合中。因此，在上述事务已提交（由“st”显示）的情况下，即使数据被该事务锁定为排他锁（由“w id”显示），也允许共享锁。因此，锁的兼容性矩阵如图5所示。这里，“之前允许的”排他锁分为两个阶段：在其事务的状态是提交或回滚的情况下，所请求的共享锁被允许;否则，它被拒绝。同样，可重复读取的事务也为查询执行共享锁定，尽管它一直保持到提交或回滚。

同样，对于排他锁，如果交易已经提交，也可以允许所请求的排他锁。然而，在这种情况下，由于下一次更新可能在图4（d）所示的当前更新完成之前开始，所以有必要将“数据1”作为该数组。因此，在本文中，独占锁定不应该如图5所示那样兼容。这里，所有的锁定和解锁操作必须由并发事务以及RDBMS顺序执行。换句话说，对每个文档的“ctl”字段的操作必须按顺序执行，包括对TP集合的查询。

**1.4 实施和评估**

为了确认指定的隔离级别的事务可以组成，我们实现了一个程序来更新图4所示的银行账户并进行实验。

1.4.1程序结构

图6显示了实验程序的结构。我们使用Java（版本1.6.0 17）来实现该程序; MongoDB Java驱动程序（Ver。2.13.0）访问MongoDB（Ver。2.6.7）。另外，我们使用Java的Thread类来实现事务的并发执行;我们使用Synchronized方法来实现事务的顺序执行，这些操作由操作来锁定和解锁。

在实验开始时，主程序将实验数据设置为MongoDB中的集合。接下来，它通过使用Thread同时访问MongoDB来启动更新程序和查询程序。我们实现了MongoDB访问类，并且通过这个类的更新和查询方法来执行对MongoDB的访问，如有必要，这些方法伴随着锁操作。更新程序查询帐户集合中目标文档的余额，然后更新文档的余额。因此，我们使用排他锁实现了更新程序的查询方法，这与SQL中“更新”短语伴随的“select”语句类似。对于图6中的处理，我们在MongoDB访问类中准备了以下方法。这里，排除了以下基本方法：连接，关闭和隔离级别设置。

int readUp（int account）

这是更新程序的查询方法，它以与读取方法相同的方式返回余额：首先，它转换状态;第二，它不同于读取并用排他锁锁定目标文件;然后，它查询余额。在执行提交或回滚方法之前，锁不会解锁。

WriteResult更新（int account，int balance）

这将指定的余额设置为指定的帐户。它以与readU p方法相同的方式锁定目标文档，然后更新文档。顺便说一句，返回值的WriteResult是一个用于表示更新结果的MongoDB Java驱动程序。

void commit（）

这提交了交易。它将TP集合的状态转换为提交（c），然后通过删除“ctl”字段中的锁定信息来解锁目标文档。文件一个接一个地解锁。此时，如果目标文件被专用锁锁定，则将“data1”的数据反映为“data0”，并且如图4所示移除“data1”。

void rollback（）

这将执行事务的回滚。其与TP收集和解锁的过程与commit方法相同。如果目标文档被排他锁锁定，则会删除“data1”字段。

1.4.2事务处理的实验

首先，为了确认交易处理特征可以通过提案方法来实现，我们进行了如下所示的实验。 这些实验在100个账户上进行。 对于以下三个查询程序的隔离级别，我们进行了两个实验：成功完成和失败。 顺便提一下，由于严格的2LP，更新程序会锁定目标文档的排他锁。

（1）未提交读取：它在不锁定的情况下逐个查询文档。

（2）读取提交：它使用共享锁逐一查询文档; 它在每个查询完成时解锁。

（3）可重复读取：它还使用共享锁逐一查询文档，并提交每10个文档执行解锁。

为了避免死锁，更新和查询程序都按帐号顺序访问帐号集合。更新程序启动后，每10次更新一次提交一次锁;然后，查询程序在适当的延迟之后启动。为了使它们之间发生冲突，后者在实验中超越了前者。因此，我们将两个程序都配置为延迟，如表2所示。这里，Read未提交的查询程序即使未提交，也需要查询已更新的数据。因此，相应的更新程序被配置为在“更新后”延迟。至于其他隔离级别，它需要在其提交和下一次更新操作之间取代更新程序。因此，相应的更新程序被配置为在“提交后”延迟。

在这种情况下，如果程序被锁定延迟，则在50毫秒后执行重试。为了便于确认实验结果，我们执行了以下程序：将查询程序的查询结果存储到Result集合中，这些结果在没有事务处理的情况下执行;一些类型的信息被输出到控制台。后者是锁定信息，更新数据，重试信息等等。预先设定的每个账户的余额由等式1,000 + 1,000×帐号表示。然后，更新程序通过在这些实验中添加20,000来更新每个帐户的余额。

对于Read未提交的查询程序（1），图7（1-1）和（1-2）分别显示了在成功完成和失败的情况下的查询结果。由于更新程序的提交每10次更新一次，垂直轴表示提交点。在成功完成（1-1）的情况下，直到查询程序超过更新程序，前者在更新之前查询数据;否则，它会在更新后查询数据。此外，由于它查询未提交的数据，所以变化点与提交点不一致。另一方面，在失败（1-2）的情况下，在31和34之间以及41和44之间的账户中，查询程序在更新后查询数据。关于前者的原因，当查询程序查询了账户34时执行回滚。对于后者，由于查询程序已经取代了更新程序，所以查询回滚之前的数据。

类似地，图7（2-1）和（2-2）显示了Read committed（2）的实验结果。至于成功完成（2-1），由于没有发生脏读，所以查询的数据在提交点被更新的数据之前和之后被分割;至于失败（2-2），所有查询的数据都在更新之前。另外，由于共享锁是在查询程序中执行的，所以在共享锁中发生冲突所致的延迟时间为3.6倍，平均排除锁时为0.3倍。这里，前者发生在查询程序中;后者发生在更新程序中。

对于可重复读（3），由于必须确认在同一事务中查询的数据没有变化，所以实验查询程序组成如下：它依次查询10个数据;延迟（25毫秒）后，它再次以相同的方式查询数据;接下来，它执行提交;经过延迟（50毫秒）后，它开始处理下一个10个数据。在图7（3-1）和（3-2）中，实线表示第一个查询结果;虚线显示第二个查询结果。结果，两者都是相同的，即查询结果即使再次被查询也没有改变。这里，查询结果就像（3-1）和（3-2）中的（2）一样。另外，由于冲突导致的等待时间在共享锁中出现了4.8次，平均而言在独占锁中出现了1.4次。

其次，我们检查了“读未提交”隔离级别和“findOne”方法之间的性能比较。后者是MongoDB查询指定单一数据的标准语句。在这个实验中，预先在Account表中保存了指定数量的数据，并且所有的数据都被逐一查询。两者之间没有显着差异，如图8所示。此外，为了评估同时更新两个以上文档的性能，我们进行了实验以更新1024个数据。如图9所示，提交是针对指定号码的每次更新执行的。通过这个实验，我们得到了在这种情况下同时更新超过30个文档的效率更高的结果。

**1.5 注意事项**

如图7所示，所提出的方法实现了隔离级别的可重复读取，因此可以解决目标问题。也就是说，可以通过该方法将多个文档更新为交易。而且，可以为每个事务指定隔离级别。因此，对于实验中使用的银行账户，可以按照以下方式执行具有业务所需的隔离级别的每个事务。

首先，在余额查询不需要确切顺序的情况下，可以有效地执行Read未提交的情况。特别是，对于查询事务，由于它不锁定目标数据，所以不会发生由冲突引起的延迟。其次，在两个账户之间的银行账户转账的情况下，具有读取提交的交易可以保持ACID属性的一致性。也就是说，他们的金额被查询为一定的金额。但是，在这种情况下，假设每个账户通过单次操作更新，例如增加或减少指定的金额。第三，在交易首先查询余额并决定提取金额的情况下，在查询之后有必要防止其从其他交易中更新。因此，在这种情况下，应该使用可重复读取来执行事务。

而且，对于这种方法的Read未提交的查询，其性能大致相当于图8所示的findOne方法。顺便说一下，我们应该考虑以下内容：访问通常在文档中逐一执行MongoDB的;这种方法可以同时执行不同隔离级别的事务。因此，通过利用这种方法，可以认为所需的交易处理可以在整个系统保持效率的情况下实现，如下：具有常规访问的交易是在未读取未执行的情况下执行的;为了同时更新多个文档，事务处理必须具有必要的隔离级别，即Read committed或Repeatable read。

而且，如图6所示，该方法可以作为MongoDB上的应用程序来实现。而且，如图4所示，锁定信息被保存在数据收集的单独文档中;交易信息同样存储在TP集合中。所以，这个方法可以通过MongoDB的常用方法来实现。换句话说，这种方法不仅提供了用于更新MongoDB的多个文档的事务处理，而且还具有通过利用函数作为MongoDB的分布式数据库（例如分片和复制）来维持可伸缩性的效果。

**1.6 结论和未来的工作**

尽管MongoDB可以灵活地操纵各种数据，但存在的问题是它不能在维护事务的ACID属性的情况下操作复数文档。针对这个问题，我们提出了MongoDB的事务处理方法，该方法利用了灵活的数据结构的优点，没有模式：每个文档都有更新前后的数据，并根据事务状态查询有效数据。此外，通过实验，我们确认了以下隔离级别可以由此方法组成：读取未提交，读取提交和可重复读取。

我们正在设计使用监控摄像机视频来测量工厂库存的系统。股票更改时，其视频数据存储在MongoDB中。但是，由于数据由机器和多个工作人员同时添加和更新，所以事务处理对于执行并发控制是必需的。因此，未来的研究将通过实际操作将重点放在此应用和评估上。

**2 A proposal of transaction processing method for MongoDB（原文）**

At present, to deal with a large amount of variety data in the database, various NoSQL databases have been proposed and put to practical use. However, since most of them support the transaction processing only on the single data, there is the problem that the plural data cannot be updated in a lump with maintaining the ACID properties. To solve this problem, in this paper, we propose a method to process plural data as a single transaction for MongoDB, which is a kind of document oriented NoSQL database. Concretely, each data has both of the before and after update fields, and the state of the transaction is managed. Then, in the case of before the commit, the former data is queried; after the commit, the latter data is queried. By this method, we show that the plural data can be updated as a transaction with the specified isolation level.

**2.1 Introduction**

With the development of big data, various information of the real world is published in the databases. Here, the relational database management systems (below, RDBMS) target the specific company, and mainly deal with the value and character data predefined by the scheme to maintain the high consistency. However, nowadays, the databases are accessed by the worldwide users, and its data is also spread to the various range such as documents and videos. So, it has become necessary to adapt to the feature called 3V, that is, Volume (huge amount), Velocity (speed), Variety (wide diversity). Thus, the various database management systems called NoSQL database, which is different from the conventional RDBMS, have been proposed and put to practical use.

Here, as for the RDBMS, the concurrency control is executed. So, even in the case where a number of transactions access the database at the same time, they are performed as if they are performed one after another. As a result, the ACID properties of the transaction, that is, Atomicity, Consistency, Isolation and Durability, are maintained. On the other hand, in order to deal with the 3V feature, many of the NoSQL databases maintain only the BASE property, that is, Basically Available, Soft state and Eventually consistent. As for this property, the above-mentioned ACID properties are maintained only in the case of updating a single data. So, in the case of updating the plural data, the data is updated one after another, and finally they become to be updated. That is, there is the anomaly in the midst of this updating, such as one data has been updated and another has not been updated.

The BASE property is effective for the transactions that do not need the strict concurrency control. For example, there is the net shop which goods are replenished. Here, even in the case where plural transactions purchase the same goods at the same time, it is not the problem that either of the transactions are executed first . On the other hand, in the case where the stock is limited, such as the hotel reservation, it becomes the problem. Thus, the level of the concurrency control required for the database depends on the business process.

For this problem, as for the RDBMS, the multiple isolation levels are defined in the SQL standard, and the level is selected for each transaction individually. Similarly, as for the NoSQL databases, it is also known that such a transaction processing is required for some business processes. So, the two phase commit method is used for MongoDB, which is a kind of document oriented NoSQL database. This method manages the plural data updating. And in the case of abort, it performs the compensation transaction to recover the data before updating. However, since it is based on the BASE property, the above-mentioned anomaly remains in the midst of updating and recovering data.

Our goal of this paper is to propose a transaction processing method having the specified isolation level for MongoDB. Concretely, each data has both of the before and after update fields, and the state of the transaction is managed. Then, in the case of before the commit, the former data is queried; after the commit, the latter data is queried. More- over, we implemented this method as the experimental programs using Java, and confirmed that the database accesses could be performed concurrently as a transaction with the specified isolation level.

The remainder of this paper is organized as follows. Section 2 shows the problem of the transaction processing of MongoDB, and we propose the method for this problem in Section 3. Section 4 shows the implementation of this method, and the experimental results. Section 5 shows the considerations, and Section 6 concludes this paper.

**2.2 Problem of transaction processing in MongoDB**

2.2.1 Transaction processing in RDBMS

To clarify the problem of the transaction processing in MongoDB, we show the overview of the transaction pro- cessing in RDBMS. As for the transaction processing in RDBMS, the ACID properties are defined as the following 4 properties: Atomicity means the transaction updates completely or not at all; Consistency means the consistency of database is maintained after it is updated; Isolation means each transaction is executed without effect on the other transactions executing concurrently; Durability means the update results survive the failure.

Furthermore, in order to perform plural transactions simultaneously with maintaining the ACID properties, the concurrency control is performed. As a result, even though many transactions are performed at the same time, their update results are as if they have been performed one after another. This concurrency control is generally performed by the lock method. That is, by performing the lock before accessing the data, the conflict access to this data from the other transactions are protected. As for the transaction processing, it has been shown that the two phase locking protocol (below, 2PL) is required, in which all the locks are executed before any unlock; to protect the cascade abort, the rigorous 2PL is required, which holds all the lock until the commit or abort.

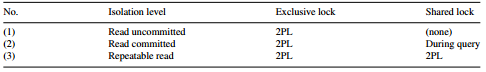
However, in the case where the conflict occurs between the transactions in the lock method, one transaction must wait for the unlock of another transaction. Thus, the long latency may occur in the following case: the conflict with the long time transaction; the access concentration to a specific data. So, the plural isolation levels of the transaction have been defined as shown in Table 1. Here, the transaction obeys 2PL to update data in every isolation level. Read uncommitted (1) allows the transaction to query the updating data before it is committed, and the lock is not executed as for the query. Read committed (2) allows the transaction to query the updated data after it is committed, and the shared lock is executed only during the query. Repeatable read (3) allows the transaction to query repeatedly the same data without changed by the other transactions, and the transaction obeys 2PL also to query data. Incidentally, as for the isolation level, there is also serializable to prevent the phantom read: the data, which is not queried by the previous query, is not queried by the next query. However, since there is not the target data to lock, this level cannot be implemented by the simple locking protocol. So, Serializable is excluded in this paper.



Fig. 1. Example of document structure in MongoDB

Moreover, it is shown that in the case where the isolation level of all the transactions are Read uncommitted at least, the isolation level of each transaction is maintained as of its own. That is, if each transaction obeys one of the locking protocol shown in Table 1, any transaction can be performed with the specified isolation level. Therefore, even in MongoDB, it is possible to execute any transaction with the specified isolation level, in the case where the other transactions were performed with the isolation level Read uncommitted.

Table. 1. Isolation level and locking protocol.



2.2.2 Problem of data update manipulations in MongoDB

Data of MongoDB is configured as the document of the JSON (JavaScript Object Notation) format shown in Fig. 1, and the document consists of the fields. For example, in Fig. 1, {“ id”: Id1 } is the field which identifier is “ id”; value is “Id1” (below, Object ID). Here, the Object ID field corresponds to the primary key of the relational database. Also, the field is possible to be nested as shown by “name” field, which has “first” field (first name) and “last” field (last name). Since the document has such a structure, it is not necessary to define the structure of the database beforehand by schema as RDBMS. So, fields of each document can be added or removed at any time. That is, it is possible that each document has different fields except “ id ”. Incidentally, the set of documents composes the “collection”, and each corresponds to the records and the table in a relational database although not strictly.

In addition, similar to SQL in RDBMS, it has CRUD (Create, Read, Update, Delete) methods: insert, find (corre- sponding to select), update and remove (corresponding to delete). However, these methods do not intend to manipulate the plural documents as a transaction similar to RDBMS. Their transaction features are provided only on the single document. In other words, in the case to update plural documents in a lump, there is a problem that the middle state of the update can be queried by the other transactions, that is, the ACID properties cannot be maintained. Incidentally, in the case of updating plural documents by the single method in MongoDB, “$isolated” option can be designated. Utilizing this option, the target data can be prevented to be updated by the other transactions on the way. However, there is the problem that this option does not ensure the atomicity. Also, it cannot be adopted to the updating com- posed of the multiple kinds of methods. Furthermore, the transaction processing based on the optimistic concurrency using the timestamp is proposed for HBase , which is a kind of NoSQL database and manages the version of each data by the timestamp. However, to apply this method to MongoDB, it is necessary to implement the similar version management functions to HBase.

For this problem, the two phase commit method is used, by which plural documents can be updated with main- taining Atomicity. In this method, for example, in the case to perform the bank account transfer from account A to account B, its transaction number is stored to the management collection. Before the documents of account A and B are updated, the number is stored to the both documents to manage the updating documents by this transaction. In the case of the successful completion, the number is removed and the processing is ended; In the case of the failure, the compensation transaction is performed to recover the data before updating.

This method is similar to the “saga” in RDBMS: in this method, the lump-sum update of many data is divided into small particle updating processes, and they are performed one after another; in the case of the failure, it performs the compensation transactions sequentially to recover the data before updating. However, it is shown that Isolation cannot be maintained by this method in the case of concurrent execution with the other transactions. Therefore, in the above case, for example, the problem occurs: even though the total amount of account A and B does not change actually, the temporarily changed total amount is queried by the other transactions. In addition, we have shown there is another problem: in the case of the data associated with each other, Consistency of the data may not be maintained even though the update completed successfully. That is, since the transaction processing of MongoDB cannot update the plural documents in a lump with maintaining the ACID properties, there is the problem that it cannot be applied to the business systems required such a data manipulation.



(a) Document of data collection



(b) Document of TP(Transaction processing management) collection

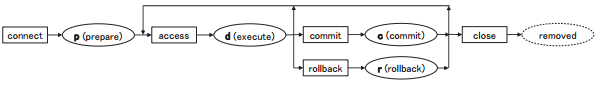


Fig. 3. State transition of transaction

**2.3 Proposal of transaction processing method for MongoDB**

2.3.1 Data structure and manipulations of the proposal method

To update plural documents as a transaction in MongoDB, the commit or rollback must be performed on all the target documents at the same time. So, in the proposal method, each document has both of the before and after update fields to conceal the update result until it is committed. Also, it has the lock information field to perform the exclusive control. In addition, TP (Transaction processing management) collection is added to manage the running transactions. Fig. 2 shows the structures of the documents of these two collections.

(a) shows a data collection. There are various kinds of data collection, and they store corresponding business data. They consist of these fields: “data0” stores the data before updating; “ctl” stores the lock status of this document; “data1” stores the data after updating. Also, “ctl” field is consisted of the following fields: “rn” stores the number of the transactions locking this data by the shared lock, which transactions are querying this document; “r id” stores the array of their Object ID in TP collection; similarly, “w id” stores the Object ID of transaction, which is locking this document exclusively. Incidentally, the underlined fields in Fig. 2 is added when the data appears, and removed when the data disappears.

(b) shows TP collection which stores the running transaction information, and its document is inserted when the corresponding transaction connects to MongoDB; it is removed when the transaction closes this connection. Its field “tno” stores the transaction identification number; “st” stores the transaction state listed below; “level” stores the transaction’s isolation level shown in Table 1. Fig. 3 shows the state transition of the transaction: p (prepare) shows the transaction connected to the database; d (execute) shows the first data manipulation was done; c (commit) shows it began to commit due to the successful completion; r (rollback) shows it began to rollback due to the failure.

Fig. 4 shows an example of data manipulation due to this method, in which a bank account transfer from an account to another account is performed as a transaction. From (a) to (e) of Fig. 4, upper two data show the documents of Account collection, which is a kind of the data collection of (a) in Fig. 2. In both of “data0” and “data1” fields, “ac” is the field of the account number; “bal” is its balance. The rest document shows TP collection. Here, the underline shows the changed fields,the enclosed fields show they are the queried data.In addition,although Fig.4 shows the simple case of updating only the two documents for the sake of clarity, it is possible to update any number of documents of the Account collection as a transaction in the same way.



(a) Prepare update



(b) Exclusive lock



(c) Data update



(d) Commit



(e) After update

Fig. 4. Example of data update procedure

(a) shows the state before the update, and the update transaction information was stored in TP collection as the document having Object ID “Id30” and state “p”. (b) shows the state after the exclusive lock was completed. Here, the exclusive lock was performed to each document sequentially, so the state field of the document of TP collection transitions “d” when the first lock is performed. (c) shows the state after the update data is added, which is performed sequentially. (d) shows the state commit, and the state field transitions to “c”. (e) shows the state after the update, in which the following processes are performed in each document simultaneously: the data of “data1” is reflected into “data0”, and ”data1” is removed; the exclusive lock managed by “w id” is unlocked. This process is also performed on each document sequentially. Incidentally, in the case of the rollback, the following process is performed: at (d), the state field transitions to “r”; at (e), “data1” field is removed, and “data0” field remains as it is; the exclusive lock is unlocked as mentioned above. In addition, in the case of insertion, the document without “data0” field is inserted at (c); in the case of deletion, there is not “data1” at (c) and (d), the corresponding document is deleted at (e).

In the case of query, the data manipulation is different by the isolation level as shown in Table 1. First, in the case of Read committed (2), the query result of the two documents in Account collection needs to be either of the following: the both are not updated; the both have been updated. So, the query is performed in the following procedure. Since the transaction has not committed, “data0” field is queried between (a) and (c). Though the documents are updated sequentially during these processes, “data0” is not updated. So, they do not affect the query results. Next, at (d), since the transaction has committed, “data1” field is queried. As above-mentioned, since the data of “data1” is reflected into “data0” in each document sequentially, the updated result can be queried as follows: both of the “data0” and “data1” are queried simultaneously; and, in the case where there is “data1”, it is queried; otherwise, “data0 is queried.

To perform such query, it is necessary to confirm the transaction state is either: it has not committed; it has committed. So, the state of this transaction is queried to TP collection using the Object ID stored in “w id” field. In the case where it is “d”, that is, the transaction has not committed, the query process obeys from (a) to (c). So, “data0” is queried. In the case where it is “c”, it obeys to (d). Here, (e) is included in it. So, “data1” is queried.

Second, in the case of Read uncommitted (1), the updated data is queried even if the transaction has not committed. Therefore, the updated data is always queried by the procedure shown in (d) in Read uncommitted. Incidentally, since the documents of the data collection are sequentially updated, it may become the following result: some data is before updating; the other data is after the updating. Lastly, in the case of Repeatable read (3), its query process is same as Read committed, but they are different on the locking protocol as shown in the next section.

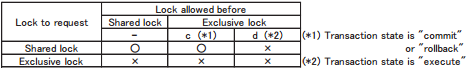


Fig. 5. Compatibility matrix for read committed and repeatable read

2.3.2 Procedure of exclusive control by the lock method

As shown in the column “Exclusive lock” in Table 1, the lock based on the 2PL is performed as for the update manipulation in all the isolation levels. Incidentally, in this method, to prevent the cascading rollback, the rigorous 2PL is performed as the alternative protocol of 2PL. That is, the unlock does not be performed until the transaction performs the commit or rollback. Therefore, in the case where there is the conflict among the update transactions, one waits until another transaction is committed.

Similarly, as shown in the column “Shared lock”, the exclusive control depends on the isolation level as for the query manipulation. First, in the case of Read uncommitted, the target data is not locked for the query manipulation. So, in the case where the transaction performs only the query manipulation, it can be performed without the latency due to the conflict. Incidentally, in the case where the transaction of Read uncommitted updates and commits each document one by one, it is similar to the transaction processing of MongoDB although not strictly. The transaction of Read committed performs the shared lock only during the query. The shared lock depends on the “rn” field as shown in Fig. 2. And, as shown in Fig. 4, in this method, the commit completion of the update depends on the both: “w id ” field, and the state (“st” field) in TP collection. So, in the case where the above-mentioned transaction has committed (shown by “st”), the shared lock is permitted even if the data is locked with the exclusive lock (shown by “w id ”) by this transaction. So, the compatibility matrix of the lock becomes as shown in Fig. 5. Here, the exclusive lock “allowed before” is divided into two stages: in the case where the state of its transaction is the commit or rollback, the requested shared lock is permitted; otherwise, it is rejected. Similarly, the transaction of Repeatable read also performs the shared lock for the query, though it is held until the commit or rollback.

Similarly, as for the exclusive lock, it is also possible to permit the requested exclusive lock if the transaction has committed. However, in this case, since the next update may begin before the current update completion shown in Fig. 4 (d), it is necessary to have “data1” as the array. For this reason, in this paper, the exclusive lock shall not be compatible as shown in Fig. 5. Here, all the manipulations to lock and unlock have to be executed sequentially by the concurrent transactions as well as RDBMS. In other words, the manipulation on “ctl” field of each document has to be performed sequentially including the query to TP collection.

**2.4 Implementations and evaluations**

To confirm the transaction of the specified isolation level can be composed, we implemented a program to update the bank account shown in Fig. 4 and conducted the experiments.

2.4.1 Program structures

Fig. 6 shows the structure of the experimental program. We used Java (Ver. 1.6.0 17) to implement the program; MongoDB Java driver (Ver. 2.13.0) to access MongoDB (Ver. 2.6.7). In addition, we used Thread class of Java to implement the concurrent execution of the transactions; and we used Synchronized method to implement the sequential execution of the transactions, which is used by the manipulations to lock and unlock.

At the beginning of the experiment, the main program set the experimental data to the collections in MongoDB. Next, it starts the update program and query program by using Thread to access MongoDB concurrently. We imple-mented MongoDB access class, and the access to MongoDB is performed by the update and query methods of this class, which are accompanied with the lock operation if necessary. The update program queries the balance of the target document from the Account collection, next updates the balance of the document. So, we implemented the query method of the update program with the exclusive lock, which is similar to “select” statement accompanied with “for update” phrase in SQL. For the processing in Fig. 6, we prepared the following methods in MongoDB access class. Here, the following basic methods are excluded: the connection, close and isolation level setting.

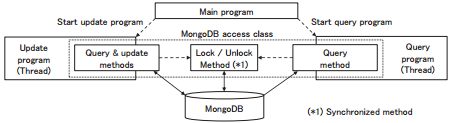


Fig. 6. Experimental program composition

• int read(int account)

This is the query method for the query program, and it returns the balance of the specified account. First, if the state of the corresponding document of TP collection is not execution (d), it transitions this state to “d”. Second, it locks the target document with the shared lock based on the specified isolation level according to Table 1. The success or failure of the lock depends on Fig. 5; in the case of failure, it does a retry; in the case of success, it queries the balance. Here, if the isolation level is Read uncommitted, it does not lock the target document.

• int readUp(int account)

This is the query method for the update program, and it returns the balance in the same way as read method: first, it transitions the state; second, it different from read and locks the target document with the exclusive lock; then, it queries the balance. The lock is not unlocked until commit or rollback method is performed.

• WriteResult update(int account, int balance)

This set the specified balance to the specified account. It locks the target document in the same way as readU p method, then it updates the document. Incidentally, WriteResult of the returned value is a class of MongoDB Java driver to express the updated results.

• void commit()

This commits the transaction. It transitions the state of the TP collection to commit (c), then unlocks the target documents by removing the lock information in “ctl” field. The documents are unlocked one after another. At this time, if the target document is locked with the exclusive lock, the data of “data1” is reflected to “data0” and “data1” is removed as shown in Fig. 4.

• void rollback()

This performs the rollback of the transaction. Its procedure as for TP collection and the unlock is the same as commit method. If the target document is locked with the exclusive lock, “data1” field is removed.

2.4.2 Experiments of transaction processing

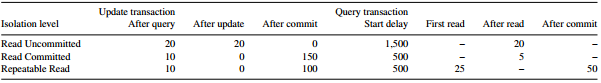
First, to confirm that the transaction processing feature can be realized by the proposal method, we conducted the experiments shown below. These experiments were conducted on 100 accounts. As for each of the following three isolation levels of query program, we conducted two cases of experiments: the successful completion and failure. Incidentally, the update program locks the target document with the exclusive lock due to the rigorous 2LP.

(1) Read uncommitted: it queries document one by one without the lock.

(2) Read committed: it queries document one by one with the shared lock; it unlocks at each query completion.

(3) Repeatable read: it also queries document one by one with the shared lock, and the commit is performed each 10 documents to unlock.

Table 2. Delay time of executing program (millisecond).



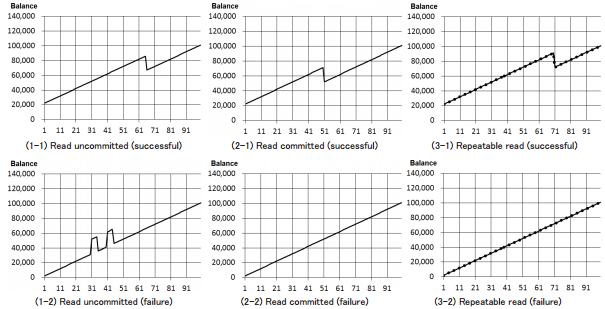


Fig. 7. Result of isolation level experiment

To avoid a deadlock, both the update and query programs access Account collection sequentially by the account number. The update program is started, and its locks are committed every 10 updates; then, the query program is started after the appropriate delay. To make the conflict among them, the latter overtakes the former in the experiment. So, we configured both of the programs to delay as shown in Table 2. Here, the query program of Read uncommitted needs to query the updated data even if it has not been committed. So, the corresponding update program is configured to delay at “After update”. As for the other isolation levels, it needs to overtake the update program between its commit and next update manipulation. So, the corresponding update program is configured to delay at “After commit”.

Here, in the case where a program was delayed by the lock, its retry is performed after 50 milliseconds. And, in or- der to facilitate the confirmation of experimental results, the following procedures were performed: the query results of the query program were stored into Result collection, which were performed without the transaction processing; some kinds of information were output to the console. The latter were the lock information, update data, retry information and so on. The balance of each account set beforehand is expressed by the equation 1, 000+1, 000×account number. Then, the update program updated the balance of each account by adding 20,000 in these experiments.

As for the query program of Read uncommitted (1), Fig. 7 (1-1) and (1-2) show the query results in the case of the successful completion and failure respectively. Since the commit of the update program is performed every 10 updates, the vertical axis represents the commit point. In the case of the successful completion (1-1), until the query program overtook the update program, the former queried the data before the update; otherwise, it queried the data after the update. Furthermore, since it queried the uncommitted data, the change point does not coincide with the commit point. In the case of the failure (1-2), on the other hand, at the accounts between 31 and 34, and between 41 and 44, the query program queried the data after updated. As for the reason of the former, the rollback was performed when the query program had queried the account 34. As for the latter, since the query program had overtaken the update program, the data before the rollback was queried.

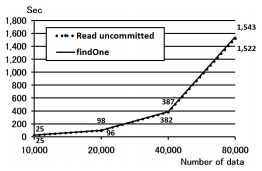


Fig. 8. Performance of READ UNCOMMITTED to findOne

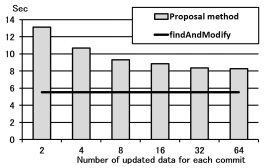


Fig. 9. Performance of update to findAndModufy

Similarly, Fig. 7 (2-1) and (2-2) show the experimental results of Read committed (2). As for the successful completion (2-1), since the dirty read did not occur, the queried data was divided to before and after the updated data by the commit point; as for the failure (2-2), all the queried data was before the update. In addition, since the shared lock is performed in the query program, the latency due to the conflict occurred 3.6 times in a shared lock, 0.3 times in the exclusive lock on average. Here, the former occurred in the query program; the latter occurred in the update program.

As for Repeatable read (3), since it must be confirmed that the data queried in the same transaction is not changed, the experimental query program is composed as follows: it queries 10 data sequentially; after delay (25 msec), it queries the data in the same way once more; next, it performs the commit; after delay (50 msec), it starts to process the next 10 data. In Fig. 7 (3-1) and (3-2), the solid line shows the first query results; the broken line shows the second query results. As a result, the both were the same, that is, the query results were not changed even though they were queried again. Here, the query results are just like (2) in the both case of (3-1) and (3-2). In addition, the latency due to the conflict occurred 4.8 times in the shared lock, 1.4 times in the exclusive lock on average.

Second, we examined the performance comparison between “Read uncommitted” isolation level and “ findOne” method. The latter is the standard statement of MongoDB to query the specified single data. In this experiment, the specified number of data had been saved in Account table beforehand, and all the data was queried one by one. There was no significant difference between the both as shown in Fig. 8. Moreover, to evaluate the performance of updating more than two documents simultaneously, we performed the experiment to update 1024 data. As shown in Fig. 9, the commit was performed for each update of the specified number. Through this experiment, we got the result that it is more efficient to update more than 30 documents simultaneously in this case.

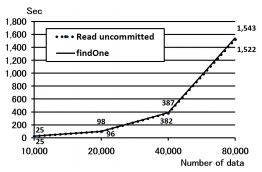


Fig. 8. Performance of READ UNCOMMITTED to findOne

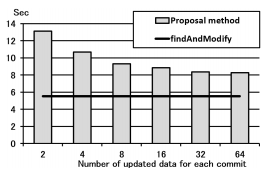


Fig. 9. Performance of update to findAndModufy

**2.5 Considerations**

As shown in Fig. 7, the proposed method achieved the isolation level Repeatable read, so the target problem could be resolved. That is, the plural documents can be updated as a transaction by this method. Moreover, the isolation level can be specified for each transaction. Thus, as for the bank account, which is used in the experiment, it is possible to perform each transaction with the isolation level required by the business as follows.

First, in the case where the query of the balance does not need the exact order, it can be efficiently performed with Read uncommitted. In particular, as for the query transaction, since it does not lock the target data, the latency due to conflict does not occur. Second, in the case of the bank account transfer between two accounts, the transaction with Read committed can maintain the Consistency of the ACID properties. That is, their amount is queried as a certain amount. However, in this case, it is assumed that each account is updated by single manipulation, such as increasing or decreasing the specified amount. Third, in the case where the transaction queries the balance first and decides the withdraw amount, it is necessary to prevent its update from other transactions after this query. Therefore, in this case the transaction should be performed with Repeatable read.

Moreover, as for the query with Read uncommitted of this method, its performance is roughly equivalent to f indOne method as shown in Fig. 8. By the way, we should consider the following: the access is generally performed on document one by one in MongoDB; the transactions with different isolation level can be performed concurrently in this method. Therefore, by utilizing this method, it is considered that the required transaction processing can be realized with maintaining the efficiency as a whole system as follows: the transaction having conventional access is performed with Read uncommitted; to update plural documents simultaneously, the transaction is performed with the necessary isolation level, that is, Read committed or Repeatable read.

And, as shown in Fig. 6, this method can be implemented as an application on MongoDB. Moreover, as shown in Fig. 4, the lock information is saved in the individual document of the data collection; the transaction information is stored in TP collection similarly. So, this method can be implemented by the usual methods of MongoDB. In other words, this method not only provides the transaction processing for updating multiple documents of MongoDB, but also has the effect to be able to maintain the scalability by utilizing the functions as a distributed database of MongoDB, such as the sharding and replication. 、

**2.6 Conclusion and future work**

Though MongoDB manipulates a variety of data flexibly, there is the problem that it cannot manipulate the plural documents with maintaining the ACID properties of the transaction. For this problem, we propose the transaction processing method for MongoDB, which utilizes its advantage of the flexible data structure without schema: each document has both of the before and after updated data, and the valid data is queried based on the transaction state. Furthermore, through the experiment, we confirmed that the following isolation level could be composed by this method: Read uncommitted, Read committed and Repeatable read.

We are designing the system to measure the stock of the plant using the surveillance camera video. When the stock is changed, its data with video is stored in MongoDB. However, since the data is added and updated by the machine and the plural workers simultaneously, the transaction processing is necessary to perform the concurrency control. So, the future study will focus on this application and the evaluations through the actual operations.