软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.000000]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

分布式数据库中一致性与可用性的关系 *

朱 涛, 郭进伟, 周 欢, 周 烜, 周傲英

华东师范大学 数据科学与工程学院, 上海 200062

通讯作者: 周烜, E-mail: xzhou@dase.ecnu.edu.cn

摘 要: 随着各类应用在数据量和业务量上的扩展，单机数据库系统越发难以应对现实需求。分布式数据库技术在沉寂多年以后，逐步开始受到应用的青睐。近年来，分布式数据库产品层出不穷，并在互联网应用中被大量投入使用。然而，分布式数据库的系统复杂度前所未有。为了让系统可用，设计者需要在多种属性中作合理选择和折中。这造成现有的数据库产品形态各异、优缺点对比分明。至今为止，尚未有人对分布式数据库的设计空间和折中方案进行过深入分析和整理。本文作者在对多个分布式数据库产品进行深入理解之后认识到：分布式数据库系统的设计方案可以通过三个属性进行基本刻画 – 操作一致性、事务一致性和系统可用性。虽然这三个属性并不新颖，但它们在数据库语境下的含义在文献中尚未得到充分澄清。本文对这三个属性进行澄清，并通过它们对典型数据库产品的格局进行概括、对现有的分布式数据库技术进行综述。此外，本文还对这三个属性之间的相互关系进行深入分析，以期帮助未来的开发者在分布式数据库的设计过程中作出合理选择。

关键词: 分布式系统；数据库；一致性；可用性；事务处理

Consistency and Availability in Distributed Database Systems

Tao ZHU, Jinwei GUO, Huan ZHOU, Xuan ZHOU, Aoying ZHOU

School of Data Science and Engineering, East China Normal University, Shanghai 200062, China

Abstract: The rapid growth of data and workload makes centralized database systems less and less favorable to today's applications. While the technology of distributed database has been around for decades, it has not gained much attention from applications until recently. Since the needs for distributed DB became apparent, an increasing number of products have emerged and been adopted by the Web. However, due to the complexity of distributed DB systems, their designers have to trade off among several desired properties, resulting dramatical difference in their designs and advantages. To the best of our knowledge, no one has performed a comprehensive analysis on the design space and the tradeoff choices of modern distributed DB systems. After reviewing and understanding a significant number of real world DB products, we believe that a distributed DB system can be generally described using three dimensions – operational consistency, transactional consistency and availability. While these dimensions are not new, their concepts are somehow blurred in the literature. In this paper, we clarify the three concepts in the context of database, based on which we manage to draw a sensible landscape of the existing products and technologies. We also provide an analysis of the relationship among the three dimensions, which we wish can help developers make right choice when designing new distributed DB systems.

Key words: distributed system; database; consistency; availability; transaction

* 基金项目: 国家自然科学基金(00000000, 00000000)
Foundation item: National Natural Science Foundation of China (00000000, 00000000)
收稿时间: 0000-00-00; 修改时间: 0000-00-00; 采用时间: 0000-00-00; jos 在线出版时间: 0000-00-00
CNKI 在线出版时间: 0000-00-00

随着数据量的爆发式增长以及应用负载的快速增加，传统关系型数据库所采用的单一服务器模式越来越难以应对当今应用对数据存储和事务处理的需求。一旦数据管理功能被扩展至计算机集群和跨区域的分布式系统，数据库系统的复杂程度被明显抬升。因此，单纯的硬件扩展未必能够等量级地获得更高的并发度和性能。系统设计者还必须考虑架构的改变（从单机到分布式系统）对系统稳定性、数据一致性、可用性等性质的影响，因为这些性质与系统的扩展性和性能密切相关，甚至相互制约。

近年来，号称能够提供良好扩展性的分布式数据管理技术和产品星罗棋布，并且被广泛运用于各种互联网应用中。按照当今比较流行的分类方式，它们可以分为以 HBase[3], BigTable[28]，Cassandra[7], MongoDB[8], CouchDB[9], Dynamo[10], Neo4J 等为代表的 NoSQL 数据库，以及以 VoltDB[11], Spanner[12], MemSQL[4], Clustrix[1], NuoDB[5], eXtremeDB[2]等为代表的 NewSQL 数据库。前者被认为是通过弱化了对数据一致性的保护能力（如抛弃了 ACID 原则[13]，而退而求其次地遵循 BASE 原则[23]）而获得更强的系统扩展能力和系统可用性。后者被认为是针对不同领域的应用而定制的特殊数据库系统；它们通过更贴近应用的设计而获得更好的性能和扩展能力。然而，这样的划分方式无疑太过于简化。具体系统设计所遇到的问题并非是 NoSQL 和 NewSQL 的理念所能解决的，还需要设计者能够对数据库系统的各个重要属性以及它们之间的制约关系有清晰的理解。

系统研究的历史经验告诉我们，复杂系统的构建往往难以做到面面俱到，而需要在多个重要指标之间做取舍和折中。例如功能和易用性、性能和通用性都是难以同时兼顾的。Eric Brewer 曾在 ACM 分布式计算大会上提出了著名的 CAP 理论[14]，即分布式系统不可能同时满足一致性(Consistency)，可用性(Availability)和分区容错性(Partition Tolerance)，而最多只能同时满足其中两个。例如，在存在网络分区的情况下，支持高可用性的系统不能同时保证数据的强一致性。这一理论被广泛用于对分布式系统的设计考量。由于分区容错性被认为在分布式环境下无法避免，各分布式系统需根据应用的需求选择性地保证强一致性或者高可用性。例如，传统的主备数据库牺牲一定的可用性来保证数据的强一致性，而 Riak[6]选择了数据最终一致性(Eventual Consistency)[24]来支持高可用性。

虽然 CAP 理论对系统设计者而言具备较强的实践意义，但在数据库系统的设计中它仍然显得比较局限。一方面，CAP 理论对一致性和可用性的定义与传统数据库对这两个属性的定义并不完全一致。这给分布式数据库研究者和设计者带来了一定的困惑。例如，Spanner 号称自己是能够严格保证数据一致性的高可用分布式数据库。这看似与 CAP 理论相矛盾，但事实上是由于两者对一致性和可用性的定义有所区别。另一方面，CAP 理论并未对自身涉及的三个属性进行量化。这导致系统设计者难以对属性之间的制约关系进行考量 – 比如当发生网络分区时，分布式数据库到底需要牺牲多少可用性来换取事务 ACID 属性的保证。

本文立足于数据库系统，致力于在数据库的语境下对数据一致性和系统可用性的关系进行分析，并且对由这两个属性所构成的系统设计空间（Design Space）进行全面考量，以期帮助分布式数据库开发者在未来的系统设计过程中作出合理选择。

时至今日，数据库领域已经有大量工作集中于权衡数据库事务一致性与性能的关系和衡量分布式系统一致性与可用性的关系。事务隔离级别是常见的对数据一致性的强弱划分。ANSI SQL-92[16]根据事务执行过程中出现的三种异常现象定义了不同的事务隔离级别，如快照隔离(Snapshot Isolation)、游标稳定性(cursor stability)等。Adya[16]和 Cerone[17]采用不同的一致性模型重新定义了分布式事务一致性级别。近年来，还有不少学者提出了各种新的事务一致性级别，如原子性读(Read Atomic)[18]、一致性读(consistent read)[19]、Parallel Snapshot Isolation(并行快照隔离)[20]等等，以及不同类型的分布式事务一致性协议，如 Spanner、VoltDB、Calvin[21]等。Peter Bailis[22]从理论上分析了节点高可用性和事务隔离级别之间的关系。除此之外，工业界也涌现出了一大批优秀的分布式数据库系统，它们都在一致性和可用性上作了深入考虑，选择了各自合理的系统构建方案。然而，尚未有工作对这一系列的研究成果和系统成果进行系统梳理和分析。本文将把它们放在我们的一致性-可用性框架内，对这些系统和工作进行定位，并对它们的设计思想进行归类和总结，希望能够为分布式数据库的研究者和开发者从宏观视角理解这些系统和工作提供帮助。

本文的第三个目的是通过分析现有系统在一致性和可用性上的定位发现它们的共同薄弱环节，以及尚未

探索的领域，以期引出未来分布式数据库的研究和发展方向。

1 基本概念的重新梳理

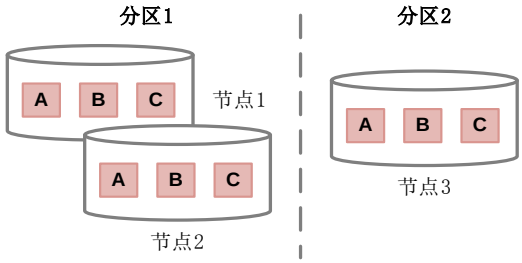

Fig.1 The replicas distribution of a distributed system under network partition

图 1 分布式系统中网络分区后的数据副本分布

分布式环境具有较高的不稳定性，任何一个部件都可能发生故障。通常我们可以将故障分类两类:1、节点故障;2、链路故障。节点故障通常导致单个或若干个节点暂时性或者永久的不可服务。产生该类故障的原因可能是服务器的硬件损坏、电力供应故障、或软件故障等。针对节点故障，通常分布式系统为每个节点增设副本。当单个节点故障时，可以由其他节点继续提供服务，保证系统的可用性。在网络链路良好的情况下， 系统可以充分同步对多个副本的写入内容，使得多个副本的存在对上层读写操作而言是透明的。然而，分布式系统也会碰到链路故障。链路故障是指多个节点间的通讯线路或交换机发生的故障。这类故障会导致网络分区[57]，即:一个分布式系统的集群被划分为若干个区域。在同一个区域中，节点间可以通信；跨区域的通信则出现困难。当多个副本分布在不同的网络分区中时，对一个副本的写入可能会无法同步到其他副本。那么，读取不同的副本将会返回不一致的结果。因此，由于分布式环境下存在的链路故障，系统通常需要在读写操作的一致性 (Consistency) 和节点的可用性 (Availability) 之间做权衡。这是 CAP 理论涉及的基本概念，也是前文提到的 NoSQL 数据库在设计时的主要权衡点之一。

然而 NoSQL 数据库的弱点是弱化了数据库系统的功能。典型的 NoSQL 系统仅提供 Key-Value 形式的存储和读取接口，不仅不支持关系模型和 SQL 语言，也不支持事务处理。它们假设这些功能由上层应用负责提供的，但这无疑导致应用的设计复杂度大大增加。应用开发者更希望分布式数据库能够提供传统数据库的完备功能。传统数据库的对外接口不再是简单的读写操作，而是具有较强表达能力的 SQL 接口。它的使用者也不再直接关注读写操作的一致性，而更多关注事务级别的一致性（即 ACID 性质）。除此之外，传统数据库的用户并不直接面对节点故障和链路故障，他们通常不在乎系统中的某个节点的可用性，而更多地关注整个系统是否可用。总而言之，CAP 理论和 NoSQL 系统所涉及的一致性 (Consistency) 和可用性 (Availability）概念在传统数据库的语境下发生了转移，不再是一个单一的非此即彼的折中问题。而在传统数据库的框架下，一致性和可用性的关系尚未被深入分析过。

鉴于 NoSQL 的局限性，工业界和学术界开始研究和实现 NewSQL 系统，力图将传统数据库的完整功能 NoSQL 系统良好的水平扩展性结合起来。Google 的 Spanner 是一款具有代表性的 NewSQL 系统。它不但实现了数据库完整的事务功能和严格的一致性保证，还具备极高的可用性和可扩展能力。最近，类似于 Spanner 的系统层出不穷，都号称能够同时提供较强的系统可用性和事务一致性。由此可见，数据库中一致性和可用性和分布式系统中的概念并不相同，不能直接套用 CAP 理论。在本节中，我们重点对 NewSQL 数据库系统中的一致性和可用性概念进行定义，并将其和分布式系统中的概念进行区分。

1.1 操作一致性

在分布式系统中，为了应对可能发生的故障，系统会为每一份数据维护多个副本，并存储在不同的节点

上。当某个节点无响应时，系统可以将服务迁移到其他节点上。上层应用的多次读写操作可以访问不同的副本。当某个副本被写入操作更新后，并且更新内容没来得及同步到另外一个副本时，对这两个副本的读取操作将会返回不同的结果。因此，读写操作是可能出现不一致的情况。为了限制不一致现象的发生，系统需要采取额外的管理机制。该机制对读写操作一致性的保障可以很强，也可以较弱。因而我们可以大致将分布式系统所提供的操作一致性分为两类：强一致性和弱一致性。

强一致性通常被等同于可线性化 (Linearlizability[25])。它要求：每个读写操作都发生于某个时间点（称为可线性化点），该时间点位于操作开始后、结束前；所有的操作的可线性化点可按时间顺序排成一个序列；每个读取操作都返回在它之间的最后一次写入操作写入的内容。如果强一致性的到保障，对上层应用而言，每个操作都是在它的可线性化时间点上即刻发生的。这可以方便应用确定读写操作的行为。然而，强一致性对写入数据在多个副本之间的同步有非常严格的要求。例如图 1 中，读取操作 R(A)在t_R时刻开始并尝试访问节点 1 中的副本，与此同时写入操作 W(A=a)修改了数据 A 在节点 3 上将 A 的取值更新为 a。可线性化的一致性模型要求：要么在t_R前将 W 的修改同步到节点 1 并能被 R 读取到，要么在t_R后才宣告 W 结束。因为根据可线性化的定义，如果 W 的线性化时间点位于 R 之前，W 的写入对 R 而言必须是可以读取到的。如图 1 所示，如果系统遇到了链路故障导致节点 1 和节点 3 无法通信，那么 W 对节点 3 上 A 的更新无法被同步到节点 1。此时，为了保证强一致性，系统要么宣告 W 失败，要么禁止所有操作再对分区 1 中的 A 进行访问。

弱一致性[63]放松了可线性化对读写操作的要求。通常在弱一致性中，一次写入操作 W 完成后，系统无法保证之后开始的读取操作都能获得 W 写入的内容。在弱一致的前提下，系统仍然可以采取措施减少不一致所带来的异常。典型的方案是保证最终一致性(Eventual Consistency[24,55]，弱一致性的一类)。在最终一致性中，即便多个副本的状态没有充分同步，每个副本依然可以处理读写操作。为了确保多个副本都收敛到相同的数据状态，最终一致性的系统通常使用某种规则在多个差异的副本状态中选择一个作为最终状态。例如，系统可以以最后写入的版本作为数据的最终状态。使用最终一致性需要容忍一段时间内可能发生读取数据的不一致，但在一段时间后多个副本能够被充分同步，使得读取操作最终能够返回相同的数据。如图 1 所示，当发生分区时，即使 W 对分区 1 中 A 的更新无法被同步到分区 2，最终一致性仍然允许后续操作访问分区 2 中的 A，这可以保证较高的系统可用性。在微博等社交网站中，最终一致性常常被采用；由于同一个账户的数据通常在同一个时刻只被一个客户端更新，双重更新发生概率很小。

1.2 事务一致性

操作一致性考虑的是单个数据项上的读写操作需要满足的一致性语义。事务一致性考虑的是多个数据项上读写操作序列需要满足的一致性语义。一个事务是由多个数据项上读写操作所组成的程序。严格的事务一致性要求所有事务之间存在一个次序，事务的实际执行效果等价于按照该次序顺序执行这些事务，即可串行化（Serializability）。例如，存在事务 T_1：$W_1(A), W_1(B)$以及 T_2：$R_2(A), R_2(B)$，每个事务依次执行各自的操作。严格的事务一致性仅允许以下两种实际调度次序的：1）$W_1(A),W_1(B),R_2(A),R_2(B)$，即 T_2 在 T_1 完成后开始；2）$R_2(A),R_2(B),W_1(A),W_1(B)$，即 T_1 在 T_2 完成后开始。但事务的可串行化与操作的可线性化不同。如果时间上 $W_1(A)$发生在 $R_2(A)$之前，那么操作可线性化要求 $R_2(A)$一定能读到 $W_1(A)$写入后的数据；但事务可串行化则允许 $R_2(A)$读取 $W_1(A)$写入前的数据，即将 $R_2(A)$调度到 $W_1(A)$前，只要最后的执行效果与 $R_2(A),R_2(B),W_1(A),W_1(B)$等价。与此同时，满足操作可线性化的调度可以是：1) $W_1(A), W_1(B), R_1(A), R_2(B)$、2) $W_1(A), R_2(A), W_1(B), R_2(B)$、3) $W_1(A), R_2(A), R_2(B), W_1(B)$；其中仅有调度 1）是事务可串行化的。事实上，事务一致性定义的是事务之间需要满足的顺序语义，而操作一致性定义的是操作之间需要满足的时序语义。有时候，应用对系统既有事务一致性的要求又有操作一致性的要求。例如，严格可串行化[46]（Strict Serializability）和外部一致性（External Consistency）既要求事务的可串行化又要求操作的可线性化。

传统的事务一致性要求系统满足原子性 (Atomic)、一致性 (Consistency)、隔离性 (Isolation) 和持久性 (Durability)[13]四个性质。其中，原子性要求一次事务的读写操作或者全部完成，或者全部失败；一致性要求被

事务修改后数据库的完整性 ⌣不会被破坏；持久性要求事务写入的数据不会丢失；隔离性要求事务的调度满足可串行化或其他隔离级别。传统数据库系统依赖日志[33,47,49,51,59]保证事务的原子性和持久性；对于数据一致性，通常是由应用定义的；为了保证一致性，系统会对事务执行产生的数据库中间状态进行完整性检查，若完整性被破坏则中止事务的执行。以上三点与本文要讨论的可用性没有明显的相关性，而且数据库系统都有成熟的机制进行保证，因此此后的章节不再过多涉及。本文重点要讨论的是隔离性，即上面提到的事务之间的顺序语义。后文所谓的事务一致性都仅仅指隔离性。

隔离性的保证依赖于并发控制技术[40,41,42]。严格的隔离性，如可串行化，在实际应用会造成并发事务之间更多的冲突和阻塞，从而造成性能损失。在实际运用中，隔离性通常会被放松，从而弱化了事务一致性。依据隔离性被弱化的程度，出现了不同的隔离级别，包括可串行化、可重复读、快照隔离、读已提交、读未提交[15]等。可串行化保证了严格的事务一致性。其余的隔离级别均允许特定的调度异常出现，因此都可以称为弱一致性。不同隔离级别允许的异常调度将会在第 3 节中阐述。根据对异常调度的排除程度，图 2 列出若干隔离级别之间的强弱关系。越强的隔离级别遇到的异常调度种类越少，从而保证了更强的事务一致性。

与操作一致性一样，事务一致性也会受到系统故障的影响。以图 1 的场景为例，假设两个事务 T_1：$W_1(A)$, $W_1(B)$和 T_2：$R_2(A)$, $R_2(B)$先后发生。由于链路故障的出现，T_1 在分区 1 上的修改无法被同步到分区 2。为了保证事务的可串行化，T_2 既可以在分区 1 上执行又可以在分区 2 上执行。如果 T_2 在分区 1 上执行，事务的串行顺序为 T_1T_2。如果 T_2 在分区 2 上执行，事务的串行顺序可理解为 T_2T_1。但是，如果在发生链路故障时，T_1 对 A 的修改已经同步到了分区 2，而对 B 的修改来不及同步到分区 2，那么 T_2 将无法在分区 2 执行，分区 2 将暂时变得无法使用。

可串行化 > 可重复读 / 快照隔离 > 读已提交 > 读未提交

Fig.2 The relationships among different isolation levels

图 2 隔离级别间的关系

1.3 系统可用性

可用性是指当发生节点或链路故障后，系统中的所有节点或部分节点是否可以继续服务上层业务的请求。根据故障对系统服务能力产生的影响，本文将不同系统所能提供的可用性分为以下三个级别；节点级高可用、服务级高可用、人工介入后可用。为了方便讨论，我们只考虑链路故障，即出现网络分区的情况。节点故障可视为一种特殊的网络分区--故障节点除了和其余节点分离之外，与用户也发生了分离。

节点级高可用：当出现网络分区后，所有节点都能继续对外提供服务。例如，图 1 中 3 个节点间发生了网络分区，将集群划分为两个集合。此时，节点级高可用要求节点 1、节点 2、节点 3 均能够响应用户请求。显然，某个写操作修改了节点 3 上数据项 A 的值，本次修改将无法同步到节点 1 和节点 2。之后发生的读取操作无法从节点 1 或者节点 2 上读取到之前的写入，即无法保证可线性化。针对这种情况，CAP 理论形式化的证明了：一个分布式系统在考虑网络分区的情况下，不可能同时保证操作的强一致性和节点高可用性[14]。可以说，CAP 理论和 NoSQL 系统谈及的可用性概念是“节点可用”的概念。

服务级高可用：当出现网络分区后，只要分区的严重程度有限（比如某个分区中存在多数派），那么整个系统任然可以继续对外提供服务。服务级高可用不要求故障后所有节点都可用，但要求一定存在部分节点可用，而且要求切换到可用节点的过程是全自动的，这样可以保证服务不中断。例如，图 1 中集群发生网络分区后，服务级高可用要求分区 1 和分区 2 中至少有一方能够响应用户请求。如果系统选择分区 1 继续提供服

⌣ 完整性定义了对数据库状态的一组约束。当这些约束均能够满足时，数据库状态是正确的。

务，那么可以中止分区 2 的服务行为；这样做的好处是可以避免两边同时服务造成的数据不一致。

对于数据库系统而言，服务级高可用几乎能够满足绝大部分应用的需求。因此，传统数据库和 NewSQL 系统所谈及的可用性概念通常是“服务可用”的概念，而非“节点可用”的概念。服务级高可用要求可用节点的切换是全自动的。在此前提下，为了实现数据的一致性，系统通常需要采用分布式共识协议 (如：Paxos[26,58,61,62], Raft[27])。这些协议要求：当系统中由超过半数节点构成的子集群通信正常时，系统才能对外提供服务的。在图 1 的例子中，节点 1 和节点 2 所在的网络分区是一个包含超过半数节点的分区，分布式共识协议能够保证该分区中的节点依然能提供一致性保障的数据服务。因此，服务级高可用能够容忍的故障规模是有限的。若集群发生了严重的链路故障，服务级高可用的系统同样会停止服务。例如，若图一中节点 1 与节点 2 的通信也存在异常，此时任意节点间都无法互相通信，系统可能不得不让三个节点都停止服务。

人工介入后可用：当出现网络分区后，系统暂时不可对外服务，只有在人工介入确认系统状态或采取某种补救措施后，系统才能继续提供服务。相比于前两类，这类系统在发生故障后会出现服务中断。这种中断通常是为了保证数据的一致性不得已而为之的。例如，当图 1 中的分区出现后，我们可能需要人工确认分区 1 和分区 2 各自的状态，再决定应该将服务切换到哪个分区才能保证数据的一致性。在没有共识机制的情况下，这样的人工干预是必要的。因此，系统的故障恢复时间也取决于人对故障的反应速度和应对速度。这类系统的典型代表是采用主从复制技术的传统数据库系统。当节点或者网络故障时，需要 DBA 确定故障对系统造成的影响，并采用特定的对策恢复系统。

综上所述，NoSQL 和 CAP 理论谈及的可用性与 NewSQL 和传统数据库谈及的可用性在概念上是有所区别的。前者所谓的高可用是指节点级高可用，后者所谓的高可用是指服务级高可用和人工介入后可用。这导致 CAP 并不能直接套用于数据库对可用性的讨论。只有在明确概念之后，数据库可用性与一致性的关系才能讨论清楚。

2 系统分类

在 CAP 理论看来，由于存在链路故障和节点故障的可能性，分布式系统需要在操作一致性和可用性之间进行取舍。系统对操作一致性的支持程度又会影响其对事务一致性的支持。本节对现实的数据库系统在操作一致性，事务一致性和可用性上进行定性分析。图 3 通过可用性，操作一致性和事务一致性对常见的数据库系统进行了分类。可以看到，不同系统对各个属性的支持程度是不同的，这既取决于应用的需求，又取决于各属性之间的制约关系。在下文中，我们将列举和分析每一个类型的典型系统。

2.1 类型I

第一类系统在操作一致性方面提供强一致性，但不支持事务，因而不保证任何级别的事务一致性。此类系统的代表有 BigTable[28]、HBase 等。这类系统通常以扩展性和较强的操作一致性为首要目标，在此基础上尽可能提升系统的可用性。然后，根据 CAP 理论，由于分布式环境下存在网络分区的故障，为了保证操作的强一致性，系统必然无法达到节点级别的高可用。

BigTable 由 Google 公司开发，支持结构化数据的存储。HBase 是开源社区对 BigTable 的仿制。BigTable 主要由三部分构成：Chubby[29]，Tablet Server[50]和 GFS[30]。BigTable 将数据库划分为多个 Tablet。Chubby 将每个 Tablet 分配到唯一的 Tablet Server 上。因此，对 Tablet 的多次操作会由相同的 Tablet Server 服务，从而保证了数据访问的强一致性。BigTable 能够做到服务级高可用性，这主要得益于 Chubby 和 GFS 提供的高可用性保障。

Chubby 是一项分布式元数据管理服务，它利用 Paxos 协议维护多个副本，能够应对半数以下的节点故障或节点通信故障，因此 Chubby 可以帮助其他系统实现服务级的高可用。Chubby 会持续监测集群中的 Tablet Server 是否发生了故障。当一个 Tablet Server 故障或者与 Chubby 无法联系时，Chubby 会指定一个其他 Tablet Server 替换故障节点。新的 Tablet Server 能够保证恢复所有故障节点上提交的数据。当发生网络分区时，若某个 Tablet Server 与 Chubby 之间出现通信困难，此 Tablet Server 会自动停止工作，Chubby 则会认为当前的节

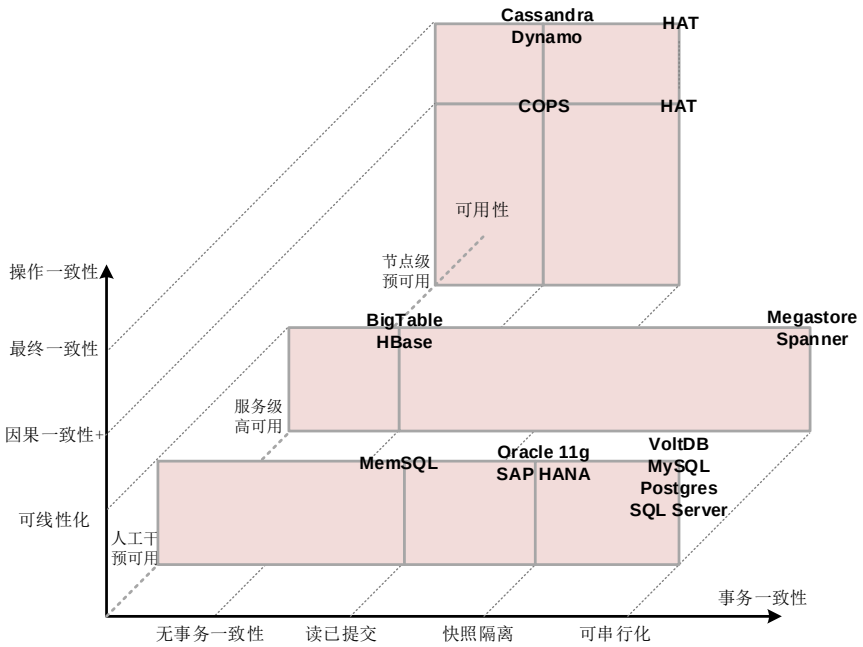

Fig.3 A taxonomy of different systems based on their consistency and availability

图 3 基于一致性，可用性的系统分类

点发生了故障，并指定一个新的节点恢复 Tablet 的数据并继续提供服务。

GFS 是一个高可用的分布式文件系统，用于保存 BigTable 的数据文件和 Redo 日志文件。在 GFS 中，所有的文件都存在多个副本，以保证其可用性。任何写入操作在修改数据前都需要向 GFS 写入 Redo 日志。当发生 Tablet Server 故障后，Chubby 会要求将故障节点上的 Tablets 服务迁移到新的节点上。新节点读取 GFS 中的 Redo 日志将对应的 Tablets 恢复至最新状态。由于，所有的数据修改会在 GFS 上写入 Redo 日志，因此新的 Tablet Server 能够通过 Redo 日志恢复出故障节点上所有已提交的数据。另外，GFS 也是一个服务级高可用的系统。在 GFS 中，同一份数据会在多个节点上保存副本。一个全局的 Master 节点会监控节点的故障并指定新的节点替换故障节点。全局 Master 节点的故障需要人工修改。在 GFS 的新版本 Colossus 中存在多个 Master 节点同时服务。这些节点由 Chubby 管理，保证服务级可用性。

BigTable 中不支持事务级别的一致性。一方面，Google 内部的很多应用并不使用下层系统提供的事务级别一致性保障。另一方面，提供事务一致性的支持会增加系统的复杂度和开发难度。因此，BigTable 在设计之初放弃了对事务一致性的支持。后续，Google 研发团队陆续推出了 Megastore[31]以及 Spanner[12]，以提供对事务一致性的支持。

2.2 类型II

第二类系统提供较弱的操作一致性，并且也不提供事务一致性保证。此类系统的主要代表有 Cassandra[7], Dynamo[10]等。这类系统以提升系统的可用性为第一要务，同时尽量提供能够满足业务需要的操作一致性性。当以系统可用性为主要优化目标时，这些分布式系统会支持最高的节点级高可用。即，这类系统能保证任何在线节点都能对接收到的请求做出响应，即便在网络分区的情况下。他们因而会在操作一致性上作出一些让步。

Cassandra 和 Dynamo 具有类似的系统架构。我们以 Cassandra 为例介绍它们的设计。系统中所有的节点都是对等的。与 BigTable 等系统不同，Cassandra 不依赖于特定节点提供元数据管理服务，数据通过一致性哈

希被分配到不同节点中。系统中每个节点都可以服务读写操作，不作主备节点的区分。节点之间通过 Gossip 协议同步写入数据。事实上，Cassandra 并不是纯粹的提供最终一致性和节点级高可用。它可以让使用者在两者之间作选择。用户有如下的系统配置选项：1）某个数据的副本数量（N）完成一次写入操作；2）数据需要被同步到的最少副本数量（W）；3）完成一次读取操作，至少需要访问的副本数量（R）。当$W+R>N$时，读取操作返回的数据必然是最近一次写入的内容，即保证了强一致性；当$W+R\leq N$时，读取操作不能保证返回最近写入内容，因而只能保证最终一致性。相应，W 和 R 的值越大，系统的节点可用性越弱。

COPS[32]是由学术界设计和实现的原型系统，用于验证当满足节点级的高可用下，分布式系统是否能够支持高于最终一致性的操作一致性。COPS 论证了支持节点级高可用的系统可以实现一种强化的因果一致性。首先强化的因果一致性具有最终一致性的特点，即对同一个数据项的并发写入最终会收敛到相同的状态。此外，这种一致性具备因果一致性[34]的特征。因果一致性首先定义了两个操作间的因果关系：如果两个操作若由同一个客户端发起，或者一个读操作读取了另一个写操作的写入，那么两个操作存在因果关系；并且因果关系具有传递性。因果一致性保证：对于存在因果关系的多个操作，它们的实际调度顺序与因果关系一致。在该模型下，若操作之间存在因果关系，上层应用是可以确定这些操作在系统内的实际调度次序。

2.3 类型III

第三类系统支持事务，它既提供最强的操作一致性也提供最强的事务一致性，与此同时还提供服务级高可用性。这类系统通常以事务一致性作为设计的首要目标。其代表有 Google Spanner 和 Megastore。由于对事务一致性的要求，它们难以提供的最高级别的可用性（即节点级高可用），但可以利用共识机制达到较高级别的可用性 – 服务级高可用。

Megastore 是 Google 开发的分布式数据库系统，它使用 BigTable 存储数据，提供操作的强一致性。同时，它通过构建事务管理层提供可串行化的事务一致性保证。在此前提下，它还提供服务级高可用性。Megastore 的操作强一致性是由 BigTable 保证。因此 Megastore 所有的读写操作都是通过 BigTable 完成的。

Megastore 将数据库划分为多个实体组（entity group）。本质上，一个实体组对应了数据库的一个数据分区。每个分区的事务管理都有一个节点完全负责。每个分区节点都构成一个独立的数据库。当事务仅访问单一分区时，Megastore 利用乐观并发控制保证了可串行化的隔离级别。当事务需要访问多个分区时，通常需要两阶段提交协议保证提交的原子性。其代价是增加事务处理的延迟和冲突。因此，Megastore 提出了一种基于消息队列的事务管理机制。该机制将事务分解为若干个在单个分区内独立执行的子事务。这提升了性能，但牺牲了跨分区事务的一致性。

为了保证服务级高可用，Megastore 要求为每个分区节点设置多个副本。事务首先在主副本上执行。在提交阶段，事务首先在主副本上写 Redo 日志，然后使用 Paxos 协议将日志同步到半数及以上的备副本中。当主副本故障时，Paxos 协议会将某个备副本提升为新的主副本并延续事务管理的服务。

Spanner 是 Google 设计的跨地域、多副本的数据库系统。Spanner[45]保证了操作的强一致性以及可串行化的事务一致性，同时能够提供服务级高可用。Spanner 为每个数据记录维护了多个副本，并设定其中一个为主副本，其余的为备副本。对数据记录的所有写入操作到发生在主副本上。类似于 Megastore，Spanner 将修改内容通过 Paxos 协议同步到备副本上，并以此实现了服务级的高可用性。读取操作存在两种执行策略：1）直接读取主副本；2）访问主副本获得最新的数据版本信息，从备副本读取指定的数据版本。在两种情况下，读取操作都会返回最新的写入数据。因此，保证了操作的强一致性。

Spanner 将数据库水平划分为多个数据分区。系统采用两阶段锁隔离并发写事务。当写事务需要访问多个数据分区时，Spanner 使用两阶段提交协议[52]保证提交的原子性。在写事务提交时，Spanner 能够为其分配提交时间戳。该时间戳不仅对应了正确的事务串行化顺序，同时对应了写入操作的可线性化顺序。在此基础上，只读事务的可串行化仅需要通过比较时间戳来保证，而无需使用两阶段锁。这避免了只读事务对其他事务造成阻塞。

2.4 类型IV

第四类系统提供较强的操作一致性和事务一致性，但在可用性上做了一定妥协，只提供人工介入下的可用性。这类系统包括以 VoltDB[11]为代表的 NewSQL 数据库以及传统的基于主从复制技术的关系型数据库。

VoltDB 是新型的无共享、分布式内存数据库系统。它保证了可串行的事务一致性，但不提供服务级高可用，而只实现了人工介入下的可用性。在 VoltDB 中，当写事务尝试修改数据项时，系统要求写事务在修改同步到数据项的所有副本后再提交，因此 VoltDB 保证了操作级的强一致性。

VoltDB 提出了一种新型的事务执行模型。在 VoltDB 中，数据库水平划分为若干个分区。每个分区都绑定到固定的物理线程上。访问相同分区的所有事务都会在对应线程的任务队列中排序后依次执行。在一个分区上，线程总是执行完当前事务后才开始执行下一个事务。相对于其他事务型数据库系统，VoltDB 的事务执行模式不依赖于并发控制技术。由于这种事务调度模式总是在事务执行前确定它们的调度顺序，因此这种策略通常称为确定性执行策略[43,44,48]。对于需要访问多个数据分区的事务，VoltDB 依赖于两阶段提交协调多个线程共同完成事务的执行。

VoltDB 为每个数据分区维护了多个副本。同一个数据分区的多个副本会分布到不同的物理节点上。对任意一笔写事务，VoltDB 首先确定它需要访问的数据分区，而后会将该请求同步到所有副本节点上。多个副本会并行的执行该请求。由于在确定性执行策略下，同一个分区的多个副本上会接受到相同的事务请求序列。因此，分区的多个副本在处理完相同的事务序列后依然会保持相同的数据状态。VoltDB 能够容忍部分节点故障。若在存活的节点上所有数据分区依然保留有副本，VoltDB 便能够对外提供服务。但是系统本身无法区分节点故障和链路故障。当链路故障发生后，可能会有多个网络分区同时对外提供服务，数据库的状态会产生分叉，导致数据的不一致性。因此，VoltDB 依赖于外部的网络分区检测工具判定是否发生了链路故障。若检测到网络分区发生，外部工具会仅保留拥有节点数量最多的那个网络分区，并强制关闭其他网络分区中的节点。外部工具通常依赖 DBA 控制和维护，因此 VoltDB 只能提供了人工干预下的可用性。

传统的关系型数据库（例如 Oracle，MySQL）通常使用主从复制技术提升系统的可用性。在这些系统中，所有的读写操作都是在主数据库中完成的。因此，这些系统实现操作一致性和事务一致性的工程代价和性能代价都相对较低。在理论上，这些系统可以支持可串行化的事务一致性。但是，保持最严格的事务一致性会产生较大的性能代价。因此，实际中的数据库系统也兼容较低级别的事务一致性。甚至一些系统抛弃了可串行化的隔离级别。例如，Oracle 11g 最高仅支持快照隔离，MemSQL[4]仅支持读已提交。部分应用对隔离级别要求不高；通过弱化隔离级别，系统可获得整体性能的提升。

传统数据库为了提升系统的可用性，需要使用主从复制技术维护若干备数据库。通常，在写事务提交阶段，主库不仅需要将事务的日志写入本地磁盘，还需要将他们复制到备数据库。日志复制包含两种模式：异步模式和同步模式。异步模式允许在日志写入备库前向客户端确认事务提交成功。该模式的优点是事务提交延迟低。但当主库发生故障并将服务迁移到备库时，部分未复制到备库的写入数据将丢失。同步模式要求在日志写入备库后向客户端响应事务提交成功。该模式下，系统将服务从主库切换到备库不会导致数据丢失，但事务提交延迟将变高。

在保证较高一致性的要求下，简单的主备复制方案仅能达到人工介入下的可用性。通常这种方式无法自主地应对故障。例如，当主备间发生链路故障时，备库无法确定故障的原因和主库的状态。即便主库已经因为故障停止服务了，备库也需要有外部确认后才能开始服务上层业务。

事实上，传统的数据库系统可以通过改进日志复制技术来提供服务级的高可用。例如，MySQL Replication Group（MGR）是针对 MySQL 设计的日志同步组件，替换了原有的主从复制技术。MGR 使用了 Paxos 共识协议在 MySQL 的多个副本间同步事务日志。当集群中超过半数的 MySQL 实例间通信正常时，系统可以对外提供服务。

2.5 类型V

第五类系统提供操作的弱一致性，同时提供一定程度的事务一致性。此类系统为了支持节点级高可用性，弱化了操作一致性。事务是由操作组成的，较强的事务一致性需要建立在较强的操作一致性基础之上。对于如何在较弱的操作一致性上尽可能提升事务一致性，Peter Balis[22]在他的 HAT 论文中进行探讨。根据他的分析，在操作弱一致性和其他一些限定条件下，事务一致性最高能够达到读已提交的级别。因此，HAT 系统可视为提供较弱的操作一致性和事务一致性，但具备节点级高可用能力的系统。然而，HAT 只处于理论阶段，当下尚没有系统的工程实践案例。因此，此类系统是否在工程上和实际应用中可行尚未可知。

3 讨论

本文以操作一致性、事务一致性和可用性作为分布式数据库分类的坐标。本章对着三者之间的关系进行分析和讨论。

链路故障和可恢复性的节点故障本质上均可认为是导致网络分区的故障（节点故障造成的分区导致故障节点成为孤岛）。本章将讨论在系统产生分区的情况下，一致性与可用性之间的关系，一致性主要涉及事务的隔离级别与操作一致性。为了简化描述，假设系统中最多有两个分区，每一分区内部的副本节点之间可以互相通信，而分区之间则无法进行通信。我们称副本节点数据较多的那个分区为主分区，另一个称之为副分区。如图 1 所示的系统模型，系统中共有 N=3 个副本节点，每个节点均存储全量数据，所有的节点被分为两个集合，分区 1={节点 1，节点 2}，分区 2={节点 3}，分别为主分区和副本区。节点 1 和节点 2 之间可以互相通信，但与节点 3 无法连接。

在该系统模型下，客户端与服务端将采用会话的机制。具体来说，客户端向服务端发送读写操作之前，需要与某个分区上的副本节点建立会话连接。连接建立成功后，客户端与服务端将维持一个特有的会话，该客户端后续的读写操作均属于该会话。客户端的所有操作均串行执行，即发起新的操作的前提是已收到前一个操作的服务端响应。当连接断开后，客户端与服务端的会话结束。如果客户端想要继续读写数据，则需要重新建立新的会话。

本文的事务模型将采用读-修改-写（Read-modify-write）。具体来说，一个客户端上的事务先从服务端读取一个数据项，然后根据读的结果在本地修改某一个数据项，最后将修改后的结果写回到至服务端。我们假设一个事务可以包含多个操作，但最多仅包含一个写入操作，且该写入操作为这个事务的最后一个操作。例如，如果一个事务包含两个操作，则该事务的第一个操作为读取操作，后一个操作为写入操作。写入操作默认包含对本事务的提交，并且系统保证所有单一操作的原子性。我们用 $R_1(A=a)$表示事务 T1 读取数据项 A 的操作并获取数据 A=a，用 $W_1(A=a)$表示事务 T1 更新数据项 A 为 a 并进行提交。如图 4 所示的执行序列：

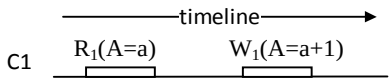

Fig.4 An example for the operation sequence of a transaction

图 4 事务执行序列样例

事务 T1 运行在客户端会话 C1 上，其包含两个操作，并先执行 R_1 再执行 W_1。在下文中，如果没有特别说明，我们将默认客户端会话 C1 连接在主分区上，客户端会话 C2 连接在备分区上。

3.1 操作一致性与事务一致性的关系

一致性需要保证不同操作之间（不同事务之间）满足某些条件或者避免某些异常。为了满足不同的需求，一般将一致性分为多个级别。在操作一致性中，一致性级别为不同的一致性模型；而在事务一致性中，一致

性级别为不同的隔离级别。本节将主要介绍操作一致性和事务一致性的衡量方式以及它们之间的关系。

3.1.1 操作一致性的衡量

在分布式系统中，操作一致性一直是热点问题，最近 Viotti 等人对非事务系统中的一致性进行了详尽的综述[35]。我们将通过操作特性来描述不同的一致性模型，表 2 展示了典型的一致性模型与操作特性之间的关系。通常，以下特性越能得到保障，操作一致性就越强。

最新性（Recency）：一个客户端会话向服务端发起对某数据项的读取操作，总能够获取到该数据项的最新版本。

单调读（Mononic Reads）：一个客户端会话的任意两个读取操作，后一个读取操作从服务端获取的数据版本不旧于前一个读取操作获取的数据版本。

单调写（Mononic Writes）：一个客户端会话的所有写操作在服务端可见的顺序将按照这些写操作的提交次序。

读后写（Read Your Writes）：一个客户端会话 C1 先执行读取操作，获取到另一个客户端会话 C2 写入操作 W_1 的结果，然后再执行写入操作 W_2，则如果一个客户端会话能够看到 W_2 的结果，那么他也能够看到 W_1 的结果和 C2 上任何早于 W_1 的写入操作。

写后读（Writes Follow Reads）：一个客户端会话先执行对某个数据项进行写入操作，再读取该数据项，则该读取操作读到的结果不旧于自己写入的数据。

Table 2 The relationship between operational consistencies and properties

表 2 操作一致性与特性之间的关系

	最新性	单调读	单调写	读后写	写后读
线性一致性	是	是	是	是	是
因果一致性	否	是	是	是	是
PRAM	否	是	是	是	否

单调读、单调写、读后写和写后读均属于会话特性。因果一致性模型满足所有的会话特性，前面提到的线性一致性模型不仅满足所有的会话特性，还满足最新性。

3.1.2 事务一致性的衡量

ANSI SQL-92 根据异常现象定义了四种隔离级别，Bernstein 等人对此进行了详细的分析，并增加了一种新的隔离级别，即快照隔离[15]。由于读未提交隔离级别在现实生产系统中极少使用，本文主要关注读已提交、可重复读、快照隔离和可串行化，表 1 展示了事务的隔离级别与异常现象之间的关系。总体而言，越强的事务一致性能够规避越多的异常现象。

脏读（Dirty Read）：两个并发的事务 T1 和 T2，T1 修改了某一个数据项，然后 T2 读取该数据项并获取到了 T1 对其修改的结果，如果 T1 由于某些原因被回滚，则 T2 读到了一个不存在的结果。

不可重复读（Non-repeatable Read）或模糊读（Fuzzy Read）：两个并发的事务 T1 和 T2，T1 先读取了某个数据项，T2 再修改或者删除了该数据项并提交，然后 T1 再一次的读取这个数据项，它会发现该数据项的值发生了变化或者该数据项已被删除。

幻读（Phantom）：两个并发的事务 T1 和 T2，事务 T1 根据某个谓词读取了一个数据集合，事务 T2 新建了一个数据项并提交，该数据项满足之前事务 T1 的读取谓词，然后事务 T1 再次执行读取操作获取满之前谓词的数据集合，并发现前后两个数据集合不相等。

丢失更新（Lost Update）：两个并发的事务 T1 和 T2，T1 先读取了一个数据项，T2 修改了该数据项并提交，然后 T1 修改了这个数据项并提交。

写倾斜（Write Skew）：两个事务的读写集合互有交集，且不满足读写反依赖。例如，在避免脏读的事务系统中存在两个并发事务 T1 和 T2，T1 和 T2 均未提交。事务 T1 读 X 写 Y，事务 T2 读 Y 写 X，因此 T1 和

T2 的读写集合互有交集。在数据项 X 上，T1 和 T2 的顺序为<T1, T2>；在数据项 Y 上，T1 和 T2 的顺序为<T2, T1>。因此，事务 T1 和 T2 不满足读写反依赖。

Table 1 The relationship between transactional isolation levels and anomalies

表 1 事务隔离级别与异常之间的关系

	脏读	不可重复读	幻读	丢失更新	写倾斜
读已提交	无	有	有	有	有
可重复读	无	无	有	无	无
快照隔离	无	无	无	无	有
可串行化	无	无	无	无	无

3.1.3 操作一致性与事务一致性之间的关系

事务一致性的级别可以看做是不同的隔离级别；操作一致性的级别可以看做是不同的操作一致性模型。在分布式数据库中，操作一致性和事务一致性都需要被实现，两者关系密切。

首先，操作一致性是事务一致性实现的基础。当系统无法满足某种程度的操作一致性时，相应程度的事务一致性也难以得到保障。例如，如果系统无法满足“写后读”的会话特性，那么系统将无法避免“丢失更新”的异常 – 假设同一个会话连续提交两个事务 T1 和 T2；由于“写后读”无法保证，T2 可能读到 T1 修改前的数据版本并对其进行重复修改，造成 T1 的更新丢失。又如，当系统无法满足“读后写”的会话特性时，可串行化的隔离级别是无法保障的。目前，我们只知道操作一致性会制约事务一致性，但两者的具体关系尚不清晰，有待进一步的研究工作澄清。Peter Balis 在关于 HAT 的论文中阐述了：当操作一致性受节点可用性的制约而无法被完全保障时，事务一致性只能达到读已提交的级别；这算是对二者关系的一个局部分析结论。

其次，即使操作一致性得到了全面保障，这并不能说明事务也能得到保障。为了说明这一点，我们关注操作的线性一致性和事务隔离级别之间的关系，即当系统可以保障所有事务的操作满足线性一致性的时候，该系统依旧无法为事务提供可串行化隔离级别。这是因为事务的一致性不仅需要考虑操作之间的关系，还需要考虑不同事务之间的关系，而操作一致性仅需要考虑不同操作之间的关系即可。因此，即使系统提供线性一致性模型，也无法保证避免掉某些事务的异常现象，如不可重复读和丢失更新，因而无法提供可串行化隔离级别。如图 5 所示的执行序列：

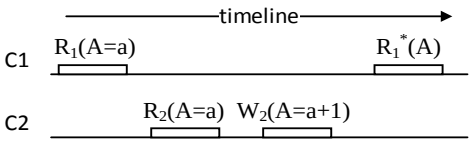

Fig.5 A schedule of the unrepeattable read anamoly

图 5 不可重复读的异常调度

事务 T1 和事务 T2 分别在客户端 C1 和客户端 C2 上执行，按照线性一致性模型，操作在服务端上执行的序列为 $R_1(A=a)R_2(A=a)W_2(A=a+1)R_1^*(A)$,因此事务 T1 的 $R_1^*(A)$应该获取到 A=a+1,这和该事务前一次的读取操作 $R_1(A)$返回的结果并不相等，属于典型的不可重复读异常。

事务系统为了避免这些异常，通常需要采取额外的手段来调整事务操作的执行序列。例如，系统可以采取两阶段封锁（Two-phase locking）的机制，构建一个共享的锁表。当事务想要执行一个操作的时候，只有从所表中获取对应数据项的锁权限后，才能真正执行该操作。

综上，操作一致性是事务一致性的必要条件。在实现操作一致性的基础上，事务一致性还需要进一步的并发控制机制才能得以保证。David Lomet 等研发的 Deuteronomy 系统[37] [38] [39]正是为了在保障操作一致性的

NoSQL 数据库上实现事务一致性。

3.2 节点级高可用与一致性的关系

在节点高可用系统中，无论副本节点在主分区还是备分区，当其收到到客户端的请求后，均能进行处理。由于不同分区间无法进行通信，整个系统无法维护一个全局时钟。但每个副本节点可以维护一个本地的局部时钟，该时钟在跨分区操作中作用有限。

3.2.1 节点级高可用与操作一致性

本小结将讨论节点级高可用系统能够实现的操作一致性模型。由于操作的特性与一致性模型有直接的关系，我们接下来分析在节点高可用的系统中，有哪些操作的特性可以满足。

最新性：不可满足。由于分区的存在，并且分区的时间可以任意长。因此当某个分区上的副本节点收到读取操作的请求后，如果请求的数据项在其他分区上有更新版本的数据，则该副本上的请求数据项无法满足最新性。如下面图 6 所示的执行序列：

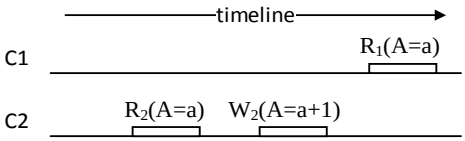

Fig.6 A schedule of the lost recency anamoly

图 6 缺少最新型的调度

C1 和 C2 分别连接到主备分区，由于系统为节点级高可用，C2 的写操作 W_2 可以顺利执行，但 A=a+1 并没有同步到主分区，因此 C1 上后续的读取操作 $R_1(A)$仍获取的是较旧的版本。

单调读、单调写和写后读：可满足。由于在同一个会话中的所有操作均由同一个副本节点负责处理，那么操作在服务端执行的顺序可以完全按照在会话中发起的顺序。因此，节点可用性系统可以保证同一个会话内的所有操作满足单调读、单调写和写后读。

读后写：可满足。副本节点在收到写入操作的请求后，并不立即将其结果作用于本地，而是将其缓存在一个单独的空间，客户端的读取操作暂时无法获取该写入的结果。当系统不存在分区后，所有的副本节点根据因果关系协商缓存的结果，并达成一个一致的结果。此时，客户端再访问任意一个副本节点，总能获取相同的结果。

由于节点级高可用系统无法满足最新性，因此无法对外提供线性一致性的服务。而单调读、单调写、写后读和读后写均可以得到满足，因此节点可用系统可以对外提供因果一致性和 PRAM 一致性的服务。

3.2.2 节点级高可用与事务一致性

本小结将讨论节点级高可用系统能实现的隔离级别。我们接下来分析，节点级高可用系统在分区的情况下，其能够避免哪些事务的异常现象。

脏读：可避免。在本文的事务模型中，由于每个事务仅包含一个写操作，并且系统保证该写操作的原子性，也就是说，当某个副本节点收到写入操作的请求后，需要保证该事务提交后，其对应的结果才可以对外可见。因此，在本文的系统模型下，系统可以避免脏读，可以提供读已提交的隔离级别。

不可重复读：可避免。由于一个事务的所有操作只能转发到同一个副本节点上，副本节点只需要为事务保存相应数据读取的版本，当事务再次请求相应的数据项的时候，副本节点返回保存的版本即可。当事务执行完成后，其对应的版本便可以删除。

幻读：可避免。由于同一个事务仅与一个副本节点保持连接，因此我们可以让副本节点为事务读取的数

据保存其对应的版本。当某个事务再次读取某个数据项的时候，可以将之前读取的版本直接返回。

丢失更新：不可避免。两个事务分别在主副两个分区上执行，这两个事务读取相同的数据项并获取到了同一个版本的数据，然后都修改了该数据项并将结果写回至各自的副本节点。由于分区间无法通信，事务在提交的时候无法对另一个分区上的并发事务进行检测。为了保证可用性，这两个事务都最终提交成功。如下图 7 所示的执行序列：

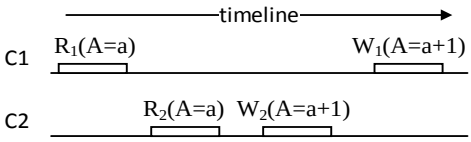

Fig.7 A schedule of the lost update anamoly

图 7 丢失更新的异常调度

事务 T1 和事务 T2 分别在 C1 和 C2 上，由于系统为节点节点级高可用，T1 和 T2 均提交成功，最终 A=a+1，但该结果并不符合 T1 和 T2 任何串行化执行的结果，即 A=a+2。

写倾斜：不可避免。丢失更新针对的是同一个数据项，如果同一个事务的读写发生在不同的数据项上，节点可用系统将会产生写倾斜异常。

我们可以发现，节点级可用系统可以避免脏读异常，因此这类系统可以对外提供读已提交隔离级别。上文中提到的隔离级别，除了读已提交，均需要避免丢失更新异常，而节点级高可用系统无法避免丢失更新，因此该类系统无法提供可重复读、可串行化等更高的隔离级别。

3.3 服务级高可用与一致性的关系

在我们的例子中，最多存在两个分区，并且两个分区的节点数目并不相等。如果系统为服务可用，那么客户端的请求只能在主分区的某个副本节点上得到处理。也就是说，副分区上的副本节点将拒绝处理客户端的读写请求。

因为所有的读写请求均发送到主分区的副本节点，且主分区的副本节点之间可以进行通信，我们可以采取全局时钟的方法为每一个操作请求分配全局唯一且递增的编号。当其中某节点在处理一个客户端操作请求的时候，需要保证该请求前面的操作均已执行。因此，服务可用的系统可以保证一致性模型里面所有的特性，可以对外提供线性一致性。但是，每当系统收到一个操作请求的时候，主分区内的节点均需要进行一次协商来为其分配编号。通常，协商/共识协议需要额外的节点间交互，这增加了每一个操作的延迟，从而会影响系统整体的性能。

事务系统需要维护额外的信息来保证事务的隔离性，我们可以在主分区的所有副本节点上维护一张全局的锁表，当其中的一个副本节点收到事务的一个操作请求后，就会对共享的锁表进行更新（获取锁相应数据项的锁权限）。当更新成功后，其他的副本节点均可以发现锁表的变更。因此，服务可用的系统可以采取两阶段封锁的机制来实现可串行化隔离级别。然而，一方面，主分区的副本节点之间需要通过协商来同步锁表的更新；另一方面，当事务操作的数据项已被其他事务获取锁权限的时候，该事务的操作会被阻塞。我们已经知道，一个事务执行的时间会影响与其他事务的冲突的概率[36]。在服务可用的系统中，由于事务执行时间变长，系统的整体性能会受到较大的影响。

因此，系统在满足服务级高可用的情况下，可实现线性一致性和可串行化隔离级别。但是，协商/共识协议的实现增加了系统工程开发的难度；由于协商/共识协议引入了额外的开销，与节点高可用系统相比，其性能降低。

3.4 人工介入后可用与一致性的关系

在传统的基于复制的分布式数据库系统中，在所有的副本节点中存在一个主副本节点，其他副本节点为备份节点。客户端只能与主副本节点建立会话连接，也就是说，所有客户端的操作请求均会发送给主副本节点，由该节点负责处理执行。因此，该主备份点可以对所有的操作请求进行调度，从而可以保障最新性以及会话特性，对外提供线性一致性服务。

我们可以在主副本节点上维护锁表信息，事务的操作请求需要先从锁表中获取对应数据的锁权限，再在主副本节点上执行处理。因此，人工介入后可用的系统可以采用两阶段封锁的策略保证事务的串行化执行，可以对外提供可串行化的隔离级别。

为了保证数据的持久化，该类系统通常采用积极机制（Eager）来复制数据。具体来说，当主备份节点收到事务的提交请求后，需要确保该事务在所有副本节点上都执行成功后才能提交该事务并响应客户端。但是，当系统产生分区的时候，主副本节点无法将数据同步至某些备份节点，这将导致不能及时对客户端的写入请求做出响应，因此系统将不再对外可用。如果主备份节点在主分区上，管理员需要为主分区添加新的备份节点；如果主备份节点在备分区上，管理员需要为主分区添加新的备份节点并选从中选择新的主节点。删除备分区中的节点后，新的主分区将继续对外提供服务。

我们可以看到，该类系统与服务级高可用系统均可以对外提供最强的一致性。而且，与服务高可用系统相比，人工介入后可用系统不仅在工程实现上更加简单，并且可以避免协商产生的代价，从而提高了系统整体的性能。但是当其出现分区的时候，需要管理员介入来恢复系统的服务，这大大降低了系统对外的可用性。

综上，对于可用性和一致性的关系，如果系统只提供服务级可用或更低的可用性级别，无论操作一致性或事务一致性都可以得到充分保障。但这样的一致性保障需要以性能为代价。在构建分布式数据库时，系统设计者还需要考虑到性能与可用性或一致性之间的平衡。例如，Spanner 同时保证了较高的可用性和一致性，但复杂的共识协议难免会给它造成性能上的损失。传统数据库系统则削弱了系统的可用性，从而获得了更好的性能。而大部分 NoSQL 数据库则放弃了一定的一致性，以获得可用性与性能的保障。

4 未来研究方向探讨

本文通过一致性和可用性对分布式数据库产品的格局进行了刻画。从这两个维度，我们清晰地看到不同产品之间的差异和差距，以及设计者的考虑和选择。由此，我们也看到一些重要却尚未被充分探索过的领域。对研究者而言，它们都是有价值的未来研究方向。本章选取了三个笔者认为重要的方向加以阐述。

（1）如何提高服务级高可用系统的读写以及事务性能

服务级高可用系统既具备较高可用性又可以提供强一致的数据服务。相比于其它系统，此类系统在功能上具备最高的普适性。但是由于协商/共识协议本身需要额外的开销，该类系统通常性能有限，妨碍了它成为通用的系统。因此，如何提高服务级高可用系统的性能在最近几年收到了研究者的关注[54]，也理应是未来研究的热点。

在跨数据分区的服务级高可用系统架构下，例如 Spanner，分布式事务的并发控制和数据更新的共识协议通常是两个相对独立的模块。事务在提交的时候，并发控制层需要采用多次交互（如 2PC 协议）来保证事务的原子性，协商层使用 Paxos 需要至少两次网络交互来保证数据的一致性。这使得系统在单次提交分布式事务时就至少需要四次网络交互。一方面，是否可以将多个事务进行批量处理，减少系统整体上的网络开销？另一方面，是否可以将 2PC 和 Paxos 的执行流程进行结合，减少单次分布式事务的网络交互次数？这些问题的答案尚不清晰。因此，在服务及高可用系统下，如何减少网络开销从而提升系统性能，将是未来研究者关注的热点

在同一数据分区内部，我们可以选取具有 leader 角色的副本节点来降低协商/共识的代价；该节点可以由本数据分区内部的所有复本节点通过 Paxos 协议选举出来并采用租约机制保持身份。但在该机制下，leader 节点将负责所有客户端的读写请求，使得其它副本节点不能被充分利用。一方面，是否可以通过降低一致性级

别（例如 Spanner 采用的快照读取操作），使得副本节点无需进行协商即可提供服务？另一方面，是否可以增加处理层次（例如，中间件），使得请求可以直接转发至满足一致性要求的副本节点？这些问题都值得探索。因此，在服务级高可用系统模型下，如何有效的利用副本节点的资源来提高系统整体的性能，也将是未来研究的热点之一。

（2）可用性的多样化与量化

数据库系统的高可用性对上层应用而言是非常重要的一个特性。例如，在一些电商或银行应用中，系统短时间的不可服务都会造成不可忽视的经济损失，甚至产生较大的社会影响。然而，提升系统的可用性可能会弱化系统的一致性。如上文所述，服务级高可用系统的性能在一致性和可用性方面达到了一定的平衡。一方面，它采用共识算法确保系统在发生局部故障是能够自主的在多个副本间迁移服务，避免了由人工介入来恢复系统服务。另一方面，它不需要牺牲系统的一致性。然而，服务级高可用系统中的协商/共识算法仅能容忍少数副本发生节点故障或者规模较小链路故障。因此，提升系统的高可用性可以考虑如何让服务级高可用的系统能够容忍半数以上的副本无法访问。这或许会牺牲一定的操作一致性或事务一致性，但这样的牺牲未必是不能容忍的。如何在服务级高可用和节点级高可用之间进行折中？这是一个值得探索的问题。

高可用性的支持通常伴随着比较高的性能代价。例如，在服务高可用的系统中，一次操作的修改需要持久化到半数以上副本中才认为提交完成了。这较大的增加了完成一次操作的延迟。多数情况下，配置良好的服务器集群在大部分时间中都会处于正常状态，仅在较短一段时间内会发生局部故障。为了应对小概率的故障，花费巨大性能代价是否值得？那么，是否能够在一定程度上弱化数据同步的要求来提升一般情况下系统的性能？例如，仅要求数据同步到某个副本的内存中。这可能导致系统在故障时不具备百分之百的自主恢复能力。但是，如果自主恢复在大部分情况下是可行的，对大部分应用是可接受的。如果在极少数情况下系统服务无法恢复，仍然可以述诸于人工介入。如何在服务级高可用和人工介入后可用之间进行折中？这也是值得研究的。

此外，系统的高可用性目前是一种很难具体量化的指标。本文从故障对系统产生的影响程度对高可用性进行了一定的分类。然而，对应用而言，它可能更加关心系统在实际部署后能够达到的可用性。通常，应用开发人员可以根据某个系统在真实案例中的表现作出评估。然而，实际系统的可用性是受到故障的类型、发生的频率等因素影响的。这些因素在不同的硬件环境中都是不同的。从研究的调度，我们是否可以设计出一套对高可用性的基准测试方案？它能够模拟不同类型故障发生的频率，并且统计系统可用的时间。最终，它能够输出一个系统可用性的报告，供开发人员参考。

（3）如何为应用选择系统

商业应用对系统一致性、可用性和性能的要求各不相同。例如，很多互联网应用对系统操作一致性和事务一致性要求相对低，但希望系统的可用性高；而面向金融的应用常常希望系统既支持较高的一致性又保证高可用。另一方面，即便应用对数据一致性有较高要求，它也可以选择对一致性支持较弱的系统，而通过应用逻辑保证一致性；这让应用既可以收获较高可用性又可以减少性能损失。如何根据应用的特点从种类繁多的分布式数据库系统中做出合理选择？这是大多数软件开发人员面临的一大难题。到目前为止，大多数研究都从分布式数据库设计角度出发探讨了系统可能适合的互联网应用，但这样的论述缺少对应用本身的深入分析，也很难做到包罗万象。对于一个特定的应用，如果它的功能和性能需求已知，那么它在不同系统上构建方式和构建代价到底如何？目前这方面的知识和经验总结还处于缺失状态。这也是一个重要且艰巨的研究课题。

5 总结

本文从一致性和可用性的维度分析和总结了分布式数据库系统的设计空间，并对当今分布式数据库的产品格局进行了刻画。本文的目的并非提出新的概念、想法或技术，而是对现有技术进行梳理，从而对它们的能力边界有一个宏观认识。从这个意义上讲，本文是一篇综述文章，希望能够为当今的数据库系统设计者提

供参考。由于分布式数据库系统的研究历史很长，文献可谓汗牛充栋，笔者没有能力将所有与一致性或可用性相关的技术全部纳入到文章中，甚至可能遗漏掉一些重要文献，也希望读者阅读时持开放怀疑的态度。

References:

[1] Clustrix: https://www/clustrix.com/.

[2] eXtremeDB: https://www.mcobject.com.

[3] HBase: https://hbase.apache.org/.

[4] MemSQL: https://www.memsql.com/.

[5] NuoDB: https://www.nuodb.com/.

[6] Riak: https://basho.com/products/.

[7] Lakshman A, Malik P. Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev., 2010, 44(2):35–40.

[8] Plugge E, Hawkins T, Membrey P. The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing. Apress, 2010.

[9] Anderson JC, Lehnardt J, Slater N. CouchDB: The Definitive Guide: Time to Relax. O'Reilly Media, Inc., 2010.

[10] DeCandia G, Hastorun D, Jampani M, et al. Dynamo: Amazon's highly available key-value store. In Proc. of 21st SOSP, 2007, 205-220.

[11] Stonebraker M, Weisberg A. The VoltDB Main Memory DBMS. IEEE Data Eng. Bull., 2013, 36(2): 21-27.

[12] Corbett JC, Dean J, Epstein M, et al. Spanner: Google's Globally Distributed Database. ACM Trans. Comput. Syst., 2013, 31(3): 8:1--8:22.

[13] Gray J, Reuter A. Transaction Processing: Concepts and Techniques. Morgan Kaufmann, 1993.

[14] Gilbert S, Lynch NA. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News, 2002, 33(2): 51-59.

[15] Berenson H, Bernstein P, Gray J, et al. A critique of ANSI SQL isolation levels. In Proc. of SIGMOD. 1995. 1-10.

[16] Adya A, Liskov B H. Weak consistency: a generalized theory and optimistic implementations for distributed transactions. Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1999.

[17] Cerone A, Bernardi G, Gotsman A. A framework for transactional consistency models with atomic visibility[C]//LIPIcs-Leibniz International Proceedings in Informatics. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015, 42.

[18] Bailis P, Fekete A, Ghodsi A, et al. Scalable atomic visibility with RAMP transactions. ACM Trans. Database Syst., 2016, 41(3): 15:1--15:45.

[19] Ganesh A, Bamford R J. Consistent read in a distributed database environment: U.S. Patent 7,334,004[P]. 2008-2-19.

[20] Ardekani M S, Sutra P, Shapiro M. Non-monotonic Snapshot Isolation: Scalable and Strong Consistency for Geo-replicated Transactional Systems. In Proc. of 32nd SRDS. 2013. 163—172.

[21] Thomson A, Diamond T, Weng S C, et al. Calvin: fast distributed transactions for partitioned database systems. In Proc. of SIGMOD. 2012: 1-12.

[22] Bailis P, Davidson A, Fekete A, et al. Highly available transactions: Virtues and limitations. In Proc. of VLDB. 2013. 181-192.

[23] Brewer E A. Towards robust distributed systems. In Proc. of PODC. 2000, 7.

[24] Vogels W. Eventually consistent. Commun. ACM, 2009, 52(1): 40-44.

[25] Herlihy M, Wing JM. Linearizability: A Correctness Condition for Concurrent Objects. ACM Trans. Program. Lang. Syst., 1990 12(3): 463-492

[26] Lamport L. Paxos Made Simple, Fast and Byzantine. In Proc. of the 6th OPODIS. 2002. 7—9.

[27] Ongaro D, Ousterhout JK. In Search of an Understandable Consensus Algorithm. In Proc. of USENIX ATC. 2014. 305—319.

[28] Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber R. Bigtable: A Distributed Storage System for Structured Data. ACM Trans. Comput. Syst., 2008,26(2):4:1--4:26.

[29] Burrows M. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In Proc. of 7th OSDI. 2006. 335—350.

[30] Ghemawat S, Gobioff H, Leung ST. The Google File System. In Proc. of 19th SOSP. 2003. 29—43.

[31] Baker J, Bond C, Corbett JC, Furman JJ, Khorlin A, Larson J, Leon JM, Li YW, Lloyd A, Yushprakh V. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In Proc. of CIDR. 2011. 223—234.

[32] Lloyd W, Freedman MJ, Kaminsky M, Andersen DG. Don't settle for eventual consistency. Commun. ACM, 2014, 57(5), 61—68.

[33] Mohan C, Haderle DJ, Lindsay BG, Pirahesh H, Schwarz PM. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. ACM Trans. Database Syst., 1992, 17(1), 94—162.

[34] Lamport, L. Time, clocks, and the ordering of events in a distributed system. Commun. of the ACM, 1978, 21(7), 558-565.

[35] Viotti P, Vukolić M. Consistency in Non-Transactional Distributed Storage Systems. ACM Comput. Surv., 2016, 49(1): 19:1-19:34.

[36] Gray J, Helland P, O'Neil P, Shasha D. The Dangers of Replication and a Solution. In Proc. of SIGMOD. 1996. 173-182.

[37] Justin J. Levandoski, David B. Lomet, Mohamed F. Mokbel, Kevin Zhao: Deuteronomy: Transaction Support for Cloud Data. CIDR 2011: 123-133

[38] Justin J. Levandoski, David B. Lomet, Sudipta Sengupta, Ryan Stutsman, Rui Wang: High Performance Transactions in Deuteronomy. CIDR 2015

[39] Justin J. Levandoski, David B. Lomet, Sudipta Sengupta, Ryan Stutsman, Rui Wang: Multi-Version Range Concurrency Control in Deuteronomy. PVLDB 8(13): 2146-2157 (2015)

[40] Bernstein PA, Hadzilacos V, Goodman N: Concurrency Control and Recovery in Database Systems. Addison-Wesley. 1987.

[41] Kung HT, Robinson JT: On Optimistic Methods for Concurrency Control. In Proc. of VLDB. 1979. 351.

[42] Fekete A, Liarokapis D, O'Neil PE, Shasha DE: Making snapshot isolation serializable. ACM Trans. Database Syst, 2005, 30(2), 492-528.

[43] Thomson A, Abadi DJ: The Case for Determinism in Database Systems. PVLDB, 2010, 3(1), 70-80.

[44] Stonebraker M, Madden S, Abadi DJ, Harizopoulos S, Hachem N, Helland P: The End of an Architectural Era (It's Time for a Complete Rewrite). In Proc. of VLDB. 2007. 1150-1160.

[45] Bacon DF, Bales N, Bruno N, et al. Spanner: Becoming a SQL System. In Proc. of SIGMOD. 2017. 331—343.

[46] Papadimitriou CH: The serializability of concurrent database updates. JACM, 1979, 26(4), 631—653

[47] Malviya N, Weisberg A, Madden S, Stonebraker M: Rethinking main memory OLTP recovery. In Proc. of 30th ICDE. 2014. 604—615.

[48] Thomson A, Diamond T, Weng SC, Ren K, Shao P, Abadi DJ, Calvin: fast distributed transactions for partitioned database systems. In Proc. of SIGMOD. 2012. 1—12.

[49] Yao C, Agrawal D, Chen G, Ooi BC, Wu S: Adaptive Logging: Optimizing Logging and Recovery Costs in Distributed In-memory Databases. In Proc. of SIGMOD. 2016. 1119—1134.

[50] O'Neil PE, Cheng E, Gawlick D, O'Neil EJ: The Log-Structured Merge-Tree (LSM-Tree). Acta Inf., 1996, 33(4), 351-385.

[51] Lomet DB, Tzoumas K, Zwilling MJ: Implementing Performance Competitive Logical Recovery. PVLDB, 2011, 4(7), 430—439.

[52] Samaras G, Britton K, Citron A, Mohan C: Two-Phase Commit Optimizations and Tradeoffs in the Commercial Environment. In Proc. of ICDE. 1993. 520—529.

[53] Zhang I, Sharma NK, Szekeres A, Krishnamurthy A, Ports DRK. Building consistent transactions with inconsistent replication. In Proc. of SOSP. 2015. 267-278.

[54] Mu S, Nelson L, Lloyd W, Li J. Consolidating Concurrency Control and Consensus for Commits under Conflict. In Proc. of OSDI. 2016. 517-532.

[55] Bernstein PA, Das S: Rethinking Eventual Consistency. In Proc. Of SIGMOD. 2013. 923—928.

[56] Abadi D. Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. IEEE Computer, 2012, 45(2): 37—42.

[57] Davison SB, Molina HG, Skeen D. Consistency in Partitioned Networks. ACM Comput. Surv., 1985, 17(3): 341—370.

[58] Lamport L: The Part-Time Parliament. ACM Trans. Comput. Syst., 1998, 16(2), 133—169.

[59] Wang JH, Cai P, Qian WN, Zhou AY. Log replication and recovery in cluster-based database system. Ruan Jian Xue Bao/Journal of Software, 2017,28(3):476−489.

[60] Lin ZY, Lai YX, Lin C, Xie Y, Zou Q. Research on cloud databases. Journal of Software, 2012,23(5): 1148−1166.

[61] Chandra TD, Griesemer R, Redstone J: Paxos made live: an engineering perspective. In Proc. of PODC. 2007. 398—407.

[62] Gray J, Lamport L. Consensus on transaction commit. ACM Trans. Database Syst, 2006, 31(1): 133—160.

[63] Atul A: Weak Consistency. A Generalized Theory and Optimistic Implementations for Distributed Transactions. Ph.D. Dissertation, 1999.

附中文参考文献:

[59] 王嘉豪,蔡鹏,钱卫宁,周傲英.集群数据库系统的日志复制和故障恢复.软件学报,2017,28(3):476−489.

[60] 林子雨,赖永炫,林琛,谢怡,邹权.云数据库研究.软件学报,2012,23(5):1148-1166.