# Fast General Distributed Transactions with Opacity

## ABSTRACT

事务可以通过对应用程序开发人员隐藏数据分布、并发性和故障来简化分布式应用程序。理想情况下，开发人员会看到单个大型机器的抽象，该机器按顺序运行事务并且永不失败。这需要事务子系统提供不透明性opacity（对提交和中止的事务都具有严格的序列化能力），以及具有**高可用性的透明容错**。如果性能不佳，即使是最好的抽象也不太可能使用，因此系统还必须提供**高性能**。

现有的分布式事务设计要么削弱了这种抽象，要么**不是**为了在数据中心内获得最佳性能而设计的。本文扩展了 FaRM 的设计（它只为提交的事务提供严格的可串行化）以提供opacity，同时在现代数据中心内保持 FaRM 的高吞吐量、低延迟和高可用性。它使用基于真实时间的时间戳排序，时钟在集群中能在几十微秒内同步，并使用故障转移协议来确保时钟主机故障的正确性。当在具有 3 路复制的 90 台机器上运行 TPC-C 事务组合时，具有不透明度的 FaRM 每秒可以提交 540 万个新订单事务。（平均一个节点才6万的IOPS）

## INTRODUCTION

云数据中心提供了许多相对较小、个别不可靠的服务器。尽管个别服务器出现故障，云服务需要在此类服务器的集群上运行以保持可用性。他们还需要向外扩展以增加超出单个服务器的吞吐量。对于需要将数据保存在主内存中的延迟敏感应用程序，还需要横向扩展以超出单个服务器的内存限制。

挑战在于分布式应用程序，尤其是有状态的应用程序，比单线程甚至多线程应用程序更难编程。我们的目标是通过提供一台大型机器的抽象来使它们更易于编程，该机器一次运行一个事务并且永不失败。这需要具有以下属性的分布式事务系统：

• **可串行化**：所有执行都相当于提交事务的某种串行排序。

• **严格性**：这种排序与真实时间一致。

• **快照读取**：所有事务都会看到数据库的一致快照，直到它们提交或中止。

• **高可用性**：系统从服务器故障中透明地恢复，停机时间短到足以表现为性能暂时下降。

前三个属性的组合也称为不透明度 [13]。直观地说，**不透明度将严格可序列化的属性扩展到中止的事务**，即这些事务还在与真实时间排序一致的时间点看到一致的快照，直到它们中止。

由于即使是最好的抽象，如果它们表现不佳也不太可能被使用，因此系统还必须提供可扩展性和高性能。现有的设计要么削弱了这种抽象，要么不是为了在数据中心内实现最佳性能而设计的。 Spanner [6] 是一个地理分布式数据库，它提供不透明和可用性，但不提供数据中心的低延迟和高吞吐量。几个事务系统 [4, 10, 20, 35] 利用每台服务器大量廉价的 DRAM、快速商品网络硬件和 RDMA 来在数据中心实现良好的性能。与 TCP [9] 相比，RDMA 可以将网络吞吐量和延迟提高几个数量级。与使用 RDMA 的双向消息传递相比，单侧 RDMA 可以进一步降低 CPU 成本和延迟，因为它绕过了远程 CPU。几个分布式事务协议 [4, 10, 35] 使用单侧 RDMA 来发送更少的消息并实现比数据中心内的两阶段提交 (2PC) 更高的性能。

当前使用单面 RDMA 的设计不提供不透明度。FaRMv1 [9, 10] 和 DrTM [4, 33] 为已提交但不为中止的事务提供可扩展性、可用性和严格的可序列化性。在这些系统中乐观地执行事务可能会读取不一致的状态，并保证此类事务最终会中止。（乐观并发都具有这个问题，除非提供多版本。）NAM-DB [35] 提供读取快照，但不提供严格性、可序列化性或高可用性。

在本文中，我们描述了 FaRMv2，它扩展了 FaRMv1 的原始设计和实现，为所有事务提供读取快照。 FaRMv2 使用一种新颖的时间戳排序协议，该协议利用 RDMA 的低延迟来同步时钟。时间戳基于真实时间，可以很好地扩展，因为它允许机器**使用本地时钟来生成时间戳**。然而，由于时钟不是完全同步的，交易协议在生成读写时间戳时**必须“等待不确定性”**，这会引入延迟。 FaRMv2 利用低延迟、CPU 高效的基于 RDMA 的通信来频繁地同步网络上的时钟，以实现数十微秒的不确定性，比 Spanner [6] 低两个数量级。

与 Spanner 不同，FaRMv2 不需要原子钟或 GPS。相反，服务器使用 CPU 周期计数器并与从系统中的所有服务器中选出的时钟主master同步（时钟领导者）。使用时钟主机故障转移协议在时钟主机故障之间维护时间戳排序。我们的设计和实现还支持多版本控制，可提高只读事务的性能。旧版本通过有效的分配和垃圾收集保存在内存中。

该论文做出了以下新颖的贡献：

• 一种利用RDMA 将时钟同步到几十微秒以内的机制。

• 一种不透明的事务协议，使用全局时间和单边 RDMA。

• 时钟故障转移协议，可在时钟master故障期间保持时间戳单调，而无需原子钟或 GPS 等特殊硬件。

• 用于多版本控制的高效线程本地、基于块的分配器和垃圾收集器。

FaRMv2 在 90 台机器的集群上运行 TPC-C 事务混合时，每秒可以提交 540 万个新订单事务，具有 3 路复制以实现容错。它保留了 FaRMv1 的高可用性，可以在服务器发生故障的几十毫秒内恢复到满吞吐量。我们相信 FaRMv2 在任何提供不透明性和高可用性的系统中具有最高的已知吞吐量。

## MOTIVATION

可串行化是开发人员易于理解的隔离级别，因为它避免了许多异常。我们发现严格性和不透明性对于使用事务性平台的开发人员也很重要。

严格的可序列化[28]意味着事务的序列化顺序与真实时间相对应。如果 A 在 B 开始之前完成，那么任何正确的执行都必须等同于 A 出现在 B 之前的串行执行。当客户端使用系统外部的某个通道进行通信时，这一点很重要，例如，当其他系统分层在数据库之上时。

不透明度 [13] 是事务执行对于中止事务和提交事务严格可序列化的属性。这通过确保在事务执行期间保持不变来简化编程。许多现有系统通过使用带有读锁的悲观并发控制（例如 Spanner [6]）或通过使用时间戳排序在执行期间提供读取快照（例如 Hekaton [7, 22]）来提供不透明性。但是许多使用乐观并发控制 (OCC) [21] 的系统不提供中止事务的读取快照，例如 FaRMv1 [9, 10] 和 DrTM [4, 33]。这种设计决策可以提高性能，但会给开发人员带来很大的负担。由于开发人员不能假设不变量成立，他们必须通过在事务代码中明确检查不变量来进行防御性编程。关系数据库通过提供在每个 SQL 语句之后自动检查约束的机制来减少这种负担，但这会增加一个不小的性能开销，它仍然需要开发人员编写所有相关的约束。

FaRM 和 DrTM 提供低级事务内存模型而不是关系模型。这允许开发人员编写高性能 C++ 代码来操作事务中的任意指针链接数据结构，但这种灵活性是有代价的。在每个 C++ 语句之后进行高效的自动约束检查是不可行的。此外，缺乏不透明度会导致违反内存安全。例如，事务可能会读取已释放和重用的内存，这可能会导致崩溃或无限循环。我们通过讨论在 FaRM 之上的哈希表和 B 树索引的实现来说明没有不透明性的编程的困难。

FaRM 哈希表 [9] 使用链式关联跳房子哈希。每个键查找读取两个相邻的数组桶和零个或多个溢出桶。 FaRMv1 确保以原子方式读取这些对象中的每一个，但它们可能不会全部从同一个一致的快照中读取，因为 FaRMv1 不确保不透明度。这会导致一些异常，例如，一个并发事务可能在删除键 B 的同时将 keyA 从溢出桶移动到数组桶，导致查找事务错误地错过了 A。 FaRMv1 通过向所有对象头添加 64 位化身来解决这个问题

，在所有溢出桶指针中复制它们，并向每个数组桶添加额外的版本字段。这增加了复杂性和开销，这可以通过提供不透明度来避免。

FaRM B-Tree 实现在每个服务器上保存内部节点的缓存副本以提高性能。Fence键 [12, 24] 用于检查父子遍历的一致性。通过始终读取未缓存的叶节点并将它们添加到事务的读取集中来维护严格的可序列化性。缓存的内部节点在所有线程之间只读共享，无需制作额外的线程本地或事务本地副本。这是非常有效的，因为大多数查找都需要一次未缓存的读取，并且不会复制内部节点。

在使用这个 B 树时，缺乏不透明度会导致几个异常，例如，一位开发人员报告说他们发现了一个错误，因为“他们在 BTree 中插入一个键，（然后）在同一个事务中查找相同的键，但它并不存在。”在调查中，我们发现当（另一个）并发事务在 B 树中创建一个分裂，将有问题的键 (A) 迁移到一个新的叶对象，删除 A，同时运行原始事务的服务器从缓存中驱逐了从根到新叶的路径上的一些内部节点，这种并发是有可能的。即使在这种情况下原始事务会中止，程序员仍然需要在事务中止之前对执行进行推理。推理像这样的复杂极端情况是很困难的。 Opacity 通过为中止的事务提供强大的隔离保证来简化编程。

在本文中，我们的贡献是为 FaRM 添加不透明度以提高可编程性，同时保持良好的性能。很难量化提供更好的开发人员体验的好处。

根据我们的部署经验——两年多的 FaRMv1 和两年多的 FaRMv2——我们可以说我们的开发人员称赞了不透明度的增加，我们不再看到由于违反不透明度而导致的错误报告。我们还能够删除哈希表中每个数组桶的额外版本字段，并将溢出链的“胖指针”转换为普通指针，从而简化了代码并减少了空间使用。