# **周烜：华东师范大学的数据库系统研究**

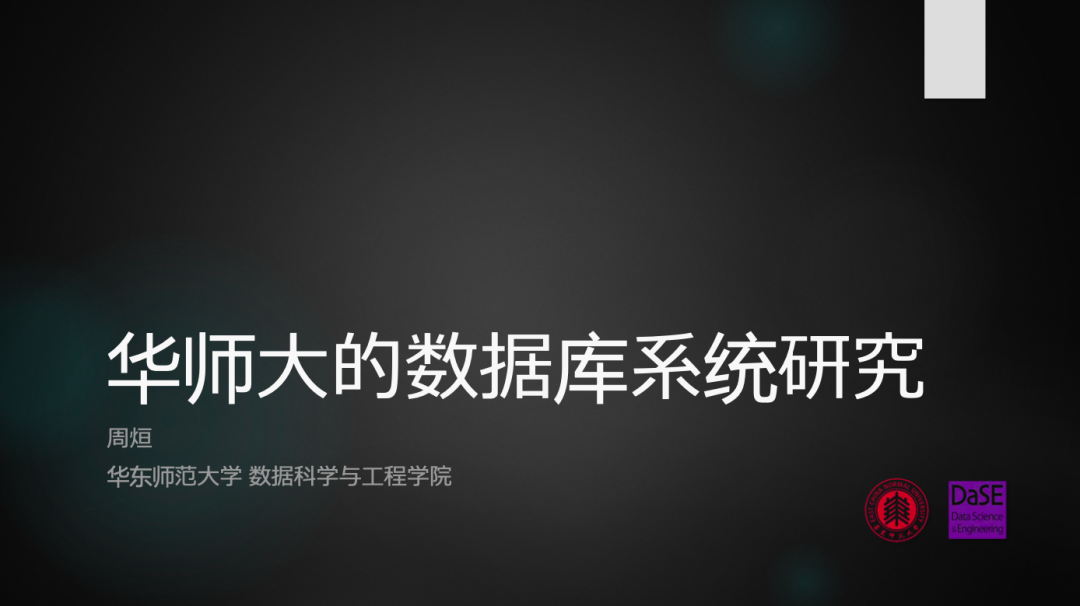
文章来源：infoq周烜



****/报告摘要/****

华东师范大学是国内为数不多长期坚持数据库内核技术研究的高校，在学术界和工业界均建立了较好的声誉。本次讲座将分享华东师范大学数据库团队近期的一些科研思路和研究成果。首先分析驱动数据库技术发展的主要因素，谈一谈未来有价值的研究方向。再聊一聊团队近来取得的一些有趣的研究成果，领域包括新硬件的数据库适配、分布式事务处理、HTAP、系统实现模块化（Modularization）等等。

## **00 引言**



今天我代表华东师范大学的数据库团队，来分享一下对数据库这种技术或者这种产品的一些研究心得以及当前的研究成果。其实，高校跟企业实际上是处在两个不同的生态领域当中，高校更关注关于研究理论的一些问题，但是企业更多的关注企业本身的产品以及用户，所以两者面向的目标是不太一样的。但是经过我们在华师大这么多年的一些摸索，特别是我们自己研究上的一些摸索，以及跟企业合作的经历，我们觉得实际上高校和企业应该一起来做我们称为数据库系统或者基础软件的研究，因为只有这样才能够更好的推动这个行业本身的发展。

我的报告内容会分成两个部分。首先分享一下我们对数据库这种技术发展动态的看法，我们团队在数据库这个领域也做了很多年，包括我之前在中国人民大学也是做数据库系统的，我在这个领域里面有可能不到 20 年的积累。我希望能够提出我们的看法，并且能够得到大家的一些反馈，纯粹是做一种探讨，因为对技术的发展方向的探索，没有标准的答案。我觉得大家应该集思广益，共同去探讨，才能把这个问题看得更清楚，这是讲座前面一部分的内容。

后面一部分的内容我会聚焦到我们的一些研究成果上，这些研究成果可能就会比较技术细节了，会更适合搞技术、数据库、系统的这些同学，但是我会尽量把讲座的形式变得更大众化一点，尽量用更通俗的方式去给大家介绍。我希望通过这种细节的介绍，也能够让大家了解一下，我们在设计系统的时候，通常一个工程师、一个研究者或者一个学者，他的思路大概是怎么样的，我不能代表所有的人，但是因为我们这个团队是一个典型的做系统的团队，所以这个思路可能仅代表了一部分做系统的学者的思路。

## **01 数据库系统的形态变化**

### **1.1 什么引起了数据库系统的形态变化？**

首先问大家一个问题，什么引起了数据库系统的形态变化？我们知道数据库系统实际上是一个有很长历史的系统，是现代软件开发的一个核心部件。任何应用都离不开某种数据库，但是我们其实也可以看到，如果你有一定的经验，比如大概 10 年的工作经验，你会看到数据库系统的形态是在发生变化的，10 年前用的很流行的东西，现在不见得普遍被采用。这个系统虽然有很长很长的历史，也很成熟了，但是它的形态还是在发生变化的。

什么在驱动数据库系统的形态变化？这是一个很重要的问题，也是比较有趣的问题，但并不是一个好回答或者能够全面的回答的问题。我这里就直接抛出我们对这个问题的一些看法，我们觉得数据库系统的形态变化，主要来自三个方面的推动力：

* 第一个方面是 ****应用需求**** 的变化。我们的软件产品一直在变得越来越丰富，应对的场景越来越多，实际上应用需求是在变化的，它的变化提出了不同的要求需求，它在促使数据库系统的形态在发生改变。
* 第二个方面在学术界其实比较少去触及的，就是 ****软件开发模式**** 的变化。数据库是70年代80年代设计的，面对的是当时的软件开发模式，我们知道在九十年代以后，软件开发模式实际上发生了很大的改变。软件开发的模式的变化，也在引起数据库使用方式的改变，也在推进数据库系统的演变。
* 第三个方面就是 ****硬件平台**** 的革新，硬件在改变处理器、存储器件，整个平台以前是一个大型的计算机，大型机、中型机，然后现在其实就是云平台，这些东西的变化实际上是在影响数据库本身的形态的改变。

我们看到的推动力主要就是这三个。我们认为未来推动数据库变化的原因也不出于这三个，我们可以通过这个东西去预知未来应该朝什么方面去推进我们的数据库技术的进步。下面我就大概做一个简单的展开。

#### **1.1.1 应用需求的变化**

首先第一点就是应用需求的变化。我们认为这实际上是在推动底层技术，像数据库系统这种技术变化的一个主要原因。

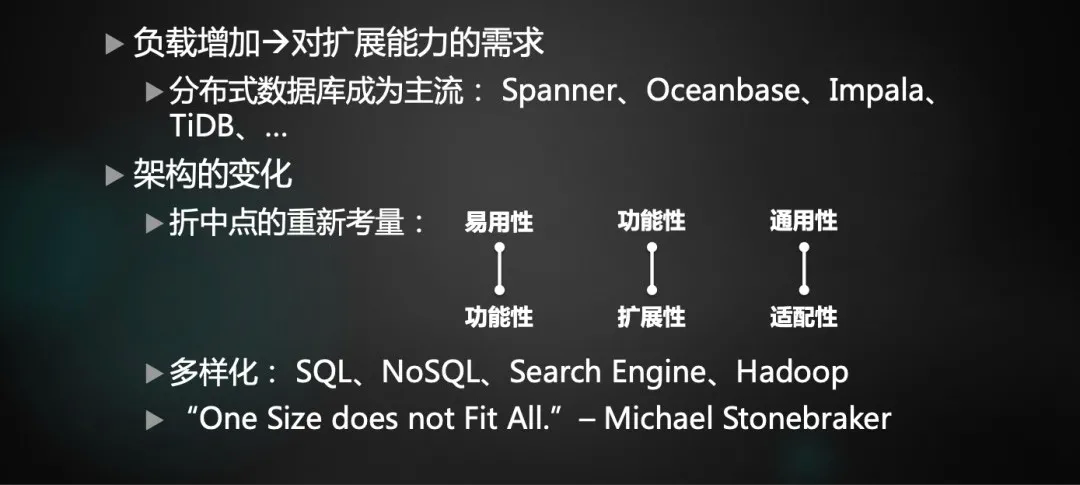


通过上面这三幅图大家能够很容易去理解数据库诞生的年代，那个时候磁盘作为一个存储介质，刚刚在市场上推广，磁盘取代了磁带，数据的随机访问变的可能了。那个时候对数据管理功能的需求一下子就增长了，当时出现了各种各样的数据库，包括网状数据库，包括后面的关系数据库，那个时候是我们叫前互联网时代就出来了。但那个时候应用的规模并不大，数据库的用户量一般，终端用户其实很少，对于一个银行来讲就是那些银行职员在使用数据库，普通用户在银行排队，他们不是终端的使用者。

后来进入到互联网时代之后，我们发现终端的使用者一下子就爆炸性的增长了。现在我们每个人有一个手机，随时都在使用手机上的 App，然后这个 App 他随时都会把请求发送给后台的数据库。我们看到应用规模在最近 20 年有一个很大的增长，如果往后看的话，未来我们认为增长有可能还会继续。我们的工业互联网、物联网使用的话，我们的终端会变得更多，他可能对数据管理系统的压力会变得更大。

所以我们看到应用不断的扩张，给数据库这种底层系统有一个持续增长的压力，要应对这样的压力，以前对数据库的设计，它逐渐就变得不太实用，必须去革新，必须去改变它。

这个是我们看到的一个最主要的推动力。但除了应用规模的扩展，当然就还有一个应用领域的扩展，最开始的数据库它就是金融领域或者电信领域去使用，并不会在互联网、销售或者传媒等等这些领域去大规模的使用，但是我们发现 IT 的渗透到各行各业之后，它的应用范围也在增加，应用范围增加对传统的系统来讲是不友好的，或者是说你的传统数据库系统对这些应用是不友好的，所以这个也在推动它的一个变革。



应用需求带来什么变化？我们回顾历史的话，我们看到分布式数据库现在变得越来越被大家所需要，大家看到了谷歌的一些分布式数据库的产品，它现在作为一种标杆的产品，然后国内也有一些分布式数据库的场景，包括美团在内，听说美团内部也在研制自己的分布式数据库，实际上是在对需求的负载增加的一个应对。

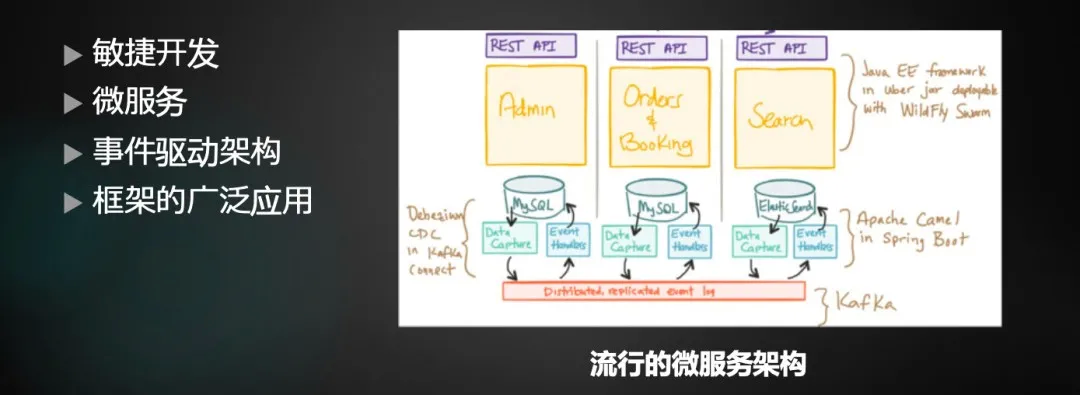
应对它实际上并不是一个很简单的事情，如果要增加你的数据库的扩展性，有时候你必须要重新去设计你的数据库的架构。我们通常做系统的同学应该了解，其实在做系统的时候，你需要做很多折中的考量的，有些东西是不能兼顾的，比如说你的功能性和应用性对吧？一个东西特别简单去使用的话，它功能性有时候就是比较简单，它复杂的功能处理不了，或者是你的功能很复杂的时候，你的扩展性又上不去。

在系统设计的时候你必须去做一种权衡，去获得一部分这方面的能力，你就必须丢失其他的一些能力。当你需要它的扩展性非常强的时候，你有可能就要去重新考量它，你要去丢弃什么东西，你要去忍受什么其他的一些东西，然后这个时候就产生架构的变化，这样的话我们就会发现有新的系统出来，比如说以前的是 SQL，我们现在有讲 NoSQL 大家认为它的扩展性容易做的更好一些，然后会有其他形态的一些数据管理产品。

针对这个问题，学界工业界都有很多的讨论，Michael Stonebrake 大概 10 多年前发表的一些言论，就说“One size does not fit all”，你不太能指望某一个系统能够能够处理所有的应用需求。因为不同的应用需求，你可能必须要做不同的折中、重新的考量，你只能顾此，你顾此只能失彼，所以这样就产生了一个多样化的形态。

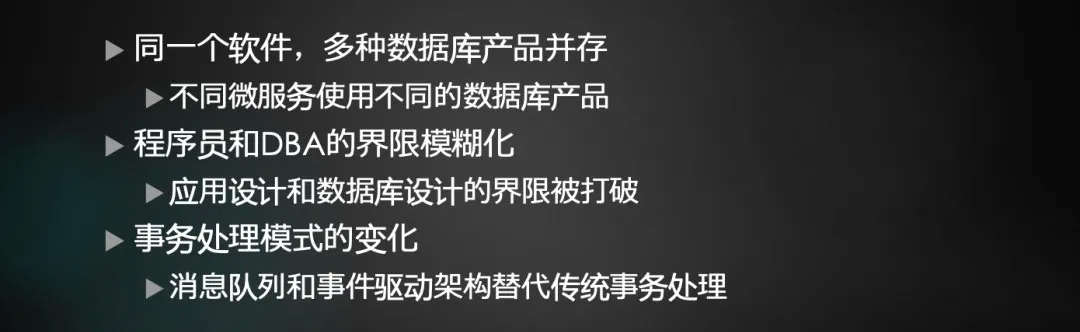
我们现在看到的数据库，如果你们在使用的话，你会面临很多选择，你到底用 MySQL 还是 MongoDB 对吧？你的分析的时候你要用什么？用 Hadoop 还是用传统的数仓 MPP 产品，有不同的需求，可能有不同的考量。这个我们看到应用需求它的变化，它的增加，它在实际上对这个系统起到了一个很重要的推动的这个作用，这是我展开的第一点。

#### **1.1.2 软件开发模式的转变**



然后，第二点就是我们发现软件开发模式的变化，也在促进数据库本身的改变。就像我刚才提到的，现在用到的很多软件开发的模式跟以前不一样。以前关系数据库这样的产品刚刚被广泛应用的时候，当时的软件开发它是以数据为中心的，一个数据库设计出来，有好多应用都会去用它，它是一个 Shared 底层系统。那个时候数据库的设计过程，它是相对独立的，DBA 根据 App 开发人员的需求、用户的需求，以 DBA 的方式去设计数据库，按照对数据模型的理解，把它设计得非常的规整，要满足各种各样的范式，然后给不同的应用去用。但现在我们发现一旦用到微服务这种新的开发模式的时候，很多时候这种横向的分割变得不是那么重要了。原本数据是一层、应用逻辑是一层，这两层之间的解耦是很重要的，但现在不是了。

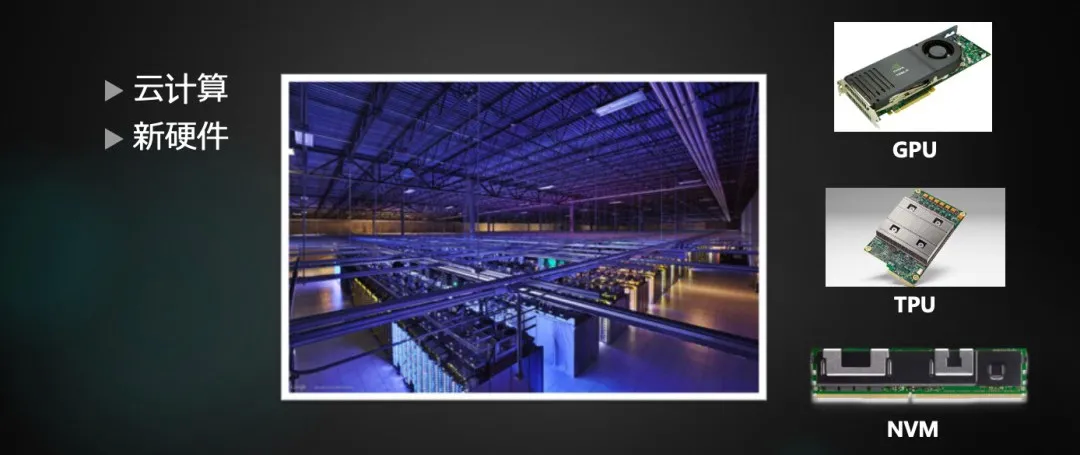
现在是微服务的形式，是纵向的切割，把业务整个分成一块一块的，每一块里面都有单独的数据库，有单独的功能设计，这弱化了数据库跟应用之间的界限，强化了应用里面不同模块的界限。这样的话，每个模块可以用不同的数据库产品，比如说一个设备用 MySQL，另外一个设备可能用 MongoDB，第三个设备可能用 ES，这些模块之间的数据有同步有交互，可以用一些比如像事件驱动的架构，像 Kafka 这种 MQ (Message Queue)去连接在一起，形成一个总体的架构。



这种设计跟以前的数据库是不太一样的，对于数据库本身的要求也不太一样。不同的 Service 会用不同形态的数据库产品，根据需求或者根据软件开发者的习惯，去采用各种缓存、消息队列等，去把这些东西给嫁接在一起。这样的形态对数据库有不同的要求，所以 NoSQL 被很多人接受。

在某些软件开发的场景下，NoSQL 就是比关系数据库使用起来更简单。然后事务的处理方式也变得很不一样，现在的消息队列在事务处理中，它的权重非常的高，而不是完全依赖于传统数据库内部支持事务处理的模式。这个是我们也看到这样的一些变化，这个是软件开发模式带来的一些改变。

#### **1.1.3 硬件平台的革新**



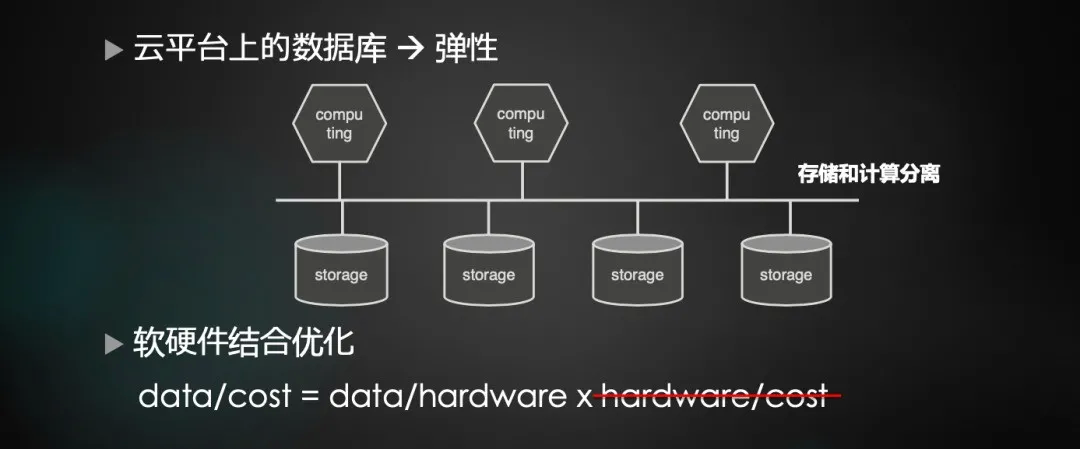
第三个方面就是平台的革新。其实我们不会对传统的数据库的硬件平台灵活性有太多的关注，但现在数据库产品面向的基本上都是云平台，不是以前的 IBM 大型机了，不是 Oracle 那个时候的硬件平台。我们面对的是一个云平台，云平台自身是在发展的。以后的云平台其实可以预见得到，它不单单是现在我们看到的，是由一个个虚拟机或者是一个个容器组成的一个计算平台，还有可能就像一个大型的计算机一样，只是这种大型计算机它的资源非常的丰富，需要调用什么样的资源都可以获得，需要更多的内存、CPU、存储，都可以直接的获取。

云平台通过比较高效的网络方式把这些资源全部连在一起，然后给用户的接口也很简单，很多维护功能是在云平台内部去实现的。数据库要扎根在这样的一个云平台上面，其实对数据库系统就会有新的要求。以前的数据库，大家都记得有几种架构可以选择，叫 Shared Everything、Shared Nothing、Shared Disk 这样的一些架构。

但是我觉得在“云”层面上，数据库其实不再是那么简单去划分的，就是说数据库系统的产品，必须要做到具有很好的弹性。任何的资源在短缺的时候，可以通过云的这种方式很快的把资源调度过来，从而增加数据库的能力。对用户的话，只是提供一种数据库的服务，用户用多少？就提供多少。这样的一个云架构下的数据库，形态可能跟以前要有一些变化。除了这种云的体系之外，其实还有一些新硬件出现，硬件种类也变得越来越多。不同的种类的硬件，在不同的条件下，算力也在不断增加。当然除了计算，还有存储也发生很多变化，把这些放到云的里面去，云的资源变得更加丰富，对数据库的要求也会变得更高。

因此，我们看到这种硬件平台的变革，它实际上在推动数据库的一些发展。现在大家很常见的就是这种计算与存储分离的这种架构的数据库。我把数据库本身这种系统，它的计算层跟存储层完全的分割开，计算层可以自己扩展，存储层也可以自己扩展，两层就通过云这种高速的、互联的通道能够连接在一起，这种实际上就是针对云的一个特殊的处理。我们知道存储是便宜的，所以在存储需要扩展的时候，没有必要在扩展存储的基础上去加 CPU 的资源，因为 CPU 比较贵。如果分开扩展的话，这样确实会在成本上有极大的提升。

未来各种资源加进去之后，它都有可能有扩展的需求。比如说缓存，新的存储器，新的内存加进去之后，有可能需要它跟底层的存储分开，进行一个隔离，再分别去扩展，这都是有可能的。但现在的数据库产品其实面对这种扩展能力，实际上是非常有限的，存储和计算分开扩展到一定程度，实际上它的能力就达到一个峰值了。那么怎样去推动它进一步的这种弹性的增加，实际上是一个挺难的问题，但是也是挺有趣的问题。



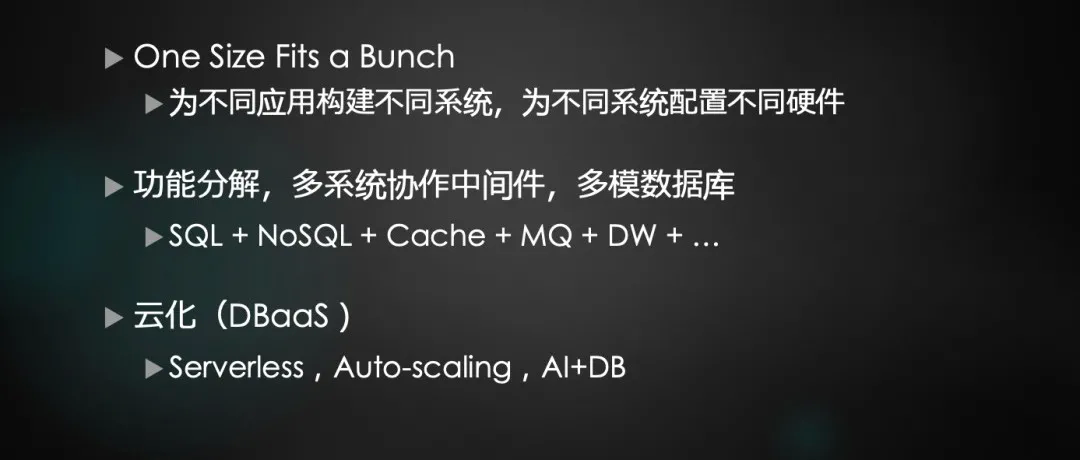
新硬件对数据库产品的影响，这里有一个简单的公式，公式的左边叫做 Data/Cost，单位数据处理，单位代价上面可以处理多少数据。这个是我们需要提升的，因为可以想象以后的数据量会越来越大，如果不把单位价格上面能处理的数据这个值提升的话，应对数据的能力就没办法提升。数据越多，需要花费的资源或者代价就越多，这个是我们不希望看到的。希望左边这个式子中 Data/Cost 的值随着技术的进步，它可以逐渐的提升。

然后把左边这个式子它分解一下，分解成 Data/Hardware 和 Hardware/Cost 的一个乘积。这其实是很简单的一个因素分解，大家能够看得很明白。我们发现后面这个式子 Hardware/Cost，实际上它的增长在逐渐的趋缓甚至停滞，主要的一个问题就是单位价格能够买到的硬件资源，要在这上面做更进一步的提升，会变得非常的困难。

最后，如果想实现 Data/Cost 的提升的话，只能去提升左边这个式子 Data/Hardware，这个是未来一个很明显的趋势。怎么样能够提升 Data/Hardware？就是单位硬件下处理数据的能力要怎样提升？我们认为只有两种途径，第一种是硬件定制化，面对不同的应用需求，需要为这种应用需求做特殊的硬件。其实我们能看到像 GPU、TPU 这种出现，其实就已经在揭示这种规律了，专用硬件效率总是要比通用硬件好的。然后软件是一样的，专用的软件的效率肯定比通用的软件好。

这个趋势我觉得可以从长期来看，短期可能并不是那么的明显，但长期来看的话，这个过程应该是不可阻挡的，也就是说我们可能会面临要去为应用去定制系统，要为专门的这种系统配置适合它的硬件。

### **1.2 数据库系统的未来发展趋势**

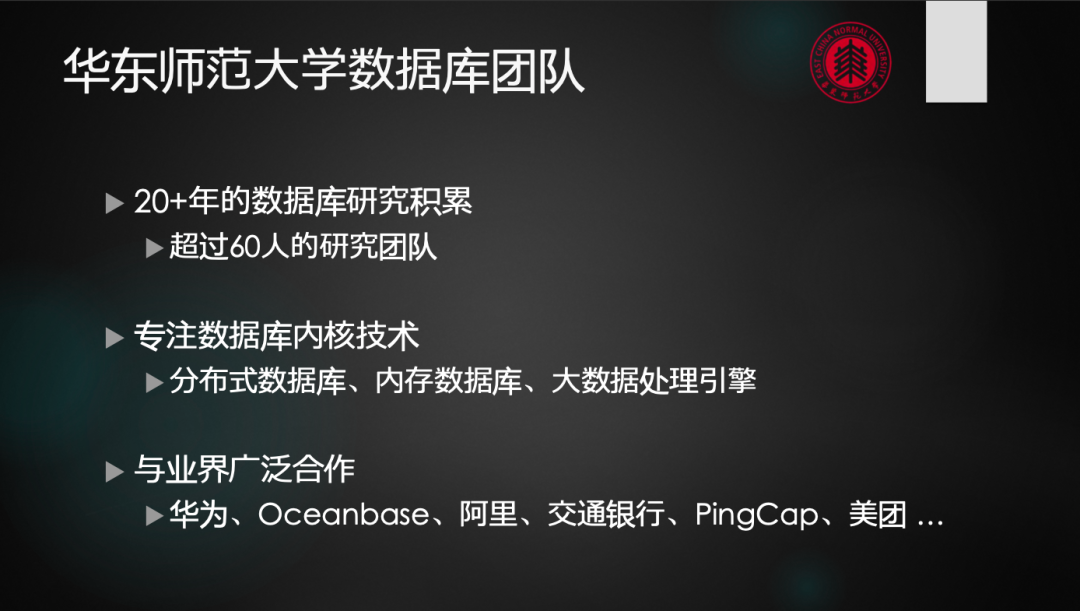


我们刚才讲了三点，第一点就是应用的变化在推动系统的演进；第二个是软件开发模式的变化实际上也在带来系统的功能的一种变革；最后是硬件平台。因此，我们觉得未来数据库发展的趋势：

* 第一个是“One size fits a bunch”，为不同应用构建不同系统，为不同系统配置不同硬件。就像Michael Stonebraker说的那样，数据库不是一种系统就能应对所有的应用了。我觉得Stenberg当时是针对的是应用负载的增加带来的问题而提出的观点，但是硬件的发展瓶颈到来的时候，同样会引领我们认同这个观点，“One size does not fit all”，不可能为所有应用产生一个系统，应该是一类应用对应一套系统的，这个是我们看到的一个很明显的发展的趋势。
* 第二个发展的趋势是现在的系统变多了之后，它需要协同，会有各种中间件，还会有各种形态的数据库，我们需要把它协同起来，这些数据库形态太多，对程序开发人员，维护人员都是一个Disaster，代价会变大，怎么更好地把它协同起来，这也是未来一个发展的方向，我们会看到这些系统相互之间会变得越来越配合，越来越融合。
* 第三个就是云平台成为一个主流的硬件平台之后，我们看到数据库会朝云这个方向有更深入这种发展，它会更适合云这种形态，它的弹性，自我维护、自我修复的能力是会进一步提升的。这个是我们对未来发展的展望。

## **02 华师大的数据库系统研究**

### **2.1 研究团队**



然后介绍一下我们现在这个团队，华东师范大学数据科学工程学院大概有 20 多个老师，大概 10 个老师是从事数据库内核的研究的。整个团队的历史是超过 20 年的，我们近 10 年其实做了非常多的系统内核研发的工作，跟业界的很多的公司也有合作。我们的学生其实也做了很多工程性的工作。

去年我们有拿到一个国家科技进步二等奖，这个是基于我们当时和某银行一起做的一款数据库产品，是我们基于 OceanBase 的一个早期开源的版本上实现的一个系统，具体的这种内容我就不做过多的介绍，这个系统实际上在某银行得到了比较深入的应用，是我们比较引以为豪的一个研究成果。

### **2.2 研究成果**

我们团队的研究其实还是蛮广泛的，我们在事务型数据库和分析型数据库其实都有研究，但我们更多的精力还是集中在事务型数据库上面。然后接下来，我会介绍一些典型的研究成果，让大家了解一下我们的研究是在做一些什么事情，主要分三个部分，第一个是分布式事务，第二个是数据库系统解耦合，第三个是新硬件。

#### **2.2.1 分布式事务**

分布式事务是一个几十年来大家都在探讨的话题，实际上也是有一定的争论：分布式事务到底合不合用？我们在使用事务处理这种功能的时候，是不是应该去规避分布式事务？还是我们应该进一步去增强数据库支撑分布式事务的能力，让程序员不要刻意去规避分布式思维？这实际上是一个疑问，目前没有一个明确的答案。

如果分布式事务确实是不行的，那我们就应该做一些其他方面的处理，来弥补分布式事务本身的缺陷，这样做有两种方式：

* 第一种是干脆不要数据库提供分布式事务功能，将事务处理推给应用，根据应用的特点去规避这种分布式事务的一些缺陷。
* 第二种是尽可能提升集中式事务处理的能力，集中式事务能够达到和分布式同样的效果，就不再需要分布式事务了。

我们认为现在以 NoSQL 为代表的这些系统的推动者，实际上是持这样的观点的。比如典型的 NoSQL 客户系统 MongoDB，它的一般的事务处理或者数据库的访问，都是 Single Document 一个文档一个文档去处理的。实际上，就是如果真要进行复杂的事务处理，那就到上层应用去处理，就用最定制化的方式去应对这种事务，它的效率可以比较高。

另外一种观点认为分布式事务本身应该是可行的，我们应该提升数据库处理分布式事务的能力，解放开发者。

这样的观点就是那些推动 NewSQL 这一类系统的人所持的观点，比如说谷歌的 Spanner 、TiDB、OceanBase。那么想要把分布式事务做好，怎样去优化，提升分布式事务的能力？首先要处理异常，分布式事务最害怕的一个问题就是出现异常，出现异常之后，如果是一个分布式事务，一旦事务锁掉一个节点上的数据，另一个节点出现故障的话，就会很麻烦。那为了处理异常，然后以 Spanner 为代表，使用了很多高可用的系统架构，用 Paxos/Raft 创建这种在云平台，在这种廉价计算机上同样能够有高可用能力的基础设施，在这个上面我可以去规避分布式事务所遇到的这种异常的问题。

但除了异常问题之外，实际上还有分布式事务的扩展性的问题。虽然分布式事务可以比较放心地应用于可靠的、高可用的系统上，但是它的性能会比较差。因此，我们必须要优化它的性能，为此学术界也做了很多尝试，对于团队来讲，最近几年，我们也做过一些研究和尝试。接下来我就大概讲一下我们的大概思路。

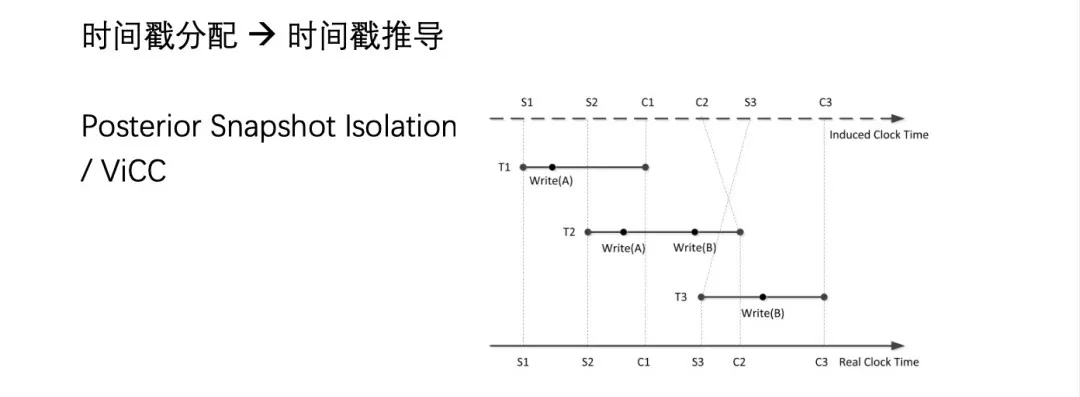
首先，我们要了解到底是什么限制了分布式事务的扩展能力，大家是比较公认的一种观点是制约分布式事务扩展的主要是事物之间的这种阻塞 Blocking。你可以想象一下，当一个事物它去访问一个数据，特别是修改一个数据之后，它会加锁，然后在加锁的过程中，又要去跟其他的节点进行各种通信。

其实这种通信有时候是很耗时的，有时候甚至要跟异地的节点，比如说要做高可用，就需要跟异地的节点建立通信。在加锁的过程中去通信，由于通信的过程很长，然后就会把加锁的时长变得特别的长，阻塞就会变得很严重。一旦事务处理的阻塞时间增加，它的事物的吞吐有可能会受到很严重的影响，特别对于有一些热点的数据出现的话，不是时长增加一倍，性能就降低一倍的，有时候时长增加一倍，性能可能会降低若干倍，这是一个很麻烦的问题。

所以说，真的想要解决分布式的扩展能力的话，在已经有高可用的前提下，我们最重要的目标就是要降低阻塞。怎样降低阻塞？其实可以用到很多的技术。比如说 MVCC/OCC、 MVCC 是多版本的数据管理，它可以降低阻塞，这个很直观，就是我一个数据有多个版本，当我的一个事务去改动数据的时候，我直接产生一个新的版本，这样就不用去阻止别人读你的旧版本。

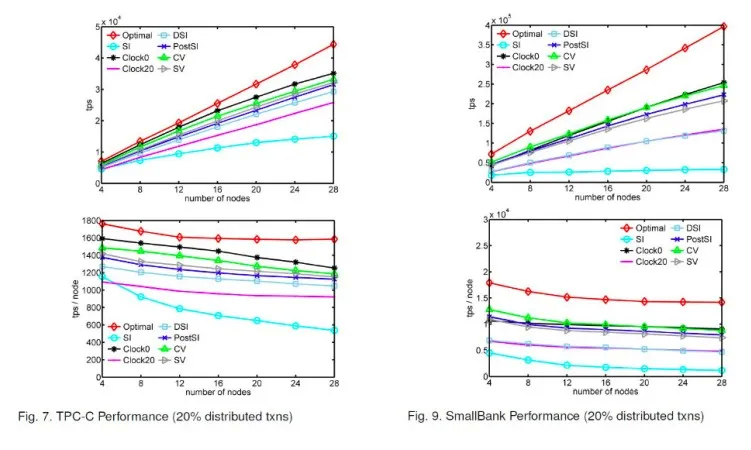
因为你有多个版本，新版本在产生的过程中，你没有办法去读，可以去读旧版本不用阻塞，这就是 MVCC 的使用。然后 OCC 也是一样的，就是 OCC 就是乐观并发控制，也就是说默认不需要加锁，到最后再来检测这个事务是不是执行正确，如果执行不正确推翻就行了。然后还有一些锁的优化，通过各种各样的技术，来把阻塞的现象把它尽量的减少，从而来提升分布式事务的扩展能力。

****MVCC 时间戳分配去中心化****



在 MVCC 上我们做过一些工作。当然 MVCC 有一个时间戳分配的问题，就是说它来判断一个事务或者一个数据或者一个数据的版本，它是不是应该由某一个事务去读取的话，它要通过一些时间戳的判断来做。时间戳的分配很麻烦，按道理来说，它是应该有一个中心的时钟，大家都去中心的时钟去拿时间戳，这样就可以保证事务处理正确无误。但通常如果有一个中心的时钟的话，那扩展性就受限了，比如谷歌的 Spanner 用一些原子钟去规避这个问题。然后我们做的一个研究就是去中心化，去优化了 SI 的隔离级别。

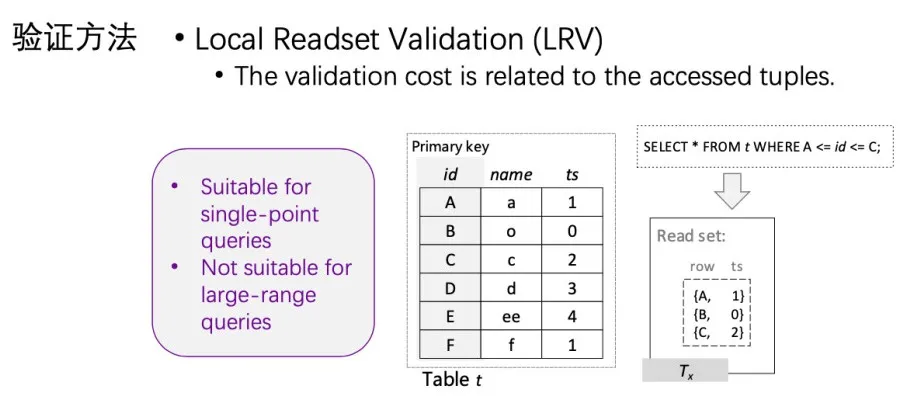
SI 是一种典型的 MVCC 的隔离算法或者并发控制的算法，它是需要时间戳的。我们做了一个去中心化。想法是我在给事务分配时间戳的时候，不是在事务开始的时候分配，而是在事务要结束的时候，根据它跟其他事务的关系、冲突情况，来给它指定一个合适的时间戳，所以叫后验时间戳，Posterior SI。这种方式使得我们可以不需要用一个中心的时间戳去做这个事情，但这个东西做起来其实蛮复杂的，我们大概 5 年前做了这件事情，最后有一些实验没有实现在现实的系统里面去，因为实现相对来说比较复杂，而实际应用中使用一个统一的中心的时间戳，基本的应用还是可以满足的。

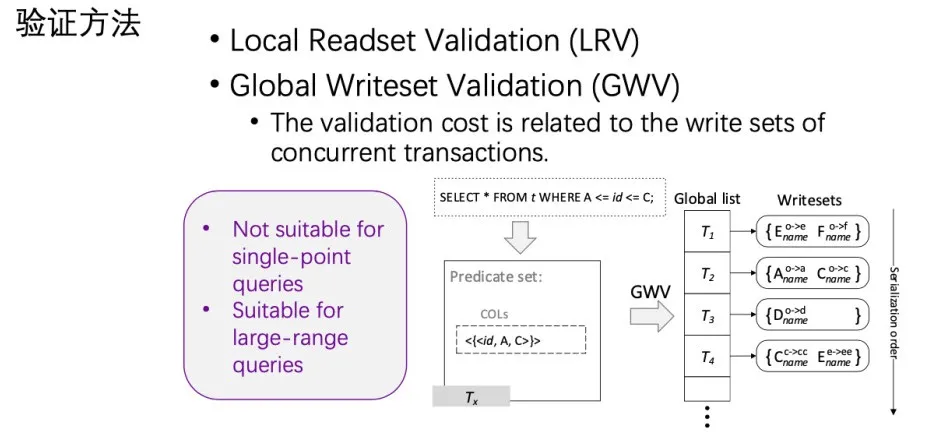


我们做了很多的实验，实验结果可以表明我们的方法，当扩展到一定程度的时候，这种时间戳的分配是不会成为一个扩展性的瓶颈的。

****降低 OCC 的阻塞时间****

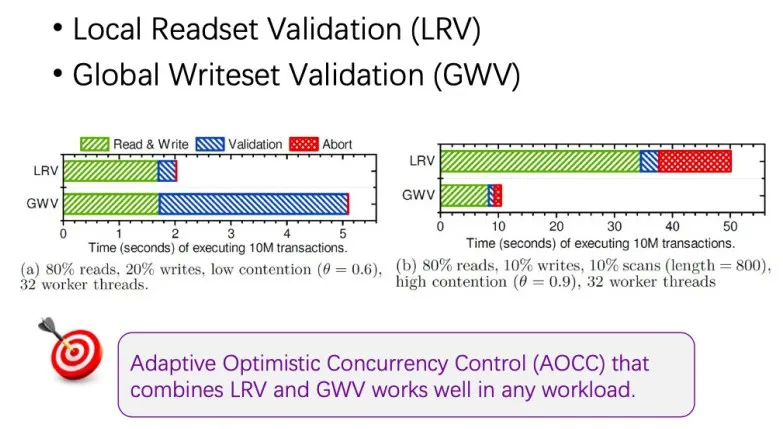
我们也做了一些 OCC 的工作。OCC 在事务访问数据的时候，就放开让事务去访问，访问完了事务要结束的时候会做一个验证叫 Validation，做完验证，再决定这个事务是提交还是回滚。这种方式也是降低事务之间阻塞的一种方法。其实这种事务的最后的正确性验证，有时候会挺耗时间的，所以在这个上面做了一个优化。



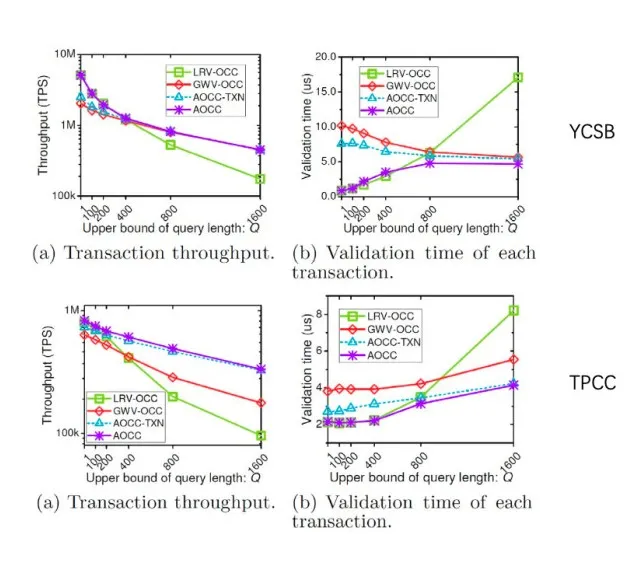


我们认为做正确性验证的方式有两种，一种主要的方式叫 Local Readset Validation，每个事务把它读过的数据记录下来，最后再去查读过的数据有没有被改动过。这种方式的缺点是当读取的东西特别多的时候，它的代价就会相当的大。

然后，还有一种方式叫 Global Writeset Validation，这种方式就是我不去记录每个事务读过的数据，只记录现在有多少正在运行的事务改动了那些数据，也就是说记录的是那些被改动的数据。然后读的时候，观察它的范围有没有包括改的数据，如果包括了验证就失败。这种方式对读取数据内容比较多的事务是友好的，但对那种小的、短的事务并没有那么友好。



所以我们就做了一个叫 AOCC，Adaptive OCC，就是把这两种方式给结合起来，我们会判断一个事务的运行情况，如果读取的数据很多，就用 Writeset Validation；如果读取得很少，就用 Local Readset Validation，这样的话就把两种方式的优点结合起来。



我们做了一些实验，实际上结果确实证明这种方式没有极端的情况的缺陷。因为以前的那种就是读集 Receipt Validation 和 Receipt，德行在在各自的极端情况下都会呈现出特别差的一个性能。但是我们这种方法实际上它是比较均衡的。

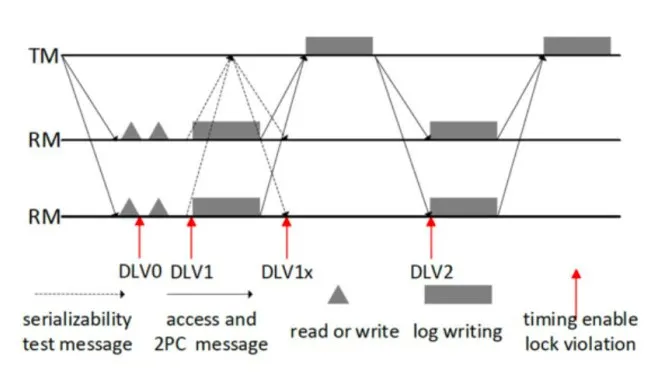
****跨区域高可用系统的锁时长****

然后第三个工作也是关于分布式事务的，就是我们做了类似于 Spanner 这样的系统，跨区域的高可用的系统。



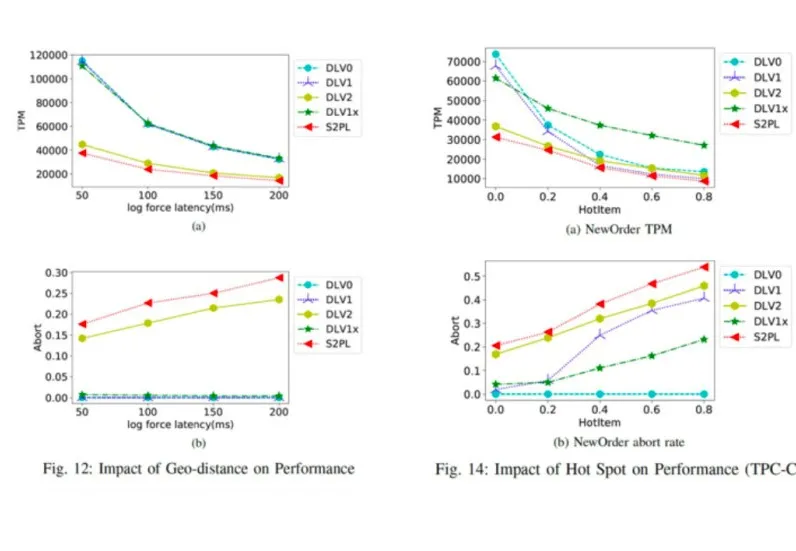
就像我刚才提到的，一旦加锁，加锁的过程中，出现了跨区域数据的通信，这个持锁的时间就会特别的长。这里是一个例子，在跨区域高可用的系统上做的一个两阶段提交。可以看到红色的线是一个加锁的过程，这个过程已经是在一个正常的事务里面最短的加锁过程了，它是在事务提交开始之前加锁，一直到事务提交完成之后释放锁。

对一个普通的事务来说，它本身就是要加锁的，但这个加锁的时间可以看到，在加锁过程中，Prepare 阶段会有大量的本地节点跟异地节点之间的同步，然后在 Commit 阶段，同样的也有大量的本地节点跟异地节点之间的同步，这样的一个同步是很耗时间的，如果在这个时间上去做加锁的话，一旦遇到热点的数据访问，这个事务处理的性能就会极度的下降。所以在这样的条件下，我们就想可不可以用提前释放锁的方式去规避加锁，缩短加锁的长度，直接降低阻塞概率。



然后我们就设计了叫 DLV，LV 的意思是 Lock Violation，实际上就是提前释放锁。DLV 的话我们叫 Distributed Lock Violation，同样是一个两阶段提交的一个协议。我们就看在什么地方释放所，它的效率是好的。我们选择了四个时间点：

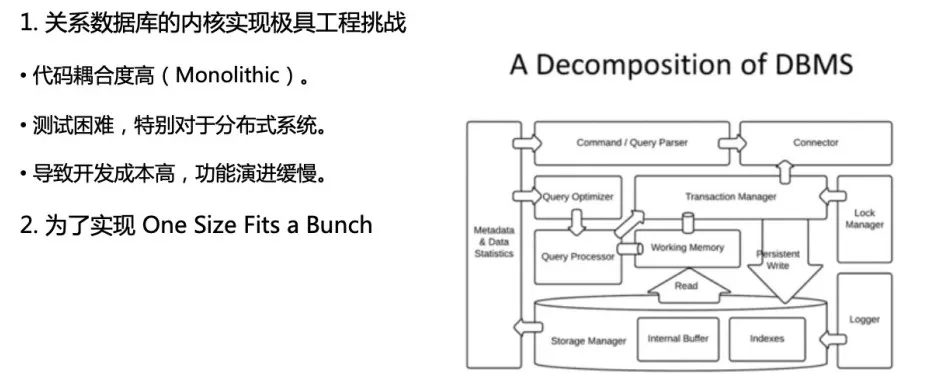
* 第一个是在 prepare阶段之前访问数据的时候，访问一个数据就放一个数据锁，相当于就不加锁，这是一种最极端的方式。但这种方式到后面的回滚率会非常的高，只能通过验证的方式来判断事务是否正确的执行，因此遇到死锁的情况也会很多，然后事务不正确的情况也很多。
* 第二个时间点就是我们叫DLV1，就是在事务基本上数据访问的差不多了，但是协调节点还不太清楚，就是说所有的事务节点是不是已经完全做完了不太清楚，但是所有的事务节点各自都认为它自己做完的时候，这个时候释放锁。
* 第三个时间点就是多加了一个协调的过程，协调的节点会跟所有的事务处理的节点做一个通信，通信完了之后，它认为这个时候所有节点都做完了，这个时候释放锁。
* 第四个时间点就是两阶段提交的第一个阶段结束之后，再释放锁，这个是最安全的。



然后我们就实验了这些不同的加锁和释放锁的方式，得到的一个结论，左边这两张图（横轴是远程通信的时长）说明两个计算机中心离得越远，它通信时间越长，然后用这种传统的方式，会看到时长越长，它的性能就会越差。在高冲突的情况下，分布式事务处理性能就会比较差。但是如果使用提前释放锁的方式，性能就是绿色蓝色的线，表示着它的性能会有一个比较大的提升，这个就说明提前释放锁是有用的。

但什么时候提前释放锁最合适呢？右边这个图我们做的一个实验最后的结论是第三个时间点就是 DLV1x 这种方式，协调节点跟事务处理阶段有一个短通信，这个是本地通信，不是异地通信，通信之后确认所有节点都做完了，这个时候释放锁，这样的负面作用是最少的，而且它的加锁时间也会很短，这种方式它的效率是最高的。

#### **2.2.2 数据库系统解耦合**



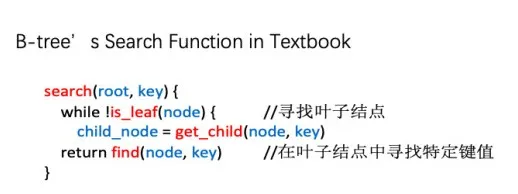
我刚才介绍了三项研究，都是关于分布式事务处理的，然后我接下来再讲一讲我们在数据库系统解耦上面的一些研究，这也是非常有趣的一个问题。什么叫数据库系统的解耦？实际上教科书会把数据库系统拆成一个一个的模块，比如说这个是 collector 就是跟应用对接的连接器，还有 Query Optimize、Query Evaluator 或者叫 Query Processor，就是查询处理查询优化的模块，New Storage Manager、New Transaction Manager，还有各种日志、Lock、Manager 等等，这个是我们教科书上面对数据库一个模块的切分。

但实际上当我们去真正的看一个数据库系统的实现，就会发现这些模块之间实际上没有切分的那么干净的，而且有时候是实际上模块之间很高耦合的，很紧的耦合在一起。对于一个刚开始做数据库的人会觉得跟我们学的东西会不完全一样。然后很少的人真的去探究为什么会是这样。我们在实现数据库系统的时候，实际上这种高耦合的系统架构给我们带来了很大的困扰。

我们当时在某银行改那个 OceanBase 的系统时，改动一个数据类型，我记得好像花了好几个月的时间，很多人去做这个事情。这实际上一听上去会让人比较诧异，但实际上你去看系统的实现，它就是这么回事。一个数据类型好多地方都会用到，必须把每一个地方都清除掉，这个时候就必须花很多时间去读代码去理解去测试的。其实我们觉得如果一个系统的耦合度能够变低，模块之间能够分得很清楚，实际上对系统工程来讲是有很大收益的。

我们回顾刚才讲的一个数据库的发展趋势，叫“One size fits a bunch”，也就是说我们认为以后系统会变得很定制化。如果一个系统的模块化做得很好的话，去定制去改动这个系统也会变得很简单。一旦一个新的硬件出现的时候，我们要去使用新的硬件去对这个系统进行优化，会变得更简单。

其实一个数据库系统的实现，是有需要去做进一步的解耦合，这里面其实有很多问题。我们去做了一些探索，但其实是非常有限的探索，我觉得这个工作其实可以有更多的人去做。我们做的探索，就是说想把并发控制直接从数据库的存储层抽离出来，然后让存储的代码跟并发控制的代码尽量的互不相关。

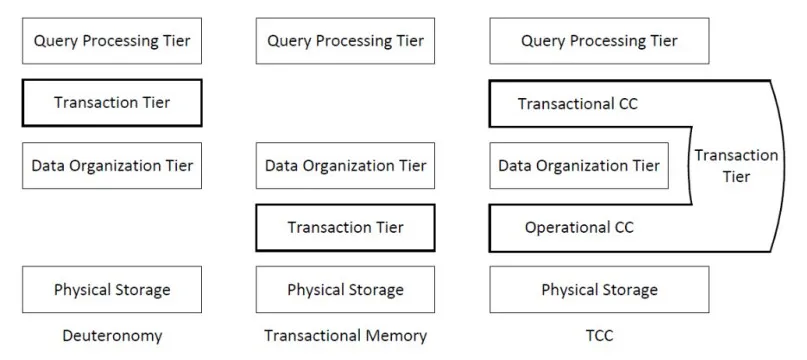


这是一个 B-Tree 的 Search Function 的例子，在教科书里面，关于 B-Tree 的 Search，你可能会看到这样一个代码，非常简单。



但实际上一个 B-Tree 的 Search 没有那么简单，可以看到这里面有好多的东西，这还只是一个例子。如果对开源系统比较熟悉的话，一般的一个开源系统的 B-Tree，差不多要将近十万行代码，非常复杂。

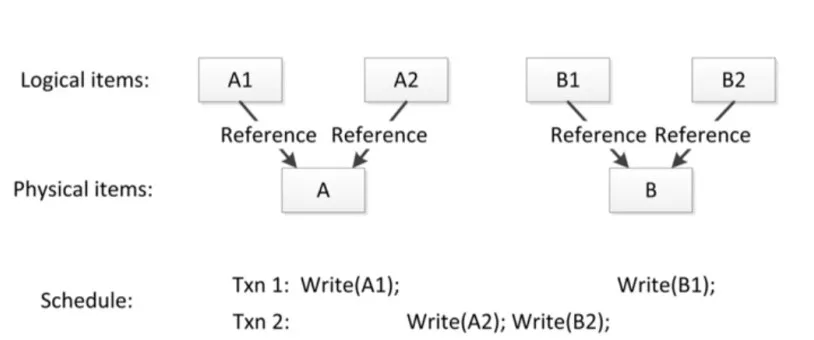
这个代码为什么这么复杂？可以看到 B-Tree 里面有很多的锁，有比如 Latch、Lock 之类的很多东西，它实际上是在做并发控制。当然并发控制只是导致代码复杂的原因之一，但还有其他的原因，并发控制把这个代码变得远远的复杂于 B-tree 本身功能的程度。其实这就是数据库解耦合的一个动机，如果可以把耦合度解开，并发控制可以交给一个单独的模块去做，B-Tree 的代码就可以像第一个例子那样写，事情就变得很简单。



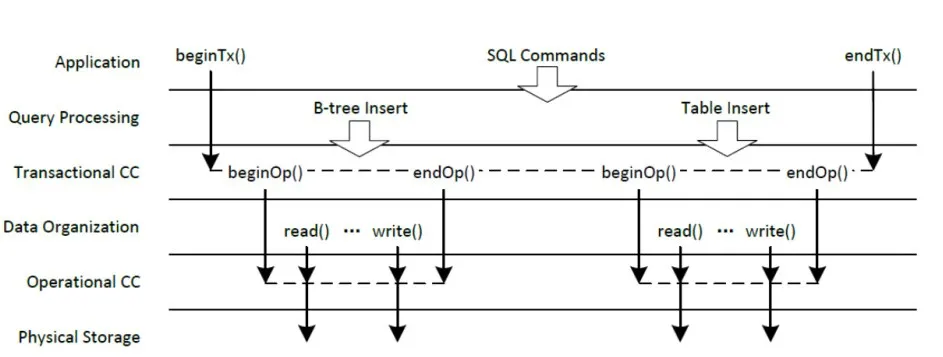
于是，我们就想如果把 CC 就是并发控制从数据库这种体系里面解耦出来应该怎么做？有很多种方式，最左边的这种其实是比较传统的方式，这种方式实际上并没有让数据库存储层变得更简单，只是在存储地上面做了一个事务处理层，这是一个比较浅的做法。中间的这种就是一个很暴力的做法，它是在物理的存储上面加一个 Transaction Tier，然后在上面做存储做运算。

大家应该听说过 Transactional Memory，就是事务内存，这种就是直接用事务内存做事务处理，是一种很暴力的方式。最右边是我们提出来的方式，把并发控制的层次分成了两层，一种是我们叫操作层的并发控制，一种是事务层的并发控制，把它们合在一起变成一个新的模块。实践下来肯定是我们这种方式效果明显地更好，这种很暴力的事务内存的方式，实际上性能是不可以接受的。

我们其实看到现在有一些做存储的同学，他有一些比较天真的想法，他认为把事务做到存储的最底层，然后上面就不需要关心事务了，实际上那是不行的。事务是跟系统的功能是有很多耦合的因子在里面的，不能完全把它抛弃掉。然后最左边这种方式的结果是不彻底的，实现 B-tree 的时候还是会挺复杂的。然后我们提出的这种方式，可以清楚的把 CC 给抽离出来。

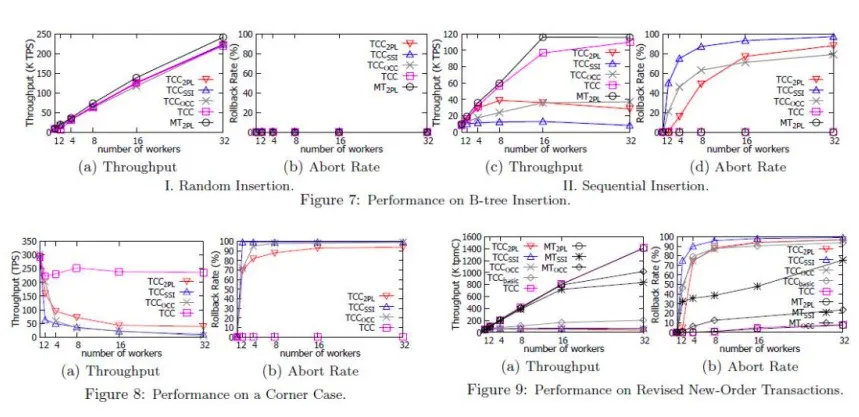


这个事情其实并不是那么的简单，上面是一个很简单的例子，A 和 B 是两个物理数据，然后在数据库的数据结构里面，定义两个引用（Reference），A1 和 A2，B1 和 B2，A1 和 A2 都是指向 A 的，B1 和 B2 都是指向 B 的。上面的事务层实际上是对 A1、A2、B1、B2 进行访问的。这个时候如果只是通过逻辑层去决定事物跟事物是否冲突的话，是会出错的。因为这里的逻辑层跟物理层，有一个重复引用的关系，可以看到下面这个事务处理的 Schedule，从 A1、A2、B1、B2 这种方式去看，好像这两个事务这样处理是没问题的，它是有这种可串行化的能力的，但实际上并没有，因为 A1 和 A2 指向的是同样一个数据。



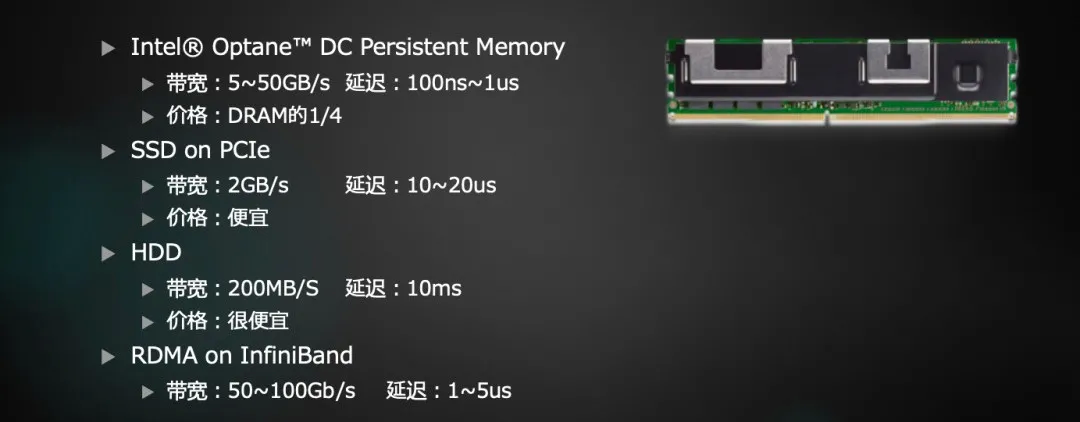
这种实际上就是说如果真的要去把这个事务抽离出来的时候，会有很多的问题需要去解决，我这里没有办法深入地探讨，总之我们做了这样一个尝试，我们叫 Transparent Concurrency Control（TCC），就是透明的并发控制的一种模式。

我们把这个事务层抽成两层，一个是事务层次的并发控制，一个是操作层次的并发控制，让这两种东西能够配合起来使用。然后用户去编写程序的时候，他可以不要去关心操作层面的并发控制，他直接去写他的 B-Tree 就行了。但是写好 B-Tree 之后，在事务层次上的这种并发控制，需要提供一些语义的信息说明哪种操作跟哪种操作之间实际上是不会冲突的，这样整个事务处理过程就可以保证正确性和高效性。



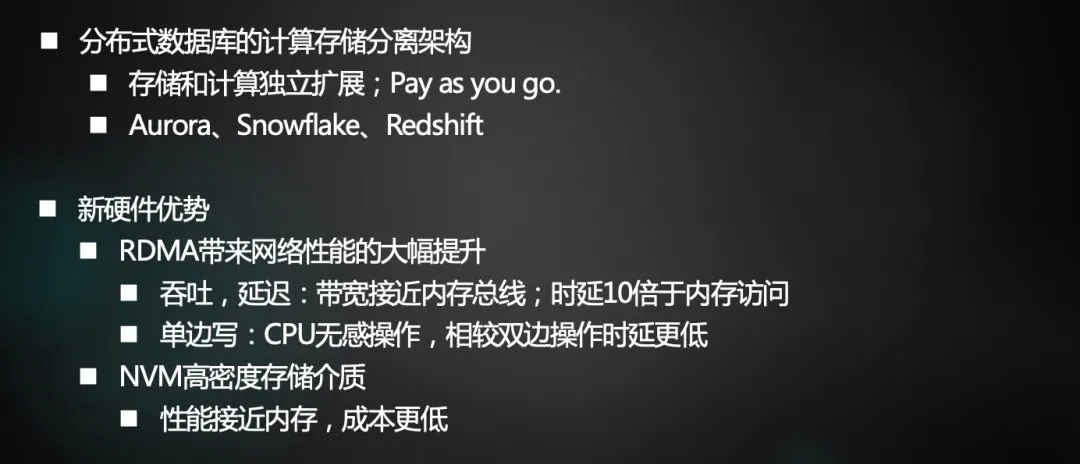
我们做了一些实验，去验证这样的一个解耦。右边的这些图，这种圆圈的线代表原始数据库实现的性能，可以看到我们的方式（TCC）的性能在很多情况下可以接近原始数据库的性能，这是解耦之后数据库的表现。很多时候解耦之后的表现可以接近原始数据库的性能，所以我们觉得这种解耦实际上还是可行的。但如果真的要把它用到一个现实的系统当中，其实并没有那么简单。这是我介绍的第二个研究工作，就是我们在数据库解耦上面的一些有趣的发现。

#### **2.2.3 新硬件**

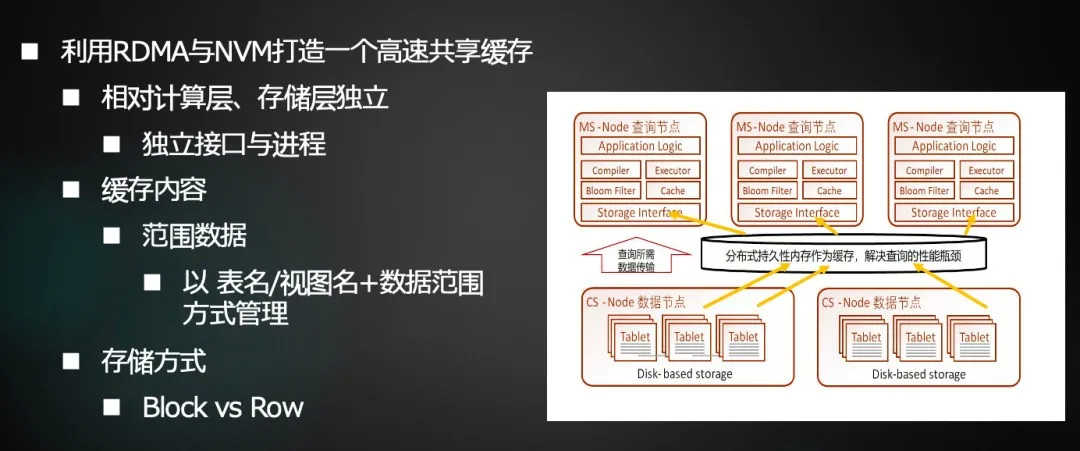


最后我再谈一谈新硬件，就是这种非易失内存。英特尔的傲腾是现在市面上唯一的一个真正的非易失内存产品，图中是产品的相关指标。对于存储器件的话，我们一般看两个指标，一个是带宽，一个是延迟。

可以看到它的带宽和延迟都是远远超过 SSD，当然更加超过这种硬盘。它的价格会比 SSD 和硬盘要昂贵不少，但相对于内存而言，它还是便宜的，我们可以预测它后面会越来越便宜，它跟 SSD 之间的一个价格的差异会变得越来越小，所以以后它有可能会取代 SSD 这种固态硬盘，但是不会太早。这种新的存储，它的性能更好，又比内存的造价低，所以它以后在系统当中肯定是有很重要的一个位置的。现在有了这种硬件之后，我们需要讨论在数据库系统里面，这个硬件到底起什么作用，一个新的数据库的架构应该怎么去使用它？怎么定位它的价值和位置？



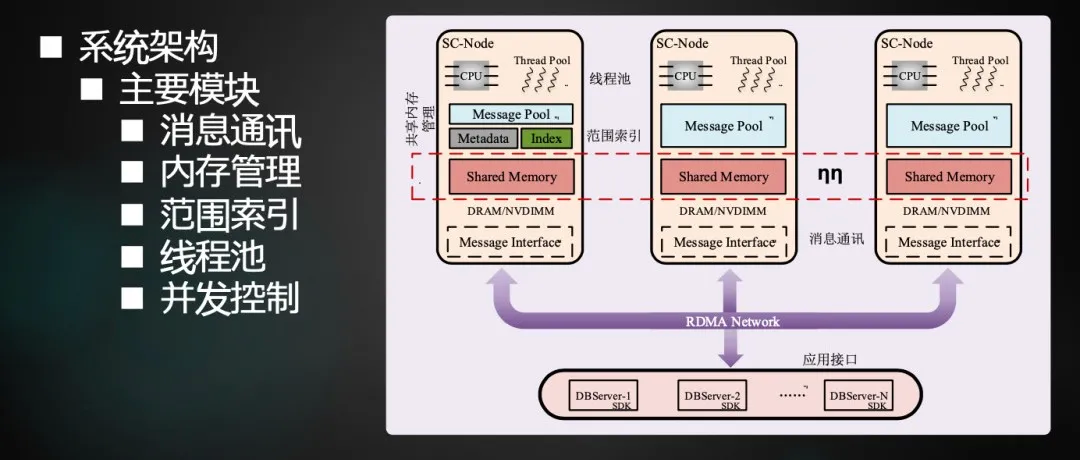
我们团队对这个东西讨论了很长的时间，最后有一个这样的设计，首先非易失内存这个东西，当然要用到数据库里面，数据库有各种形态的数据库，不同的形态的数据库使用方式是不一样的，但我们最后把它定位在云的数据库上面，因为我们知道云是未来的最主要的架构。在云的数据库上面怎么去用这个东西，我们觉得它跟 RDMA 的使用应该结合在一起。

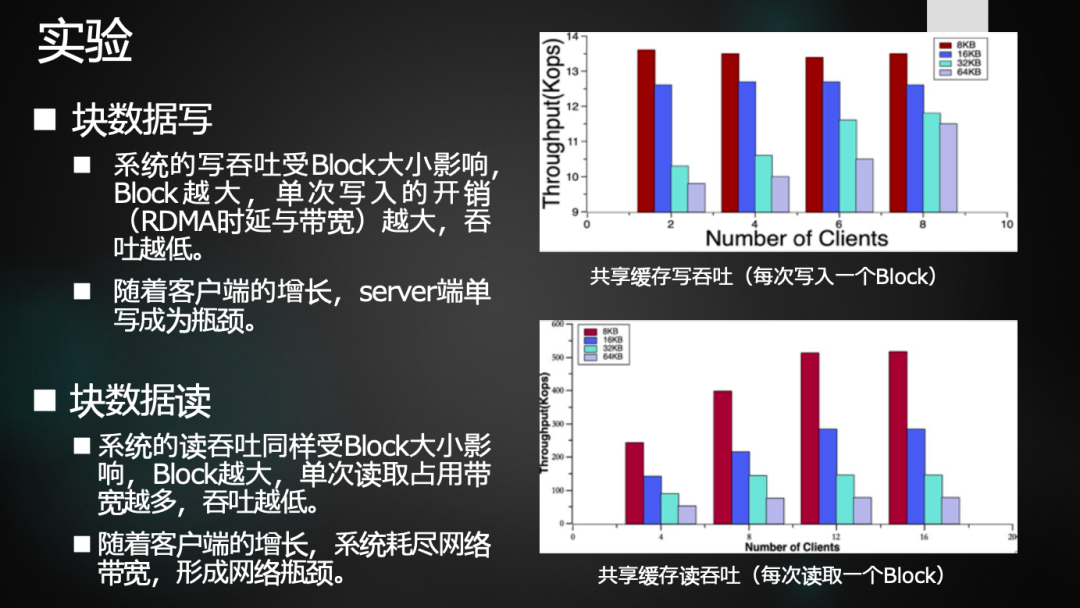


现在的云数据库变成一种计算节点跟存储节点是相对分开的架构方式。然后一旦 NVM 加进来之后，我们希望它成为计算节点跟存储节点之间的一个缓存。我们觉得现在暂时不能用它来做全量数据的存储，因为它的价格实在是比较昂贵，很多冷的数据，完全没有必要存在这样昂贵的存储里面，所以它作为一种缓存，比较合适的。

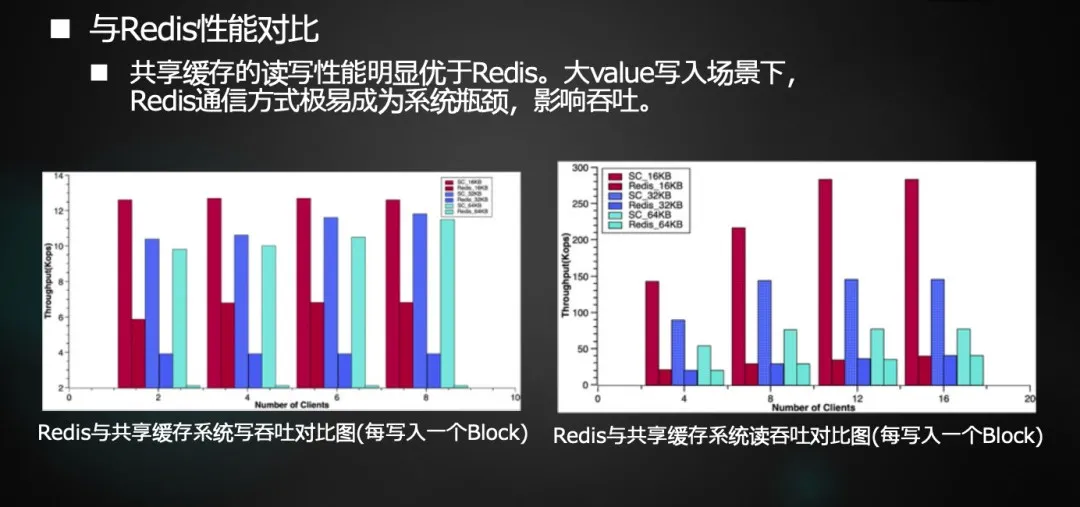
另外一方面，它作为一个缓存，不应该是一个割裂的节点，因为我们去看它的性能指标，可以看到实际上这种非易失内存的吞吐、延迟，和在高速网络上的 RDMA 的吞吐和延迟是比较接近的。如果比较接近的话，这个器件是通过 RDMA 的远程去访问，还是通过本地访问的速度差异有可能并不会很大。如果这个速度的差异并不会很大的话，我们实际上是可以把多个节点的 NVM 联合在一起，作为共享的缓存，缓存共享有非常多的优势，省去了很多缓存数据同步的代价，然后还可以让系统的负载均衡变得更好。

我们决定去设计这样一个系统架构。这个是 NVM，我们叫存储节点和计算节点之间的一个缓存。





我们这个事情正在做的过程中，所以目前为止我们是实现了一个分布式的缓存，可以作为共享缓存来用。我们开始讨论它应该是作为数据库而言，是行级别的缓存还是块级别的缓存，最后我们的选择是块级别的缓存，主要的原因还是因为实现起来更简单。我们先试一试，如果做到行级别的缓存的话，是有很多的工作量的，我们后期可能还会去尝试。



然后缓存的基本的测试，我们觉得我们的实现基本已经到位了，就是它的带宽的瓶颈基本上压到了 RDMA 访问带宽的瓶颈，如果要对它进行读写的话，它的瓶颈基本上就是 RDMA 远程访问的瓶颈，然后它的性能是远远高于像 Redis 这样的一个系统的，我们希望用这个缓存把它放在数据库里面，去提升这种云数据库的一个性能，但这个过程我们还在实现当中，我们有一些初步的结果，它是有一些效果的，特别是面对底层是 SSD 或者磁盘这样的系统的时候。我们希望后面有更明确结果的时候，再给大家介绍。

## **03 相关论文列表**



这是我们实验室的一些代表性论文，不是很全，我刚才讲的部分技术并不在列，因为还没有公开发表。如果感兴趣的话，大家可以阅读一下。

## **写在后面**

华东师范大学周烜教授也是 2020-2021 年度美团科研课题合作学者。当前美团技术团队与超过 30 位来自国内外高校和科研院所的学者建立了科研课题合作。美团科研合作计划，基于美团在生活服务领域全场景里提炼出的科研命题，面向学术界征集前沿解决方案。

我们致力于与学术界“一起解决真实世界的问题”，愿与学术界共同推动产学研成果落地。2021 年将更加精彩纷呈，敬请期待。

****作者介绍****：

****周烜****，华东师范大学教授，数据科学与工程学院副院长

2001 年本科毕业于复旦大学，2005 年在新加坡国立大学取得博士学位，2005 年至 2010 年期间先后在德国 L3S 研究中心和澳大利亚联邦科工组织从事科研工作，随后在中国人民大学信息学院任教 6 年，于 2017 年 3 月加入华东师范大学。研究兴趣包括数据库系统和信息检索技术。曾参与和负责多个国内外的科研项目和工业合作项目，积累了丰富的数据管理系统研发经验。研究成果被发表于众多国际一流的学术会议和期刊。凭借在分布式数据库领域的成果转化获得国家科技进步二等奖和教育部科技进步一等奖。入选教育部“新世纪优秀人才”支持计划。

****本文转载自公众号美团技术团队（ID：meituantech）。****

****原文链接****：

[周烜：华东师范大学的数据库系统研究](https://mp.weixin.qq.com/s?_<i>biz=MjM5NjQ5MTI5OA==&amp;mid=2651754717&amp;idx=2&amp;sn=4a697a0ad231ce17477565320f558c08&amp;chksm=bd1249908a65c0860e3ef8dc31faddf2dbc2700f71307da3ff1eb7884b6d7e792272a2582669&amp;scene=27" \l "wechat</i>redirect" \t "https://cloud.tencent.com/developer/news/_blank)