分布式存储技术

存储引擎

存储引擎是分布式存储技术中的关键组成部分之一，负责处理数据的持久化、管理和访问。

存储引擎概述：

存储引擎是数据库系统或分布式存储系统中负责数据存储和管理的组件。

存储引擎定义了数据如何组织、存储、索引和访问。

存储引擎的选择会对数据的读写性能、可扩展性和可靠性产生重要影响。

存储引擎的特性：

数据模型：存储引擎可以支持关系型数据模型、文档模型、键值模型、列族模型等不同的数据模型。

数据持久化：存储引擎负责将数据持久化到磁盘或其他存储介质，并恢复数据以确保数据的可靠性和持久性。

并发控制：存储引擎需要提供并发控制机制，以支持多个客户端同时对数据进行读写操作。

索引技术：存储引擎使用索引来提高数据的访问速度，常见的索引结构包括B树、B+树、哈希索引等。

分布式存储：一些存储引擎设计用于分布式环境，能够在多个节点上存储和管理数据，提供高可用性和可扩展性。

存储引擎的选择因素：

数据访问模式：根据应用的读写比例、数据大小和访问模式等因素选择适合的存储引擎。

性能需求：根据应用的性能需求选择具有高性能读写能力的存储引擎。

数据一致性要求：根据应用的数据一致性要求选择提供合适的事务支持的存储引擎。

可扩展性需求：根据应用的数据增长和负载变化预测选择具有良好可扩展性的存储引擎。

存储系统=存储引擎+数据模型

存储系统由存储引擎和数据模型两个组成部分构成。

1. 存储引擎：

   存储引擎是存储系统的核心组件之一，负责数据的存储、管理和访问。它定义了数据在底层存储介质上的组织方式、存储结构和访问方法。存储引擎负责数据的持久化，将数据写入磁盘或其他存储介质，并提供读取和更新数据的接口。不同的存储引擎可能使用不同的数据结构、索引技术和存储策略，以满足不同的性能、可扩展性和一致性要求。

2. 数据模型：
   数据模型定义了数据在存储系统中的组织方式和表示方式。它描述了数据的结构、关系和约束，以及对数据的操作和查询方式。数据模型可以分为不同的类型，常见的包括关系型数据模型、文档模型、键值模型、列族模型等。每种数据模型都有其特定的数据结构和操作方式，适用于不同类型的应用场景。

   - 关系型数据模型：使用表格（二维表）来表示数据，关系由表之间的关联和键值来建立。

- 文档模型：将数据组织为文档（如JSON或XML）的集合，每个文档可以具有不同的结构。
- 键值模型：将数据存储为键值对的集合，通过唯一的键值来访问数据。
- 列族模型：将数据组织为列族的集合，每个列族包含一组相关的列，适用于大规模的分布式存储。

数据模型定义了数据的逻辑结构和操作方式，而存储引擎负责将数据模型映射到底层的物理存储结构，并提供对数据的读写和查询操作。存储引擎和数据模型紧密配合，共同构成了完整的存储系统。

哈希存储引擎

用于分布式存储系统中，通过哈希函数将数据分散到不同的节点上进行存储和访问，实现数据的均衡分布和快速查找。

哈希函数：

哈希存储引擎使用哈希函数将数据转换成固定长度的哈希值。哈希函数将数据块映射到存储结构中的一个位置，使得数据可以快速定位和访问。哈希函数通常具有以下特性：

输入数据的变化会导致哈希值的不可预测性和不可逆性。

相同的输入数据始终生成相同的哈希值。

不同的输入数据尽可能生成不同的哈希值（避免哈希冲突）。

哈希表：

哈希表：哈希表是一种以哈希值为索引的数据结构，用于存储和访问数据。哈希表通常使用数组来实现，每个数组元素称为哈希槽（bucket）。数据被哈希函数映射到相应的哈希槽，可以直接在哈希槽中进行数据的读写操作。哈希表具有快速的插入、查找和删除操作，适用于需要高效访问的场景。

哈希存储引擎的特点：

快速访问：由于使用哈希函数进行数据映射，哈希存储引擎可以实现快速的数据访问和定位。

平均查询时间复杂度为O(1)：在理想情况下，哈希存储引擎的平均查询时间复杂度为常量级别，即O(1)。

不支持范围查询：哈希存储引擎主要适用于等值查询，不支持范围查询（如大于、小于等条件）。

哈希冲突处理：由于哈希函数的输出空间有限，不同的数据可能会生成相同的哈希值，导致哈希冲突。哈希存储引擎需要采取冲突解决策略，如链表法或开放寻址法，来处理哈希冲突。

支持随机读不支持顺序读：

随机读：哈希存储引擎通过哈希函数将数据映射到特定的位置，使得通过给定的键值可以快速定位到对应的数据。因此，当你拥有键值时，可以直接通过键值进行查询，这被称为随机读。随机读具有快速的查询速度，平均时间复杂度为常量级别（O(1)）。

顺序读：顺序读是指按照数据在存储中的物理顺序进行顺序访问。然而，由于哈希存储引擎将数据根据哈希函数分散存储在不同的位置，它们的物理顺序可能是不连续的。因此，无法直接通过顺序读的方式获取存储在哈希存储引擎中的全部数据，因为数据在物理上可能是分散的。

综上所述，哈希存储引擎支持通过键值进行快速的随机读取，但不支持对存储中的数据进行连续的顺序读取。如果需要按照顺序读取数据，可能需要使用其他存储引擎或数据模型，例如基于索引的存储引擎（如B+树索引）或支持顺序访问的数据模型（如关系型数据库中的表格）。

Bitcask

Bitcask 是一种常见的哈希存储引擎，它被广泛应用于键值存储系统中，旨在提供快速的数据读写和高可用性。它的设计目标是在具有大量随机写入和随机读取操作的负载下保持低延迟和高吞吐量。

Bitcask 架构主要组件：

内存哈希索引（Memory Hash Index）：在内存中维护一个哈希索引，用于快速查找键值对的存储位置。

数据文件（Data Files）：存储实际的键值对数据。

日志文件（Log Files）：记录写入操作的日志文件，用于数据的持久化和崩溃恢复。

合并过程（Merge Process）：定期或根据配置触发的合并操作，用于将数据文件和日志文件合并并去除重复的键值对。

特点：

高性能：Bitcask 的写入和读取操作都具有很高的性能，由于数据是追加写入和顺序读取的，可以实现较低的写入延迟和高吞吐量的读取操作。

内存索引：Bitcask 使用内存中的哈希索引，实现快速的键值查找，从而提高读取的效率。

持久化：Bitcask 将写入操作追加到日志文件中，保证数据的持久性，同时可以通过日志文件进行崩溃恢复。

合并过程：Bitcask 定期或根据配置触发合并操作，将数据文件和日志文件进行合并，去除重复的键值对，保持存储空间的紧凑性和读取性能的稳定性。

Bitcask 具体存储方式：

内存中的哈希索引：维护一个哈希表，将键映射到数据文件中的偏移量。

数据文件：存储实际的键值对数据，数据文件是不可变的，每个数据文件都包含一个索引块（Index Block），用于加速键值的查找。

日志文件：将写入操作追加到日志文件中，包括键、值、时间戳和 CRC 校验和。

日志结构流派：

日志结构流派是一种在计算机存储系统中常见的设计范式，它的核心思想是将所有的写入操作追加到顺序写入的日志（log）中，而不是直接修改原始数据结构。这种设计范式的目的是最大化写入操作的性能，并减少磁盘随机写入的开销。日志结构流派的典型代表包括 Bitcask、LSM树等。

以下是日志结构流派的一些特点和优势：

1. 顺序写入性能优势：日志结构流派将所有写入操作追加到日志中，这可以通过顺序写入磁盘来实现高性能。相比于随机写入操作，顺序写入操作通常更快，因为它充分利用了磁盘的预读和顺序写入的优化。
2. 崩溃恢复简单：由于所有的写入操作都追加到日志中，崩溃恢复非常简单。在系统崩溃或重启后，可以通过重新执行日志中未完成的操作来恢复到一致的状态。
3. 数据持久性和不可变性：日志结构流派通过追加写入操作到日志中，可以保证数据的持久性和不可变性。一旦写入操作完成，数据将被追加到日志中并不再改变，从而提供了数据的持久性和可靠性。
4. 合并操作优化：为了维护较小的存储空间和提高读取性能，日志结构流派通常会使用合并操作。在合并操作中，将多个日志或数据文件合并为更大的文件，并去除重复的键值对。这样可以减少存储空间的占用，并提高读取操作的效率。

各种操作：

写入操作：

将写入的键值对追加到日志文件的末尾。

将键值对写入内存中的哈希索引，更新键的位置信息。

读取操作：

在内存中的哈希索引中查找键的位置信息。

根据位置信息，从数据文件中读取相应的键值对数据。

合并操作：

选择需要合并的数据文件和日志文件。

从日志文件中读取写入操作，将新的键值对追加到新的数据文件中。

去除重复的键值对，更新内存中的哈希索引。

崩溃恢复：

在启动时，读取最新的日志文件，恢复未合并的写入操作。

重建内存中的哈希索引。

恢复到最新的一致状态。

合并：

Bitcask 使用合并（Merge）过程来管理数据文件和日志文件，以减少存储空间的占用和提高读取性能。合并操作会将较小的数据文件和日志文件合并为更大的文件，并去除重复的键值对和删除标记。

下面是 Bitcask 合并数据的一般流程：

1. 选择要合并的数据文件和日志文件：
   Bitcask 根据一定的策略选择一组较小的数据文件和日志文件进行合并。这个策略可以基于文件的大小、文件的数量、合并的时间间隔等因素。
2. 创建新的数据文件：
   首先，Bitcask 创建一个新的数据文件，作为合并的目标文件。
3. 合并数据文件：
   Bitcask 逐个读取选定的数据文件，并将其中的键值对逐个写入到目标文件中。在此过程中，Bitcask 需要检查键值对的有效性，去除重复的键值对，以及处理删除标记。
4. 合并日志文件：
   Bitcask 接着读取选定的日志文件，并将其中的写入操作逐个应用到目标文件中。这样可以确保合并后的文件包含最新的写入操作。在此过程中，Bitcask 需要忽略已经存在于数据文件中的键值对，只添加数据文件中不存在的键值对。
5. 更新内存中的哈希索引：
   在合并完成后，Bitcask 需要更新内存中的哈希索引，以反映合并后的数据文件的新的索引信息。
6. 删除旧的数据文件和日志文件：
   合并完成后，Bitcask 可以安全地删除已经合并过的较小的数据文件和日志文件，以释放存储空间。

通过合并操作，Bitcask 可以维护存储空间的紧凑性，减少文件数量，提高读取性能，并且合并过程可以在后台进行，不影响主要的读写操作。

需要注意的是，合并过程是一个周期性或触发性的操作，具体的合并策略和触发条件可以根据应用程序的需求进行配置。一般来说，较小的数据文件和日志文件积累到一定程度或者一定时间过去后，会触发合并操作以保持存储的高效性。

Bitcask主要问题

Bitcask 的主要问题之一是内存中存储索引受内存容量限制。由于 Bitcask 使用内存中的哈希索引来加速读取操作，索引的大小直接影响着索引的性能和系统的可扩展性。

B树存储引擎

B树存储引擎是一种常见的数据库存储引擎，用于高效地存储和管理数据。它基于B树数据结构，并在其基础上实现了数据的持久化、索引、事务处理和并发控制等功能。

关系型数据库

B树存储引擎通常用于支持关系型数据库系统，而不是键值数据库系统。关系型数据库系统使用B树存储引擎来实现索引结构，以支持高效的数据检索和查询操作。

在关系型数据库中，表格通常由多个列组成，每个列都可以作为索引的键。B树存储引擎可以为这些索引键构建B树索引，提供快速的数据查找和范围查询能力。通过B树的多层级结构和自平衡特性，关系型数据库可以高效地管理大量数据，并保持一致的性能。

相比之下，键值数据库通常使用基于哈希表或LSM树等数据结构来实现存储引擎，以支持高速的键值对查找和写入操作。这些存储引擎在特定的应用场景下更加适用，例如缓存、会话存储和分布式存储系统等。

InnoDB

InnoDB是一个流行的关系型数据库管理系统MySQL中的存储引擎，它提供了可靠性、高性能和事务支持等特性。

特点：

页做节点：

InnoDB按照页面来组织数据是指它在磁盘上将数据划分为固定大小的数据页并进行存储。数据页是InnoDB存储引擎管理数据的最小单位。具体来说，InnoDB使用固定大小的数据页（通常为16KB）来存储数据。每个数据页都是一个连续的磁盘块，用于存储表数据、索引数据和其他相关信息。

通过按照页面来组织数据，InnoDB可以实现以下优势：

1. 磁盘IO优化：按照页面组织数据可以减少磁盘IO次数。InnoDB使用缓冲池（Buffer Pool）来缓存数据页，将经常访问的数据页保留在内存中。这样可以避免频繁从磁盘读取数据，提高数据访问的速度和效率。
2. 空间利用效率：按照页面组织数据可以更好地利用磁盘空间。由于每个数据页是固定大小的，InnoDB可以更加高效地管理和分配磁盘空间，避免了碎片化和浪费。
3. 并发控制：按照页面组织数据也有助于实现并发控制。InnoDB使用行级锁定来管理并发访问，而数据页作为锁的基本单位，可以更细粒度地控制并发操作，提高并发性能和数据一致性。

行级锁定：

InnoDB使用行级锁定来管理并发访问，提供更细粒度的数据并发控制。行级锁定允许多个事务同时读取和修改不同的行，提高了并发性能和并发控制的灵活性。

数据一致性和崩溃恢复：

InnoDB通过使用日志（redo log）来保证数据的一致性和持久性。通过将数据修改操作记录在日志中，可以在系统崩溃或异常情况下恢复数据的一致性状态。

B+树索引结构：

InnoDB使用B+树索引结构来组织和管理数据。B+树索引提供了高效的数据查找和范围查询能力，适用于大型数据集和高并发环境。

缓冲池（Buffer Pool）：

InnoDB通过使用缓冲池来管理数据页的读取和写入操作，提高了数据的访问速度。缓冲池是一个内存区域，用于缓存磁盘上的数据页，减少磁盘IO操作。

自适应哈希索引：

InnoDB引擎还支持自适应哈希索引，用于加速频繁访问的热数据。自适应哈希索引是一种内存索引结构，可以减少磁盘IO操作和提高查询性能。

日志区分（InnoDB日志与Bitcask日志结构）：

在InnoDB中，日志和数据是两个不同的概念。InnoDB的提交日志（redo log）用于记录事务的数据修改操作。当应用程序执行事务时，所有的数据修改操作都会首先记录在提交日志中，然后再将其应用到数据文件中。数据文件中存储的是实际的数据内容，而提交日志用于确保事务的持久性和一致性，并支持系统的恢复操作。

Bitcask的日志流派结构中的日志是指存储引擎在写入数据时将数据操作追加到日志文件中的过程，用于实现实现高性能的写入操作。在Bitcask中，日志实际上就是存储的数据。当应用程序向Bitcask存储引擎写入数据时，数据会被追加到日志文件中，形成有序的日志序列。

而所以，在Bitcask中，日志就是存储的数据本身，而在InnoDB中，日志是用于记录数据修改操作的一种机制，和实际的数据是分开存储的。

内存缓冲区替换策略

LRU

LRU策略的优点是简单且容易实现，适用于访问模式相对均匀且没有明显热点数据的场景。然而，LRU策略在面对访问模式不均匀、存在热点数据或周期性访问模式时可能会导致缓冲区命中率下降，性能下降。

LIRS

LIRS算法相对于传统的LRU（Least Recently Used）策略具有更好的性能。它能够更精确地识别最近被访问的数据页，并根据相互引用关系进行调整，从而更准确地保留热数据和淘汰冷数据。然而，LIRS算法的实现相对复杂，需要维护附加的数据结构来跟踪数据页之间的引用关系。

B树存储引擎仍不具备可扩展性

固定大小的索引节点限制了B树的可扩展性。当需要存储大量数据时，B树可能需要更多的索引节点来容纳数据。如果索引节点大小有限，那么为了存储大量数据，B树的高度将会增加，因为每个节点能容纳的键值对数量有限。随着B树高度的增加，查询或更新操作就需要更多的磁盘访问，导致性能下降。另外，如果键值对的大小较小，但索引节点的大小较大，那么存储引擎会浪费较多的空间来存储指针和未使用的空间。

LSM树存储引擎

LSM树是一种用于存储和管理数据的存储引擎，特别适用于写入密集型和大规模数据的场景。它采用了一种基于日志结构的存储方式，通过合并和压缩操作来提供高性能和可扩展性。

与Bitcask区分：

LSM存储引擎和Bitcask是两种常见的键值存储引擎，它们在数据组织和存储方式上存在一些异同。

相似之处：

1. 键值存储：LSM存储引擎采用了LSM树的结构，包括MemTable和多个SSTable。而Bitcask使用哈希索引和日志文件的结构。
2. 内存和磁盘结合：两者都将数据分为内存和磁盘两个层级。数据首先写入内存中的数据结构，然后根据一定的策略刷写到磁盘上的持久化存储结构。
3. 高写入性能：由于采用了追加写入的方式，LSM存储引擎和Bitcask在写入操作上都能够提供较高的性能。

不同之处：

1. 数据有序性：LSM存储引擎中的数据是有序存储的，SSTable中的键值对按照键的顺序排序。而Bitcask中的数据是无序存储的，每个键值对都追加到日志文件中，没有进行排序，**区间查询性能差。**
2. 读取操作：由于LSM存储引擎的数据是有序存储的，范围查询操作可以高效地利用SSTable的有序性。而Bitcask由于数据无序存储，范围查询需要遍历整个日志文件。
3. 内存使用：LSM存储引擎需要较大的内存空间用于存储MemTable，以支持高性能的写入操作。而Bitcask相对较小的内存占用，因为它仅使用内存索引来加速键值对的查找。
4. 写入流程：LSM先追加到内存mmtable，大小达到阈值后生成磁盘sstable；bitcask直接往磁盘的activate file里追加，active file够大了变成old file（不再修改），bitcask内存仅用于存全部索引。
5. 数据存储：bitcask的索引放内存，数据放外存，**受内存容量限制**；LSM**不在内存放索引**，索引和数据存一起放外存里。

总体来说，LSM存储引擎适用于高写入负载的场景，通过牺牲一部分读取性能来提供高效的写入操作。Bitcask适用于读取密集的场景，适合于数据集较小的情况，且对实时性要求较高。

键值模型：

LSM存储引擎在数据存储方面采用了一种特殊的索引结构，通常是基于有序字符串表（Sorted String Table）的方式进行索引。

LSM存储引擎中的索引是通过SSTable实现的。SSTable是一个有序的、不可变的数据文件，其中的键值对按照键的顺序进行排序。通过这种有序性，LSM存储引擎可以利用范围查询和快速查找的优势。

LSM存储引擎中的索引通常是基于跳表（Skip List）或B树（B-tree）等结构实现的。这些索引结构可以在SSTable文件中进行快速查找，并定位到存储键值对的位置。

需要注意的是，LSM存储引擎的索引主要用于加速读取操作，而写入操作通常是追加写入的方式，并不需要经过复杂的索引计算。写入操作首先会写入内存中的数据结构（如MemTable），然后根据一定的策略将数据刷写到磁盘上的SSTable中。因此，LSM存储引擎的索引更加关注读取性能和范围查询的优化。

结构：

写缓存（Write Buffer）：用于接收新写入的数据，通常是一个内存结构，如散列表（Hash Table）或跳跃表（Skip List）。

磁盘层（Disk Level）：由多个层次组成，每个层次包含一个或多个磁盘文件，用于持久化数据。

MemTable：将写缓存中的数据以一种有序的方式存储到内存中。

SSTables：根据键值对的键进行排序的不可变磁盘文件，每个SSTable包含多个数据块。

Bloom Filter：用于快速判断某个键值对是否存在于SSTable中，以减少磁盘查找的开销。

写入操作：

新写入的数据首先被追加到写缓存中。

当写缓存达到一定大小或一定时间间隔时，将其转换为内存中的MemTable。

MemTable以有序的方式存储数据，通常采用跳跃表或红黑树等数据结构，以提高写入性能。

MemTable达到一定大小后，将其转换为一个新的SSTable，并将其写入到磁盘层。

读取操作：

读取操作首先在写缓存中查找数据，如果找到则返回结果。

如果在写缓存中未找到，则在磁盘层的SSTables中进行查找。

从较新的最小的SSTables开始查找，一级一级往下查，SSTable会越来越大。

通过Bloom Filter可以快速判断某个SSTable中是否可能存在待查找的键值对，减少不必要的磁盘查找。

合并和压缩：

为了减少磁盘空间的占用和提高查询性能，LSM树定期进行合并和压缩操作。

合并操作将多个SSTables合并为一个更大的SSTable，以减少磁盘文件数量。

压缩操作将SSTable中的冗余数据进行删除，以减少磁盘空间的占用。

LSM树存储引擎通过将写操作追加到写缓存中，并通过批量合并和压缩操作将数据持久化到磁盘中，实现了高效的写入性能和较小的磁盘空间占用。同时，通过Bloom Filter和有序的SSTables，LSM树能够提供快速的读取操作。这使得LSM树在大规模数据和写入密集型的场景下，具有良好的**可扩展性**和性能。

MemTable

MemTable是LSM树中的内存结构，用于存储最新写入的数据。

写入操作时，新的键值对首先被追加到MemTable中，并保持有序。

由于MemTable位于内存中，读取操作可以快速地在其中查找数据，从而提供低延迟的读取性能。

SSTable

SSTable是LSM树中的磁盘文件，用于持久化数据。

每个SSTable包含多个数据块（Data Blocks），每个数据块存储一段键值对的数据。

SSTable中的键值对根据键的排序顺序进行有序存储，以支持高效的范围查询操作。

SSTable也分级，经过合并足够大之后会放到下一级。

关系模型

关系模型和键值模型

关系模型和键值模型是两种不同的数据模型，它们在数据组织和访问方式上有所区别，适用于不同类型的存储引擎。

关系模型是一种以表格（表）为基础的数据模型，其中数据以行和列的形式组织，使用结构化查询语言（SQL）进行数据操作。关系模型适用于需要复杂的数据关联和查询操作的场景，具备强大的数据完整性和一致性。

常见的关系型数据库管理系统如MySQL、Oracle和SQL Server等，使用B+树、哈希索引等索引结构来支持高效的数据检索和查询操作。这些存储引擎适用于需要事务支持、复杂查询、数据一致性和强大的数据完整性约束的应用。

键值模型是一种简单的数据模型，其中数据以键值对的形式存储，每个键对应唯一的值。键值模型适用于需要高速读写和简单数据结构的场景，具备良好的可扩展性和性能。

键值存储引擎如Redis、Amazon DynamoDB和Riak等，使用哈希表或B树等索引结构来支持基于键的快速查找和存储。这些存储引擎适用于需要高性能、低延迟、大规模数据存储和缓存的应用。

总结起来，关系模型适用于需要复杂查询和数据关联的场景，具备强大的数据完整性和一致性，适合使用关系型数据库管理系统。键值模型适用于需要高速读写和简单数据结构的场景，具备良好的可扩展性和性能，适合使用键值存储引擎。选择存储引擎时应根据应用需求和数据模型的特点进行评估和选择。

SQL和NoSQL：

NoSQL和SQL是两种不同的数据库系统和查询语言，它们在数据模型、数据组织方式和查询方式上存在一些异同。

异同点如下：

1. 数据模型：SQL数据库系统采用关系模型，其中数据以表格（表）的形式组织，具有预定义的结构和关系。而NoSQL数据库系统采用多种数据模型，包括键值模型、文档模型、列族模型和图形模型等，以适应不同类型的数据存储需求。
2. 数据一致性：SQL数据库系统通常提供ACID（原子性、一致性、隔离性和持久性）事务，确保数据的一致性和完整性。而NoSQL数据库系统在一些情况下可能提供较弱的一致性保证，如最终一致性、事件ual consistency等，以换取更高的性能和可伸缩性。
3. 扩展性：NoSQL数据库系统通常具有较好的可伸缩性和分布式处理能力，可以方便地进行水平扩展，处理大规模数据和高并发访问。SQL数据库系统在扩展性方面相对较弱，需要采用复杂的分片和集群技术来实现扩展。
4. 查询语言：SQL数据库系统使用结构化查询语言（SQL）进行数据查询和操作，具有强大的查询能力和灵活性，支持复杂的关系查询和数据关联操作。而NoSQL数据库系统通常使用自定义的查询语言或API，查询和操作方式更加灵活，可以根据具体的数据模型进行优化。
5. 数据一致性和性能权衡：SQL数据库系统通常注重数据一致性和完整性，适用于需要强一致性和复杂查询的应用场景。而NoSQL数据库系统注重性能和可伸缩性，适用于需要高速读写和大规模数据存储的应用场景。
6. NoSQL事务性较弱：在一些NoSQL数据库系统中，对数据的事务处理能力相对较弱或受到一些限制。

事务与并发控制

数据分布

数据分布是指将数据在分布式系统中的多个节点之间进行分配和存储的过程。这样可以将数据存储在多个节点上，实现数据的冗余备份、负载均衡以及提高系统的可伸缩性和性能。

哈希分布

哈希分布使用数据的哈希函数来确定数据应该存储在哪个节点上。具体而言，对于每个数据项，通过对其键或标识符应用哈希函数，得到一个哈希值，然后根据哈希值的范围或散列结果，将数据分配给相应的节点存储。

哈希分布的优点：

1. 均衡的数据分布：哈希分布可以确保数据在整个分布式系统中均匀分布，避免数据倾斜和热点问题。
2. 高效的数据查找：由于哈希函数具有确定性，可以快速计算出数据在分布式系统中的存储位置，从而实现高效的数据查找和访问。

哈希分布限制：

1. 动态扩展困难：当分布式系统需要扩展节点数量时，由于哈希函数的改变，可能导致数据重新分布和迁移，对系统造成一定的影响。
2. 数据访问的局部性：哈希分布通常只考虑键的哈希值，而不考虑数据之间的关系。这可能导致一些相关的数据在不同节点上存储，增加了数据访问的开销。

一致性哈希

一致性哈希算是一种用于数据分布和负载均衡的算法，它解决了传统哈希算法在动态添加或删除节点时导致数据重新分布的问题。

如何实现负载均衡?

通过一致性哈希算法，节点的增加或删除只会影响到环上的一小部分数据，而不会导致整个数据的重新分布，从而实现了负载均衡。当新增节点时，新的节点会分担原有节点的负载。当节点失效或移除时，其负载会被其他节点接管，保持负载均衡。

顺序分布

顺序分布按照数据的顺序将其存储在分布式系统的节点上。具体而言，数据按照其顺序或范围进行排序，然后依次分配给节点进行存储。

顺序分布的优点包括：

1. 数据的局部性：顺序分布可以保持数据的顺序性和局部性，使得一些相关的数据存储在相邻的节点上，提高数据访问的效率。
2. 易于范围查询：由于数据按照顺序进行存储，范围查询（例如，按照时间范围查询数据）可以更加高效地执行。

然而，顺序分布也存在一些限制和注意事项：

1. 数据倾斜：如果数据分布不均匀或某些范围内的数据比其他范围内的数据更多，可能导致节点负载不均衡。
2. 动态数据插入和删除：在顺序分布中，动态插入和删除数据可能需要进行数据的重新排序和迁移，对系统造成一定的开销。

选择适当的数据分布策略需要考虑具体的应用需求、数据访问模式和系统规模等因素。哈希分布适合均匀分布和快速数据查找的场景，而顺序分布适合需要数据局部性和范围查询效率的场景。

负载均衡

子表合并拆分是指将数据分成多个子表，并将每个子表分配给不同的节点进行存储。通过将数据拆分成子表，可以将负载分散到多个节点上，避免某个节点负载过重。同时，合并拆分的过程可以根据负载情况进行动态调整，以实现负载均衡。

具体来说，子表合并拆分可以带来以下好处：

1. 负载均衡：通过将数据拆分成多个子表并分配给不同的节点，可以将负载分散到多个节点上，实现负载均衡。这样可以避免某个节点成为系统的瓶颈，提高整体系统的性能和可扩展性。
2. 动态调整：通过监控节点的负载情况，可以根据需要动态地将子表合并或拆分，以实现负载的动态均衡。当某个节点负载过重时，可以将其子表拆分到其他节点上，从而平衡负载。反之，

当某个节点负载较轻时，可以将其子表合并到其他节点上，充分利用节点资源。
3. 故障恢复：如果某个节点故障或失效，可以将其子表重新分配给其他可用节点，实现负载的重新分布和故障的快速恢复。

故障恢复和容错

检查点过程中的修改操作怎么保护?

在检查点过程中，修改操作的保护是通过一些技术和策略来实现的。以下是几种常见的方法：

1. 事务日志：事务日志是一种记录数据库操作的持久化日志。在检查点之前，系统会将所有的修改操作写入事务日志中，以确保数据的持久性。即使在检查点期间发生故障，系统可以通过回放事务日志来还原到检查点之前的状态。
2. 写前日志：写前日志是一种常见的日志记录策略。在执行修改操作之前，先将修改操作的日志记录写入日志文件。只有当日志记录成功写入后，才执行实际的修改操作。这样可以确保在检查点之前，所有的修改操作都已经被记录下来，即使发生故障，也可以通过日志进行恢复。
3. 快照隔离：快照隔离是一种并发控制策略，在检查点过程中可以保护修改操作。它通过创建数据库的一致性快照来实现隔离，使得修改操作不会影响在检查点之前已经读取的数据。这样可以确保检查点的一致性，并且允许其他事务在检查点过程中继续执行读操作。
4. 多版本并发控制：多版本并发控制是一种高并发环境下的并发控制策略。它使用版本号或时间戳来跟踪数据的不同版本，并且允许并发的读操作和写操作同时进行。在检查点过程中，多版本并发控制可以通过保留旧版本的数据来保护修改操作，以便在需要恢复时可以使用旧版本。

这些方法通常结合使用，以提供更强的修改操作保护。例如，在检查点过程中，系统可以先将修改操作写入事务日志和写前日志，然后使用快照隔离或多版本并发控制来保护读操作的一致性。如果发生故障，系统可以使用事务日志和写前日志进行恢复，并且通过快照隔离或多版本并发控制保证数据的一致性。

基于同步时钟的租约机制

基于同步时钟的租约机制是一种用于分布式系统中的协调和一致性保证的机制。它通过基于时钟的协调方式，使得系统中的节点能够按照统一的时间线进行操作和协作。下面是关于基于同步时钟的租约机制的介绍：

1. 租约（Lease）：租约是指一个节点对资源的临时访问权限。在基于同步时钟的租约机制中，节点可以通过申请租约来获得对某个资源的访问权限，并在租约的有效时间内持有该权限。
2. 时钟同步：为了实现基于同步时钟的租约机制，系统中的节点需要进行时钟同步，即保证节点的时钟在一定误差范围内保持一致。
3. 租约管理：在基于同步时钟的租约机制中，通常有一个中央控制节点或协调者负责租约的管理和分配。协调者维护一个租约表，记录每个节点拥有的租约信息，包括租约的持有者、资源和租约的到期时间等。
4. 租约续约：租约机制通常包括租约的续约机制。在租约即将到期时，持有者可以向协调者发送续约请求，以延长租约的有效时间。续约请求需要在时钟同步的前提下进行，以确保续约操作的一致性。
5. 租约过期和恢复：当租约到期或持有者发生故障时，租约将会过期或失效。协调者会在租约到期或失效后将资源的访问权限收回，并将资源重新分配给其他节点。这样可以保证资源在系统中的合理分配和利用。

基于同步时钟的租约机制可以用于解决分布式系统中的一致性问题和资源管理问题。通过租约的申请、续约和过期机制，可以确保节点在一定时间范围内拥有对资源的访问权限，并保证节点之间的操作和协作按照统一的时间线进行。这有助于实现分布式系统的协调性、可扩展性和容错性。

可扩展性

设计优化

GFS 的设计优化主要集中在不支持小文件的方面。GFS 针对大文件的存储和处理进行了优化，将小文件合并成更大的块进行存储。这样可以减少元数据的开销，并提高系统的吞吐量和性能。GFS 对小文件的处理效率较低，因为小文件的元数据管理和存储分配会导致额外的开销，并且小文件的读写操作也不利于并行处理。因此，GFS 的设计目标是针对大文件的高吞吐量读写，而不是优化小文件的处理。

HDFS 的设计优化则主要集中在不支持数据更新的方面。HDFS 是一种基于写一次、读多次的模型，一旦数据写入 HDFS 后，通常情况下是不允许对其进行修改的。这种设计优化带来了一些好处，例如简化了数据的一致性和并发控制，减少了数据复制和存储开销。同时，HDFS 的不支持数据更新也使得系统能够更好地适应大规模数据处理和分析任务，因为数据一旦写入后就不会发生变化，可以更好地支持数据的批量处理和并行计算。

垂直拆分和水平拆分

垂直拆分是将数据库按照数据表或数据列的维度进行拆分。这意味着将一个大型数据库中的表或列按照某种规则，分割成多个独立的数据库或数据库实例。每个数据库实例负责处理特定的数据表或列，从而减轻了单个数据库的负载压力，提高了数据库的性能和扩展能力。

水平拆分是将数据库中的数据按照某种规则分散到多个数据库节点或服务器上。每个节点或服务器负责处理其中的一部分数据。通过将数据水平分布到多个节点上，可以提高数据库的并发性和扩展性，以处理更大的数据量和更高的负载。

分布式协议

租约协议：租约协议是一种分布式系统中用于资源管理和并发控制的协议。它通过为资源分配一个租约（lease），确定资源的使用权和有效期。租约协议通常由租约持有者和租约请求者组成，租约持有者在租约有效期内拥有对资源的独占访问权。租约协议可以用于解决分布式锁、缓存一致性、容错和故障恢复等问题。

复制协议：复制协议是一种用于分布式系统中实现数据副本的协议。它通过在多个节点上复制数据副本，提高数据的可用性、容错性和读取性能。复制协议通常涉及数据的复制策略、一致性维护、数据同步和冲突解决等方面的机制。

一致性协议：一致性协议是用于在分布式系统中实现数据一致性的协议。它确保在分布式环境下的不同节点上对数据的操作能够达到一致的效果。一致性协议通常涉及数据的读写操作、数据复制和同步、冲突解决和并发控制等方面的机制。常见的一致性协议包括分布式事务协议（如两阶段提交和三阶段提交），Paxos协议和一致性哈希协议等。

分布式事务：在分布式系统中涉及多个独立节点的事务操作。

Paxos协议

Paxos 协议主要解决的问题是如何在一个拥有多个节点的分布式系统中，使得这些节点能够达成一致并决定一个值。Paxos 的核心思想是通过选举和多轮投票的方式，使得系统中的节点能够就某个提议达成共识。

下面是 Paxos 协议的基本流程：

1. 角色：
    - 提议者（Proposer）：负责提出提议，并将提议发送给其他节点。
    - 接受者（Acceptor）：接收来自提议者的提议，并根据一定规则进行投票。
    - 学习者（Learner）：负责接收和学习最终达成的共识值。
2. 提议流程：
    - 阶段 1（Prepare 阶段）：
        - 提议者向所有接受者发送一个准备请求（Prepare Request）。

        - 接受者收到请求后，检查提议者的编号（编号唯一且递增），如果编号大于自己已经回复的最大编号，则接受请求，并回复一个承诺（Promise）。
    - 阶段 2（Accept 阶段）：
        - 提议者收到多数接受者的承诺后，将自己的提议（包括编号和提议的值）发送给所有接受者。
        - 接受者收到提议后，检查提议者的编号，如果大于等于自己已经回复的最大编号，则接受提议，并回复一个接受（Accept）。
    - 阶段 3（Learn 阶段）：
        - 当一个提议者收到多数接受者的接受后，它可以确定自己的提议值已经被接受。
        - 提议者将最终达成的共识值广播给所有学习者，使得所有节点都能学习到最终的共识值。

问题

在 Paxos 协议中，节点出错可能导致协议无法终止，即无法达成最终的共识。这是因为在经典的 Paxos 协议中，节点需要等待多数接受者的回复才能进入下一轮投票，但如果有一个或多个节点出错或宕机，无法回复或无法参与投票，那么整个协议可能无法终止。

这种情况下，Paxos 协议可能会陷入僵局，无法进一步推进，导致无法达成共识。这对于确保可终止性是一个问题，因为在分布式系统中，我们希望协议能够在有限的时间内终止。

然而，需要注意的是，尽管 Paxos 协议在节点出错场景下可能无法保证可终止性，但只要协议能够终止，它仍然能够保证正确性和一致性。也就是说，如果 Paxos 协议能够成功达成共识并终止，那么最终的共识值将在系统中被广播和学习，保证了正确性和一致性。

分布式存储系统

NFS（网络文件系统）

NFS是一种基于网络的分布式文件系统，它允许不同计算机之间通过网络进行文件共享和访问。NFS最初由Sun Microsystems开发，并成为UNIX和类UNIX系统中广泛使用的标准文件系统之一。

1. 文件共享和访问：NFS允许多台计算机之间共享文件和目录，使得文件可以在不同主机之间进行读取和写入。这样，用户可以通过NFS访问和操作远程计算机上的文件，就像它们本地存在一样。
2. 客户端-服务器架构：NFS采用客户端-服务器（Client-Server）架构。文件的共享由一个或多个NFS服务器提供，而客户端则通过网络连接到这些服务器来访问共享的文件。客户端可以是任何支持NFS协议的计算机。
3. NFS协议：NFS协议定义了客户端和服务器之间进行通信和交互的规则。NFS协议使用远程过程调用（RPC）来实现客户端和服务器之间的通信，以便进行文件的读取、写入和管理操作。
4. 透明性：NFS提供了透明的文件访问，这意味着用户可以像本地文件一样访问远程文件，而无需关注底层的网络细节。用户可以使用标准的文件系统操作（如打开、读取、写入、关闭文件），而不需要了解文件的实际位置和网络传输的细节。
5. 安全性：NFS支持基于权限和身份验证的安全性控制机制，以确保只有授权的用户可以访问文件。可以使用访问控制列表（ACL）或基于用户和组的权限来管理文件的访问权限。
6. 性能和缓存：NFS通过使用客户端端的缓存机制来提高性能。客户端可以缓存远程文件的数据和元数据，以减少对服务器的频繁访问。这样可以提高读取和写入操作的效率，并减少网络传输的开销。

集群文件系统和网络文件系统

网络文件系统是一种基于网络的分布式文件系统，它允许多台计算机之间通过网络进行文件共享和访问。NFS的主要目标是提供透明的文件访问，使得远程文件可以像本地文件一样被访问和操作。NFS通常用于连接不同的计算机，允许用户在网络上远程访问文件。

而集群文件系统是一种专为集群环境设计的文件系统，它旨在支持多台计算机组成的集群共享文件和数据。集群文件系统通常在高性能计算（HPC）环境中使用，其中多台计算节点通过高速互连网络连接在一起。集群文件系统的设计目标是提供高性能、可靠性和可扩展性，以满足大规模计算和数据处理的需求。

虚拟的文件系统

NFS的设计目标是允许远程计算机通过网络访问和操作远程文件系统中的文件，而不需要直接管理本地磁盘上的数据。NFS客户端将远程文件系统映射到本地文件系统，使得远程文件对用户和应用程序来说就像本地文件一样可见和可访问。

NFS提供了透明的文件访问，用户和应用程序可以像访问本地文件一样访问远程文件，而无需了解实际文件存储在远程服务器上的细节。NFS隐藏了底层的网络通信和文件系统管理细节，提供了一个抽象层，使得远程文件对用户来说是透明的。

客户端与服务器

在NFS中，数据存储在NFS服务器上。NFS服务器是一个运行NFS服务的计算机，它负责管理和存储文件系统中的数据。NFS服务器通常具有本地磁盘或其他存储设备，上面保存着实际的文件数据。

当NFS客户端需要访问文件时，它会通过网络连接到NFS服务器，并发送相应的文件操作请求。NFS服务器会接收这些请求，并在本地磁盘上查找、读取或修改对应的文件数据。然后，NFS服务器将数据传输回客户端，客户端可以进行相应的操作，如读取、写入或执行其他文件系统操作。

要注意的是，NFS客户端并不直接管理文件数据，它通过NFS协议与NFS服务器进行通信，并由服务器来处理文件的实际存储和管理。客户端只负责发送请求和接收服务器的响应，以实现对远程文件系统的访问和操作。

挂载NFS服务器输出目录

当客户端通过NFS协议挂载远程文件系统时，实际上是在客户端的文件系统命名空间中创建了一个挂载点，该挂载点指向远程文件系统。挂载点成为客户端文件系统树的一部分，客户端可以通过它来访问远程文件系统中的文件和目录。

当客户端访问挂载点下的文件时，NFS客户端会通过网络协议与NFS服务器进行通信，并请求相应的文件数据。NFS服务器在收到请求后，将相应的文件数据传输给客户端。客户端在接收到数据后，可以在本地进行操作，如读取或写入。

需要强调的是，NFS并不会将整个文件系统或文件全部下载到客户端本地。根据需要，NFS客户端会在访问文件时通过网络从服务器获取相应的数据。这种按需获取数据的方式使得客户端可以透明地访问远程文件系统中的文件，而无需将所有数据复制到本地。

NFS v4文件锁

NFS v4文件锁是在NFS协议层实现的，它在客户端和服务器之间进行通信，协调并控制对共享文件的并发访问，确保在某个客户端正在对文件进行操作时，其他客户端无法同时修改文件的内容，从而保证数据的一致性。文件锁的管理和协调由NFS服务器负责，客户端在需要获取或释放锁时向服务器发送相应的请求。

HDFS（分布式文件系统）

HDFS是Apache Hadoop生态系统的一个分布式文件系统，提供了可靠的数据存储和处理能力，具有高容错性和可扩展性，适合于存储和处理大规模数据集。它具有以下特点：

1. 分布式存储：HDFS将大文件切分成多个数据块（block），并将这些数据块分散存储在Hadoop集群中的多个节点上。默认情况下，数据块大小为128MB（可配置），并且每个数据块都会在集群中多个节点之间进行复制，以提供数据冗余和容错能力。
2. 主从架构：HDFS采用主从架构，其中包含一个称为NameNode的主节点和多个称为DataNode的从节点。NameNode负责管理文件系统的命名空间、数据块的元数据和访问控制等，而DataNode负责实际存储数据块。
3. 冗余和容错：HDFS通过在多个DataNode之间复制数据块来提供冗余和容错能力。默认情况下，每个数据块会有3个副本保存在不同的节点上，以防止数据丢失。如果某个节点发生故障或数据损坏，HDFS能够自动恢复丢失的数据。
4. 流式数据访问：HDFS适用于大规模数据集的批量处理场景，其中数据一次写入，多次读取。HDFS的设计目标是提供高吞吐量而不是低延迟，因此适合用于批处理作业（如MapReduce）和数据仓库等应用。
5. 可扩展性：HDFS支持水平扩展，可以轻松地添加更多的节点来增加存储容量和处理能力。
6. 顺序读性能好随机读性能差：HDFS将大文件划分为固定大小的数据块，并将这些数据块分布在集群中的多个节点上。这种数据块的存储和分布方式使得顺序读取变得高效。当进行顺序读取时，可以通过按照数据块的顺序读取节点上的数据块来最大限度地减少磁盘寻址和网络传输的开销。相比之下，随机读取需要跳跃到不同的数据块和节点来获取所需的数据，这会导致额外的寻址和网络开销。

NameNode：

1. 元数据管理：NameNode负责管理HDFS文件系统的命名空间和元数据。它维护了整个文件系统的目录结构、文件和数据块的映射关系，以及文件的属性和访问权限等元数据信息。通过记录和跟踪这些元数据，NameNode使得文件系统能够有效地组织和访问大规模的数据。
2. 数据块管理：NameNode跟踪和管理HDFS中数据块的位置信息。它记录了每个文件的数据块在哪些数据节点上存储，以及数据块的副本数量等信息。通过这种方式，NameNode可以指导客户端进行数据的读取和写入操作，并确保数据的可靠性和高可用性。
3. 客户端交互：NameNode是客户端与HDFS之间的主要接入点。当客户端需要读取或写入文件时，它会首先与NameNode通信，获取文件的元数据信息，包括数据块的位置和副本数量等。**客户端通过与NameNode的交互，可以了解文件的存储位置，并与相应的数据节点进行数据的读取和写入操作。（安全性缺失）**
4. 故障恢复：NameNode负责监控HDFS集群中各个数据节点的状态。当一个数据节点发生故障或失效时，NameNode会感知到并采取相应的措施，例如重新复制丢失的数据块，保证数据的可靠性。此外，通过备份机制，NameNode的失效也可以通过冗余的备份节点进行快速恢复，以保证HDFS的可用性。
5. 负载均衡：NameNode负责监控和管理HDFS集群中的数据块分布情况。它会根据数据节点的负载情况和存储容量等信息，动态地调整数据块的分布，以实现负载均衡和数据的高效访问。

HDFS-MapReduce框架

计算与存储耦合

HDFS-MapReduce架构中每个节点即做存储又做计算，尽量使数据的存储和计算任务的执行发生在同一节点或同一机架上，以最大限度地提高计算效率和性能。这种紧密的耦合使得数据的处理变得高效，并且能够有效地处理大规模的数据集。

任务调度

在HDFS-MapReduce架构中，为了提高计算效率和性能，任务调度的原则是将计算任务尽可能地调度到离数据所在的位置更近的节点上执行，可以充分利用数据局部性。当任务调度到离数据所在的节点上执行时，可以避免数据的网络传输开销，减少数据的传输延迟。相比于将任务调度到远程节点上，这种局部性调度可以大大提高计算速度和效率。

Hadoop调度

slot

在MapReduce中，"slot"（槽）是指计算节点上可供任务执行的计算资源单元。每个计算节点都被分配了一定数量的槽，用于并行执行Map和Reduce任务。MapReduce框架根据数据的分布和可用的槽数来决定在哪个计算节点上执行任务。任务调度器会将任务片段分配给可用的槽，并在槽上启动并行执行。

数据局部性data locality是否越高越好

在大多数情况下，高数据局部性是更好的，因为它可以提供以下优势：

1. 减少网络传输开销：将任务调度到离数据所在的节点上执行可以避免数据的网络传输，减少数据在集群中的传输开销。这样可以减少网络带宽的占用和延迟，提高作业的计算速度和效率。
2. 提高计算速度：数据局部性可以保证在执行任务时，可以直接访问本地存储的数据，而不需要通过网络传输。这减少了数据访问的延迟，并且可以充分利用本地的计算资源，加快计算速度。
3. 节约资源：通过增加数据局部性，可以减少远程节点的访问和数据传输，从而节约了集群中的带宽资源和网络负载。此外，由于任务调度到离数据所在的节点上执行，可以更好地利用这些节点上的存储和计算资源。

然而，有时候过于追求数据局部性可能会带来一些负面影响：

1. 延长作业的等待时间：如果数据所在的节点尚未就绪，为了保持高数据局部性，可能需要等待数据就绪后再执行任务。这可能会导致作业的等待时间增加，特别是在数据加载较慢的情况下。
2. 不均衡的资源利用：如果过于强调数据局部性，可能导致节点上的资源不均衡利用。某些节点上可能会积累较多的任务，而其他节点上的资源却闲置。这可能降低了整体的资源利用率。

公平性与数据本地性的冲突

追求数据本地性可能导致任务在节点之间的分配不均衡，从而降低整体公平性；而追求公平性可能会导致任务在远程节点上执行，降低数据本地性。

数据不平衡：某些节点可能存储了更多的数据块，而其他节点上的数据较少。为了保持高数据本地性，可能需要将更多的任务调度到存储较多数据的节点上，导致任务分配不均衡，降低了公平性。

资源竞争：为了保持公平性，任务调度器可能会将任务分配给负载较低的节点，而不考虑数据的本地性。这可能导致任务在远程节点上执行，降低了数据本地性。

公平调度不足

1. 数据本地性降低：为了实现公平性，调度器可能将任务分配给负载较低的节点，而不考虑数据的本地性。这可能导致任务在远程节点上执行，增加了数据的网络传输开销，降低了数据本地性，从而影响了计算性能。
2. 资源利用率下降：公平调度通常会追求任务的均衡分配，但在实际情况下，节点之间的计算资源和数据分布可能是不均衡的。如果过于强调公平性，可能导致一些节点上的资源被闲置，而其他节点上的资源过度利用。这会降低集群的整体资源利用率，并可能导致作业的执行效率下降。
3. 长尾效应：公平调度可能导致长尾效应，即一些任务可能会等待较长时间才能得到执行。这是因为公平调度会优先考虑负载较低的节点，而任务数量较多的节点可能会出现排队等待的情况。这会增加作业的整体执行时间，并可能增加用户的等待时间。

延迟调度

延迟调度是一种调度策略，旨在提高作业的数据本地性和系统资源利用率。延迟调度的主要思想是，尽可能地将任务调度到与其输入数据所在节点相同的节点上执行，以减少数据的网络传输开销。该策略允许一定的延迟，即任务可能要等待一段时间才能在本地节点上执行，但可以最大程度地提高数据本地性。

最大跳过计数

最大跳过计数是指在调度过程中允许任务跳过的最大节点数。当一个任务需要执行时，调度器首先会尝试将其调度到与其输入数据所在节点相同的节点上执行，即实现最佳的数据本地性。然而，如果该节点上的资源不足或不可用，调度器会考虑将任务调度到其他节点上执行，但最大跳过计数限制了任务可以跳过的节点数量。

当最大跳过计数较大时，调度器可以更大程度地寻找资源可用的本地节点，以提高数据本地性。这意味着任务更有可能在距离其输入数据更近的节点上执行，减少了数据的网络传输开销，提高了数据本地性。

GFS（分布式文件系统）

GFS是谷歌开发的分布式文件系统，旨在提供高可靠性、高性能和可扩展性的存储解决方案。

一个chunk一个server

下面是GFS的详细架构说明：

1. 主要组件：
    - 主节点（Master）：GFS的主节点负责协调整个系统的操作，包括元数据管理、存储空间分配、数据块划分、副本管理等。主节点是单点故障，但它的元数据通常被持久化到本地磁盘和远程备份，以确保可靠性。
    - 块服务器（Chunk Server）：GFS系统中的存储单元是数据块（Chunk），每个块都由一个或多个块服务器存储。块服务器负责存储和管理数据块，处理读取和写入请求，并与主节点通信报告状态。
    - 客户端（Client）：客户端是应用程序访问GFS的接口，它与主节点通信以获取数据块的位置信息，并直接与块服务器进行数据的读取和写入操作。
2. 数据划分：
    - 文件划分为固定大小的数据块（通常为64MB），每个数据块都分配一个唯一的标识符。
    - 数据块以逻辑块位置（chunk handle）的形式在系统中进行标识，这个逻辑块位置映射到具体的块服务器上的物理存储位置。
3. 冗余备份：
    - GFS通过在多个块服务器上保持数据块的冗余副本来提供高可靠性。
    - 默认情况下，每个数据块在系统中保存三个副本，这些副本分布在不同的块服务器上，以提供容错能力和快速的数据访问。
4. 数据一致性：
    - GFS放宽了数据一致性的要求，将其重点放在可靠性和性能上。
    - GFS假设文件一般以追加方式写入，不支持文件的随机写入。
    - GFS提供了原子重命名操作，以支持并发写入和一致性的命名操作。
5. 元数据管理：
    - 主节点负责管理GFS的元数据，包括文件和数据块的映射关系、副本位置、块服务器状态等。
    - 元数据通常存储在主节点的内存中，并定期持久化到本地磁盘和远程备份。

6. I/O操作：
    - 客户端通过与主节点通信获取文件的元数据信息，包括文件的大小、块的位置等。
    - **客户端直接与块服务器进行数据的读取和写入操作，绕过了主节点的中心化瓶颈。**
7. 故障恢复：
    - GFS通过周期性心跳机制和数据块的状态报告来监测块服务器的健康状态。
    - 主节点通过这些信息来检测故障，并采取相应的措施，如重新复制丢失的数据块、迁移数据块等，以恢复系统的正常运行。

租约机制

租约机制的作用，是在写操作中(一个主块多个副本块)用以保持副本之间的数据的一致性(通过保证修改操作的顺序一致)。Master向其中一个数据块授予租约，我们称这个数据块为主块[primary]。主块为一连串的修改操作决定一个顺序，所有的副本都将按照此顺序执行修改操作。**GFS master(主人) 将权力下放给 chunk server，缓解了一定的压力。**

网络带宽限制可扩展性

网络带宽往往是限制ChunkServer扩展性的因素之一。这是因为在BigTable中，数据被分布在多个ChunkServer节点上，并通过网络进行数据的读写和传输。

当系统规模增大，数据量增多时，数据在ChunkServer之间的读写操作和数据传输会产生大量的网络流量。如果网络带宽有限，就可能成为系统的瓶颈，限制了整个系统的性能和扩展能力。特别是在大规模的分布式系统中，网络带宽的限制可能导致数据传输的延迟增加、吞吐量下降以及节点之间的数据同步和复制变慢。

Dynamo（分布式键值对存储）

Dynamo是亚马逊开发的一种高可用、分布式键值存储系统，用于构建高度可扩展的分布式应用程序。以下是Dynamo的相关知识叙述：

数据模型：Dynamo采用了简单的**键值对**数据模型，其中每个数据项由唯一的键标识。这种数据模型非常适合于快速存储和检索数据。

一致性模型：Dynamo采用了最终一致性的数据一致性模型。这意味着在进行写入操作后，系统将在一段时间内保证最终所有副本达到一致状态，但在写入操作期间可能存在不一致的状态。

分布式存储：Dynamo通过使用**一致性哈希**算法将数据分布到多个节点上。每个节点负责存储和处理一部分数据，使得负载在整个系统中均衡分布。同时，Dynamo使用了虚拟节点的概念，使得节点的加入和离开可以更加灵活和无缝地进行。

数据复制：Dynamo采用了主动复制的机制来确保数据的冗余和可用性。每个数据项通常会被复制到多个节点上，使得在节点故障或网络问题时仍然可以访问数据。Dynamo使用了一种称为版本向量时钟（version vector clock）的技术来处理并解决复制数据之间的冲突。

节点管理：Dynamo使用了基于Gossip协议的分布式节点管理机制。每个节点定期与其他节点交换信息，共享关于节点状态、拓扑信息和数据分布的信息。这种分布式的节点管理方式可以自适应地处理节点的加入、离开和故障恢复。

CAP理论：Dynamo在设计上更加注重可用性和分区容忍性，而对于一致性进行了一定的牺牲。这符合分布式系统中著名的CAP理论，即一致性（Consistency）、可用性（Availability）和分区容忍性（Partition Tolerance）无法同时满足，需要在这三个方面进行权衡。

一致性哈希

在传统的哈希函数中，数据项通过哈希函数的输出值来确定存储的节点。然而，传统哈希函数在节点的加入或离开时，会导致大量的数据项重新映射，导致数据迁移和负载不均衡。

一致性哈希解决了这个问题。它通过扩展哈希函数的输出范围，将哈希空间形成一个环状结构，将节点和数据项都映射到环上的位置。具体过程如下：

1. 哈希环：将哈希空间划分为一个环状结构，通常使用一个32位或64位的哈希空间。
2. 节点映射：将每个节点通过哈希函数映射到环上的一个位置。可以使用节点的唯一标识（如IP地址或节点名称）作为输入进行哈希计算。
3. 数据项映射：将每个数据项通过哈希函数映射到环上的一个位置。同样，可以使用数据项的键或标识作为输入进行哈希计算。
4. 数据定位：当需要存储或查找数据时，根据数据项的哈希值在环上顺时针找到最近的节点位置。该节点即为存储或处理该数据项的节点。
5. 节点动态变化：当有新的节点加入或节点离开时，并不会导致整个哈希环重新映射。只需要将受影响的数据项在环上顺时针找到下一个节点位置，并将其迁移到新的节点上。

通过一致性哈希，节点的加入或离开只会影响其附近的数据项，而不会影响整个系统的数据分布。这大大减少了数据的迁移量，提高了系统的可伸缩性和负载均衡能力。

虚拟节点

虚拟节点是一致性哈希算法中的一项技术，用于增加节点在哈希环上的分布密度，提高数据的均匀性和负载平衡。通过引入虚拟节点，可以在不改变物理节点数量的情况下，增加系统的可伸缩性和性能。

虚拟节点的实现方式是通过将每个物理节点映射到多个虚拟节点的位置上，使得每个物理节点在哈希环上出现多次。具体的过程如下：

1. 原始节点映射：首先，将每个物理节点通过哈希函数映射到哈希环上的一个位置，与传统一致性哈希算法一样。
2. 虚拟节点映射：为每个物理节点引入多个虚拟节点，将这些虚拟节点映射到哈希环上。虚拟节点的数量可以根据系统需求进行配置，通常是固定的倍数。
3. 数据定位：当需要存储或查找数据时，根据数据项的哈希值在哈希环上顺时针找到最近的节点位置。然后，从该节点开始，顺时针选择下一个虚拟节点作为存储或处理数据的节点。

Gossip协议

Gossip(八卦)协议是一种用于分布式系统中节点之间通信和信息传播的协议。它基于节点之间的随机化通信和信息交换，通过互相交换信息来达到一致性或共识的目标。在Gossip(八卦)协议中，每个节点通过随机选择其他节点进行通信，并交换自己持有的信息。这些信息可以是节点状态、数据副本、事件更新等。节点之间的通信是随机的，即节点在每个时间步中随机选择一些节点进行通信。

具体的Gossip(八卦)协议流程如下：

1. 选择对等节点：每个节点在每个时间步中从已知节点列表或邻居列表中随机选择一些节点作为对等节点。
2. 信息交换：节点之间进行信息交换，将自己持有的信息发送给对等节点，同时接收并处理对等节点发送的信息。
3. 信息更新：节点在接收到新的信息后，可能会更新自己的状态或数据副本，或者将信息传播给其他节点。
4. 重复过程：上述步骤不断重复，节点在每个时间步中选择对等节点并进行信息交换，以实现信息的传播和共识的达成。

向量时钟

向量时钟是一种用于在分布式系统中对事件发生顺序进行逻辑时钟标记的算法或数据结构。它用于跟踪和比较事件在不同节点上的发生顺序，以实现一致性和并发控制。

Merkle树

在Dynamo中，每个存储的数据项都被分配到多个节点进行复制，以提高系统的可用性和容错性。而Merkle树被用作验证分布式复制数据的完整性和一致性。

Dynamo的Merkle树是一种树形结构，其中每个非叶子节点都是其子节点哈希值的哈希。Merkle树的叶子节点存储着具体的数据项，而非叶子节点存储着子节点的哈希值。这样，通过比较不同节点的哈希值，可以验证数据的完整性和一致性。

当数据项在Dynamo中的节点之间进行复制时，每个节点都会维护自己存储数据项的Merkle树。当节点之间需要进行数据同步或验证时，它们会交换各自树的部分或完整的Merkle树，并比较树的哈希值。如果两个节点拥有相同的Merkle树哈希值，则可以确定它们存储的数据项是一致的，无需传输具体数据；如果哈希值不匹配，则可以确定数据存在差异。

通过使用Merkle树，Dynamo能够快速检测和传输仅发生变化的数据项，而无需传输整个数据集。这减少了网络带宽的使用，并提高了数据同步的效率。此外，Merkle树还可以用于识别数据副本之间的差异，并进行版本控制，以支持数据的冲突解决和一致性维护。

代理与直接

在分布式系统中选择服务节点的策略可以采用代理模式（Proxy Mode）或直接模式（Direct Mode），这两种模式有不同的特点和适用场景。

1. 代理模式（Proxy Mode）：
    - 在代理模式中，客户端与一个或多个代理节点进行通信，而代理节点负责将请求路由到实际的服务节点。
    - 代理节点可以根据负载情况、网络状况、服务节点的可用性等因素进行动态的负载均衡决策，以选择合适的服务节点。
    - 代理节点可以维护一份服务节点的列表，并根据一定的策略（如轮询、最小连接数等）选择服务节点。
    - 代理模式可以提供更好的灵活性和可扩展性，因为客户端只需与代理节点通信，而不需要了解具体的服务节点信息，代理节点可以动态地调整节点列表以适应系统的变化。
2. 直接模式（Direct Mode）：
    - 在直接模式中，客户端直接与实际的服务节点进行通信，不经过代理节点。
    - 客户端需要了解服务节点的地址和状态信息，并根据一定的策略选择合适的服务节点进行请求。
    - 直接模式可以减少请求的转发延迟，因为客户端直接与服务节点通信，没有额外的中间节点。
    - 直接模式适用于具有固定服务节点列表、较小规模或简单的分布式系统，其中负载均衡和故障恢复等功能可以由客户端自行实现。

可扩展性

Dynamo是一种具有良好可扩展性的分布式存储系统，它通过哈希环和一致性哈希、虚拟节点、数据分片和副本复制、自动负载均衡，实现了良好的可扩展性。它可以根据需求增加节点数量来扩展存储容量和吞吐量，而不需要全局的数据迁移或系统停机。这使得Dynamo能够适应不断增长的数据量和流量，提供可靠的分布式存储服务。

BigTable（分布式表格系统）

BigTable是Google开发的一种分布式、面向列的大规模结构化数据存储系统。它用于存储和管理海量数据，并提供高可用性、高性能和可扩展性。

租约机制选主节点+主从结构+子表+LSM

chubby协调，通过租约选主节点（没有上层主节点）

行列结构

BigTable采用了面向列的数据模型，数据以行键（Row Key）、列族（Column Family）、列限定符（Column Qualifier）和时间戳（Timestamp）的形式进行组织。每个行键可以包含多个列族，每个列族下可以有多个列限定符，每个列限定符对应一个时间戳和相应的数据值。

在BigTable中，数据以行（Row）为单位进行组织和存储。每行数据由一个唯一的行键（Row Key）标识。行键在BigTable中被用作数据的主要索引，可以用来快速检索和访问数据。

BigTable中的数据按照列族（Column Family）进行组织。列族是一组相关列的集合，通常包含具有相似特征和语义的数据。例如，在一个电子邮件应用中，可以将邮件的元数据（如发件人、收件人、主题等）存储在一个列族中。列族在BigTable中被视为数据模式的一部分，并具有相同的前缀。

每个列族下可以包含多个列限定符（Column Qualifier）。列限定符用于进一步标识和区分数据。列限定符是列族内部的唯一标识符，它们通常表示具体的属性或字段。例如，在邮件应用的列族“metadata”中，可以有列限定符“sender”和“subject”来表示邮件的发件人和主题。

除了列限定符，每个单元格还包含一个时间戳（Timestamp）。时间戳用于标识数据的版本，允许在同一个行键、列族和列限定符下存储多个时间点的数据。这对于维护历史数据记录和支持版本控制非常有用。

系统架构

1. Master节点：Master节点是BigTable系统的控制节点，负责整个系统的元数据管理和协调工作。它维护全局的表格模式（Schema）和表格位置信息（Tablet Location），并负责处理客户端的元数据操作请求。Master节点还负责监控和管理ChunkServer节点的状态，以及负载均衡和故障恢复等任务。
2. ChunkServer节点：ChunkServer节点是实际存储和处理数据的节点。每个ChunkServer负责管理一部分数据，并提供对这些数据的读写操作。ChunkServer节点之间相互独立，它们存储的数据被分成多个分片（Tablet），每个分片存储一部分数据。ChunkServer节点还负责数据的复制和恢复，以提供数据的容错性和可用性。
3. 分布式文件系统：BigTable使用分布式文件系统来存储数据。文件系统将数据划分为多个块（Block），并将这些块分布在多个ChunkServer节点上。分布式文件系统提供了高可靠性和高吞吐量的数据存储和访问，确保数据的持久性和可扩展性。
4. 分布式协调服务（Distributed Coordination Service）：BigTable依赖分布式协调服务来协调和管理系统中的各个组件。协调服务负责分布式锁管理、配置信息的发布和订阅、节点的健康状态检测等功能，确保系统的一致性和可靠性。常用的分布式协调服务包括Chubby或ZooKeeper。

chubby锁服务

分布式锁，是控制分布式系统之间同步访问共享资源的一种方式。Chubby是一种面向松耦合的分布式系统的锁服务，通常用于为一个由适度规模的大量小型计算机构成的的松耦合的分布式系统提供高可用的分布式锁服务。锁服务的目的是允许它的客户端进程同步彼此的操作，并对当前所处环境的基本状态信息达成一致。因此，Chubby的主要设计目标是为一个由适度大规模的客户端进程组成的分布式场景提供高可用的锁服务，以及易于理解的API接口定义。

推荐的部署模式是五台部署，因为数据一致性需要通过一致性的算法来进行确定，采用的Quorum机制（即需要获得绝大多数的从节点同意才算数据提交成功），所以奇数台机器可以快速的有效的进行判读数据是否提交成功。

在运行过程中，只有master节点才能够进行数据的修改创建操作，操作完成的数据通过一致性算法同步到follower节点。客户端在连接到服务端的过程中，如果连接的是master服务器则接着进行数据交互，如果连接的是从节点，则从节点会返回master服务器的信息，客户端直连master节点，从而为后面提交修改数据，并且可以获取订阅服务的修改与变更。如果在运行过程中，master节点崩溃，从节点会在租期到期之后进行重新选举，重新选择出主节点，从而客户端会重新连接到新的master节点继续运行。

SSTable

SSTable是BigTable中存储数据的文件格式，用于持久化数据到磁盘。SSTable将键值对按照键的顺序进行排序，并以一种紧凑的文件格式进行存储。SSTable文件通常具有较大的容量，因此可以存储大量的键值对。

当MemTable中的数据达到一定大小阈值或时间间隔时，会将其刷写到磁盘上生成一个新的SSTable文件。在刷写过程中，MemTable中的数据会按照键的顺序写入SSTable文件。虽然MemTable中的数据本身已经是有序的，但在刷写到SSTable文件时，需要确保整个SSTable文件中的键值对按照键的顺序排列，以满足SSTable的有序性要求。

在合并过程中，需要将多个SSTable文件中的键值对进行排序合并，以生成新的有序的SSTable文件。