1 - MapReduce

1.1 MapReduce 过程

1. Map 映射
- map 函数将对每个元素根据特定标准映射，输出键值对
2. Shuffle (Group) 排序分组
- 将所有键值对进行排序，并根据键值对中的键(key)进行分组，具有相同键的值会聚集在一起
3. Reduce 归约
- reduce 函数对每个键进值进行操作，将它们合并为更小的集合，reduce 函数会收到(word, [1,2,3,...])形式的键值对，将值列表合并成单一的数字，例如全部相加。

1.2 Partitioner 分区器

- 负责确定在 shuffle 阶段键值对应该发送给哪个 reducer
- 一般情况下，分区器使用键的哈希值对 reduce 任务的数量取模决定 reducer，用户可以自定义
- 需要确保相同的键被发送到同一个 reducer

1.3 Combiner 组合器

- 在 map 阶段将中间键值对合并，减少发送到 reducer 的键值对数量，提高效率
- 可以被看作是一个安放在 map 阶段的 reducer
- Combiner 的输出键值对必须和输入键值对有**相同的结构（键值类型）**

1.4 数据传输过程

1. Mapper：输入数据存储在分布式文件系统中。mapper 从**本地磁盘**读取数据，处理他们，在**内存**中生成键值对并缓存。如果键值对过多，超出了可用的内存时，会被排序后写入**本地磁盘**。Mapper 的输出会被**写入本地磁盘**
2. Combiner：从**本地磁盘**读取 mapper 的结果，处理数据的过程在**内存**中进行，输出会被写入到**本地磁盘**
3. Partitioner：发生在 mapper 节点上，从**本地磁盘**读取 mapper 或 combiner 的输出数据，在**内存**中处理数据，输出保存在**本地磁盘**
4. Shuffle：reducer 节点会从 mapper 节点的**本地磁盘**中拉取自己需要的数据，通过网络发送。这些数据被保存在**内存**中，如果数据超出内存空间，则剩余的部分保存在**本地磁盘**
5. Reducer：如果未完成排序和合并，则这些操作会在**本地磁盘**进行。在内存中应用 reduce 函数生成输出。最后 reducer 的输出会被写回分布式文件系统中

1.5 Secondary Sort

在 MapReduce 框架中，数据是以键值对(key-value pairs)的形式进行处理的。在这个过程中，Map 阶段生成的键值对会被分组(grouped)和排序(sorted)，然后传递给 Reduce 阶段。默认情况下，MapReduce 只会按键（key）进行排序，这称为“自然排序”。

如果我们需要以一种额外的方式对值进行排序，则这被称作为二次排序"secondary sort":

- 定义一个新的“复合键”（composite key），格式为 (K1, K2)，其中 K1 是原始键（也称为“自然键”），K2 是我们希望用于排序的变量。在这种情况下，复合键将会影响如何对数据进行分区和排序：

Partitioner: 需要自定义分区器，使其仅按 K1 进行分区，而不是按复合键(K1, K2)分区。这样可以保证相同的 K1 会被发送到同一个 reducer，但是在 reducer 内部，数据会根据 K2 的值进行排序。

这样，每个 reducer 接收的数据就会首先根据 K1 分组，然后在每个组内根据 K2 排序，实现了二次排序的目的

2 - NoSQL

2.1 Broadcast (Map) Join

1. **选择较小的数据集**：在两个需要连接的数据集中，选择较小的那个进行广播。这意味着这个数据集会被发送到集群中的每个节点，以便于本地连接操作。
2. **广播到所有节点**：所选择的较小数据集会被复制并广播到集群中的每个节点。在 Spark 中，这通常通过 `broadcast` 函数实现。
3. **本地连接**：大数据集不动，较小的数据集被广播到每个节点后，大数据集的每个分区会与本地节点上的小数据集进行连接操作。由于小数据集已经在每个节点的内存中，这大大减少了网络传输的需求，并且可以并行地在每个节点上快速完成连接。
4. **减少数据洗牌**：使用 broadcast join 可以避免在网络中进行昂贵的数据洗牌操作，因为只有小数据集在节点间移动，而大数据集则保持静止。

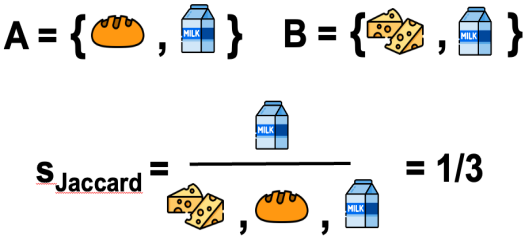
2.2 Reduce-side (Common) Join

Reduce side join 是在 MapReduce 编程模型中用于处理大规模数据集连接的一种机制。这种类型的 join 操作涉及到 Map 和 Reduce 两个阶段，在这个过程中，两个数据集都参与到数据的洗牌和排序过程中。这种 join 在处理两个大数据集的连接时尤其有用，因为它**不需要将整个数据集加载到内存中**。

以下是 Reduce side join 的基本步骤：

1. **Map 阶段：**
- 两个数据集（比如数据集 A 和数据集 B）都会被读取，并在 Map 阶段被处理。
- Map 函数会对每条记录产生中间的键值对（key-value pairs）。键通常是两个数据集中用于连接的共同字段，而值则是包含原始数据和数据来源标识（例如来自数据集 A 或 B）的记录。
2. **Shuffle 阶段：**
- Shuffle 过程将 Map 输出的中间键值对根据键进行排序和分组。所有共同键的值都会被集中到一起。
3. **Reduce 阶段：**
- 在 Reduce 阶段，每个 Reduce 任务接收到了包含相同键的所有值的列表。这个列表中会同时包含来自数据集 A 和 B 的记录。
- Reduce 函数然后遍历这些值，根据键值对中的标识将来自数据集 A 和 B 的记录分开，并执行连接操作。
- 连接后的结果会被写出到最终的输出文件中。

Reduce side join 缺点: 它需要大量的数据在网络中移动（即数据洗牌），这可能会导致较高的网络传输开销，并且如果连接的键有很多重复值，也可能会在 Reduce 阶段产生瓶颈。



2.3 Similarity Check

相似度量：距离越小 = 相似度越高

2.3.1 Jaccard Similarity

- Jaccard Similarity

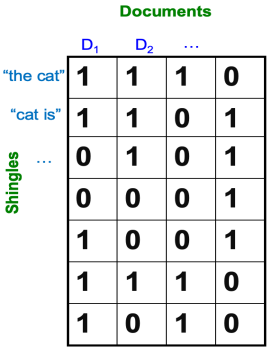
$$S_{Jaccard}(A,B) = \frac{|A \cap B|}{|A \cup B|} \qquad (1)$$

- Jaccard Distance

$$d_{Jaccard}(A,B) = 1 - s_{Jaccard}(A,B) \qquad (2)$$

2.4 Similarity Document Check

Step 1: Shingling



分词：这一步将文档转换为一组短语（称为“shingles”或“k-grams”）。每个 shingle 通常是文档中连续的 k 个项（可以是字、词或字符）。例如，对于句子“The quick brown fox jumps over the lazy dog”，如果我们使用 2-grams（bigrams）作为 shingles，那么一个可能的 shingle 集合包括{"The quick", "quick brown", "brown fox", ...}。这一步的目的是创建文档的特征集，以便于比较。

对于两个文档，我们使用一个矩阵将两个文档D_1, D_2中的分词表示出来，1 代表存在与文档中，0 则没有

Step 2: Min-Hashing

最小哈希：这一步的目的是将上一步得到的 shingle 集合转换为文档的“签名”（signature），这些签名在压缩数据的同时保留了文档间的相似性信息。签名是一个较短的数据块，它代表了文档内容的摘要。Min-hashing 算法通过对每个文档的 shingle 集合使用哈希函数，将其转换为一个较短的哈希值序列（即签名），而且这一转换过程保留了原始 shingle 集合间的相似度结构。具有相同或相似签名的文档被认为是“候选对”（candidate pairs），这意味着它们很可能是近似重复的文档。

NoSQL Pros and Cons

相对于传统关系型数据库，NoSQL 数据库的**优点**包括：

- **可扩展性**：更容易扩展到多个服务器。
- **灵活性**：可以适应多变的数据模型和不断变化的数据类型。
- **高性能**：特别是在处理大量数据和高并发请求时。

NoSQL 数据库的**缺点**可能包括：

- **一致性**：可能牺牲事务的严格一致性来获取性能和可扩展性。
- **复杂的数据关联**：对于需要复杂关联的数据，关系型数据库可能更加适合。

2.5 BASE and ACID

2.5.1 ACID

- Relational DBMS provide stronger (ACID) guarantees
- **ACID** 是传统关系型数据库的设计理念，它强调的是数据操作的可靠性和一致性：

1. **原子性（Atomicity）**：事务中的所有操作都是一个不可分割的工作单位，要么全部完成，要么全部不做。
2. **一致性（Consistency）**：事务执行结果必须使数据库从一个一致性状态转变到另一个一致性状态。
3. **隔离性（Isolation）**：并发执行的事务之间不会互相影响。
4. **持久性（Durability）**：一旦事务提交，其所做的修改将永久保存在数据库中。

2.5.2 BASE

In many NoSQL system provide weaker "BASE" approach

BASE 则是许多 NoSQL 数据库系统遵循的理念，它更强调系统的可用性和容错性：

1. **基本可用（Basically Available）**：系统保证可用性，但可能因为响应时间的延迟或系统功能的减少而不是完全可用。
2. **软状态（Soft state）**：系统的状态可能会随时间而改变，即使没有输入，系统状态仍然有可能变化（例如，由于数据复制而导致的状态变化）。
3. **最终一致性（Eventual consistency）**：系统保证，如果没有新的更新操作，数据最终将达到一致状态。

Pros：

1. **高可用性**：
   - NoSQL 数据库通常可以在部分系统故障时继续工作，它们避免了单点故障，提供了更高的可用性。
2. **弹性扩展**：
   - NoSQL 数据库设计之初就考虑到了水平扩展，它们可以通过添加更多的服务器来处理更多的数据和负载，而不需要昂贵的单体服务器。
3. **灵活性**：
   - 无模式或者灵活模式的数据存储，使得 NoSQL 数据库可以轻松应对结构变化和不同类型的数据。
4. **性能**：
   - 在某些操作上，尤其是那些不需要复杂事务支持的操作上，NoSQL 数据库可以提供更好的性能。

Cons：

1. **一致性问题**：
   - 最终一致性模型意味着在数据同步过程中可能存在不一致性的时间窗口，这可能不适合对实时一致性要求很高的应用。
2. **复杂性**：
   - 开发者可能需要在应用程序层面处理一致性问题，这可能增加应用程序逻辑的复杂性。
3. **无事务性**：
   - 传统的事务特性（如 ACID）在很多 NoSQL 数据库中是不支持的，或者只有部分支持，这对于需要强事务性的系统来说是一个限制。
4. **数据冗余**：
   - 为了提供高可用性和性能，NoSQL 数据库可能会存储数据副本，这可能导致数据存储的冗余

2.6 NoSQL Types

2.6.1 键值存储（Key-Value Stores）：

最简单的 NoSQL 数据库，以键值对的形式存储数据。

键值存储的一些特点包括：

1. **无模式**：键值存储通常不具备固定的模式或结构，数据可以以任何形式存储为值，如字符串、JSON、BLOB 等。
2. **无关联**：它们不提供原生的方式来直接关联不同的键值对或模仿 SQL 中的表间连接。关系必须由应用逻辑来管理。
3. **单一的数据集合**：虽然某些键值存储系统可能允许你创建类似于“表”的不同命名空间或数据集合，但这些通常不提供连接功能。

1. **自定义索引**：在键值存储中，创建复杂索引需要应用层面的设计，比如通过维护一个特殊的键，它的值包含了需要被索引的数据项的键的列表。

键值存储的优势：

1. **性能**：键值存储提供非常快速的读写能力，因为它们通过键直接访问值，通常这些操作可以在 O(1)时间内完成。
2. **可扩展性**：键值存储通常设计为易于水平扩展，能够处理更多的负载通过简单地增加更多的节点。
3. **简单性**：由于其简单的数据模型，键值存储通常更易于设置和维护。
4. **灵活性**：键值存储不需要预定义的模式，所以你可以随意存储不同结构的数据。
   - 例子：Redis, Amazon DynamoDB, Riak。

2.6.2 文档存储（Document Stores）：

以下是文档存储 NoSQL 数据库的一些关键特点：

1. **灵活的数据模型**：文档可以包含嵌套的数据结构，如数组和子文档。由于没有固定的模式，文档的结构可以动态更改。
2. **自描述性**：文档存储通常是自描述的，意味着数据结构描述包含在文档本身中，这使得数据的解析和理解变得直观。
3. **查询能力**：大多数文档数据库提供了强大的查询语言，允许用户执行复杂的搜索、聚合和过滤操作。
4. **索引**：为了提高查询性能，文档数据库支持在一个或多个文档的属性上建立索引。
5. **扩展性**：文档数据库也设计为易于水平扩展，允许通过增加更多的服务器来增加数据库的容量和吞吐量。
6. **API 接口**：文档数据库通常提供丰富的 API 用于交互，这些 API 可以是 RESTful 的，也可以是数据库专有的查询语言。

- 存储半结构化数据的文档，通常是 JSON 或 XML 格式。
- 文档数据库的一个主要优势在于其灵活性。它们允许开发者在不需要预先定义表结构的情况下存储和查询数据，这对于快速开发和迭代、以及处理非结构化或半结构化数据非常有利。
- 然而，文档数据库也有其局限性，如它们可能不支持像传统 SQL 数据库那样复杂的事务管理，而且当涉及到多个文档或集合时，维护数据一致性可能会更加复杂。

例子：MongoDB, CouchDB, Firestore。

2.6.3 宽列存储（Wide-Column Stores）：

以列族为中心存储数据，允许存储大量数据。

以下是宽列存储的一些核心特点：

1. **列族（Column Families）：**

- 数据被存储在列族中，每个列族是一个容器，存储着相关的列。
- 列族内的列可以在每一行中不同，允许每行有不同的列数和类型，这带来了极大的灵活性。

2. **行键（Row Keys）：**

- 每一行由一个唯一的行键（Row Key）标识，可以用来快速访问和检索数据。

3. **动态列：**

- 每行可以有数千甚至数百万列，列可以在运行时动态地增加到任何行中，不需要预先定义模式。

4. **可扩展性：**

- 宽列存储设计用于水平扩展，可以通过增加更多的服务器节点来提高容量和吞吐量。

5. **优化读/写性能：**

- 通过将相关数据存储在相同的列族中，宽列存储可以优化数据的读取和写入性能。
  - **分布式架构：**
    - 它们通常自带分布式架构，能够处理大规模数据分布在多个物理位置。
    - 例子：Apache Cassandra, HBase, Google Bigtable。

2.6.4 图形数据库（Graph Databases）：

使用图结构存储实体以及实体之间的关系，适合复杂的关系数据。

核心概念：

1. **节点（Nodes）：**

- 节点代表实体，如人、业务、账户、计算机等。
- 每个节点可以有一个或多个标签（Labels）来表示不同的类别或类型。
- 节点可以包含多个属性（键值对），用以存储关于实体的信息。

2. **边（Edges）：**

- 边代表节点之间的关系。
- 每条边都有一个类型，表明连接的节点之间的关系性质，如“朋友”、“属于”或“访问”。
- 边也可以有属性，提供有关关系的更多信息，如权重、成本、距离等。

1. **属性（Properties）：**
   - 节点和边都可以有属性，这些属性以键值对的形式存在。
   - 属性为图数据添加了丰富的语义。
2. **索引（Indexes）：**
   - 图形数据库通常支持通过索引来加速对节点和边的查询。

图形数据库的特点：

1. **关系优先**：图形数据库将关系作为一等公民，这与其他数据库系统不同，在那里关系通常是通过外键或特殊的索引来表示的。
2. **性能**：对于深度连接查询和复杂的关系网络，图形数据库可以提供卓越的性能。
3. **灵活性**：图结构的自然灵活性使得添加新的关系和节点不需要更改现有的数据模式。
4. **直观性**：图形数据库的结构使得数据模型和现实世界的网络直观对应，方便理解和查询。

- 例子：Neo4j, JanusGraph, Amazon Neptune。

2.6.5 矢量数据库（Vector Databases）

- 矢量数据库（Vector Databases）是专门设计来存储和查询矢量空间数据的数据库系统。在这个上下文中，“矢量”通常指的是多维的数值数组，它们代表了数据点在特定的特征空间中的位置。这种类型的数据库在处理大规模机器学习和人工智能任务中尤为重要，尤其是在执行相似性搜索时。

核心概念：

- **特征向量（Feature Vectors）：**

在机器学习和搜索领域，数据项经常被转换成特征向量，这些特征向量表示了数据项的特性或属性。

- **相似性搜索（Similarity Search）：**

矢量数据库的主要功能之一是快速找到与给定查询向量相似的向量。相似性度量通常使用余弦相似度、欧几里得距离等方法。

- **索引和优化：**

为了高效地进行相似性搜索，矢量数据库使用多种索引和优化技术，如树结构、哈希技术或分区策略。

矢量数据库的特点：

1. **高效的搜索性能：**
   - 矢量数据库能够在高维空间中快速执行 k 最近邻（k-NN）搜索，这对于实时推荐系统、图像或视频检索等是至关重要的。
2. **大规模数据处理：**
   - 它们可以处理数以亿计的向量，并且在这样的规模上仍能保持查询的响应时间。
3. **机器学习集成：**
   - 矢量数据库经常与机器学习模型和流程紧密集成，以便直接利用模型生成的特征向量。

2.7 Consistency

Strong consistency

- 任何数据的更新操作完成后，后续的任何读取操作都将**立即看到这个更新**。换句话说，系统确保所有节点上的数据在**任何时间点都是一致的**。
- 这通常意味着系统需要在更新数据时进行一定的协调，以确保所有的复制节点都同步更新，这可能会导致写操作延迟增加。
- 强一致性模型适合对数据一致性要求极高的场景，如金融交易系统。

Eventual consistency

- 数据的更新**不需要立即反映到所有节点上**。系统只保证如果没有新的更新发生，那么最终所有的复制节点将会达到一个一致的状态。
- 这意味着在达到一致性状态之前，不同的节点可**能会看到不同版本的数据**，从而允许系统在某个时间点上存在**数据不一致**的情况。
- 最终一致性模型提供了**更高的可用性和分区容错性，但牺牲了实时一致性保证**。
- 这种模型适合对可用性要求高，但可以容忍短时间内数据不一致的应用，如社交网络中的时间线更新。

2.8 Duplication (Denormalisation)

- 去规范化（Denormalization）是数据库优化的一个过程，特别是在关系型数据库的上下文中。高度规范化可能导致性能问题，因为复杂的查询可能需要多个表之间的连接操作，这在大型数据库中可能会非常耗时。去规范化涉及减少数据库的规范化级别，通常通过合并表格、添加冗余数据或组合字段来实现。其主要目的是提高数据库的查询性能，尤其是在大数据量和复杂查询的情况下。
- 去规范化的策略包括：

1. **添加冗余列：**在一个表中包含来自另一个表的数据，以避免连接操作。
2. **合并表：**将多个相关的表合并为一个表，以减少查询中的连接数量。
3. **预计算聚合：**存储计算结果（如总和、平均值等）而不是在每次查询时都重新计算。
4. **创建冗余索引：**创建额外的索引来加速查询，即使这些索引会占用更多的存储空间。

- 去规范化的缺点是可能导致数据更新、插入和删除操作的复杂性增加，因为需要维护额外的冗余数据的一致性。此外，它也增加了存储需求，因为相同的数据会在多个地方存储副本。

2.9 Data Partitioning

2.9.1 Table Partitioning

- 表分区（Table Partitioning）把一个大表被分解为多个更小、更易于管理的逻辑分区，但在逻辑上仍然作为单个表对外呈现。
- 每个分区可以存储在不同的物理位置，且可以单独优化和维护。
- 表分区通常用于提高查询性能、优化数据加载、提高数据维护效率以及改善备份恢复操作的速度。
- 通过表分区，一张表被横向分割成多个小表，每个小表都有和大表相同的列数量，但是行数量被分割

2.9.2 Horizontal Partitioning

- 水平分区（Horizontal Partitioning）也称作分片（Sharding），以及如何选择合适的分区键（Partition Key）或分片键（Shard Key）。

1. **不同的元组存储在不同的节点：** 这意味着在一个分布式数据库系统中，表中的每一行（或称作元组）根据某种规则，被分散存储在不同的数据库节点上。这些节点可以是同一个数据中心内的不同服务器，也可以是分布在不同地理位置的服务器。
2. **Partition Key 分区键**（或 shard key 分片键）： 分区键是用来决定每个元组存储位置的变量。根据分区键的值，数据库管理系统将元组分配到不同的节点上。拥有相同分区键值的元组会被存储在相同的节点上。
3. **如何选择分区键：** 选择分区键是一个重要的决策，因为它会直接影响查询的效率和系统的扩展性。理想的分区键应该满足以下条件：
   - **查询过滤：**如果某个列经常被用作查询条件（WHERE 子句），那么这个列可能是一个好的分区键。
   - **分组统计：**如果经常需要按某个列进行分组（GROUP BY 子句）进行聚合运算，那么这个列也可能是一个好的分区键。
   - **负载均衡：**分区键应该能够确保数据和负载在各个节点间均匀分布，避免某个节点数据量过大或查询负载过高。

- 水平分区将表中不同的行按照**分区键**的不同分割到不同的节点中。例如我们使用 city 为分区键，则 city 为"New York"的所有行都被分到节点 A，city 为"Singapore"的所有行被分到节点 B
- 如果分区之后的查询操作经常以城市做查询，则这个分区键是**高效率的**
- 如果每个城市的行数相差特别大，则节点的存储会**失衡**
- 如果城市的数量很少，我们称之为 **low cardinality 低基数**

2.9.3 Range Partitioning

- 范围分区是通过确定**键值的范围**来实现的。数据库系统根据预设的键值范围，把数据分散到不同的分区。例如，user_id 在 1 到 100 的用户记录可能存储在分区 1，而 user_id 在 101 到 200 的记录存储在分区 2。

- 如果经常需要执行基于范围的查询，例如查询 user_id 小于 50 的所有用户，那么范围分区非常有用。在这种情况下，查询时可以跳过不包含相关数据的分区（如上例中的分区 2），这种方法称为“分区裁剪”（Partition Pruning），它可以显著节省查询处理的工作量。
- **可能导致分区不平衡：** 范围分区可能会导致数据分布不均衡。例如，如果大量行的 user_id 都是 0，那么这些行都会被存储在同一个分区中，这会导致该分区数据过多，而其他分区数据不足。
- **自动的范围划分：** 通常，分布式数据库系统会有一个“平衡器”（Balancer）功能，自动调整分区范围，试图保持各个分区的数据量平衡。这意味着系统会监控数据的分布情况，并在必要时重新划分分区范围，以保持分区之间的均衡。

2.9.4 Hash Partitioning

- 哈希分区根据特定的键值的哈希函数输出来确定其分到哪个节点
- **如何进行分区：**
  每个节点在这个圆环上有一个“标记”（通常可以想象为一个矩形或点），代表其在哈希空间上的位置。
  每个元组（数据项）根据其哈希值被放置到圆环上的某个位置，然后分配给顺时针方向上的第一个节点标记。
- **删除节点：**
  当需要删除一个节点时，圆环上的这个节点标记被移除，原本分配给这个节点的所有元组会被重新分配给顺时针方向上的下一个节点。
- **添加节点：**
  相似地，添加一个新节点时，在圆环上为其增加一个新的标记，并将现在应该属于这个新节点的元组重新分配给它。
- **简单复制策略：**
  可以通过在顺时针方向上的几个额外节点中复制元组来实现元组的简单复制，以增加数据的可用性和耐久性。
- **多重标记：**
  每个节点可以在圆环上拥有多个标记。对于每个元组，依然是分配给顺时针方向上最近的标记。
  这样做的好处是，当删除一个节点时，其元组不会全部重新分配给同一个节点，这有助于更好地平衡负载。
- **优势：**
  哈希分区在应对根据被哈希的键查询一个特定数据项时，可以直接计算出该项被存储在哪个节点中。负载均衡，水平扩展很容易
- **劣势：**
  如果某些键非常频繁，则可能会造成数据倾斜；可能的哈希碰撞造成某个节点存储了过多的数据；顺序访问困难；减少分区数量需要大量计算

2.10 MongoDB

2.10.1 Routers

- 路由器在 MongoDB 中通常指的是 `mongos` 实例。`mongos` 的作用是作为前端服务，接受客户端的数据库操作请求，并将这些请求路由到正确的数据分片上。
- 客户端不直接与存储数据的节点通信，而是通过 `mongos` 来进行。当一个查询被执行时，`mongos` 会确定需要访问哪些分片，并将查询转发到这些分片上。
- 在一个拥有多个分片的大型系统中，可能会有多个 `mongos` 实例来分散客户端请求的负载。

2.10.2 Config Server

- 配置服务器存储了整个 MongoDB 集群的元数据和配置信息。这包括分片的信息、路由策略、副本集的配置等。
- 在集群中，通常有三个配置服务器实例来保证高可用性和数据一致性。
- `mongos` 查询这些配置信息来了解数据的分布情况，并据此将客户端请求路由到正确的分片。

2.10.3 Replica Sets

- 副本集是 MongoDB 提供数据冗余和高可用性的方式。一个副本集包含了多个数据节点，其中一个是主节点，其他是从节点。
- 主节点处理所有的写操作，而从节点则复制主节点的数据变更。这样可以在主节点出现故障时自动切换到从节点，继续提供服务，无需数据丢失的风险。
- 副本集也可以用于读取分离，即读操作可以在从节点上进行，分担主节点的读取压力。

2.10.4 Read or Write Query

For example, a query `find({'class': 'cs5425'})` is pushed from the app

1. Query is issued to a **router** (`mongos`) instance
2. With help of **config server**, `mongos` determines which shard (**replica set**) to query
3. Query is sent to the relevant shards (partition pruning)

- 分区裁剪（Partition Pruning）是数据库查询优化器用来提高查询效率的一种技术。当查询操作针对一个分区表执行时，查询优化器会分析查询条件，以决定是否有些分区可以被排除在查询之外，因为它们不包含符合条件的数据。这样，数据库在执行查询时就不会扫描这些不相关的分区，从而节省了时间和计算资源。

Example: when reading a specific value of the shard key, the config server can determine that the query only needs to go to one shard (the one that contains the value of the shard key); writes are similar

- But if the query is based on a key other than the shard key, which is relevant to all shards, and the query will go to all shards

Shards run query on their data, and send results `{'name': 'bob', 'class': 'cs5425'}` back to `mongos`
`mongos` merge the query results and returns the merged results to the application

2.10.5 Replication

Common configuration: 1 primary, 2 secondaries
Write:

- 主节点接收所有的写操作
- 写入操作被记录到“operation log”
- 从节点复制“operation log”然后应用到本地的数据副本中

Read:

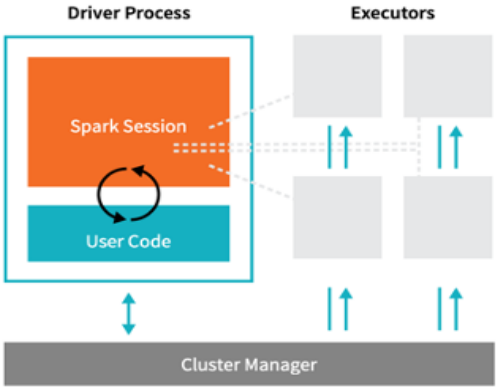
- 用户可以指定从主节点还是子节点读取
- 从子节点读取可以做到负载均衡和可能的更低延迟，但可能会出现读取到过时的数据

3 - Spark

3.1 Hadoop vs Spark

- Hadoop 在 MapReduce 的过程中，**中间数据需要被写入到磁盘**，并在机器间进行数据洗牌，这个过程是缓慢的。因为每次任务运行完之后，**输出都需要写到磁盘**，再被下一个任务读取，这造成了高磁盘开销。
- 同时，MapReduce **不适合迭代处理**，迭代处理指的是多次地对数据集进行操作，每次只修改一小部分数据。在 Hadoop 中，迭代处理的每一步都会被创建为一个**独立的 MapReduce 任务**，这使得效率变得低下。
- **Spark** 设计了一种不同的数据处理模型，它能够将大部分中间结果(RDD)存储在内存中，这使得数据处理速度大大提升，尤其是对于需要多次迭代的计算任务，例如图算法或者机器学习算法。因为这些任务需要多次读取和处理数据，使用 Spark 可以显著减少读写磁盘的次数，从而提高速度。
- 当内存不足以存储所有中间结果时，Spark 会将数据**"溢出"到磁盘**，这仍然需要磁盘 I/O，但这样的设计意味着只有在必要时才会访问磁盘，而不是像 Hadoop 那样的频繁磁盘读写。

3.2 Spark architecture



驱动进程(Driver Process):

- 驱动进程是 Spark 应用程序的主控制节点。它负责响应用户的输入，管理 Spark 应用程序的生命周期（如启动、停止），并且负责将工作分配给执行器。
- 驱动进程执行用户编写的主程序，并且创建出一个 `SparkContext` 对象。这个 `SparkContext` 会与集群管理器(Cluster Manager)通信，申请资源并在资源被分配后，将代码任务分发给集群中的执行器（Executors）。

执行器(Executer):

- 执行器是在集群中的工作节点上运行的进程，它们负责执行由驱动进程(Driver Process)分配给它们的代码，并返回计算结果。
- 每个执行器负责处理分配给其的数据，并执行任务。执行器还负责存储它们计算的结果数据，这些数据可能是 RDDs（弹性分布式数据集）的一部分，或者是广播变量和累加器。

集群管理器（Cluster Manager）：

- 集群管理器负责在 Spark 应用程序请求时分配计算资源。。
- 集群管理器的主要角色是在计算资源（如 CPU 和内存）和集群中可用的物理机器之间进行资源调度。

本地模式（Local Mode）：

- 当 Spark 在本地模式下运行时，上述所有的进程（驱动进程、执行器、甚至是模拟的"集群管理器"）都会在同一台机器上运行。

3.3 RDD

3.3 1RDD 介绍，特性

RDD（Resilient Distributed Dataset）是 Spark 中的一个基本概念，是一个不可变的、分布式的数据对象集合，能够并行操作。RDD 可以跨集群的多个节点分布存储数据，提供了一种高度的容错性、并行性和灵活性。

- RDD 一般可以被看作**是所有中间数据的统称**

1. **不可变性**：一旦创建，RDD 的数据就不可以被改变。这有助于容错，因为系统可以根据原始数据源重新构建 RDD。
1. **弹性**：RDD 能够在节点失败时重新构建丢失的数据分区，因为 RDD 的操作都是基于转换的，这些转换是可以记录的，并且是确定性的。这意味着如果某个节点的数据丢失，Spark 可以使用原始数据和转换操作日志来重新计算丢失的数据分区。
2. **分布式**：RDD 的数据自动被分散到集群中的多个节点上，可以在这些节点上并行处理。
3. **基于转换的操作**：RDD 的操作是通过转换（如 `map`、`filter`、`reduce` 等）来实现的，每个转换操作都会生成一个新的 RDD。转换是懒执行的，也就是说，只有在需要结果的时候才会执行。
4. **容错性**：RDD 通过记录转换的 lineage（血统信息）来提供容错能力。如果由于某种原因某个分区的数据丢失，Spark 可以通过这个 lineage 来重新计算丢失的分区数据。
5. **内存和磁盘存储**：RDD 可以存储在内存中，也可以存储在磁盘上，或者两者的组合。根据 RDD 的存储和持久化策略，可以优化性能。

3.4 Transformations 转换

- 转换（Transformation）是**对数据集进行操作的函数**，它接收当前的 RDD，应用一个计算函数，并返回一个新的 RDD。转换是**惰性**执行的，也就是说，它们不会立即计算结果。只有在行动（Action）操作请求时，例如当需要将数据保存到文件或者将数据集聚合计算结果返回给驱动程序时，转换才会被触发执行。
- **Narrow Transformation**: 窄转换是指不需要跨分区数据混洗（shuffle）的转换操作。在这种转换中，每个输出分区只需要一个或少数几个输入分区的数据。这意味着这些操作可以在单个节点上独立、高效地完成，不需要网络通信。
  - `map`：对数据集中的每个元素应用一个函数。
  - `filter`：过滤出满足特定条件的元素。
  - `flatMap`：将数据集中的每个元素转换为多个元素。
  - `mapPartitions`：对每个分区应用一个函数。
  - `range`: 使用 range 函数创建一个数据集
- **Wide Transformation**: 宽转换是指需要跨分区数据混洗的转换操作。在这种转换中，每个输出分区可能依赖于所有输入分区的数据。因此，这些操作通常需要跨节点的广泛数据交换和网络通信。
  - `groupBy`：根据某个键将数据集分组。
  - `reduceByKey`：根据键值对数据集进行聚合操作。
  - `join`：将两个数据集根据共同的键连接起来。
  - `sortBy`：按照某个或某些键对数据集排序。

3.5 Actions 执行

Actions trigger Spark to compute a result from a series of transformations

Examples of actions: `show`, `count`, `save`, `collect`

3.6 Caching and DAGs

3.6.1 Caching

- 在处理文件时，如果我们需要**多次处理**一个文件才能得到正确的输出结果时，最好在开始处理（即开始action）之前将文件**写入内存**

```
# Load file from HDFS (Hadoop distributed file system), then create an RDD
# sc for SparkContext
lines = sc.textfile("hdfs://...")
# Search for the Error line, create a new RDD with Error line
errors = lines.filter(lambda s: s.startwith("Error"))
# Split error line by tab, and extract the third sentence into a new RDD
messages = errors.map(lambda s: s.split("\t")[2])
# Store messages into memory
messages.cache()
# Filter mysql in messages, then count them (from the memory, faster)
messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "Spark" in s).count()
```

- `message.cache()`将 RDD 写入内存
- `message.persist(option)`可以选择将 RDD 写入内存，硬盘，或 off-heap memory

3.7 Narrow and Wide Dependencies

等同于 narrow and wide transformation

3.8 DAG

DAG 和执行阶段

- 在 Spark 中，任务的执行通过一个有向无环图（DAG）来表示，DAG 中的节点代表 RDD，边代表转换操作（即依赖关系）。
- 连续的窄依赖被组织成为一个“阶段”（Stage）。在这些阶段内，Spark 可以连续地在同一台机器上执行多个转换，而不需要在节点之间移动数据。
- 不同的阶段之间，由于宽依赖的存在，需要进行数据的“洗牌”（Shuffle），即跨分区交换数据。这个过程类似于 MapReduce 中的 shuffle，并且通常涉及到将中间结果写入磁盘。
- 由于数据洗牌是一个耗时的过程，涉及到网络传输和磁盘 I/O，所以在 Spark 程序中尽量减少洗牌是提高性能的一个重要实践。这意味着尽可能地利用窄依赖，以及在不可避免需要进行洗牌的宽依赖时，尽量减少需要交换的数据量

3.9 Lineage and Fault Tolerance

容错机制对比

- 在 Hadoop 的 MapReduce 中，容错是通过在磁盘上复制数据来实现的。如果一个数据节点失败，系统可以从副本中恢复数据。
- Spark 采取了不同的方法。由于 Spark 尝试将所有数据保存在内存中以提高速度，而内存资源相比磁盘更有限且成本更高，因此它不依赖于数据的复制来实现容错。

血统（Lineage）方法

- Spark 的 RDD 有一种内建的血统记录，即记录了它是如何从其他 RDD 转换来的。
- 当一个工作节点（Worker Node）发生故障，Spark 会启动一个新的工作节点来替代它。
- 利用 DAG（有向无环图），Spark 能够重新计算丢失的分区数据。DAG 记录了 RDD 之间的所有转换关系，所以 Spark 可以通过血统信息追溯到原始的数据源，并且只重新计算丢失分区的 RDD，而不需要重新计算整个数据集。
- 这种方法效率很高，因为它避免了不必要的数据复制，并且只在数据丢失时才重新计算数据。
- 在 Hadoop 中，一份数据会被复制三份以防止数据丢失。然而在 Spark 中，由于大部分数据都保存在内存中，在内存中复制数据会非常昂贵

3.10 DataFrames and Datasets

3.10.1 Dataset

- DataFrame 提供了一个类似于关系数据库中表格或者 Python 的 pandas 库中 DataFrame 的概念。它代表了以行和列组织的数据集，其中每列有一个名称和数据类型。

DataFrame 的特点

1. **表格形式的结构：**
   - DataFrame 提供了一个丰富的数据结构，每列都有固定的数据类型，而且它们可以容纳复杂的数据类型，如结构体和数组。
2. **SQL 类操作：**
   - DataFrame 支持多种操作，这些操作类似于 SQL 语言，如选择（select）、过滤（filter）、聚合（aggregate）等。
3. **性能优化：**
   - DataFrame 的操作是通过 Catalyst 优化器进行优化的，这是 Spark SQL 引擎的一部分。Catalyst 优化器会生成高效的执行计划。
4. **易用性：**
   - 相比于低级的 RDD API，DataFrame 提供了更简单、更直观的操作接口，对于数据分析师和数据科学家来说更加友好。
5. **兼容性：**
   - DataFrame 可以从多种数据源创建，如 Hive 表、数据库、JSON、CSV 文件等。
6. **内存计算：**
   - 类似于 RDD，DataFrame 也可以持久化到内存中，这对于迭代算法或多个操作中需要重用 DataFrame 的场景非常有用。

DataFrame 与 RDD 的关系

- **DataFrame API 是 RDD 的封装：**
  - DataFrame 的操作最终会映射到 RDD 上。Spark 在执行计划中将 DataFrame 的操作转换为 RDD 操作。
- **推荐使用 DataFrame API：**
  - Spark 官方推荐使用 DataFrame API 进行数据处理，因为 DataFrame API 提供了更好的性能和易用性。
- **RDD 仍然有其用处：**
  - 尽管 DataFrame API 被推荐用于大多数任务，但在某些需要精细控制的场景中，直接使用 RDD 仍然是有用的，比如进行一些定制化的数据转换和操作。

Transformation: 可以使用 **SQL 命令**直接对 DF 进行转换

3.10.2 Dataset

DataSet API 被认为是类型安全的（type safe），这是因为它提供了一个强类型的接口，允许编译器在编译时检查类型错误。这与 DataFrame 相对，DataFrame 是一个非类型安全的接口，因为它的列类型只有在运行时才被知晓和检查。

3.11 Machine Learning with Spark ML

3.11.1 Problem Setup

- **Classification**: Categorise samples into classes, given training data
- **Regression**: predict *numeric* labels, given training data

3.11.2 Data Processing

Handle missing values:

- Delete rows with missing values

Fill in the missing value based on:

- mean/median
- fitting a **regression** model to predict
- **Dummy variables**: optionally insert a new column which is 1 if the variable was missing, and 0 otherwise

Categorical Encoding

- Convert **categorical** feature to **numerical** features
- E.g., the risk rating [Low, Medium, High] will be converted into [0, 1, 2]
- 可以展示这个类别中暗示的数学关系，可以在 regression 中使用
- 也有可能引入不想要的数字关系 numerical relationship

One Hot Encoding

Convert **discrete feature** to a series of **binary features**
会移除数值关系

Normalisation

在数据预处理中进行归一化（Normalization）是为了调整数值型数据的尺度，使得所有的特征都被统一到一个固定范围内，通常是[0, 1]或者[-1, 1]。归一化的原因和好处包括：

1. **提高收敛速度**：在梯度下降等优化算法中，归一化可以帮助加快收敛速度。如果不同的特征具有不同的尺度，那么优化过程可能会变得很慢，因为小尺度的特征需要更大的权重变化才能在损失函数中产生相同的影响。
2. **消除量纲影响**：归一化可以消除不同特征的量纲影响，使得模型不会因为特征的尺度而偏向于某些特征。
3. **提高算法精度**：某些算法，如 K-最近邻（K-NN）和主成分分析（PCA），是基于距离的算法，如果不同的特征有不同的尺度，那么距离计算可能会被尺度大的特征主导，导致模型性能下降。
4. **避免数值计算问题**：过大或过小的数值在计算机中可能会导致数值溢出或下溢，归一化可以避免这些数值问题。
5. **满足模型的假设**：一些模型对数据有特定的假设，例如线性回归和逻辑回归假设所有的特征都是同等重要的，归一化可以帮助满足这些假设。

3.11.3 Training & Testing

Sigmoid Function

The sigmoid function $\sigma(x)$ maps the real numbers to the range (0,1):

$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad (3)$$

3.11.3 Evaluation

其中，表中的四个区域代表：

- TN (True Negative): test correct, (test) output negative
- TP (True Positive): test correct, output positive
- FN (False Negative): test wrong, output negative
- FP (False Positive): test wrong, output positive

我们可以用这四个数据计算不同的性能指标：

- Accuracy: fraction of correct predictions, $\frac{TN+TP}{TN+TP+FN+FP}$
- Sensitivity: fraction of positive cases that are detected, $\frac{TP}{FN+TP}$
- Specificity: fraction of actual negatives that are correctly identified, $\frac{TN}{TN+FP}$

3.11.4 Estimator

- **Estimator** 是一个算法，它可以基于给定的数据集学习或拟合出一些模型参数。换句话说，它是一个学习算法或者任何一个可以拟合或训练数据的对象。
- 在 Spark MLlib 中，Estimator 抽象表示一个学习算法，或者更具体地说，是一个 `fit()`方法。当你对一个数据集调用 `fit()`方法时，它会产生一个模型，这个模型就是一个 Transformer。
- 举个例子，一个用于分类的逻辑回归或者决策树算法，在训练数据上训练完成后，会变成一个 Estimator。

3.11.5 Transformer

- **Transformer** 是一个转换器，它把一个数据集转换成另一个数据集。通常，在机器学习中，转换器用来改变或预处理数据，比如进行归一化、标准化或者使用模型进行预测。
- 在 Spark MLlib 中，Transformer 表示一个 `transform()`方法，该方法接受一个 DataFrame 作为输入并产生一个新的 DataFrame 作为输出。通常，这个输出会包含预测结果、转换后的特征等。
- 例如，一个训练好的模型，比如逻辑回归模型，可以用作 Transformer 来对新数据进行预测。

3.11.6 Evaluate Regression Model

Mean Absolute Error (MAE)

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y_i}| \qquad (4)$$

Mean Squared Error (MSE)

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})^2 \qquad (5)$$

Root Mean Squared Error (RMSE)

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})^2} \qquad (6)$$

R Squared Value

The closer to 1, the better the model fits the data

$$SS_{\text{res}} = \sum_i (y_i - f_i)^2 = \sum_i e_i^2$$

$$SS_{\text{tot}} = \sum_i (y_i - \bar{y})^2$$

$$\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$$

$$R^2 = 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}}$$

4 - Streaming

4.1 Stateful Streaming Processing

Stateful stream processing 是指在流处理中**跟踪和更新状态信息的能力**。流处理是一种处理实时数据流的技术，数据流是一个连续的、快速的、无限的事件序列。与无状态流处理不同，无状态的只对单个消息进行操作，stateful stream processing 允许在处理数据流的时候考虑历史信息。在 stateful stream processing 中，可以保存关于过去事件的信息，这种信息被称为状态（state），并且可以在处理当前和未来事件时使用这些状态。

4.2 Spark

4.2.1 Micro Batch Stream Processing

Micro-batch stream processing 是一种处理实时数据流的方法，它将连续的数据流分割成小的、有序的时间窗口，这些窗口被称为"micro-batches"。每个 micro-batch 包含了一个时间段内到达的数据，并作为一个批次进行处理。这种方法介于传统的批处理和纯粹的流处理之间。

在 micro-batch 流处理模型中：

1. **数据分批处理**：实时数据流被分割成连续的小批次数据。这些批次按照它们被收集的时间段进行处理。
2. **定期执行**：每个 micro-batch 都在定期的时间间隔内被处理，例如，每隔几秒或几分钟。
3. **容错和重放**：由于每个 micro-batch 是独立处理的，这种模型可以容易地实现容错机制，例如如果处理失败，可以重新执行失败的 micro-batch。
4. **状态管理**：虽然每个批次独立处理，但 stateful 操作可以跨批次维护状态，例如，通过在连续的 micro-batches 间保持状态信息，可以计算滑动窗口聚合。
5. **延迟与吞吐量的权衡**：Micro-batch 处理模型允许在处理延迟和系统吞吐量之间进行权衡。减少 micro-batch 的大小可以降低延迟，增加批次大小可以提高吞吐量。

- **Spark** 中的 Spark Streaming 是实现微批次流式处理的一个例子

4.2.2 Structured Streaming Processing Model

1. **流处理的广义定义**：Structured Streaming 将流处理视为一个更宽泛的概念。在这个模型中，实时数据流被看作是一个无界的表（unbounded table），即一个持续增长的表，新数据不断追加到表的末尾，就像流水线上不断推送的数据流。
2. **无界表的概念**：在 Structured Streaming 中，数据流被视为一个无界表，开发者可以像查询静态表一样查询这个无界表。这种抽象简化了流处理的开发，因为处理无界表的查询与处理有界表（传统的静态数据集）的查询在概念上是一致的。

4.2.3 Five Steps to Define a Streaming Query

1. **Define input sources**
   - 选择你的数据来源，这可能是实时日志文件、消息传递系统如 Kafka、数据管道如 Amazon Kinesis 或其他支持的流数据源。
   - 为你的流数据定义一个输入架构，使得数据可以被流处理框架所解析和处理
2. **Transform data**
   - 应用转换操作来处理流数据，比如筛选、聚合或者与其他数据集的连接等。
   - 这些转换操作将原始输入数据转换成你希望在最终输出中看到的形式。
3. **Define output sink and output mode**
   - Processing details (how to process data and how to recover from failures)
   - 确定你的数据最终将被输出到哪里，这被称作输出汇（sink）。输出汇可以是文件系统、数据库或其他存储系统。
   - Output writing details (where and how to write the output): 选择输出模式，这可以是完全覆盖已有数据、只追加新数据、更新改变的数据等。
   - 配置输出写入的具体细节，比如文件格式、目录结构等。
4. **Specify processing details**
   - **触发细节(Triggering details)**: 定义何时触发查询处理新的数据。这可以是基于时间的（如每隔一定时间），或者尽可能快地处理新数据
   - **检查点(Checkpoint Location)**: 设置一个位置来存储流查询的进度，以便在故障发生后可以从上次的进度恢复。
5. **Start the query**
   - 一旦所有的细节都被指定，最后一步是启动查询。
   - 启动查询后，流处理系统将持续运行，不断处理新的数据流，直到被停止或遇到错误。

4.2.4 Incremental Execution of Streaming Queries

- **增量处理**：在这种执行模式下，系统不会在每次有新数据到达时重新处理整个数据集，而是**仅仅处理自上次查询以来新到达的数据**，这就是"**增量 incremental**"的概念。

4.2.5 Data Transformation

Stateless Transformation

- Process each row individually **without needing any information from previous rows**
- Projection operations: `select()`, `explode()`, `map()`, `flatMap()`
- Selection operations: `filter()`, `where
- **定义**：Stateless transformations 是指在处理数据时不需要考虑之前的数据或结果的转换。每个数据项都独立于其他数据项进行处理，转换的输出仅仅依赖于当前的输入数据项。
- **例子**：一个例子是 `map` 操作，它将一个函数应用于数据流中的每个元素，输出结果只取决于当前元素。其他例子包括 `filter`（过滤数据流中的元素）和 `flatMap`（将数据流中的每个元素转换为零个或多个输出元素）。

Stateful Transformation

- A simple example: `DataFrame.groupBy().count()`
- In every micro-batch, the incremental plan adds the count of new records to the previous count generated by the **previous micro-batch**
- The partial count communicated between plans is the **state**
- The **state** is **maintained in the memory** of the Spark executors and is check pointed to the configured location to tolerate
- **定义**：Stateful transformations 是指在处理数据时**需要考虑之前的数据或状态的转换**。这意味着转换的输出不仅取决于当前的输入数据项，**还取决于过去收到的数据**。
- **例子**：一个典型的例子是 `reduceByKey` 操作，它会跨多个数据项聚合值（例如，计算总和或平均值）。

这需要跟踪状态，因为每个新数据项都可能影响最终的聚合结果。其他例子包括 window 操作（在定义的时间窗口内聚合数据）和 join 操作（可能需要等待匹配的元素到达）。

4.2.6 Stateful Streaming Aggregations

在静态的（非流式的）DataFrame 上，你可以使用像 count()或 reduce()这样的直接聚合操作，它们会立即计算并返回最终的聚合结果。然而，在流式 DataFrame 上，这样的操作**是不可行的**，原因如下：

1. **连续更新的需求**：在流式环境中，数据是连续不断到来的，这意味着聚合的结果也需要随着新数据的到来而不断更新，而不是计算一次最终结果。
2. **聚合 API 的限制**：由于需要连续更新聚合结果，流式 DataFrame 不支持立即返回结果的聚合操作。因此，你不能在流式 DataFrame 上直接使用 count()和 reduce()这样的操作。
3. **使用分组聚合**：要在流式 DataFrame 上执行聚合，你需要使用 groupBy()或 groupByKey()方法。这些方法允许你定义一个或多个聚合操作，这些操作随着数据流的进行而持续执行，并且可以返回一个新的流式 DataFrame，其中包含了到目前为止的聚合结果。
4. **输出模式的选择**：在使用 groupBy()或 groupByKey()进行流聚合时，你还需要选择一个输出模式，例如“完整模式”（输出当前所有聚合的完整结果）或“更新模式”（仅输出自上次触发以来更改的聚合结果）。

4.2.7 Time Semantics

Event Time

- 事件时间将处理速度和结果完全分离。基于事件时间的运算是可预测的，其结果也是确定的。
- 由于数据到处理节点需要事件，我们使用该事件真实发生的时间来判断是否计算这个数据。无论数据流的处理有多快，或事件会何时到达运算器，事件时间窗口计算都会产生相同的结果

Watermark

水印（watermark）的概念是为了处理实时数据流中的延迟数据（late data）问题，并提供一种机制来指定何时可以安全地关闭一个时间窗口的聚合。

事件时间是指数据生成的实际时间，与处理数据的时间（处理时间，processing time）不同。由于网络延迟、系统故障或数据源的不规律发送行为等原因，数据可能会不按顺序到达处理系统，即使这些数据带有它们的事件时间戳。

水印是一个与时间相关的阈值，通常设置为“当前已观察到的最大事件时间减去一定延迟量”。它提供了一个处理延迟数据的策略：

1. **容忍一定的延迟**：通过水印，系统可以等待一段时间来处理迟到的数据，这允许在某个时间窗口内的聚合结果中包含这些迟到的数据。
2. **触发窗口计算**：当水印超过了某个 **时间窗口(Event-time Window)** 的结束时间时，可以认为该窗口不再会有更多的数据到达，因此可以安全地触发该窗口的计算并输出结果。
3. **管理状态大小**：流处理系统通常需要维护状态来处理窗口聚合。水印可以作为一个信号，告知系统何时可以清理某个时间窗口的状态，从而控制状态的增长。
4. **提高结果的确定性**：引入水印可以帮助系统更准确地确定何时可以输出最终结果，减少因为数据乱序到达而导致的结果不确定性。

- 在实际使用过程中:

```
(sensorReadings
.withWatermark("eventTime", "5 minutes")
.groupBy("sensorID", window("eventTime", "10 minutes", "
5 minutes"))
.count())
```

- 第二行代表 watermark 的有效时间为 5 分钟
- 第三行表示 event time window 窗口是 10 分钟，每 5 分钟统计一次窗口，如 11:15-12:05, 12:00-12:10, 12:05-12:15
- 假设最后一次事件的发生时间为 12:08，接下来收到了三个事件，事件分别为 11:54, 12:02, 12:13
  - 对于发生在 11:54 的事件，由于最早**可被接受的事件时间为: 12:08 - 5min = 12:03, 对应 event time window 为 11:55-12:05 和 12:00-12:10**，均超过 11:54，所以该事件不被接受
  - 对于发生在 12:02 的事件，**其处在 event time window 中，即使此事件事件在 watermark 可被接受的范围之外，但由于 event time window 的存在，处于该 window 中的所有事件都会被接受**
  - 对于 12:13 的事件，会被接受

4.2.8 Spark Checkpoint

检查点是一种容错机制，他将 RDD 的当前状态 State 保存到可靠的存储系统中(HDFS, S3).使用 checkpoint()保存

Checkpoint 的作用：

- **断点恢复**：当发生故障时，checkpoint 使 Spark 能够从最近的检查点恢复数据，而不是从头开始重新计算整个流水线
- **清理 Lineage**：一旦 RDD 被 checkpointed，其 lineage 信息被清除。在 Spark 中，每个 RDD 都保留有血统（lineage）信息，即它是如何从其他 RDD 转换来的记录。随着转换操作的增加，血统链可能变得很长，导致恢复时间和资源消耗增加。Checkpoint 可以切断这个血统链，通过保存当前 RDD 的物理状态来减少需要重新计算的步骤。
- **防止堆栈溢出**：在进行深度依赖的迭代计算时，长 Lineage 可能导致堆栈溢出，检查点可以减少这种风险

优点：

- 减少故障恢复时间，不需要重新计算整个 Lineage
- 避免长 Lineage 可能引起的堆栈溢出

缺点：

- 检查点的写入需要涉及磁盘 I/O，可能会影响程序性能
- 需要外部存储系统，增加系统的复杂性

由于检查点会将数据写入磁盘，它通常在必要时才使用，例如在迭代算法中，其中同一个 RDD 会被多次计算。在这些场景下，使用 checkpointing 可以帮助提高效率和确保数据处理的可靠性。

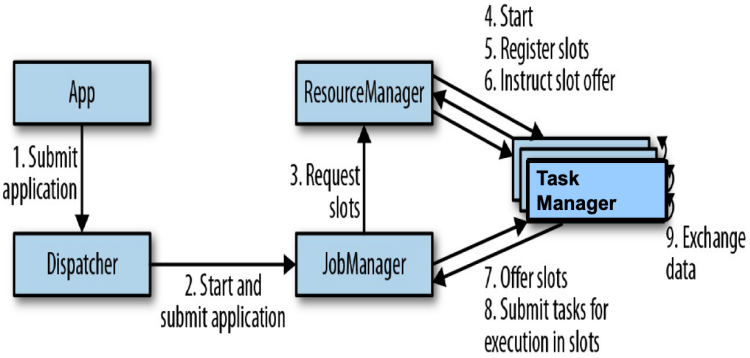
4.3 Flink

Apache Flink 是一个开源流处理框架，用于在高吞吐量和低延迟的要求下处理无界和有界的数据流。Flink 被设计用于运行在所有常见的集群环境上，以及执行任意规模的数据处理任务。

Flink 的一些关键特点包括：

1. **真正的流处理**：Flink 提供了真正的流处理能力，而不是微批处理（micro-batching）。它处理事件流的方式是一次处理一个事件，提供了更低的延迟和更高的处理效率。
2. **事件时间和水印**：Flink 支持事件时间（event time）概念和水印（watermarks），这使得它能够处理乱序事件并在分布式系统中提供一致的结果。
3. **状态管理和容错**：Flink 提供了先进的状态管理系统，可以在分布式环境中维护大量状态，同时支持容错机制，如保存检查点（checkpoints）和保存点（savepoints）。
4. **可扩展性**：Flink 能够扩展到数千个节点，处理大规模的数据流。
5. **丰富的 API**：Flink 提供了 DataStream 和 DataSet API 来处理实时和批处理任务，以及 Table API 和 SQL API 来执行关系型操作。
6. **多种部署选项**：Flink 可以在各种环境中部署，包括在本地、在大数据集群（如 Hadoop YARN、Apache Mesos）以及云平台上。
7. **支持复杂事件处理（CEP）**：Flink 内置了复杂事件处理的功能，可以用于模式匹配和事件序列的识别。
8. **集成**：Flink 提供了与其他存储系统和消息队列（如 Apache Kafka、Amazon Kinesis、Elasticsearch）的连接器，以便于数据输入和输出。

4.3.1 Flink Architecture



4.3.2 Task Execution

Task Manager 的角色

1. **并发执行任务：**
   - Task Manager 能够同时执行多个任务。这些任务可以是：
     - 相同操作符的不同实例，实现数据并行性（data parallelism）。
     - 不同操作符的任务，实现任务并行性（task parallelism）。
     - 不同应用（job）的任务，实现作业并行性（job parallelism）。
2. **处理槽（Processing Slots）：**

- Task Manager 提供了一定数量的处理槽（slots）。每个处理槽能够控制 Task Manager 能够并发执行的任务数量。
- 处理槽的数量通常是配置参数，它决定了 Task Manager 能够并发处理的任务上限。

3. **应用的执行切片：**

- 一个处理槽可以执行应用的一个切片，即应用中每个操作的一个并行任务。
- 在 Flink 中，一个应用通常被分解为多个操作，每个操作可以进一步分解为多个并行任务。每个处理槽负责其中的一个任务。

处理槽的工作方式

- 每个处理槽是应用的执行资源的一个单位。在分布式系统中，为了有效地利用资源和提高吞吐量，任务通常会在多个处理槽上并行执行。
- 处理槽数定义了 Task Manager 能够执行的任务的并行度。例如，如果一个 Task Manager 有四个处理槽，它可以同时执行四个任务。
- 处理槽的概念类似于多线程编程中的线程，每个处理槽能够独立执行一个任务，而多个处理槽则允许多个任务并行执行。

4.3.3 Data Transfer in Flink

数据传输的特点

1. **连续交换数据**：Flink 应用中的任务在运行时会连续地交换数据。这通常涉及到数据在不同操作之间的转移，例如，在 `map` 操作产生的数据被传递给 `reduce` 操作。
2. **任务管理器的数据传输职责**：Task Manager 负责管理数据传输过程，它处理从发送任务到接收任务的数据运输。这个过程可能包括序列化数据，网络传输，以及反序列化数据供接收任务使用。
3. **网络组件和缓冲区**：Task Manager 的网络组件会在发送数据之前，将记录收集在缓冲区中。这意味着数据不是一条一条立即发送的，而是积累到一定量后作为一个更大的数据块发送，以提高网络效率和减少延迟。

4.3.4 Event-Time Processing

事件时间处理的关键概念

1. **时间戳（Timestamps）：**
   - 每条记录都必须附带一个事件时间戳，这个时间戳表示事件实际发生的时间。在 Flink 中，时间戳通常是记录中的一个字段，它可以在数据进入 Flink 系统时附加，或者是从数据本身中提取的。
2. **水印（Watermarks）：**
   - 水印是 Flink 事件时间应用中的一个关键概念。它们是特殊的记录，用于表示在某个时间点之前的所有数据都已经到达。换句话说，水印是 Flink 用来处理乱序事件和实现窗口操作的机制。
   - 水印允许 Flink 估计事件时间的进度，即使事件数据是乱序到达的，也可以基于事件时间来处理。
3. **水印的实现：**
   - 在 Flink 中，水印作为特殊的记录实现，它们携带一个长整型（Long）值的时间戳。水印在常规记录流中流动，它们可以告诉 Flink 系统特定时间的数据是否已经到达。

4.3.5 State Management

在流处理任务中，状态是关键的概念，尤其是对于有状态的流处理任务。这些任务会保持数据状态，以便进行复杂的计算。状态可以是任何与任务相关的数据，比如计数器、窗口的内容、或者用于复杂事件处理的部分结果。

Operator State（操作员状态）：

- 操作员状态是特定于单个操作任务的。
- 在一个任务内处理的所有记录都可以访问相同的状态。
- 操作员状态对于同一个或不同操作的其他任务是不可见的。

Keyed State（键控状态）：

- 键控状态为每个键值维护一个独立的状态实例。

- 它将所有具有相同键的记录分配给维护该键状态的操作任务。
- 这允许任务对每个键执行独立的处理，例如在使用键/值数据进行聚合时。

4.3.6 Checkpoints

一致性检查点：

- 这与 Spark 的微批处理检查点类似。
- 它涉及以下步骤：

1. 暂停所有输入流的摄取。
2. 等待所有正在处理的数据完全处理完毕，也就是说，所有任务都已经处理了它们的输入数据。
3. 通过将每个任务的状态复制到远程持久存储来创建检查点。当所有任务完成复制时，检查点就完成了。
4. 恢复所有流的摄取。

4.3.7 Comparison between Spark and Flink

Spark:

- Micro-batch streaming processing (with latency of a few seconds)
- Spark 对每个微批次数据执行检查点操作，这是**同步进行的**。每次微批次完成时，它会触发检查点机制，而**下一个微批次必须等到检查点完成之后才能运行**，这可能会增加整体延迟。
- Watermark: a configuration to determine when to drop the late event

Flink:

- Real-time streaming processing (with latency of milliseconds)
- Flink 的检查点是**分布式且异步进行的，即在后台进行**。这使得检查点操作更高效，对数据处理的影响更小，从而降低了延迟。
- Flink 中的水印是特殊的记录，用于确定何时触发基于事件时间的计算。它允许系统处理有序和无序的事件，并保证即使出现乱序事件，时间窗口的结果也是正确的。
    - Flink uses late handling function (related to watermark) to determine when to drop the late events

5 - Graph

5.1 Simplified PageRank

PageRank 基于这样一个假设：重要的网页很可能会被更多的其他网页所链接。换句话说，一个网页的重要性取决于链接到它的其他网页的数量和质量。如果一个网页被许多其他重要的网页链接，那么这个网页也被认为是重要的。

- 其他网页引用此网页的链接被视为对此网页的“投票”
- 如果其他重要性高的网站引用了此网站，则此网站的重要性也会变高
- 重要性 r_j 对于某个网站 j 的公式如下

$$r_j = \sum_{i \to j} \frac{r_i}{d_i} \qquad (7)$$

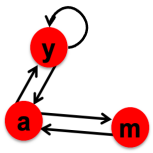
- r_i 是网站 i 的重要性，网站i引用了网站 j
- d_i 是网站 i 的所有引用其他网站的数量

5.1.1 Matrix Formulation

- The flow equation can be written as:

$$r = M \cdot r \qquad (8)$$

- M 是一个矩阵，长和宽都为网站的总数，对于网站i引用了网站j，那么在 M[j][i]的值就为$\frac{1}{d_i}$，即 j 行 n 列
- r 是一个标量，其保存所有网站
- 计算所有网站的重要性即为 M 点乘 r



	y	a	m
y	½	½	0
a	½	0	1
m	0	½	0

$r = M \cdot r$

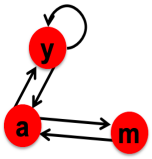
$$r_y = r_y/2 + r_a/2$$
$$r_a = r_y/2 + r_m$$
$$r_m = r_a/2$$

$$\begin{bmatrix} r_y \\ r_a \\ r_m \end{bmatrix} = \begin{bmatrix} ½ & ½ & 0 \\ ½ & 0 & 1 \\ 0 & ½ & 0 \end{bmatrix} \begin{bmatrix} r_y \\ r_a \\ r_m \end{bmatrix}$$

5.1.2 Power Iteration

Power Iteration:

- Suppose there are *N* web pages
- Initialize: $\mathbf{r}^{(0)} = [1/N,\ldots,1/N]^T$
- Iterate: $\mathbf{r}^{(t+1)} = \mathbf{M} \cdot \mathbf{r}^{(t)}$
- Stop when $|\mathbf{r}^{(t+1)} - \mathbf{r}^{(t)}|_1 < \varepsilon$



$$r_y = r_y/2 + r_a/2$$
$$r_a = r_y/2 + r_m$$
$$r_m = r_a/2$$

- **Example:**

$$\begin{bmatrix} r_y \\ r_a \\ r_m \end{bmatrix} = \begin{matrix} 1/3 & 1/3 & 5/12 & 9/24 & \\ 1/3 & 3/6 & 1/3 & 11/24 & \ldots \\ 1/3 & 1/6 & 3/12 & 1/6 & \end{matrix} \begin{matrix} \\ \\ \\ \end{matrix} \begin{matrix} 2/5 \\ 2/5 \\ 1/5 \end{matrix}$$

Iteration 0 Iteration 1 …

5.1.3 Dead End

只有一个出度的节点

5.1.4 Spider Trap

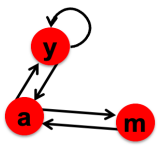
所有**能够进入环的路径**

5.2 PageRank with Teleports (Random Walk)

$$r_j = \sum_{i \to j} \beta \frac{r_i}{d_i} + (1-\beta)\frac{1}{N} \qquad (9)$$

- N 为图中所有的网站数量
- β 为阻尼系数

5.2.1 Google Matrix



$$A = \beta M + (1-\beta)\left[\frac{1}{N}\right]_{N \times N}$$

	y	a	m
y	½	½	0
a	½	0	1
m	0	½	0

N by N matrix where all entries are 1/N

	y	a	m
	⅓	⅓	⅓
	⅓	⅓	⅓
	⅓	⅓	⅓

5.3 Problems with PageRank

1. **泛化流行度问题：**
   - PageRank 通常衡量的是页面的泛化流行度，它不会根据特定主题区分页面的重要性。这意味着即使一个页面在某个主题上非常受欢迎，如果它没有足够的全局入站链接，它的 PageRank 可能仍然不高。
   - **解决方案：主题特定的 PageRank（Topic-Specific PageRank）：**
     - 这是 PageRank 算法的一种变体，它考虑了页面与特定主题的相关性。在这种方法中，随机跳转不是均匀分布到所有页面，而是倾向于与特定主题相关的页面。
2. **单一重要性度量问题：**
   - PageRank 使用单一的度量标准来定义页面的重要性，但在现实世界中，一个页面的重要性可能有多个维度。
   - **解决方案：Hubs-and-Authorities（枢纽和权威）：**
     - 这是一个不同的链接分析算法，也被称为 HITS 算法。它识别了两种类型的页面：枢纽（hubs），这些是链接到许多权威页面的页面；以及权威（authorities），这些页面被许多枢纽页面链接。枢纽和权威值分别衡量页面作为资源列表和内容提供者的重要性。
3. **易受链接垃圾攻击问题：**
   - PageRank 可以通过所谓的链接垃圾攻击来操纵，即通过人为创建的链接拓扑结构来提高页面的 PageRank。
   - **解决方案：Trust Rank（信任排名）：**
     - Trust Rank 是一个帮助识别和过滤垃圾页面的算法。它依赖于一小部分已知的高质量页面（种子集），然后基于这些页面传递信任值来识别可信页面。这有助于降低通过链接垃圾操纵 PageRank 的效果。

5.4 Topic Sensitive PageRank

$$A_{ij} = \begin{cases} \beta M_{ij} + (1-\beta)/|S| & \text{if } i \in S \\ \beta M_{ij} + 0 & \text{otherwise} \end{cases}$$

- **A** is stochastic!
- |S|为集合 S 中网站的个数
- M_ij 为入度网站的重要性

5.5 Pregel

Pregel 是一个用于大规模图处理的计算模型，其计算过程包括一系列的超步：

1. **超步（Supersteps）：**
   - 计算由一系列的超步组成，每个超步包含一轮顶点的计算。
2. **用户定义的 `compute()` 函数：**
   - 在每个超步中，框架为每个顶点调用用户定义的 `compute()` 函数，概念上这是并行执行的。

3. **`compute()` 函数的行为：**
   - `compute()` 函数定义了在单个顶点 v 和超步 s 时的行为：
     - 它可以读取在超步 s-1 发送给 v 的消息。
     - 它可以发送消息给其他顶点，这些消息将在超步 s+1 中被读取。
     - 它可以读取或写入顶点 v 的值以及其出边的值，甚至可以添加或删除边。
4. **终止：**
   - 顶点可以选择停用自己，如果在后续的超步中接收到新消息，它会被重新激活。
   - 当所有顶点都处于非活动状态时，计算停止

Pregel 的架构

- Pregel 采用主/从（Master/Workers）架构：

哈希分区：

- 顶点默认通过哈希分区，这意味着顶点会根据它们的标识符被分散到不同的工作节点（workers），这种分布方式称为“边切割”（edge cut）。

工作节点（Workers）：

- 每个工作节点负责维护其分配部分图的状态，并将其保持在内存中，以便快速访问和计算。

内存中计算：

- 所有的计算都在内存中进行，这提高了处理速度并减少了对磁盘 I/O 的需求。

执行 `compute()` 函数：

- 在每个超步中，每个工作节点遍历其负责的顶点，并执行 `compute()` 函数，进行图算法的计算。

消息传递：

- 顶点之间的消息会被发送。如果目标顶点位于同一个工作节点，则消息直接传递；如果位于不同工作节点，则消息在本地缓冲并批量发送，以减少网络流量

容错机制

Pregel 还包括以下容错机制：

检查点（Checkpointing）：

- 定期将图的状态保存到持久存储（例如，分布式文件系统），这些检查点可以用于在发生故障时恢复图的状态。

心跳检测故障：

- 通过心跳机制来检测节点故障。如果主节点在预定时间内未收到某个工作节点的心跳，就认为该工作节点发生了故障。

重新分配和重载：

- 如果检测到工作节点损坏，会重新分配该节点的任务给其他节点，并从最近的检查点中重新加载图的状态，以继续计算。

5.5.1 PageRank in Pergel

```
Compute(v, messages):
    if getSuperStep() == 0:
        v.setValue(1 / getNumVertices()) # Set initial importan
    ce to 1/N
    if getSuperStep() >= 1:
        sum = 0
        for m in messages:
            sum += m
        v.setValue( (1-beta)/getNumVertices() + (beta)*sum )
    if getSuperStep() < iteration_number:
        sendMsgToAllEdges(v.getValue() / len(getOutEdgeIterator
    ()))
else:
    voteToHalt()
```

6 - Big Data System

6.1 Database

- 存储结构性数据（table）
- 可以通过 SQL 语句查询
- 数据遵循严格的格式
- 提供强 ACID 保证：原子性，一致性，隔离性，持久性
- 适用于 Online Transaction Processing (OLTP)

6.2Data Warehouse

- A central relational repository of integrated, historical data from multiple data sources

数据仓库的优势

1. **商业社区的支持**：数据仓库为商业用户提供了一个集中的数据存储解决方案，帮助他们从历史数据中洞察业务趋势。
2. **历史数据的存储**：能够存储来自不同来源的大量历史数据，为数据分析和决策提供基础。
3. **交易保障**：提供强大的事务性保证，符合 ACID（原子性、一致性、隔离性、持久性）原则，确保数据的准确性和完整性。
4. **标准化建模**：使用标准的星型模式（Star Schema）建模技术来模型化数据，这适用于商业智能和报告。
5. **商业智能和报告**：非常适合执行业务智能分析和生成报告，帮助企业了解历史业绩并规划未来。

数据仓库面临的挑战

6. **大数据趋势**：大数据的四个特征，即“4V”：体积（Volume）、速度（Velocity）、多样性（Variety）、真实性（Veracity），提出了对数据处理能力的新要求。
7. **数据规模的增长**：数据量的急剧增加使得数据仓库的存储和处理能力受到挑战。
8. **分析多样性的增长**：分析需求的多样化要求数据仓库支持更广泛的数据处理和分析方式。
9. **扩展成本高昂**：传统数据仓库的横向扩展（Scale Out）通常非常昂贵，特别是当涉及到存储和处理大规模数据集时。
10. **对非 SQL 分析的支持不足**：传统数据仓库主要为 SQL 查询和报告设计，对于非 SQL 的数据分析模型和算法支持不足。

6.3 Data Lake

数据湖的核心特征

1. **大规模存储**：数据湖能够存储任何规模的数据，适应数据体积的增长。
2. **分布式存储**：基于普通硬件，以分布式方式存储数据，能够轻松水平扩展。
3. **开放格式文件**：数据以开放格式保存，确保了不同的处理引擎能够通过标准 API 读写数据。
4. **存储和计算分离**：通过将存储系统与计算系统解耦，可以根据工作负载的需要独立扩展它们。
5. **组件选择自由**：组织可以独立选择存储系统（如 HDFS、S3）、文件格式（如 Parquet、ORC、JSON）和计算/处理引擎（如 Spark、Presto、Apache Hive）。

数据湖的优势

6. **灵活性**：用户可以灵活选择存储、数据格式和处理引擎。
7. **成本效益**：相较于传统数据库，数据湖是一个更便宜的解决方案，有助于大数据生态系统的快速增长。

数据湖面临的挑战

8. **缺乏 ACID 保证**：数据湖通常无法提供像传统数据库那样的 ACID 事务保证。
9. **专业技能要求**：构建和维护高效的数据湖需要专家级技能。
10. **数据转换成本高**：尽管数据湖易于数据注入，但将数据转换为能够交付商业价值的形式可能非常昂贵。
11. **数据质量问题**：由于缺乏架构强制，可能会出现数据质量问题。

6.4 Data Lakehouse

6.5 Delta Lake

Delta Lake 是一个开源存储层，用于在现有的数据湖上提供 ACID（原子性、一致性、隔离性、持久性）事务支持。它将数据湖中的大量非结构化和半结构化数据转化为一个有结构的、可靠的数据存储，从而克服了传统数据湖的某些限制，如元数据管理的不足、数据版本控制和更新的复杂性、以及数据质量问题。

Delta Lake 的关键特点包括：

1. **ACID 事务**：提供数据修改的事务支持，包括并发读写操作的隔离性保证和原子性更新，确保数据的一致性和完整性。
2. **可伸缩的元数据处理**：通过在存储层面引入元数据，Delta Lake 支持大规模数据集的快速读取和写入，同时保持对元数据的快速访问。
3. **数据版本控制**：支持数据的版本控制，允许用户访问和恢复到历史数据版本，为数据变更提供审计和回滚能力。
4. **架构演化**：为数据表提供了架构演化支持，当数据架构变化时，可以在不删除现有数据的情况下添加新字段或更改现有字段。
5. **统一批处理和流处理**：Delta Lake 可以用于批处理和流处理数据，为两者提供统一的框架和 API，简化了大数据处理流程。
6. **兼容现有的数据湖技术**：它可以无缝集成到现有的数据湖架构中，如 Hadoop、AWS S3、Azure Data Lake Storage 等，并兼容大数据处理框架，如 Apache Spark。

6.6 Parquet

Parquet 是一种开源的列式存储格式，专为大数据的性能和效率而设计。它支持复杂的嵌套数据结构，并且由于其列式的性质，非常适合于数据仓库操作，如 Apache Hadoop、Apache Spark 和 Apache Impala 等大数据处理工具。

Parquet 文件格式的关键特点包括：

1. **列式存储**：数据按列存储而不是按行存储。这种方式非常适合进行大规模的数据分析操作，因为它可以有效地压缩数据并减少读取数据的 IO 操作，特别是在查询特定列时。
2. **高效压缩和编码**：Parquet 文件支持高效的压缩和编码方案。由于列中的数据通常是相同类型的，所以可以更有效地压缩数据，减少存储空间。
3. **支持复杂的数据类型**：Parquet 支持复杂的嵌套数据结构，如结构体、列表和映射等。
4. **优化的读取性能**：列式存储使得在执行分析查询时，只需读取必要的列数据，从而优化了读取性能。

5. **兼容性**：Parquet 文件可以与许多数据处理工具集成，如 Apache Hive、Presto 和 AWS Athena 等。
6. **跨平台互操作性**：Parquet 格式支持跨平台使用，这意味着在不同的数据处理系统之间可以无缝地移动和处理 Parquet 文件。

6.7 Delta Log

1. **事务日志是有序记录**：Delta Lake 的事务日志记录了自表创建以来的每一个事务。这包括数据的添加、删除、修改等所有更改。
2. **真理的唯一来源**：事务日志作为表变化的唯一记录，是确定表历史状态和当前状态的权威来源。
3. **支持并发读写**：事务日志的主要目标是允许多个读写者同时对数据集的给定版本进行操作。这意味着用户可以同时读取和写入数据，而 Delta Lake 会协调这些操作以保证数据一致性。
4. **核心功能**：事务日志是 Delta Lake 提供许多重要功能的核心，包括但不限于以下几点：
   - **ACID 事务**：当 Apache Spark 或其他数据处理引擎需要读取或修改 Delta 表时，它们会参照事务日志来获取表的最新版本。如果操作未在事务日志中记录，那么在 Delta Lake 的视角中，该操作就被视为没有发生过。
   - **可扩展的元数据处理**：随着表和文件的数量增长，事务日志允许 Delta Lake 高效管理元数据，而无需读取整个数据集的所有文件。
   - **时间旅行（Time Travel）**：事务日志允许用户查看表的历史版本，回溯到过去的某个特定点。这种能力称为“时间旅行”，它为数据的审计和回溯提供了强大的能力。

Midterm

- Sensor readings have different formats including video and audio, this shows the **uncertainty** of data
- **Broadcast Join**: copy the smallest dataset into other nodes, **no reducer** process
- **Reduce-side Join**: Read both two dataset, and process them through map, shuffle and reduce
- When a user requests to download a file from HDFS, only the **metadata** of the file will be issued to the name node. The data will directly be downloaded from the data nodes.
- Current big data system designs mainly use **scale-out** (扩大规模) architecture rather than **scale-up** (升级现有)
- Min-Hashing use the smallest value of Hashing result as **signature**, if two document has the same/similar signature, they are **candidate pairs**
- HDFS has **three replicas** for each chunk, **two at the same rack, one in the other rack**. Thus, the system offers more **flexibility** of moving a task to the machine when a replica is stored
- Task parallelism of MapReduce will **not** improved by increase the chunk size
- The key function is used in **partitioner** to better spread out the load among different reduce tasks
- The **assignment** of workers to map and reduce tasks is run within the **Mater Node**
- Hadoop needs to repeatedly write HDFS
- K-means Algorithms:
  1. **Initialisation**: Pick K random points as centers
  2. **Repeat**:
     1. **Assignment**: assign each point to nearest cluster
     2. **Update**: move each cluster center to the average of its assigned points
     3. **Stop** if no assignments change
- Key-value store:
  • Improves scalability and efficiency – writing or reading user pages is faster.
  • No need for complex queries or based on the content of user pages – just reads and writes.
  • May be acceptable for user pages to be slightly stale – then eventual consistency is acceptable
- Document store:
  • Flexible schema may be beneficial (e.g. special types of vehicles may require different sets of fields)
  • Unlike key-value stores, document stores are more suitable for queries based on fields of a document

Final

- NoSQL databases often use **denormalised** views, also known as duplication, to improve query performance and overcome the limitations of not supporting joins. This makes it easier to query the data, as all the necessary information is available in a single table/document.
- Spark creates a **DAG** to record the transformation into few stages. We would try to **avoid transformation across stages** to improve performance. Across stage transformation needs data to be shuffled across different servers. Which is expensive.
- Through DataFrame API to write `join()` function, all the languages will achieve **similar performance**. Catalyst Optimiser is embedded in DF API, all language will use this.
- In Pregel model, the compute() function will update the state of vertex and send message to **all neighbouring vertices**
- Bottleneck of Spark operations:
  - **Join two tables**: if both tables are super big (cannot fit into the memory), and not be partitioned, it will take some time to partition two tables across **different servers** (network shuffling), and then do sort merge at each partition
  - **Aggregate(take sum) of data**: if some categories are super big (have lot of records to aggregate), it will have **task straggler issue**, i.e. certain task takes much longer time than other task to complete
  - **group by, order by, bigger than, smaller than**: depending on the number of unique number of this key. The global sorting (a wide transformation requiring network shuffling) done by `orderBy` can also be a bottleneck
  - **SELECT FROM**: depending whether the dataframe has been in the RAM, if not, reading will incur potential I/O cost
- Event time and watermark:
  - Line 1: The data variable name that need to show
  - Line 2: 定义了 Watermark。Watermark 告诉系统可以容忍最多 5 分钟的事件延迟。也就是说，系统会等待至少 5 分钟以接收延迟的数据。
  - Line 3: 根据 `sensorID` 和一个时间窗口对事件进行分组。时间窗口基于 `eventTime` 字段，每个窗口持续 10 分钟，并且每 5 分钟滑动一次（也就是说窗口开始的间隔是 5 分钟）。
  - Line 4: 获得该时间段内事件的总数

```
(sensorReadings
.withWatermark("eventTime", "5 minutes")
.groupBy("sensorID", window("eventTime", "10 minutes", "
5 minutes"))
.count())
```

- Why PageRank with Teleport **converge faster** than PageRank without Teleport?
  - Teleport is designed to solve the spider trap and dead-ends problems in PageRank. For this the problem **without spider trap and dead-ends**, Teleport still helps the random walker to **occasionally jump to the less popular/important** nodes so as to **spread out the importance factors among all the nodes in the graph**. This helps the algorithm to converge faster