Flink CDC 高频面试 13 问

(万字长文吐血总结！)

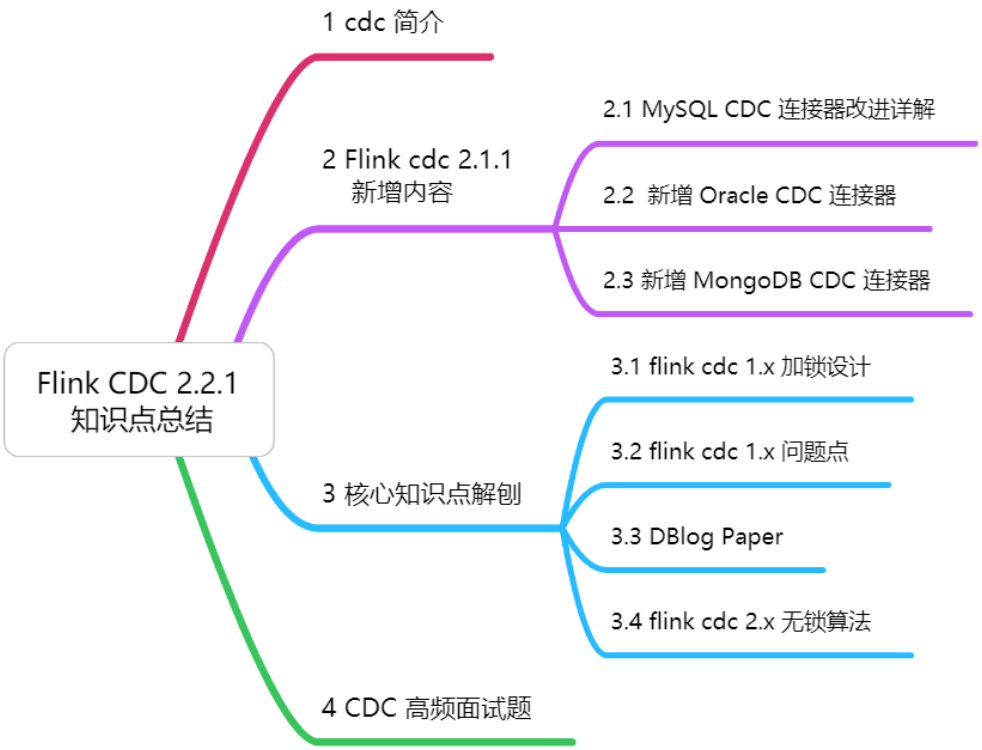
本文作者：在 IT 中穿梭旅行

本文档来自公众号：3 分钟秒懂大数据

大家好，我是土哥。

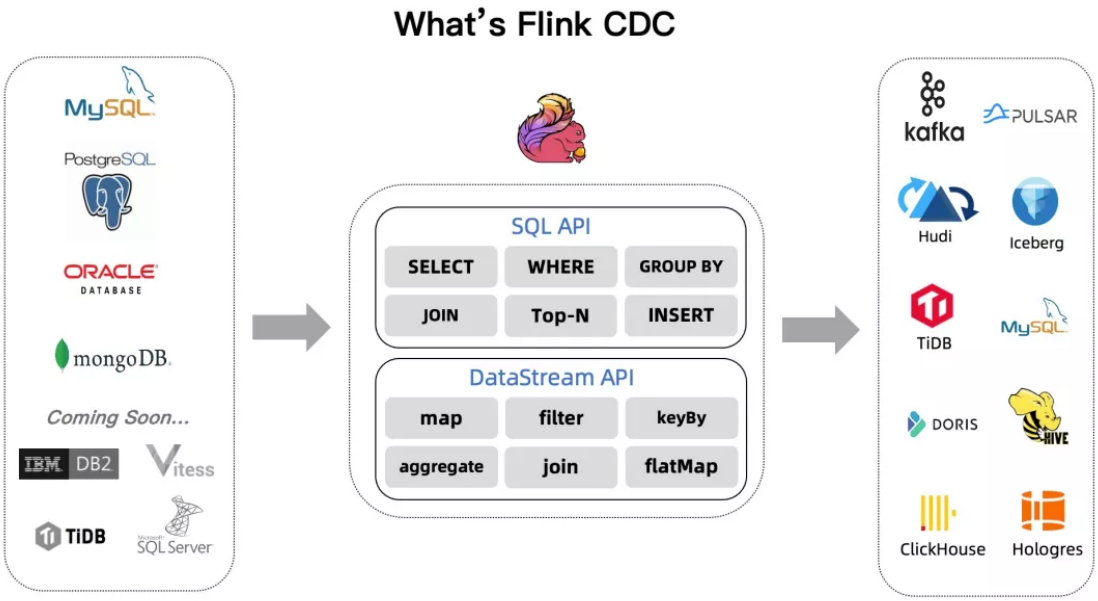
Flink cdc 2.1.1 发布后，更新了很多新功能以及知识点，今天为大家全面总结了 CDC 的知识点如 无锁算法及面试高频考点。具体内容如下：

- 1 cdc 简介
- 2 Flink cdc 2.1.1 新增内容
- 3 核心知识点解刨
- 4 CDC 高频面试题



1 cdc 简介

CDC (Change Data Capture) 是一种用于**捕捉数据库变更数据**的技术，Flink 从 1.11 版本开始原生支持 CDC 数据（changelog）的处理，目前已经是非常成熟的变更数据处理方案。



Flink CDC Connectors 是 Flink 的一组 Source 连接器，是 Flink CDC 的核心组件，这些连接器负责从 **MySQL、PostgreSQL、Oracle、MongoDB** 等数据库读取存量历史数据和增量变更数据。在 2020 年 7 月开源，社区保持了相当高速的发展，平均两个月一个版本，在开源社区的关注度持续走高，也逐渐有越来越多的用户使用 Flink CDC 来快速构建实时数仓和数据湖。

2 Flink cdc 2.1.1 新增内容

Flink CDC 2.1 版本重点提升了 `MySQL CDC` 连接器的性能和生产稳定性，重磅推出 `Oracle CDC` 连接器和 `MongoDB CDC` 连接器。新增内容如下：

（1）MySQL CDC 支持百亿级数据的超大表，**支持 MySQL 全部数据类型**，通过连接池复用等优化大幅提升稳定性。**同时提供支持无锁算法**，并发读取的 DataStream API，用户可以借此搭建整库同步链路；

（2）新增 Oracle CDC 连接器， 支持从 Oracle 数据库获取**全量历史数据**和**增量变更数据**；

（3）新增 MongoDB CDC 连接器，支持从 MongoDB 数据库获取**全量历史数据**和**增量变更数据**；

（4）所有连接器均支持 `metadata column` 功能， 用户通过 SQL 就可以访问库名，表名，数据变更时间等 meta 信息，这对分库分表场景的数据集成非常实用；

丰富 Flink CDC 入门文档，增加多种场景的端到端实践教程。

2.1 MySQL CDC 连接器改进详解

在 Flink CDC 2.0 版本里，MySQL CDC 连接器提供了**无锁算法，并发读取，断点续传**等高级特性, 一并解决了诸多生产实践上的痛点，随后大量用户开始投入使用并大规模上线。

在 Flink CDC 2.1 版本针对 MySQL CDC 连接器的改进主要包括两类：

1. 稳定性提升；
2. 功能增强。

(1) 稳定性提升

- 针对不同的主键分布，引入动态分片算法

对主键是 **非数值、Snowflake ID、稀疏主键、联合主键** 等场景，通过动态分析源表的主键分布的均匀程度，根据分布的均匀程度**自动地计算分片大小**，让切片更加合理，让分片计算更快。

动态分片算法能够很好地解决 **稀疏主键** 场景下 分片过多 的，**联合主键** 分片过大 等问题，让每个分片包含的行数尽量维持在用户指定的 chunk size，这样用户通过 chunk size 就能控制分片大小和分片数量，无需关心主键类型。

- 支持百亿级超大规模表

在表规模非常大时，以前会报 binlog 分片下发失败的错误。

这是因为在超大表对应的 `snapshot` 分片会非常多，而 binlog 分片需要包含所有 snapshot 分片信息，当 SourceCoordinator 下发 binglog 分片到 SourceReader 节点时，分片 size 超过 RPC 通信框架支持的最大 size 会导致分片下发失败。虽然可以通过修改 RPC 框架的参数缓解分片 size 过大问题，但无法彻底解决。

Flink cdc 2.1 版本里通过**将多个 snapshot 分片信息划分成 group 发送，一个 binlog 分片会切分成多个 group 逐个发送**，从而彻底解决该问题。

- 引入连接池管理数据库连接，提升稳定性

通过引入**连接池**管理数据库连接，一方面降低了数据库连接数，另外也避免了极端场景导致的连接泄露。

- 支持分库分表 `schema` 不一致时，缺失字段自动填充 `NULL` 值

(2) 功能增强

- 支持所有 `MySQL` 数据类型

包括 **枚举类型、数组类型、地理信息类型** 等复杂类型。

- 支持 `metadata column`

用户可以在 Flink DDL 中通过 db_name STRING METADATA FROM 'database_name' 的方式来访问库名(database_name)、表名（table_name）、变更时间（op_ts）等 meta 信息。这对分库分表场景的数据集成非常使用。

- 支持并发读取的 `DataStream API`

在 2.0 版本中，无锁算法，并发读取等功能只在 SQL API 上透出给用户，而 DataStream API 未透出给用户。

2.1 版本支持了 DataStream API，可通过 MySqlSourceBuilder 创建数据源。用户可以同时捕获多表数据，借此搭建整库同步链路。同时通过 MySqlSourceBuilder#includeSchemaChanges 还能捕获 schema 变更。

- 支持 `currentFetchEventTimeLag`，`currentEmitEventTimeLag`，`sourceIdleTime` 监控指标

这些指标遵循 FLIP-33 的连接器指标规范，可以查看 FLIP-33 获取每个指标的含义。其中，**currentEmitEventTimeLag 指标记录的是 Source 发送一条记录到下游节点的时间点和该记录在 DB 里产生时间点差值，用于衡量数据从 DB 产生到离开 Source 节点的延迟**。用户可以通过该指标判断 source 是否进入了 binlog 读取阶段：

即当该指标为 0 时，代表还在全量历史读取阶段； 当大于 0 时，则代表进入了 binlog 读取阶段。

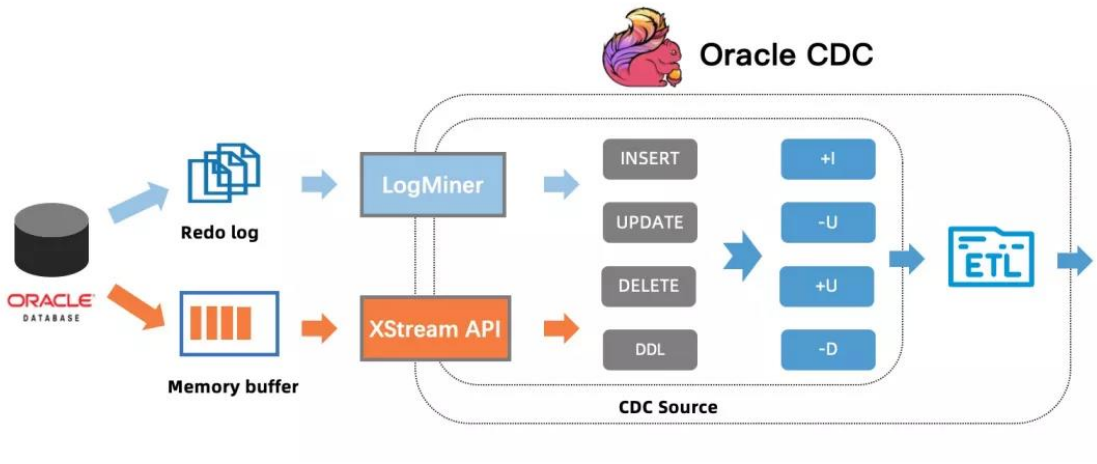
2.2 新增 Oracle CDC 连接器

Oracle 也是使用很广泛的数据库， Oracle CDC 连接器支持捕获并记录 Oracle 数据库服务器中发生的**行级变更**。

其原理是 **使用 Oracle 提供的 LogMiner 工具或者原生的 XStream API 从 Oracle 中获取变更数据**。

LogMiner 是 Oracle 数据库提供的一个分析工具，该工具可以解析 Oracle Redo 日志文件，从而将数据库的数据变更日志解析成变更事件输出。通过 LogMiner 方式时，Oracle 服务器对解析日志文件的进程做了严格的资源限制，所以对规模特别大的表，**数据解析会比较慢，优点是 LogMiner 是可以免费使用的**。

XStream API 是 Oracle 数据库为 Oracle GoldenGate (OGG) 提供的内部接口, 客户端可以通过 XStream API 高效地获取变更事件，其变更数据不是从 Redo 日志文件中获取，而是从 Oralce 服务器中的一块**内存**中直接读取，省去了数据落盘到日志文件和解析日志文件的开销，效率更高，但是必须购买 Oracle GoldenGate (OGG) 的 License。



Oracle CDC 连接器**支持 LogMiner 和 XStream API 两种方式**捕获变更事件。理论上能支持各种 Oracle 版本，**目前 Flink CDC 项目里测试了 Oracle 11，12 和 19 三个版本**。使用 Oracle CDC 连接器，用户只需要声明如下 Flink SQL 就能实时捕获 Oracle 数据库中的变更数据：

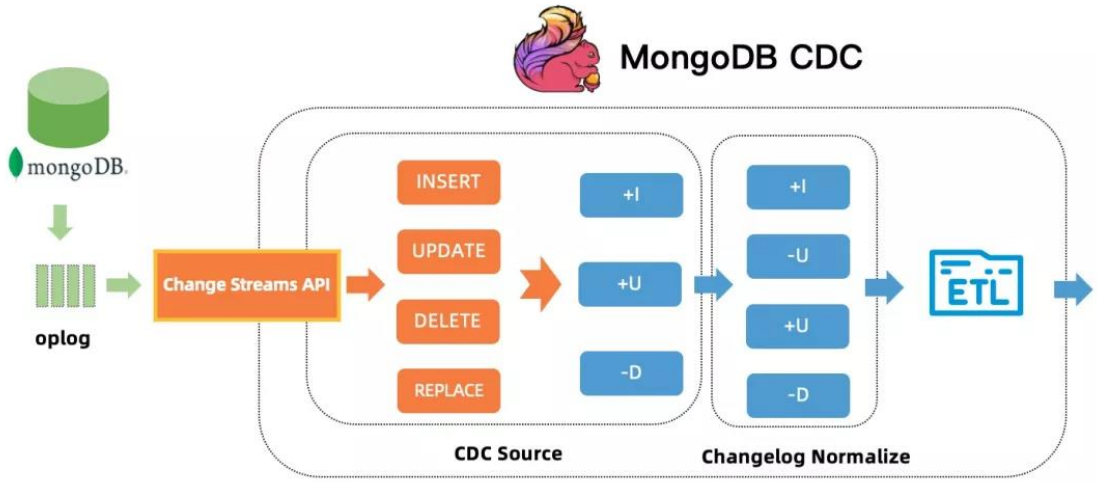
```
-- 在 Flink SQL 中声明一张 Oracle CDC 表: 'products'
Flink SQL> CREATE TABLE products (
     id INT NOT NULL,
     name STRING,
     description STRING,
     weight DECIMAL(10, 3),
     db_name STRING METADATA FROM 'database_name' VIRTUAL,   -- 获取 库名 元数据
     schema_name STRING METADATA FROM 'schema_name' VIRTUAL, -- 获取 模式名 元数据
     table_name STRING METADATA  FROM 'table_name' VIRTUAL,  -- 获取 表名 元数据
     PRIMARY KEY(id) NOT ENFORCED
 ) WITH (
     'connector' = 'oracle-cdc',
     'hostname' = 'localhost',
     'port' = '1521',
     'username' = 'flinkuser',
     'password' = 'flinkpw',
     'database-name' = 'XE',
     'schema-name' = 'inventory',
     'table-name' = 'products'
);

-- 从 products 表中读取 存量历史数据+ 增量变更数据
Flink SQL> SELECT * FROM products;
```

Oracle CDC 连接器已经将底层的 CDC 细节屏蔽，整个实时同步链路，用户只需要几行 Flink SQL，不用开发任何 Java 代码，就可以将 Oracle 的数据变更实时捕获并发送。

此外，Oracle CDC 连接器也提供两种工作模式，即读取 **全量数据 + 增量变更** 数据，和只读取增量变更数据。**Flink CDC 框架均保证一条不多一条不少的 exactly-once 语义**。

2.3 新增 MongoDB CDC 连接器

mongoDB CDC 连接器并不依赖 `Debezium`，是在 Flink CDC 项目里独立开发。MongoDB CDC 连接器支持捕获并记录 MongoDB 数据库中实时变更数据，其原理是伪装一个 MongoDB 集群里副本，利用 MongoDB 集群的高可用机制，该副本可以从 master 节点获取完整 oplog(operation log) 事件流。Change Streams API 则提供实时订阅这些 oplog 事件流的能力，可以将这些实时的 oplog 事件流推送给订阅的应用程序。



从 ChangeStreams API 获取的更新事件中，对于 update 事件，没有 update 事件的前镜像值，即 MongoDB CDC 数据源只能作为一个 upsert source。不过 Flink 框架会自动为 MongoDB CDC 附加一个 Changelog Normalize 节点，补齐 update 事件的前镜像值（即 UPDATE_BEFORE 事件)，从而确保 CDC 数据的语义正确性。

使用 MongoDB CDC 连接器，用户只需要声明如下 Flink SQL 就能实时捕获 MongoDB 数据库中的全量和增量变更数据，借助 Flink 强大的集成能力，用户可以非常方便地将 MongoDB 中的数据实时同步到 Flink 支持的所有下游存储。

```
-- 在 Flink SQL 中声明一张 MongoDB CDC 表: 'products'
CREATE TABLE products (
    _id STRING,          -- 文档id, 必要字段
    name STRING,
    weight DECIMAL(10,3),
    tags ARRAY<STRING>,  -- 数组类型
    price ROW<amount DECIMAL(10,2), currency STRING>,  -- 嵌套文档类型
    suppliers ARRAY<ROW<name STRING, address STRING>>, -- 嵌套文档类型
    db_name STRING METADATA FROM 'database_name' VIRTUAL,            -- 获取 库名 元数据
    collection_name STRING METADATA  FROM 'collection_name' VIRTUAL, -- 获取 集合名 元数据
    PRIMARY KEY(_id) NOT ENFORCED
) WITH (
    'connector' = 'mongodb-cdc',
    'hosts' = 'localhost:27017,localhost:27018,localhost:27019',
    'username' = 'flinkuser',
    'password' = 'flinkpw',
    'database' = 'inventory',
    'collection' = 'products'
);

-- 从 products 集合中读取 存量历史数据+ 增量变更数据
SELECT * FROM products;
```

整个数据捕获过程，用户不需要学习 MongoDB 的副本机制和原理，极大地简化了流程，降低了使用门槛。MongoDB CDC 也支持两种启动模式：

1. 默认的 initial 模式是**先同步表中的存量数据**，然后同步表中的增量数据；
2. latest-offset 模式则是从当前时间点开始只同步表中增量数据。

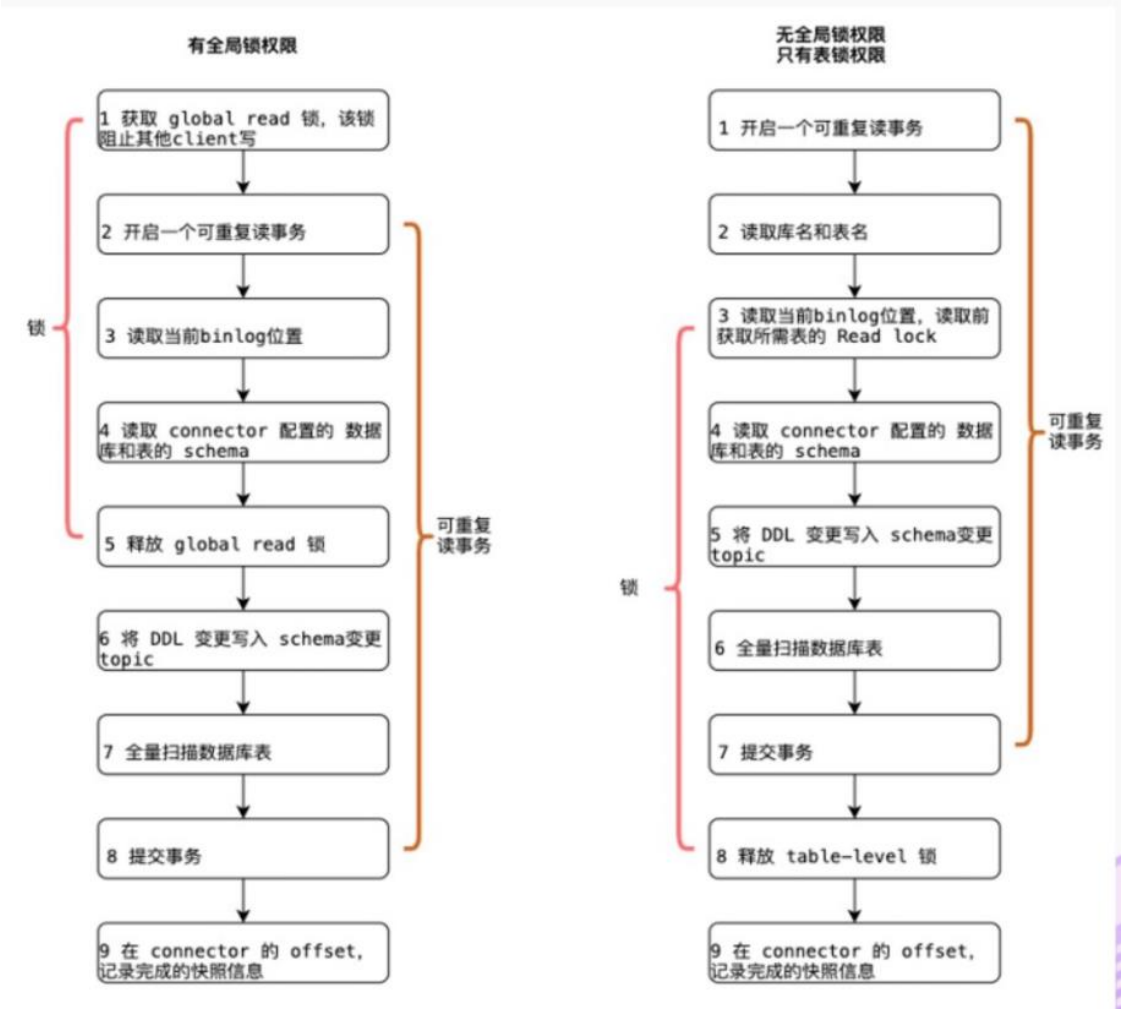
此外，MongoDB CDC 还提供了丰富的配置和优化参数，对于生产环境来说，这些配置和参数能够极大地提升实时链路的性能和稳定性。

3 核心知识点解刨

3.1 flink cdc 1.x 加锁设计

在 Flink cdc 1.x 全量 + 增量读取的版本设计中， flink cdc 底层选用 debezium 作为采集工具，Debezium 为保证数据一致性，通过对读取的数据库或者表进行加锁，**而加锁是发生在全量阶段。**

以全局锁为例，主要流程如下：



（1）首先是获取一个锁，**然后再去开启可重复读的事务**。

这里锁住操作是读取 `binlog` 的起始位置和当前表的 `schema`。

这样做的目的是**保证 binlog 的起始位置和读取到的当前 schema 可以一一对应**，因为表的 schema 是会改变的，比如删除列或者增加列。

在读取这两个信息后，`SnapshotReader` 会在**可重复读事务里读取全量数据**，在全量数据读取完成后，会启动 BinlogReader 从读取的 binlog 起始位置开始增量读取，**从而保证全量数据 + 增量数据的无缝衔接**。

3.2 flink cdc 1.x 问题点

当使用 `Flush tables with read lock` 语句时：

(1) 该命令会等待所有正在进行的 update 完成，同时阻止所有新来的 update。

(2) 该命令执行前必须等待所有正在运行的 select 完成，所有等待执行的 update 会等待更久。 更坏的情况是，在等待正在运行 select 完成时，DB 实际上处于不可用状态，即使是新加入的 SELECT 也会被阻止，这是 **Mysql Query Cache** 机制。

(3) 该命令阻止其他事务 commit。

结论： 加锁时间不确定，极端情况会锁住数据库。

3.3 DBlog Paper

针对 一致性加锁的痛点 Flink cdc 2.x 借鉴 Netflix 的 DBlog paper 设计了**全程无锁算法**

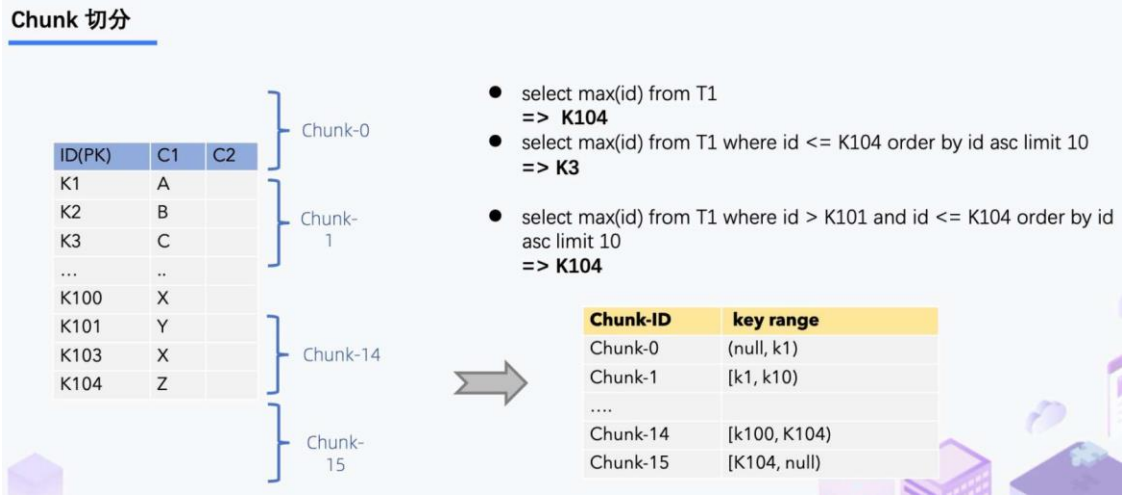
DBlog paper 论文的 chunk 切分算法

Algorithm 1: Watermark-based Chunk Selection

```
Input: table
(1) pause log event processing
    lw := uuid(), hw := uuid()
(2) update watermark table set value = lw
(3) chunk := select next chunk from table
(4) update watermark table set value = hw
(5) resume log event processing
    inwindow := false
    // other steps of event processing loop
    while true do
        e := next event from changelog
        if not inwindow then
            if e is not watermark then
                append e to outputbuffer
            else if e is watermark with value lw then
                inwindow := true
        else
            if e is not watermark then
(6)             if chunk contains e.key then
                    remove e.key from chunk
                append e to outputbuffer
            else if e is watermark with value hw then
(7)             for each row in chunk do
                    append row to outputbuffer
        // other steps of event processing loop
        ...
```

Chunk 切分算法其实和很多数据库的**分库分表**原理类似：**通过表的主键对表中的数据进行分片**。

假设每个 Chunk 的步长为 10，按照这个规则进行切分，只需要把这些 Chunk 的区间做成左开右闭或者左闭右开的区间，保证衔接后的区间能够等于表的主键区间即可。



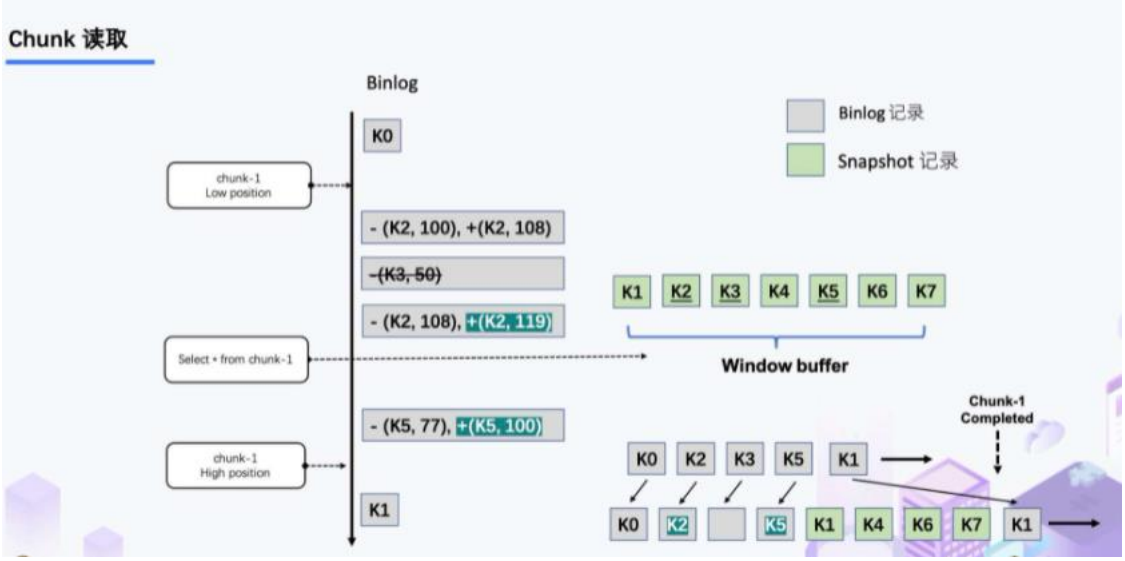
因为每个 chunk 只负责自己主键范围内的数据，不难推导，只要能够保证每个 Chunk 读取的一致性，就能保证整张表读取的一致性，这便是无锁算法的基本原理。

在 Netflix 的 DBLog 论文中

Chunk 读取算法是通过**在 DB 维护一张信号表**，再通过信号表在 binlog 文件中打点，记录每个 chunk 读取前的 Low Position (低位点) 和读取结束之后 High Position (高位点)，**在低位点和高位点之间去查询该 Chunk 的全量数据**。在读取出这一部分 Chunk 的数据之后，再将这 2 个位点之间的 binlog 增量数据合并到 chunk 所属的全量数据，从而得到高位点时刻，该 chunk 对应的全量数据。

3.4 flink cdc 2.x 无锁算法

Flink CDC 2.x 结合自身的情况，在 Chunk 读取算法上做了**去信号表的改进**，不需要额外维护信号表，**通过直接读取 binlog 位点替代在 binlog 中做标记的功能**，整体的 chunk 读算法描述如下图所示：



(1) 单个 Chunk 的一致性读:

比如正在读取 Chunk-1，Chunk 的区间是 [K1, K10]，首先直接将该区间内的数据 select 出来并把它存在 **buffer** 中，在 select 之前记录 binlog 的一个位点 (低位点)，select 完成后记录 binlog 的一个位点 (高位点)。**然后开始增量部分，消费从低位点到高位点的 binlog**。

- 图中的 - (k2,100) + (k2,108) 记录表示这条数据的值从 100 更新到 108；
- 第二条记录是删除 k3；
- 第三条记录是更新 k2 为 119；
- 第四条记录是 k5 的数据由原来的 77 变更为 100。

观察图片中右下角最终的输出，会发现在消费该 chunk 的 binlog 时，出现的 key 是 k2、k3、k5，我们前往 buffer 将这些 key 做标记。

对于 k1、k4、k6、k7 来说，在高位点读取完毕之后，这些记录没有变化过，所以这些数据是可以直接输出的；

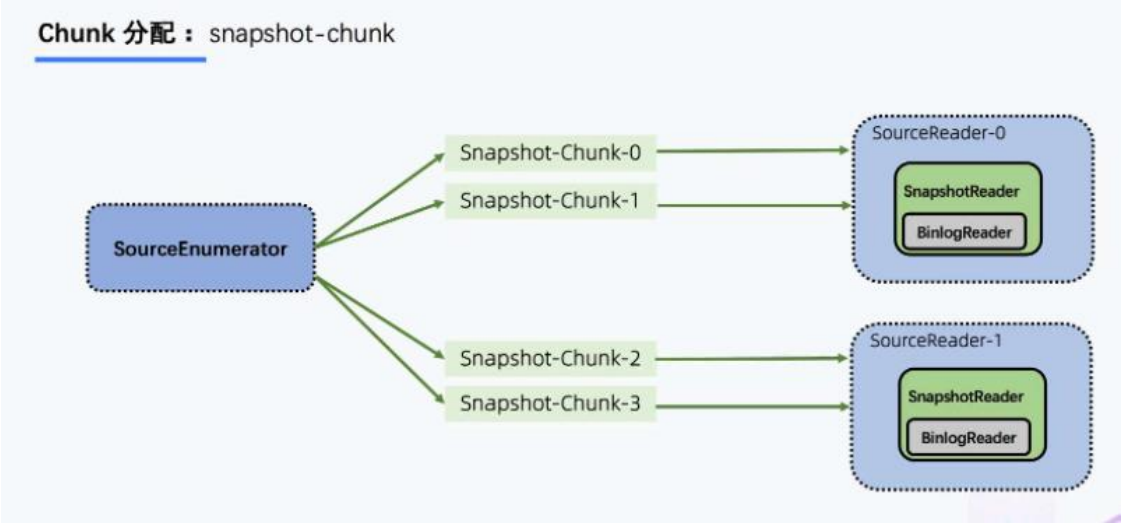
对于改变过的数据，则需要将增量的数据合并到全量的数据中，只保留合并后的最终数据

例如，**k2 最终的结果是 119** ，那么 **只需要输出 +(k2,119)**，而不需要中间发生过改变的数据。

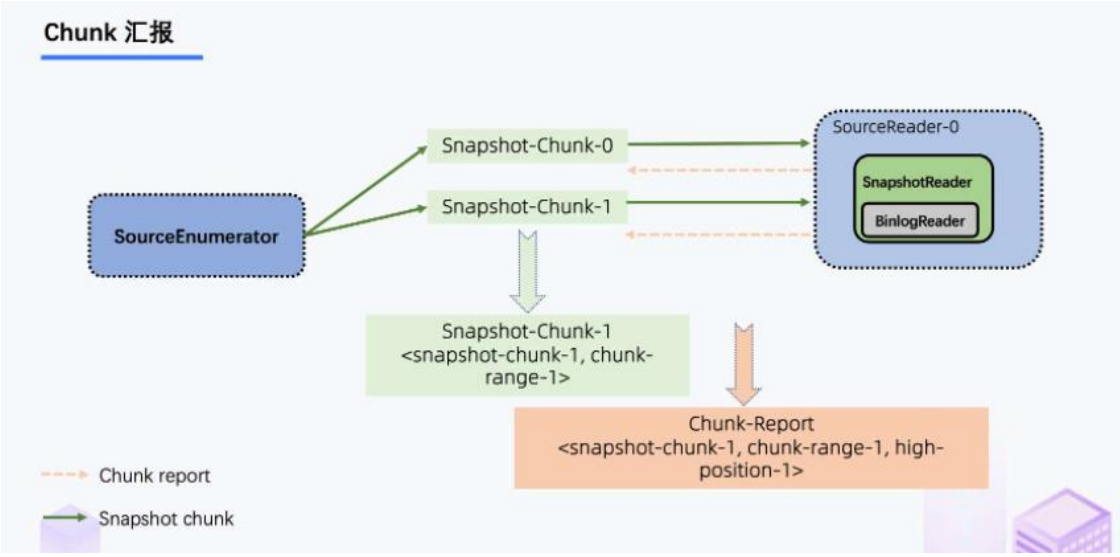
通过这种方式，Chunk 最终的输出就是在高位点是 chunk 中最新的数据。

(2) 并发读取 Snapshot Chunk

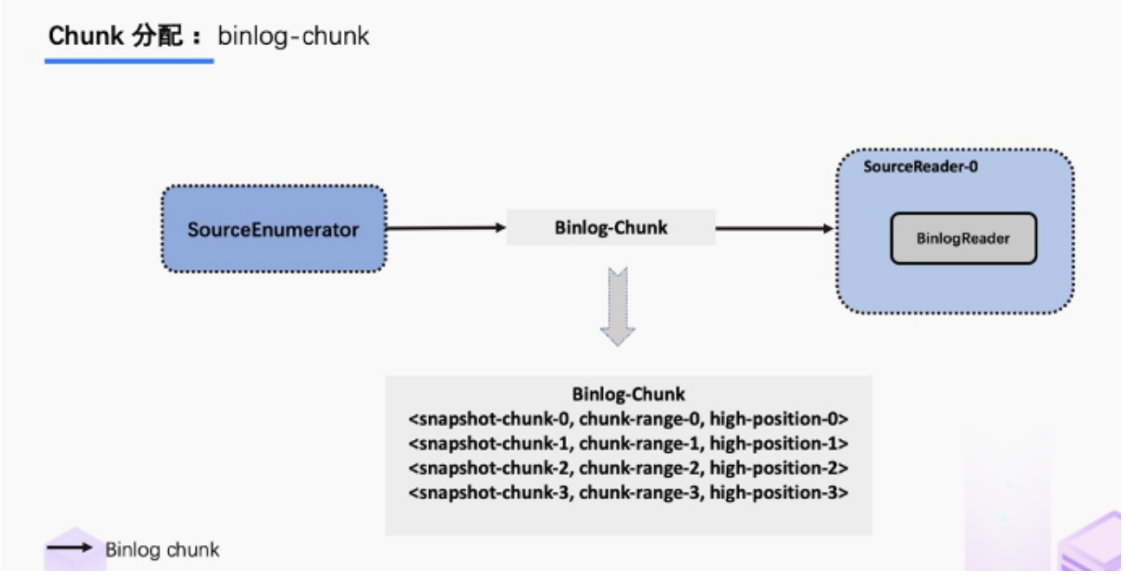
基于 FLIP-27 实现，通过下图可以看到有 SourceEnumerator 的组件，这个组件主要用来划分 chunk,将划分好的 Chunk 提供给下游的快照读取器（SourceReader）去读取，**通过把 chunk 分发给不同的 SourceReader 便实现了并发读取 Snapshot Chunk 的过程，** 同时基于 FLIP-27 方便地做到 chunk 粒度的 checkpoint。



当 Snapshot Chunk 读取完成之后，需要有一个汇报的流程，如下图中橘色的汇报信息，将 Snapshot Chunk 完成信息汇报给 SourceEnumerator。



汇报的主要目的是为了后续分发 binlog chunk (如下图)。因为 Flink CDC 支持全量 + 增量同步，**所以当所有 Snapshot Chunk 读取完成之后，还需要消费增量的 binlog**，这是通过下发一个 binlog chunk 给任意一个 Source Reader 进行单并发读取实现的。



总结如下：

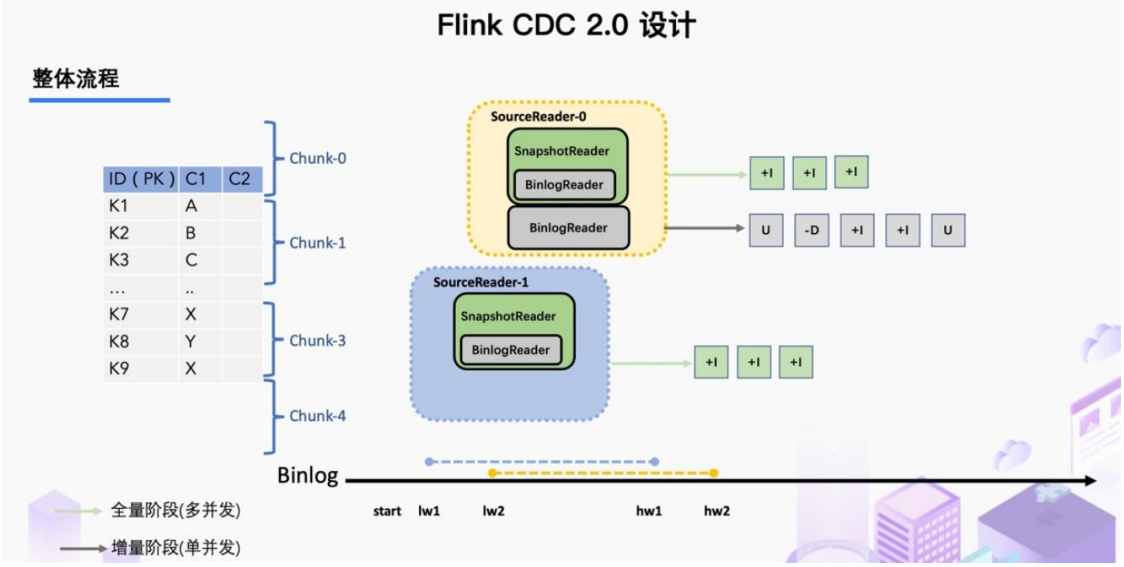
（1）在快照阶段，根据**表的主键和表行的大小**将 快照（Snapshot） 切割成多个 快照块(Snapshot Chunk) ，然后将 快照块 被分配给多个 快照读取器（SourceReader）。

（2）每个 快照读取器 使用 块读取算法（**单个 Chunk 的一致性读**） 读取其接收到的块，并将读取的数据发送到下游。源管理块的进程状态变更为（已完成或未完成），**因此 快照阶段的源可以支持块级别的检查点。如果发生故障，可以恢复源并继续从最后完成的块中读取块。**

（3）在所有快照块完成后，源将继续在单个任务（task）中读取 binlog。为了保证快照记录和 binlog 记录的全局数据顺序，binlog reader 会开始读取数据，直到 snapshot chunks 完成后有一个完整的 checkpoint，以确保所有的快照数据都被下游消费了。

（4）binlog reader 在 state 中跟踪消耗的 binlog 位置，因此 binlog phase 的 source 可以支持行级别的 checkpoint。

Flink 定期为源执行检查点，在故障转移的情况下，作业将从上次成功的检查点状态重新启动并恢复，并保证恰好一次语义。

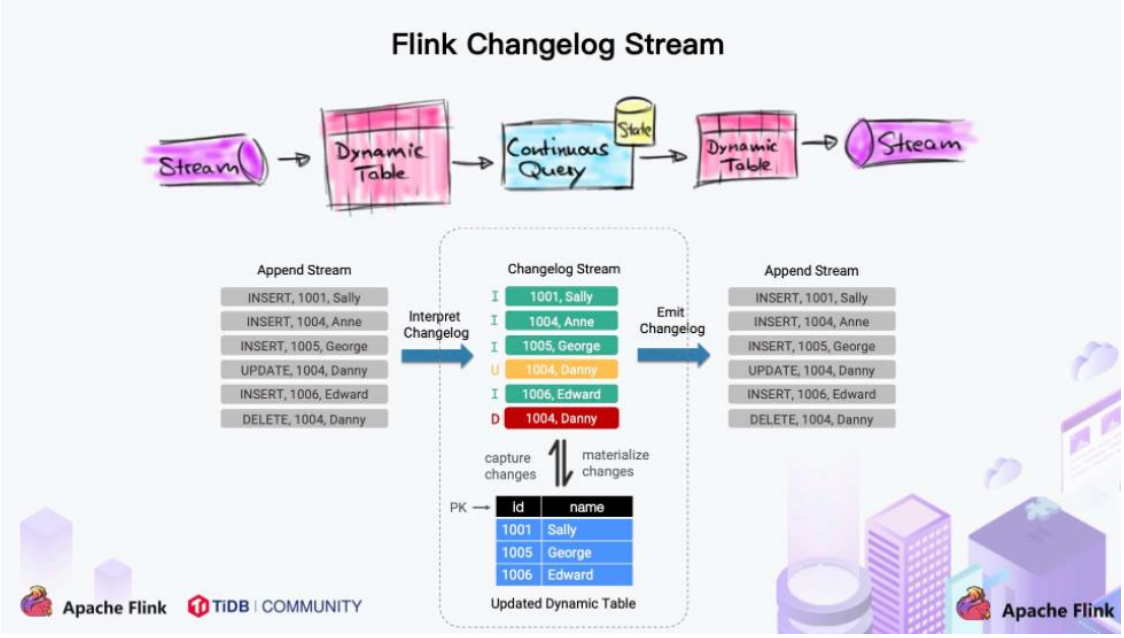


4 CDC 高频面试题

0 flink Dynamic Table & ChangeLog Stream 了解吗？

Dynamic Table 是 Flink SQL 定义的动态表，动态表和流的概念是对等的参照上图，流可以转换成动态表，动态表也可以转换成流。

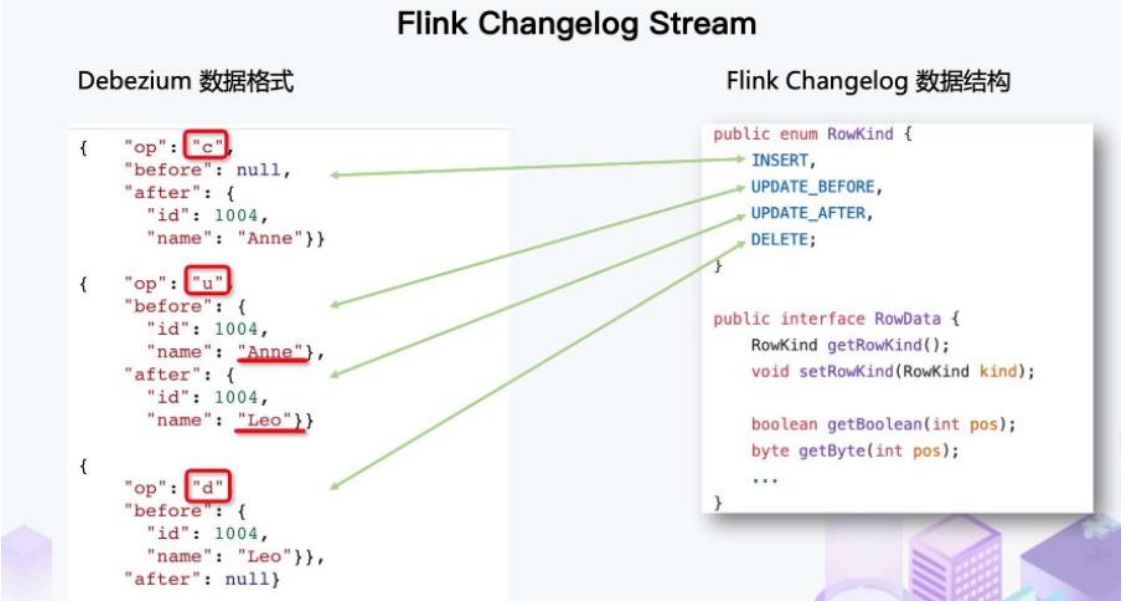
在 Flink SQL 中，数据从一个算子**以 Changelog Stream 的形式**流向另外一个算子时，任意时刻的 Changelog Stream 可以翻译为一个表，也可以翻译为一个流。



1 mysql 表与 binlog 的关系是什么?

MySQL 数据库的一张表**所有的变更**都记录在 binlog 日志中，**如果一直对表进行更新，binlog 日志流也一直会追加**，数据库中的表就相当于 binlog 日志流在某个时刻点物化的结果；**日志流就是将表的变更数据持续捕获的结果**。

这说明 Flink SQL 的 Dynamic Table 是可以非常自然地表示一张不断变化的 MySQL 数据库表。



2 flink cdc 底层的采集工具用哪个？

选择 Debezium 作为 Flink CDC 的底层采集工具，原因是 debezium 支持**全量同步**，也支持增量同步，同时也支持全量 + 增量的同步，非常灵活，同时基于日志的 CDC 技术使得提供 Exactly-Once 成为可能。

3 flink sql 与 debezium 的数据结构有哪些相似性？

通过对 Flink SQL 的内部数据结构 RowData 和 Debezium 的数据结构进行对比，可以发现两者非常相似。

（1）每条 RowData 都有一个**元数据 RowKind**，包括 4 种类型， 分别是插入 (INSERT)、更新前镜像 (UPDATE_BEFORE)、更新后镜像 (UPDATE_AFTER)、删除 (DELETE)，这四种类型和数据库里面的 binlog 概念保持一致。

（2）Debezium 的数据结构，也有一个类似的**元数据 op 字段**， op 字段的取值也有四种，分别是 c、u、d、r，各自对应 create、update、delete、read。对于代表更新操作的 u，其数据部分同时包含了前镜像 (before) 和后镜像 (after)。

两者相似性很高，所以采用 debezium 作为底层采集工具

4 flink cdc 1.x 有哪些痛点？

(1) 一致性加锁的痛点

由于 flink cdc 底层选用 debezium 作为采集工具，在 flink cdc 1.x 全量 + 增量读取的版本设计中，Debezium 为保证数据一致性，通过对读取的数据库或者表进行加锁，但是 **加锁** 在数据库层面上是一个十分高危的操作。全局锁可能导致数据库**锁住**，表级锁会**锁住表的读**，**DBA 一般不给锁权限**。

（2）不支持水平扩展的痛点

因为 Flink CDC 底层是基于 Debezium，Debezium 架构是单节点，所以 Flink CDC 1.x **只支持单并发**。

在**全量读取阶段**，如果表非常大 (亿级别)，**读取时间在小时甚至天级别**，用户不能通过增加资源去提升作业速度。

（3）全量读取阶段不支持 checkpoint

Flink CDC 读取分为两个阶段，**全量读取和增量读取**，目前全量读取阶段是不支持 checkpoint 的;

因此会存在一个问题：当我们同步全量数据时，假设需要 5 个小时，**当我们同步了 4 小时的时候作业失败，这时候就需要重新开始，再读取 5 个小时**。

5 flink cdc 1.x 的加锁发生在哪个阶段？

加锁是发生在全量阶段。

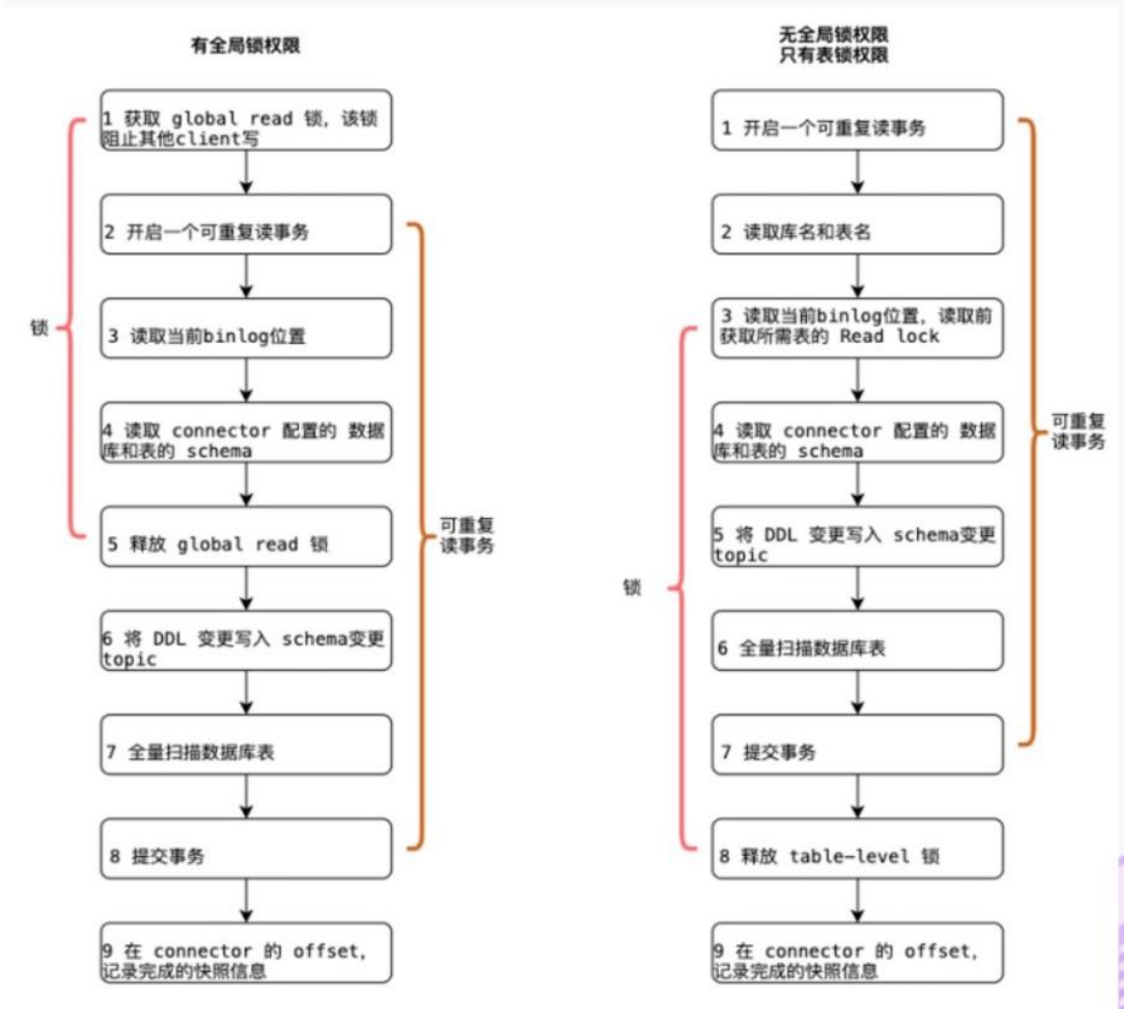
Flink CDC 底层使用 Debezium 同步一张表时分为两个阶段：

- 全量阶段：查询当前表中所有记录；
- 增量阶段：从 binlog 消费变更数据。

大部分用户使用的场景都是全量 + 增量同步，**加锁是发生在全量阶段，目的是为了确定全量阶段的初始位点**，保证增量 + 全量实现一条不多，一条不少，从而保证数据一致性。

6 flink cdc 1.x 全局锁加锁的流程

从下图中我们可以分析全局锁和表锁的一些加锁流程，**左边红色线条是锁的生命周期**，右边是 MySQL 开启可重复读事务的生命周期。



以全局锁为例：

（1）首先是获取一个锁，**然后再去开启可重复读的事务**。

这里锁住操作是读取 `binlog` 的起始位置和当前表的 `schema`。

这样做的目的是**保证 binlog 的起始位置和读取到的当前 schema 可以一一对应**，因为表的 schema 是会改变的，比如删除列或者增加列。

在读取这两个信息后，`SnapshotReader` 会在**可重复读事务里读取全量数据**，在全量数据读取完成后，会启动 BinlogReader 从读取的 binlog 起始位置开始增量读取，**从而保证全量数据 + 增量数据的无缝衔接**。

表级锁有个特征：锁提前释放了可重复读的事务默认会提交，所以锁需要等到全量数据读完后才能释放。

7 **全局锁**会造成怎样的后果？

举例： 当使用 Flush tables with read lock 语句时：

(1) 该命令会等待所有正在进行的 update 完成，同时阻止所有新来的 update。

(2) 该命令执行前必须等待所有正在运行的 select 完成，所有等待执行的 update 会等待更久。 更坏的情况是，在等待正在运行 select 完成时，DB 实际上处于不可用状态，即使是新加入的 SELECT 也会被阻止，这是 **Mysql Query Cache** 机制。

(3) 该命令阻止其他事务 commit。

结论： 加锁时间不确定，极端情况会锁住数据库。

8 Netflix 的 DBLog paper 核心设计描述一下?

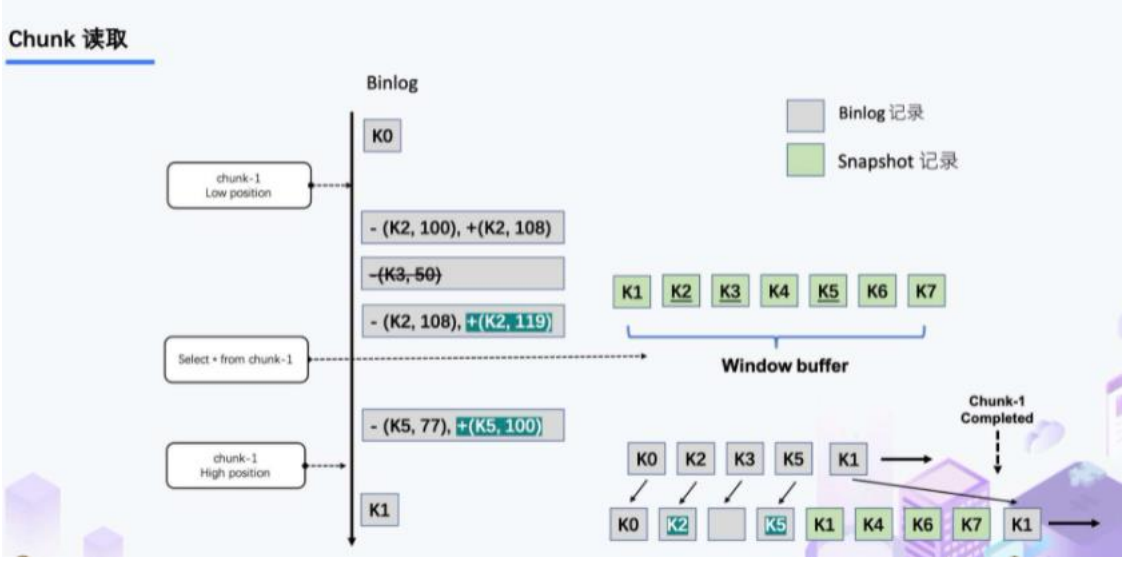
在 Netflix 的 DBLog 论文中

Chunk 读取算法是通过**在 DB 维护一张信号表**，再通过信号表在 binlog 文件中打点，记录每个 chunk 读取前的 Low Position (低位点) 和读取结束之后 High Position (高位点)，**在低位点和高位点之间去查询该 Chunk 的全量数据**。在读取出这一部分 Chunk 的数据之后，再将这 2 个位点之间的 binlog 增量数据合并到 chunk 所属的全量数据，从而得到高位点时刻，该 chunk 对应的全量数据。

9 flink cdc 2.x 如何设计的无锁算法？

Flink CDC 2.x 结合自身的情况，在 Chunk 读取算法上做了**去信号表的改进**，不需要额外维护信号表，**通过直接读取 binlog 位点替代在 binlog 中做标记的功能**，整体的 chunk 读算法描述如下图所示：

单个 Chunk 的一致性读:



(1) 记录当前 binlog 位置为 LOWoffset

(2)通过执行语句读取并缓存快照 chunk 记录 SELECT * FROM MyTable WHERE id > chunk_low AND id <= chunk_high

(3) 记录当前 binlog 位置作为 HIGH 偏移量

(4)从 LOWoffset 到 HIGHoffset 读取属于 snapshot chunk 的 binlog 记录

(5) 将读取到的 binlog 记录 Upsert 到缓冲的 chunk 记录中，将 buffer 中的所有记录作为 snapshot chunk 的最终输出（都作为 INSERT 记录）发出。

(6)HIGH 在 single binlog reader 中继续读取并发出属于 offset 之后的 chunk 的 binlog 记录。

> 10 如果有多个表分了很多不同的 Chunk，且这些 Chunk 分发到了不同的 task 中，那么如何分发 Chunk 并保证全局一致性读呢？

在快照阶段，根据**表的主键和表行的大小**将 快照（Snapshot） 切割成多个 快照块 (Snapshot Chunk) 。快照块 被分配给多个 快照读取器（SourceReader）。每个 快照读取器 使用 块读取算法（单个 Chunk 的一致性读） 读取其接收到的块，并将读取的数据发送到下游。源管理块的进程状态（已完成或未完成），**因此 快照阶段的源可以支持块级别的检查点。如果发生故障，可以恢复源并继续从最后完成的块中读取块。**

在所有快照块完成后，源将继续在单个任务（task）中读取 binlog。为了保证快照记录和 binlog 记录的全局数据顺序，binlog reader 会开始读取数据，直到 snapshot chunks 完成后有一个完整的 checkpoint，以确保所有的快照数据都被下

游消费了。binlog reader 在 state 中跟踪消耗的 binlog 位置，因此 binlog phase 的 source 可以支持行级别的 checkpoint。

Flink 定期为源执行检查点，在故障转移的情况下，作业将从上次成功的检查点状态重新启动并恢复，并保证恰好一次语义。