Deduplication: An Effective Alternative to Transaction for Eventually Consistent Use Cases of a Distributed Database

Building a distributed database is complicated and needs to consider many factors. Previously, I introduced two important techniques: sharding and partitioning for gaining throughput and performance. In this post, I will introduce another important technique, **deduplication**, that can be used to replace **transactions** for eventually consistent use cases with defined **primary keys**.

Introduction

Timeseries databases such as InfluxDB provide ease of use for clients and accept ingesting the same data more than once. For example, edge devices can just send their data on reconnection without having to remember which parts were successfully transmitted previously. To return correct results in such scenarios, timeseries databases often apply deduplication to arrive at an eventually consistent view of the data. For classic transactional systems, the deduplication technique may not be obviously applicable but it actually is. Let us step through examples to understand how this works.

Transaction

Data insert and update are usually performed in an atomic commit which is an operation that applies a set of distinct changes as a single operation. The changes are either all successful or all aborted, there is no middle ground. The atomic commit in the database is called a transaction. Implementing a transaction needs to include recovery activities that redo and/or undo changes to ensure a transaction is either completed or completely aborted in case of incidents in the middle of the transaction. A typical example of changes that need to be done in a transaction is money transferring between two accounts in which either money is withdrawn from one account and deposited to another account successfully or no money changes hands at all.

In a distributed database, transaction implementation is even more complicated due to the need to communicate between nodes and tolerate various communication problems. Paxos and Raft are common techniques used to build transactions and are well known for their complexity.

Figure 1 shows an example of a money transferring system that uses a transactional database. When a user uses a bank system to transfer $100 from account A to account B, the bank initiates a transferring job that starts a transaction of two changes: withdraw $100 from A and deposit $100 to B. If the two changes all succeed, the process will finish and the job is done. If for some reason, the withdrawal and/or deposit cannot be performed, all changes in the system so far will get aborted and a signal is sent back to the job telling it to re-start the transaction. A and B only see the withdrawal and deposit respectively if the process is finished. Otherwise, nothing will happen to their accounts.

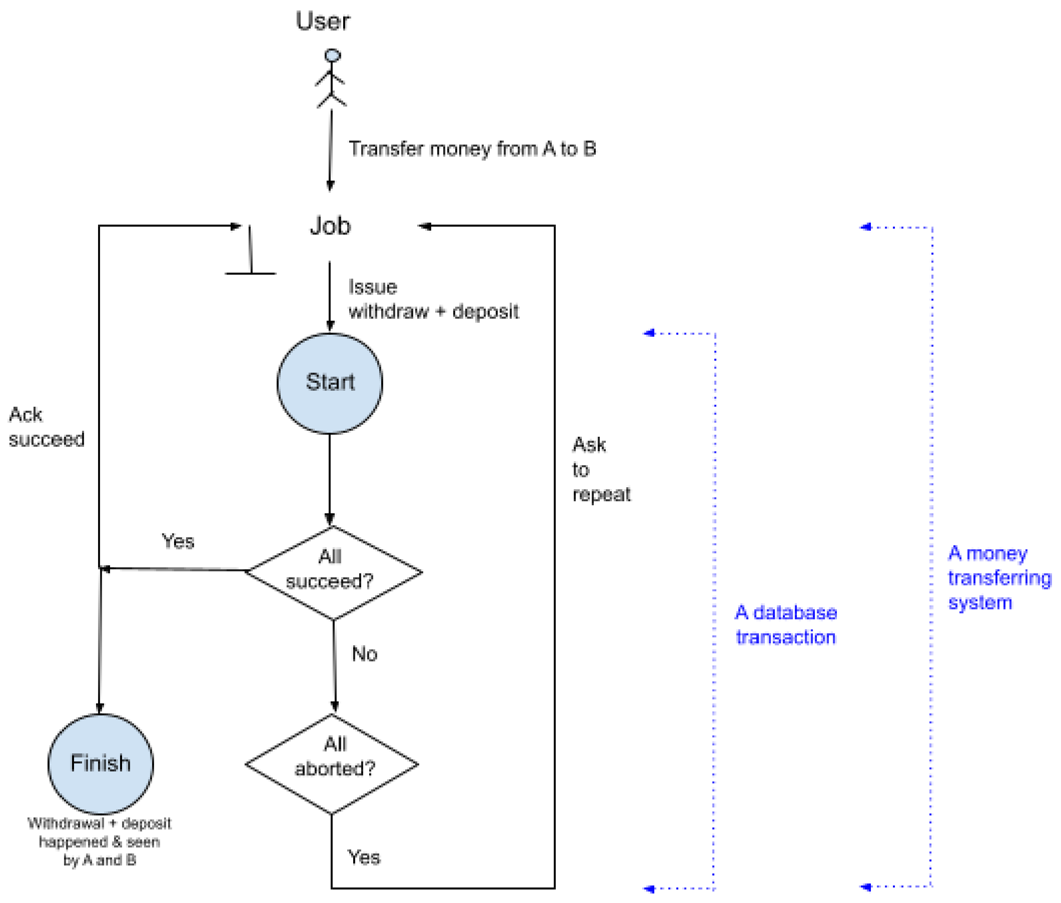

Figure 1: Transactional flow

Non-Transactional Process

Since the transactional process is complicated to build and maintain, the system can be simplified as illustrated in Figure 2. First, the job also issues withdrawal and deposit. If the two changes succeed, the job completes. If none or only one of the two changes happens; or error or timeout happens, data will be in a middle-ground state; and the job is asked to repeat the withdrawal and deposit. If none or only one of the two changes happens; or error or timeout happens during the process, data will be in “Acceptable Finish” state. The job will automatically restart issuing withdrawal and deposit at timeout.

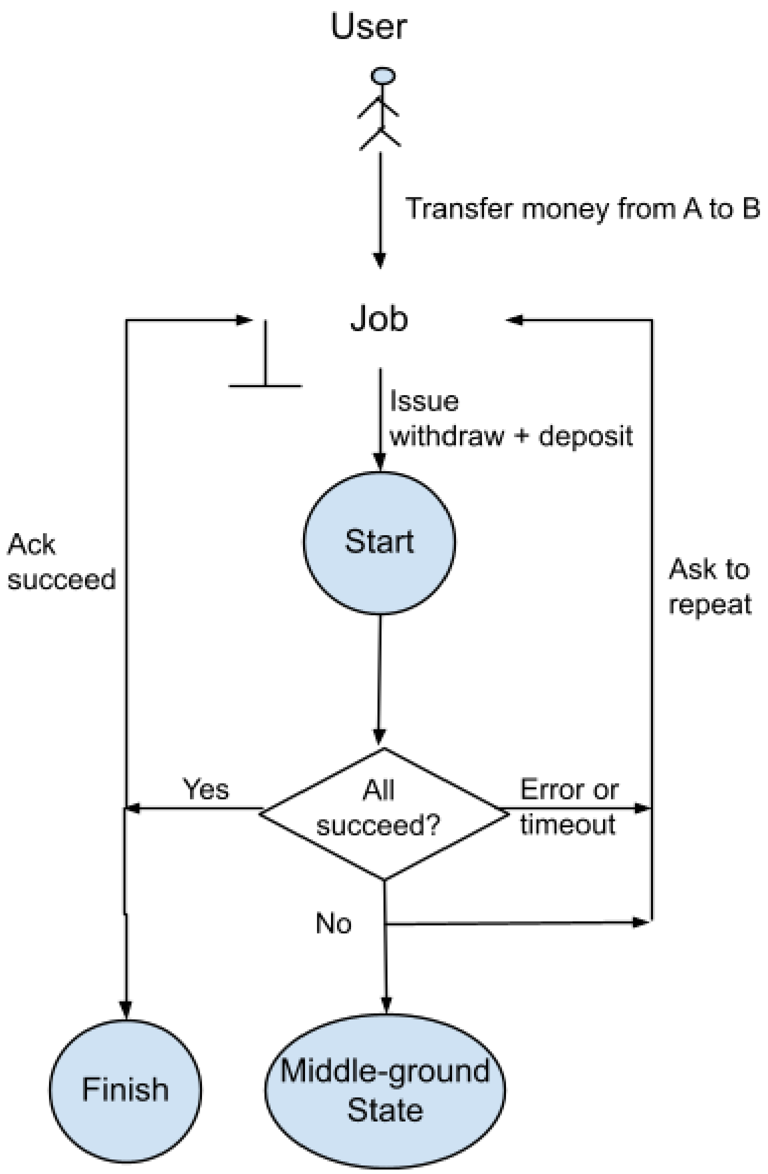

Figure 2: Non-transactional flow

The data outcomes at the “middle-ground state” can be different for various restarts on the same transfer but they are acceptable to be in the system as long as the finish state will eventually happen. Let us go over an example to show these outcomes and explain why they are acceptable. Table 1 shows two expected changes of withdrawal and deposit of the transaction if it is successful, each include four fields:

1. **AccountID** that uniquely identifies an account
2. **Activity** that is either withdrawal or deposit
3. **Amount** that is the amount of money to withdraw or deposit
4. **BankJobID** which is the ID uniquely identifies a job in a system.

AccountID	Activity	Amount	BankJobID

A	Withdrawal	100	543
B	Deposit	100	543

Table 1: Two changes of the transferring transaction

At each repeat of issuing the withdrawal and deposit of Figure 2, there are four possible outcomes:

1. No changes
2. Only A is withdrawn
3. Only B is deposited
4. Both A is withdrawn and B is deposited

For this example, let us say it takes four repeats for the job to get back the successful signal. The first repeat produces "only B is deposited", hence the system has only one change shown in Table 2. The second repeat produces nothing. The third repeat produces "only A is withdrawn", hence the system now has 2 rows described in Table 3. The fourth repeat produces both changes and makes the system data in the finish state and look like Table 4.

AccountID	Activity	Amount	BankJobID
B	Deposit	100	543

Table 2: Data in the system after the first repeat and second repeat

AccountID	Activity	Amount	BankJobID
B	Deposit	100	543
A	Withdrawal	100	543

Table 3: Data in the system after the third repeat

AccountID	Activity	Amount	BankJobID
B	Deposit	100	543
A	Withdrawal	100	543
A	Withdrawal	100	543
B	Deposit	100	543

Table 4: Data in the system after the fourth repeat and now in finish state

Data Deduplication

The four-repeat example above shows, during the job, there are three different data sets in the system as shown in Tables 2, 3, and 4. Why do we say they are acceptable? The answer is data in the system can be redundant and as long as we can identify the redundant information and eliminate them at read time, we will be able to produce the expected result. In this example, we say that the combination of AccountID, Activity, and BankJobID uniquely identifies a change and is called a key. If there are many changes with the same key, only one of them is returned during read time. The process to eliminate redundant information is called **deduplication**. Therefore, when we read and deduplicate data from Tables 3 and 4, we will get the same returned values which are the expected outcome shown in Table 1.

In the case of Table 2 that includes only one change, the returned value will be that change which is only a part of the expected outcome of Table 1. This means we do not get strong transactional guarantees but if we are willing to wait to reconcile the accounts, we will get the expected outcome. In real life, banks do not release the transferred money for us to use immediately even if we see it. In other words, this partial change is acceptable if the bank only has the transferred money available to use after a day or two. Since the process is repeated until it is successful, a day is more than enough to have that happen.

The combination of the non-transactional insert process shown in Figure 2 and data deduplication at read time provide us the results that may not be the same as the expected immediately but eventually the results will be the same as expected. This is called an ***eventually consistent system***. On the contrary, the transactional system illustrated in Figure 1 always produces consistent results but due to the complicated communication to guarantee the consistency, a transaction does take time to finish and the number of transactions per second is limited

Update in the System that Supports Deduplication

Nowadays, most databases implement an update as a delete and then an insert to avoid the expensive in-place data modification. However, if the system supports deduplication, the update can just be done as an insert if we add a field “Sequence” in the table to identify the order when data comes into the system. For example, after making the money transfer successfully as shown in Table 5, we found the amount should be $200 instead. This can be fixed by making a new transfer with the same BankJobID but higher Sequence number shown in Table 6. At read time, the deduplication will only return rows with the highest sequence number of data that have the same key, which means the data rows with amount $100 are never returned.

AccountID	**Activity**	**Amount**	**BankJobID**	**Sequence**
B	Deposit	100	543	1
A	Withdrawal	100	543	1

Table 5: Data before the “update”

AccountID	**Activity**	**Amount**	**BankJobID**	**Sequence**

B	Deposit	100	543	1
A	Withdrawal	100	543	1
A	Withdrawal	200	543	2
B	Deposit	200	543	2

Table 6: Data after the “update”

Deduplication Algorithm

Since deduplication has to compare data to look for rows with the same key, organizing data and implementing the right deduplication algorithms are critical. The common technique is having data that may duplicate with each other sorted on their keys and using a merge algorithm to find and deduplicate them. The details of how data is organized and merged depend on the nature of the data, their size and the available memory in the system. This is an example how Apache Arrow implements multi-column sort merge that is critical to perform effective deduplication.

Avoid Deduplication during Read Time

Deduplication during read time will increase the time needed to query data. To improve this, the deduplication can be done as a background task to remove redundant data ahead. Most systems already have background jobs to reorganize data such as removing data that was previously marked to-be-deleted. The deduplication fits very well in that model that reads data, deduplicates or removes them, and writes the result back. In order to avoid sharing CPU and memory resources with data loading and reading, these background jobs are usually performed in a separate server named Compactor, which is another large topic and deserves its own post.