Introduction

A known challenge in the current EoS design for the consumer's READ_COMMITTED isolation mode is the need to buffer data in the consumer until the status of a transaction has been determined. Because the COMMIT and ABORT markers are always the final messages in the transaction, the consumer must retain all messages from the transaction until that marker is read. This is particularly problematic for longer transactions or in situations with many concurrent transactions. If the data from a transaction cannot be retained in memory, it must either be persisted or re-fetched after the transaction's outcome is determined

One way to address this problem is to remove aborted data from the log prior to returning it to the client. Different approaches for this have been discussed on the dev list including Jun's "shadow log" idea. In this document, we propose an alternative. The reason the client must buffer the data is because it does not know the outcome of the transaction. If it had another way to determine the transaction outcome prior to reading the data, then the buffering would be unnecessary.

From a high level, the idea is to maintain a separate index to keep track of the aborted transactions. When a partition leader receives an `UpdateTxn` request from the transaction coordinator with the ABORT status set, it writes an entry to this index containing the corresponding PID and the range of offsets contained in the transaction. When the consumer fetches data in READ_COMMITTED mode, in addition to returning the data from the log, we return the corresponding entries from the aborted transaction index for the range of data in the log. This effectively gives the consumer the capability to "look ahead" to tell which transactions have been aborted. Any transaction not included in this list can then be assumed committed. Since all transactions are decided one way or another, there is no need for consumer buffering, and we retain the full benefit of "zero copy" transfer from the data log.

Below we provide the lower level details for this approach:

Preliminaries

LSO Tracking

In this proposal, we require the broker to track the LSO (last stable offset) for each partition. To do this, the broker must maintain in memory the set of active transactions along with their initial offsets. The LSO is always equal to one less than the minimum of the initial offsets across all active transactions.

As in one of the previous proposals, we adopt the behavior that messages from the log are only visible to consumers in READ_COMMITTED mode if they have a lower offset than the current LSO.

Aborted Transaction Index

Each log segment for a partition is given a separate append-only file to serve as an index for all aborted transactions which finished in the corresponding segment of the log. This file is created lazily upon the first aborted transaction for a log segment. We assume generally that aborted transactions are rare, so this file should stay small.

The schema for the entries in this index is the following:

```
TransactionEntry =>
  Version => int16
  PID => int64
  FirstOffset => int64
  LastOffset => int64
  LastStableOffset => int64
```

When the broker receives an `UpdateTxn` request, it must:

1. Compute the new LSO.
2. Check the transaction status. If the transaction was aborted, insert the corresponding entry with the updated LSO.
3. Remove the transaction from the active transaction set and update the LSO

The reason for including the LSO is explained below. Basically it allows us to easily find the range of commit entries corresponding to a given range of the data log.

Updated FetchRequest

We extend the FetchRequest schema to include a flag to indicate the desired isolation mode (READ_COMMITED, READ_UNCOMMITTED):

```
FetchRequest => ReplicaId MaxWaitTime MinBytes [TopicName [Partition
FetchOffset MaxBytes]]
  ReplicaId => int32
  MaxWaitTime => int32
  MinBytes => int32
  TopicName => string
  Partition => int32
  FetchOffset => int64
```

```
  MaxBytes => int32
  Isolation => READ_COMMITTED | READ_UNCOMMITTED
```

We also extend the FetchResponse to return the entries from the aborted transaction index corresponding to the fetched range:

```
FetchResponse => ThrottleTime [TopicName [Partition ErrorCode
HighwaterMarkOffset AbortedTransactions MessageSetSize MessageSet]]
  ThrottleTime => int32
  TopicName => string
  Partition => int32
  ErrorCode => int16
  HighwaterMarkOffset => int64
  AbortedTransactions => [PID FirstOffset]
    PID => int64
    FirstOffset => int64
  MessageSetSize => int32
```

As mentioned above, the fetch behavior is changed so that in READ_COMMITTED mode, only records with lower offsets than the current LSO are returned. When a consumer in READ_COMMITTED mode receives a fetch response, it caches the list of aborted transactions and proceeds to return committed data to the user and filter aborted data.

Note that we have only included the initial offset of each aborted transaction. We depend on the abort marker itself to determine the end of the transaction. This is described in more detail below.

In READ_UNCOMMITTED mode, the `AbortedTransactions` array is null.

Broker Fetch Handling

Below we outline the algorithm for fetching in READ_COMMITED mode:

1. Fetch the data according to existing logic. The initial offset is taken from the fetch request and we use the max fetch size to find the range of the log file to return in the fetch response.
2. Determine the range of offsets from the fetched data. We know the initial offset from the request, but we need to do some work to find the final offset. We propose to use the offset index to lookup an approximate upper bound for the last offset in the fetch range using the last byte position in the log file included in the fetched data.
3. Scan the aborted transaction indices of all log segments greater than or equal to the segment including the first fetched offset. We can stop when we have either reached an

entry with an LSO greater than or equal to the last offset in the fetch, or when there are no more segment indices to scan.

Why does this work?

First, we only return records from offsets earlier than each partition's current LSO. This ensures that the consumer will always know the outcome of each transaction contained in a fetch response. Aborted records can be immediately discarded and committed records can be immediately returned (even if the transaction still has data yet to be fetched).

We can exclude the indices from log segments prior to the fetched segment since they only contain transactions which were completed on previous segments. Scanning the indices for all segments greater than the current segment is clearly correct, so the only thing to check is that it is safe to stop the scan when we have reached an entry with an LSO larger than the last fetched offset. Recall that the LSO in the aborted transaction index is relative to the offset of the aborted transaction itself. All entries prior to the LSO are guaranteed to be determined (committed or aborted) if you read up to the offset of the entry itself. So by finding an entry with an LSO greater than the last fetched offset, we are guaranteed to have seen all completed transactions (keep in mind that the LSO increases monotonically).

Replica Fetching

Replicas fetch from the leader in READ_UNCOMMITTED mode and populate their own aborted transaction indices accordingly.

Log Truncation

When a log segment is truncated, we can scan the aborted transaction log and similarly truncate all entries which have final offsets greater than the new log end offset.

Example

In this example, we demonstrate how the fetch algorithm works given the following state of the partition files.

Data Log

Offset	0	1	2	3	4	5	6	7	8	9	10
PID	P1	P1	P2	P1	P2	P2	P1	P2	P1	P1	P2
Status				C		A				A	C

Aborted Transaction Index

PID	P2	P1			
FirstOffset	2	6			
LastOffset	5	9			
LSO	5	6			

Fetch #1:

Assume the fetch returns the range of data between offsets 0 and 4:

Offset	0	1	2	3	4						
PID	P1	P1	P2	P1	P2						
Status				C							

We read the entries from the aborted transaction index sequentially until we find an entry such that the LSO is greater than or equal to the last fetched offset (4). We find the aborted transaction ending at offset 5, so we include it in the fetch response and end the search without scanning the rest of the index. When the consumer receives the response, it knows to discard the transaction from PID P2 beginning at offset 2 and ending at offset 5. The transaction from P1 is assumed committed.

Fetch #2:

The fetch returns the following range of data:

Offset						5	6	7	8		
PID						P2	P1	P2	P1		
Status						A					

Again we read the entries from the aborted transaction index from the beginning. In this case, we reach the end before we find an entry with an LSO greater than our last fetched offset (8). Because both of the scanned entries are contained in our fetch range, we include them both in the fetch response. As before, the consumer then knows that the transaction ending at offset 5, as well as the one beginning at offset 6 have been aborted. The transaction at offset 7 is assumed committed.

Consumer Fetch Handling

When the consumer receives a fetch response, it must collect the aborted transactions and use them to filter the fetched records. This can be done with a simple state machine.

Initially, the aborted transactions from the fetch response are arranged in a minheap, which is ordered by their initial offsets. Separately, we maintain a set of the PIDs which have aborted transaction ranges which contain the consumer's current position. The logic to follow when processing the fetched data is as follows:

1. If the message is a transaction control message, and the status is ABORT, then remove the corresponding PID from the set of PIDs with active aborted transactions.
2. Compare the record set offset and PID with the head of the aborted transaction minheap. If the PID matches and the offset is greater than or equal to the corresponding initial offset from the aborted transaction entry, remove the head from the minheap and insert the PID into the set of PIDs with aborted transactions.
3. Check whether the PID is contained in the aborted transaction set. If so, discard the record set; otherwise, add it to the records to be returned to the user.

When seeking to a new offset, the consumer will clear the transaction state for that partition. It can always rebuild it after the next fetch response is received.

Note that we expect aborted transactions to be rare in general, so the additional overhead should be minimal.

Tradeoffs

This proposal attempts to balance the complexity on the client and the server to implement READ_COMMITTED consumption. The original EoS proposal is extremely simple for the broker, but this simplicity comes at the cost of complicated buffering logic on the client. The shadow log proposal addresses this problem, but comes with considerable complexity on the server in order to maintain all of the shadow log files. This proposal aims to keep the complexity on both sides manageable.

Pros

1. No buffering is needed in READ_COMMITTED mode for the client, which in addition to making the client implementation simpler, means we can handle long transactions much more gracefully (although they will still block consumers, nothing will run out of memory). READ_UNCOMMITTED works exactly the same way as in the current proposal.
2. The number of new files on the broker is bounded and independent of the number of aborted transactions. Additionally, we do not need any complex log segment manipulation.

Cons

1. Compared with the “shadow log” proposal, the client must do some additional work to filter the aborted entries, though this seems quite trivial. Note also that this involves some additional network overhead since the aborted entries are always delivered.
2. Compared with the current proposal, there is more work on the broker since it must maintain the aborted transaction index and do the LSO bookkeeping.

Future Work

This work supports READ_COMMITTED mode in which the records are returned in offset order. There are some advantages to returning data in transaction order instead. For example, in transaction order, a single delayed transaction will not block consumers from reading other transactions. Although there are no clear use cases at the moment, below we outline future work to enable this.

Using READ_UNCOMMITTED

Without any changes on the broker, transaction order can be supported on the client by building on top of the READ_UNCOMMITTED isolation mode. This would require client-side buffering of the fetched records. An incomplete transaction would either need to be retained in memory, persisted to disk, or re-fetched when complete (assuming it is committed).

Extending the Abort Index

Instead of only storing the aborted transactions, the index could also maintain the committed transactions. This allows the broker to scan the index quickly from a given offset to determine the next transactions that should be returned to the consumer. This potentially gives the broker some ability to skip over unneeded transaction data and return to it only when the transaction is ready to be returned to the user.