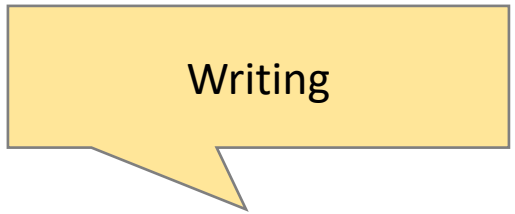
Atomic Claim Engine & Concurrency

Assumptions

Assumptions



- Primary
  - is the only system that processes commands from the domain
  - is only read by command processors
  - is append only
  - has only one technical node (it is not partitioned / not a cluster)

- Replica’s
  - are synchronised, they do not receive functional commands
  - can be partitioned

Handling dependencies in concurrent transactions

Decision: Domain specific software should pessimistically lock a ‘functionally’ relevant artifact to prevent collisions. We do not rely on optimistic locking in the database.

How to prevent collisions?

Decision:

- By pessimistically locking a ‘functionally’ relevant artifact for other writers.
- For example:
  - Lock one or more aggregates
  - Refinement: Specify operation + Lock on aggregates
    - Requires information about operations that would interfere with other operations (because they rely on the same data).

Preventing collisions is a ‘domain thing’ not a technical issue

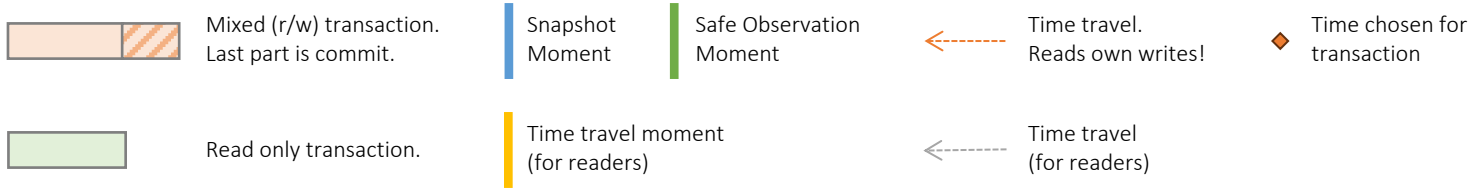
- We cannot prevent ‘semantical conflicts’ by optimistic locking.
- For example: A semantical lock could also depend on reading data.
    - Assume there is a business rule: When a person is in situation X, he or she can either apply to obtain subsidy Y or subsidy Z. Not both.
    - Two concurrent processes run. One to obtain Y and one for Z. Both read the information of the person and confirm it is in situation X, and both do not find either Y or Z. So, both will end successful. As a result, both Y and Z will be obtained.

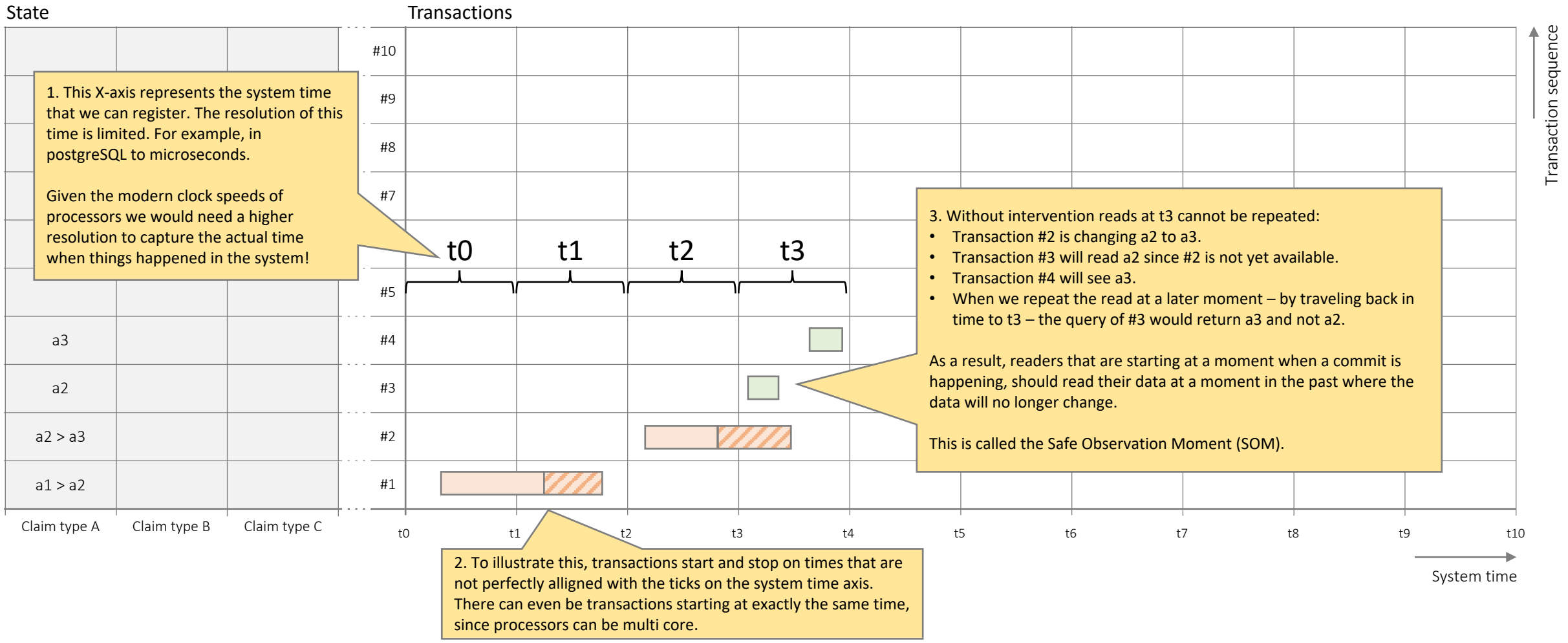
- Optimistic locking is arbitrary since it depends on the granularity of the underlying storage (and that granularity is arbitrary)
    - e.g. a json blob / table row / atomic claim

Safe Observation Moment

Decision: We always travel back to a Safe Observation Moment (for both reading and writing transactions).

Issue of limited clock resolution





Scenario:
Postponing the start of new transactions during a commit

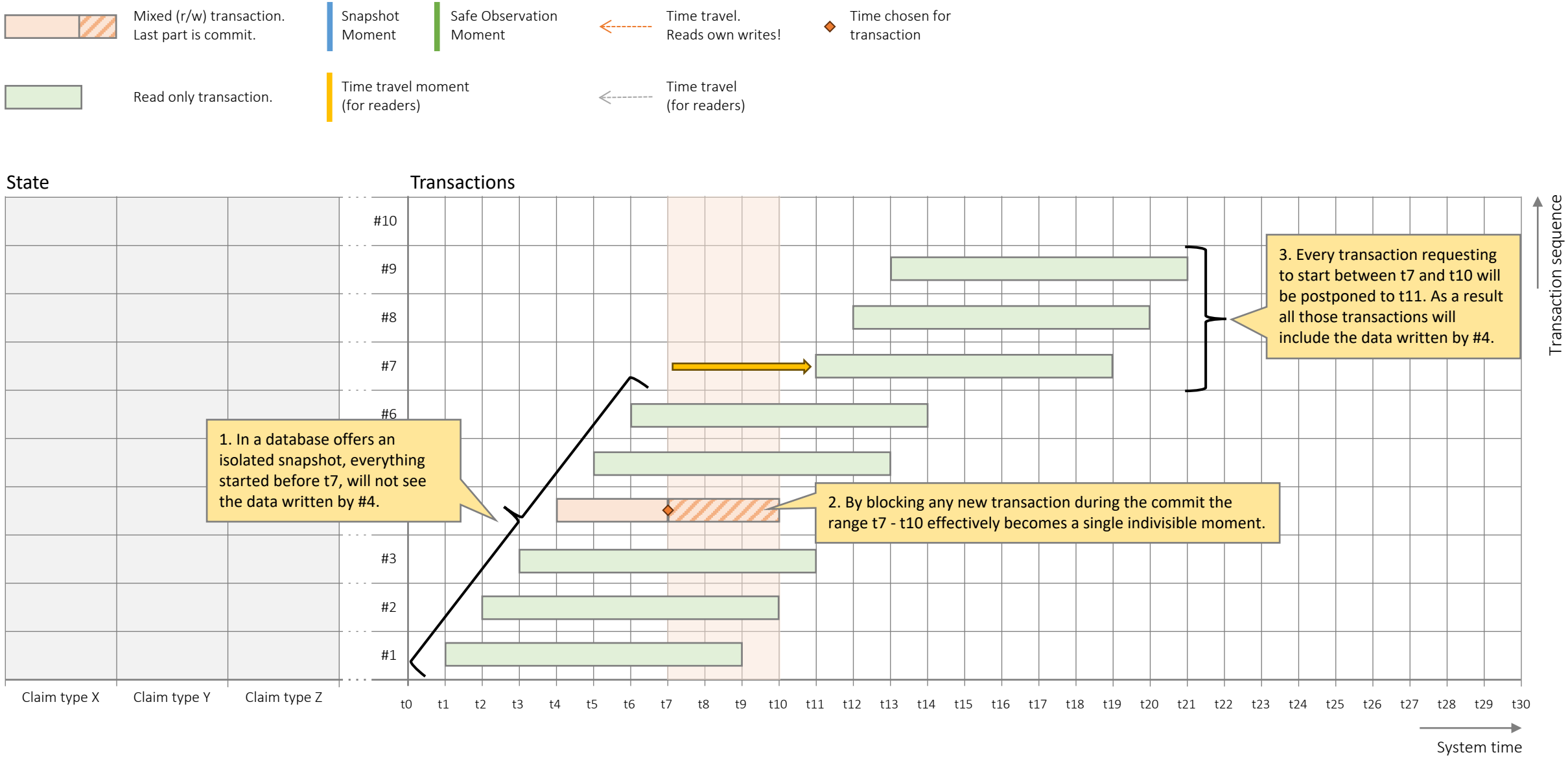
- If we could postpone the start of new transactions during a commit (until after the commit),

  and our database offers a snapshot isolation level
  (like PostgreSQL and Firebird)

  we'd always have a Safe Observation Moment.

- See explanation on next slide/page.

Blocking new transactions during the commit



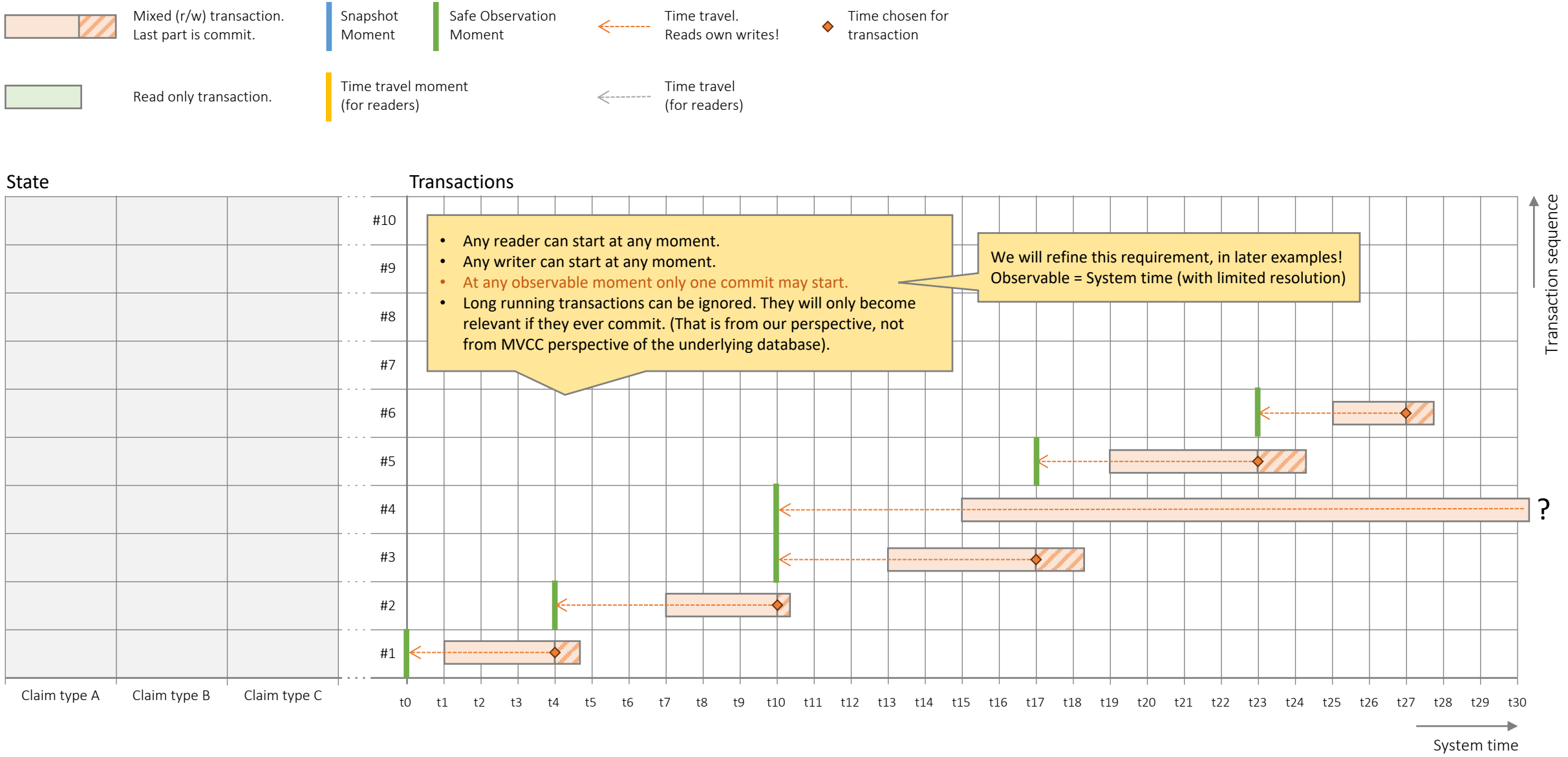
However … the bottleneck might be too large

- Writing large chunks of data could temporarily choke the system.
- Unsure: Could background processes impact the performance (like perhaps the PostgreSQL vaccuum proces).

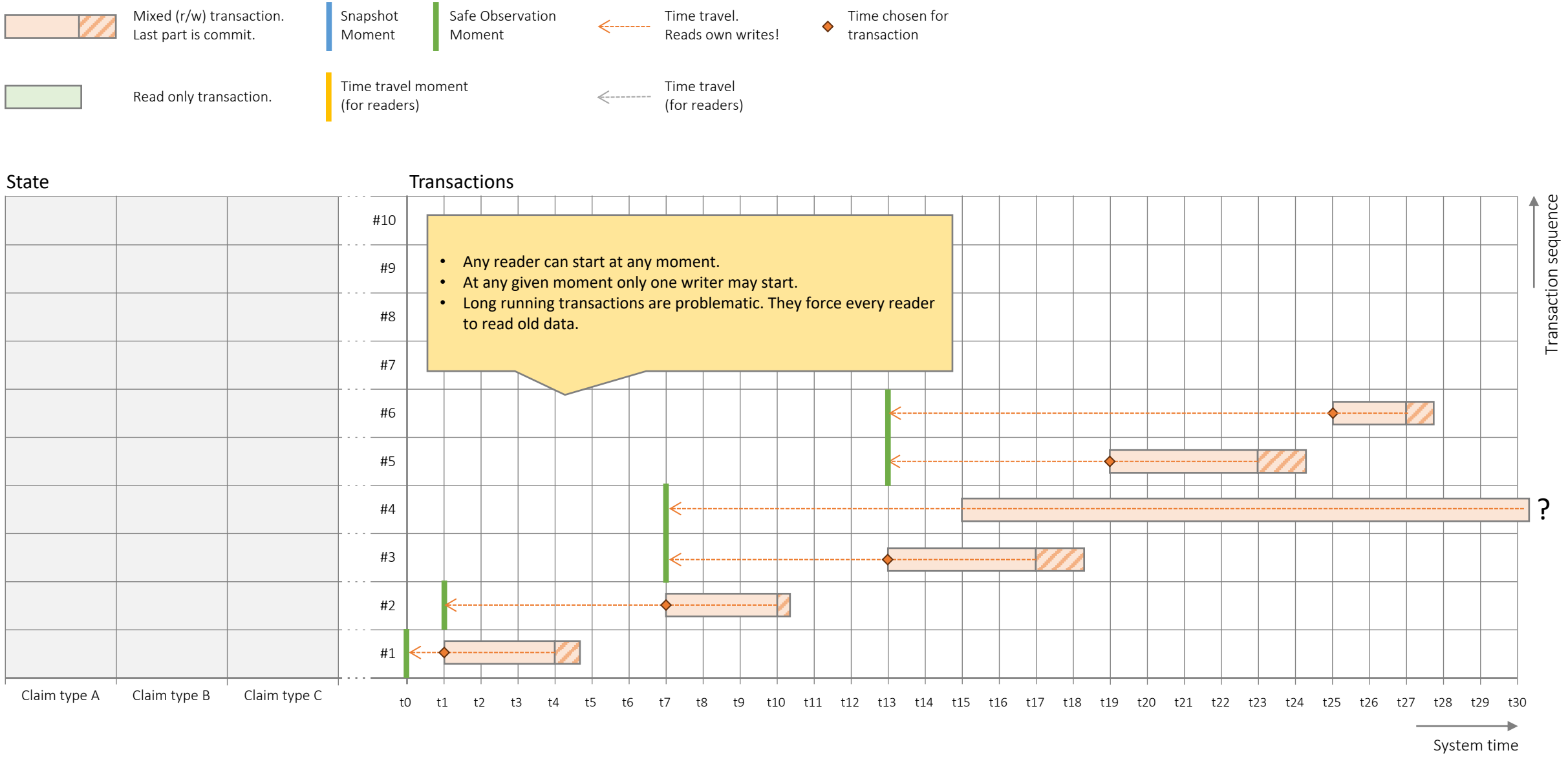
… If we do not want this bottleneck, we have to travel back to a safe moment in time!

Decision: We do not want this bottleneck!

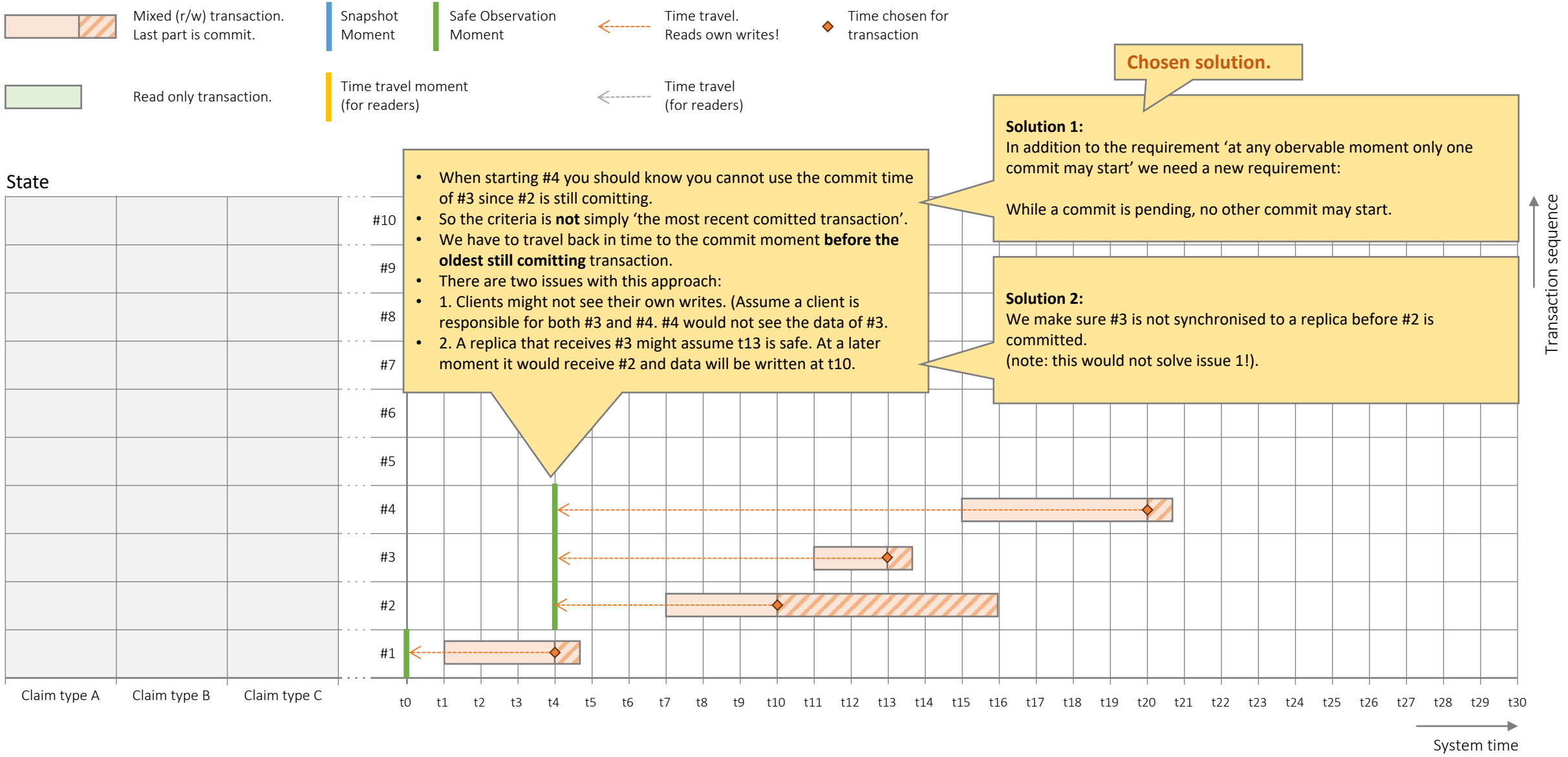
Traveling back in time to Safe Observation Moment – Transaction time = Start Commit



Traveling back in time to Safe Observation Moment – Transaction time = Start Transaction



Determining the Safe Observation Moment … (based on Transaction Time = Start Commit)



Transaction moment

Decision: We use either:

- A transaction ID, that is unique and increasing. (preferred)
- The start of the commit time.

Transaction time

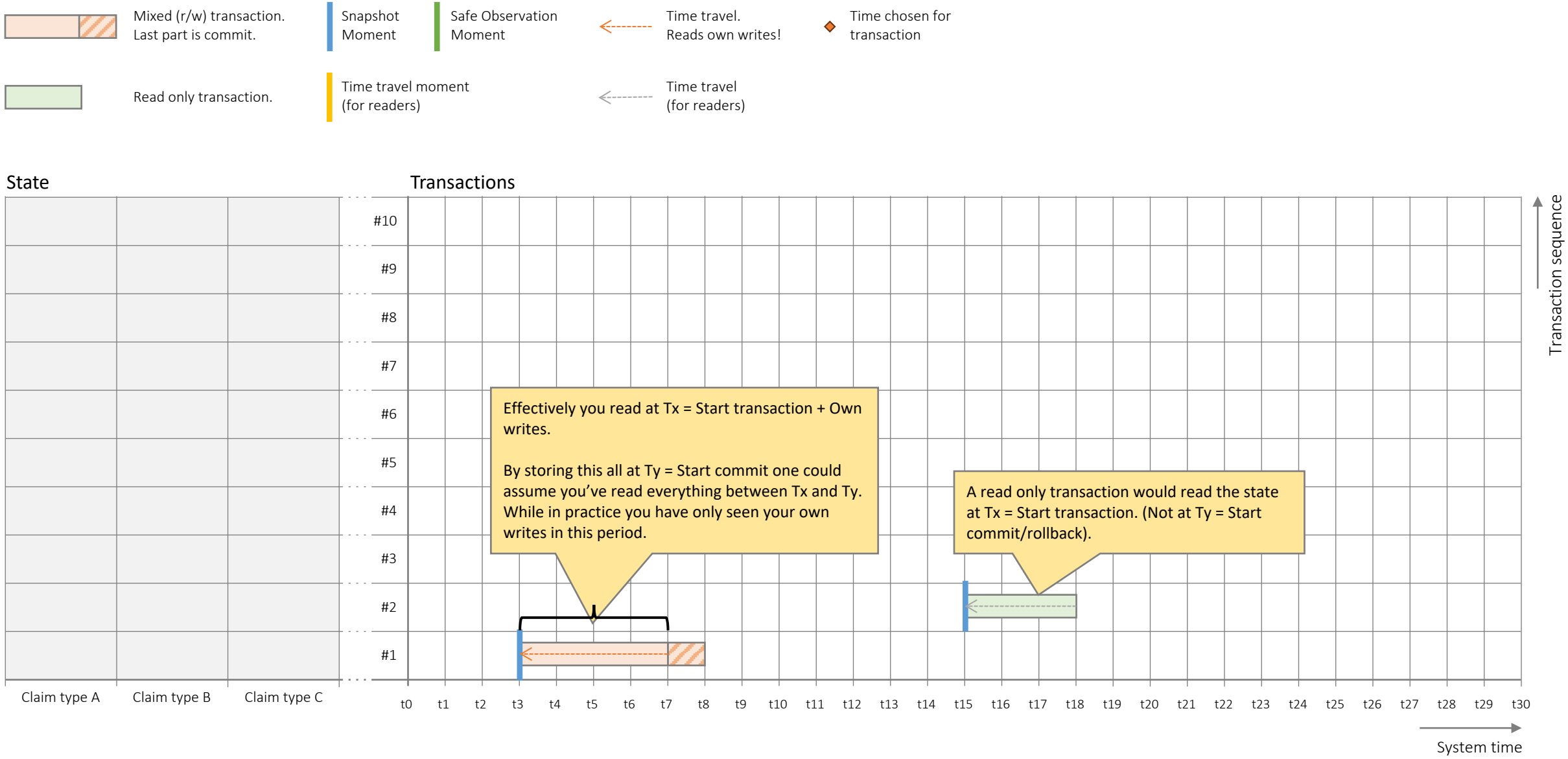
Start of transaction	Start of commit
Used in ACID databases	Used in eventually consistent (BASE) databases
Required for MVCC (actually transaction ordering, based on their start, is required. Not the start ‘time’)	
(Opiniated) It tells a better/more logical story of what a transaction ‘has seen’: assuming there is snapshot isolation, the transaction has seen the data from the start of the transaction and it’s own writes.	
This ’story’ is not perfect when concurrent transactions are mutually depending (See example).	
When working with Safe Observation Moments, the system should be protected against long running transactions.	When working with Safe Observation Moments, long running transactions are no problem.
	Easier for replication. (See example)

n.b. Transaction Time = End of commit is impossible, since the end cannot be determined when writing.

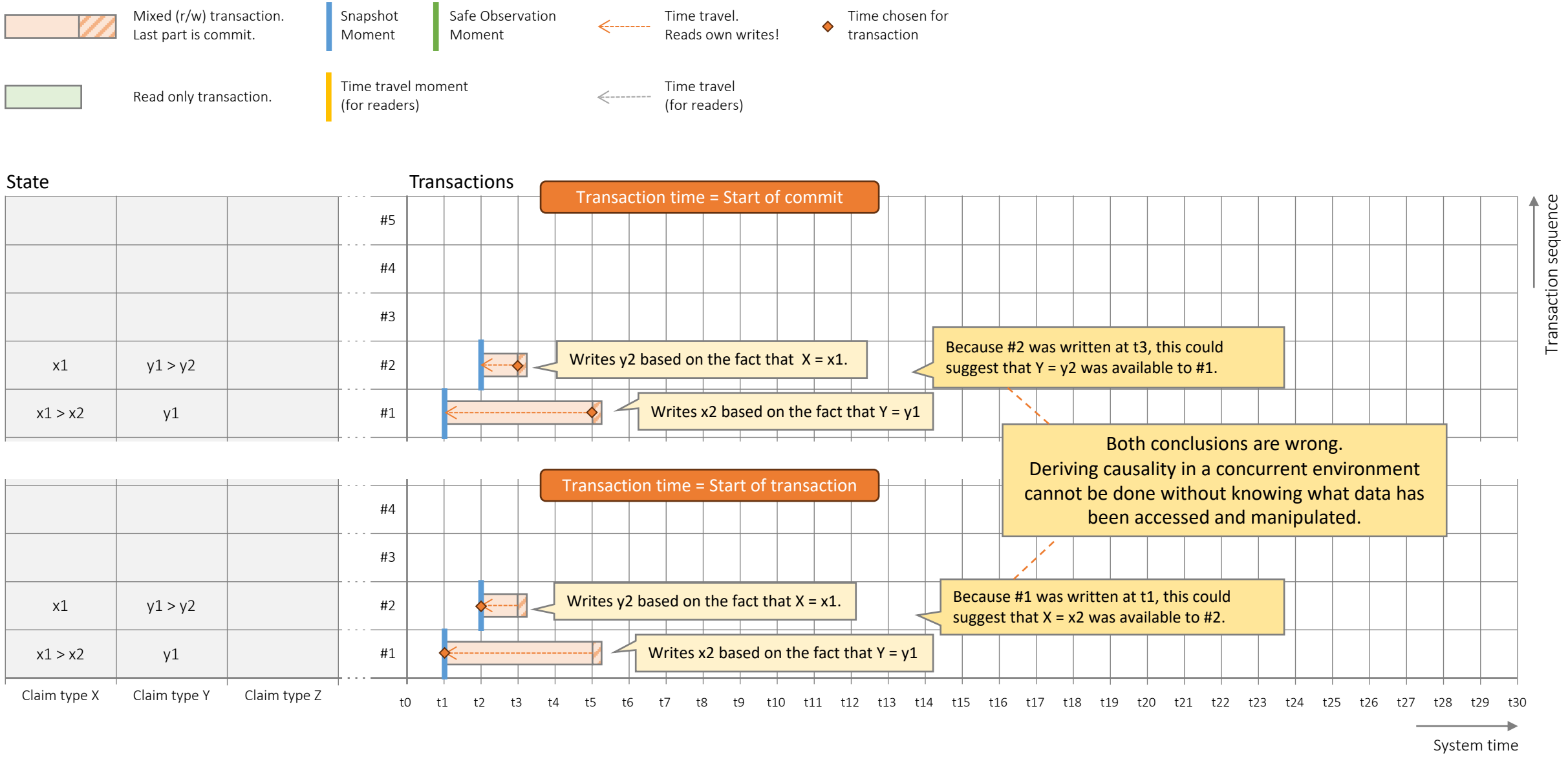
Chat GPT: Overview of Transaction Times for several vendors

Database	Timestamp Assignment	Notes
PostgreSQL	Start Time	PostgreSQL uses Multi-Version Concurrency Control (MVCC), where each transaction sees a snapshot of the database as of its start time.
MySQL (InnoDB)	Start Time	InnoDB, the default storage engine for MySQL, uses MVCC, assigning transaction timestamps at the start to determine visibility.
Oracle	Start Time	Oracle's MVCC implementation assigns transaction timestamps at the start, ensuring consistent read views.
SQL Server	Start Time	SQL Server uses a form of MVCC called Snapshot Isolation, where transactions read data as of their start time.
Firebird	Start Time	Firebird uses MVCC, assigning transaction IDs at start time. Snapshot isolation transactions see the database state as of their start time.
MongoDB (WiredTiger)	Commit Time	MongoDB's WiredTiger storage engine allows setting commit timestamps, determining when writes become visible to other transactions.
Cassandra	Commit Time	Cassandra uses timestamps provided by clients or generated at commit time to resolve write conflicts and determine the latest data.
FoundationDB	Commit Time	FoundationDB assigns a commit version at commit time, ensuring a global ordering of transactions.
Google Spanner	Commit Time	Spanner assigns commit timestamps at commit time, which can be accessed using the PENDING_COMMIT_TIMESTAMP() function.
CockroachDB	Commit Time	CockroachDB uses Hybrid Logical Clocks (HLC) to assign commit timestamps, ensuring consistency across distributed transactions.
TiDB	Commit Time	TiDB obtains the commit timestamp during the commit phase of a transaction, ensuring consistency in distributed environments.

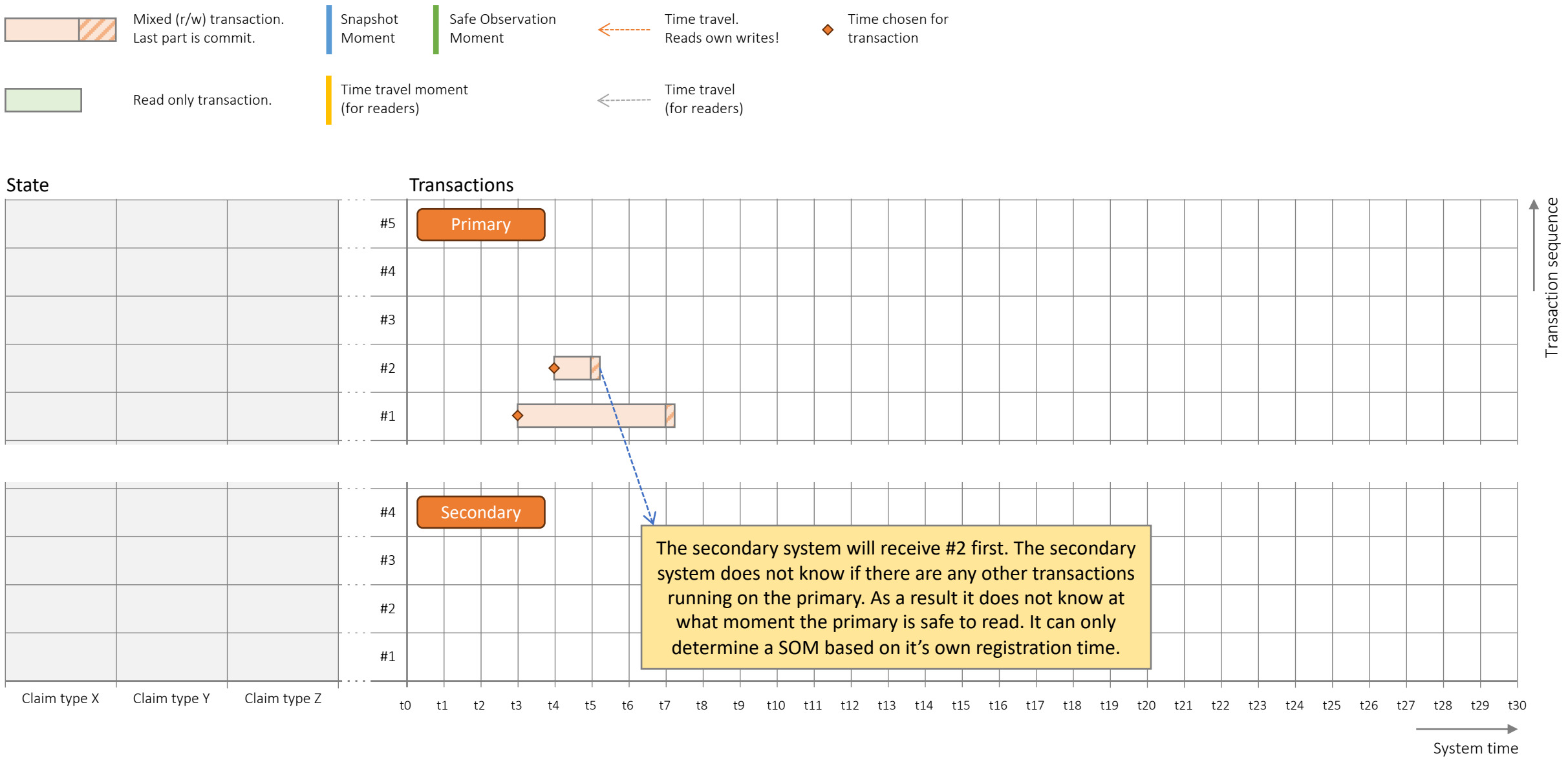
Assumptions about data that has been seen by the transaction



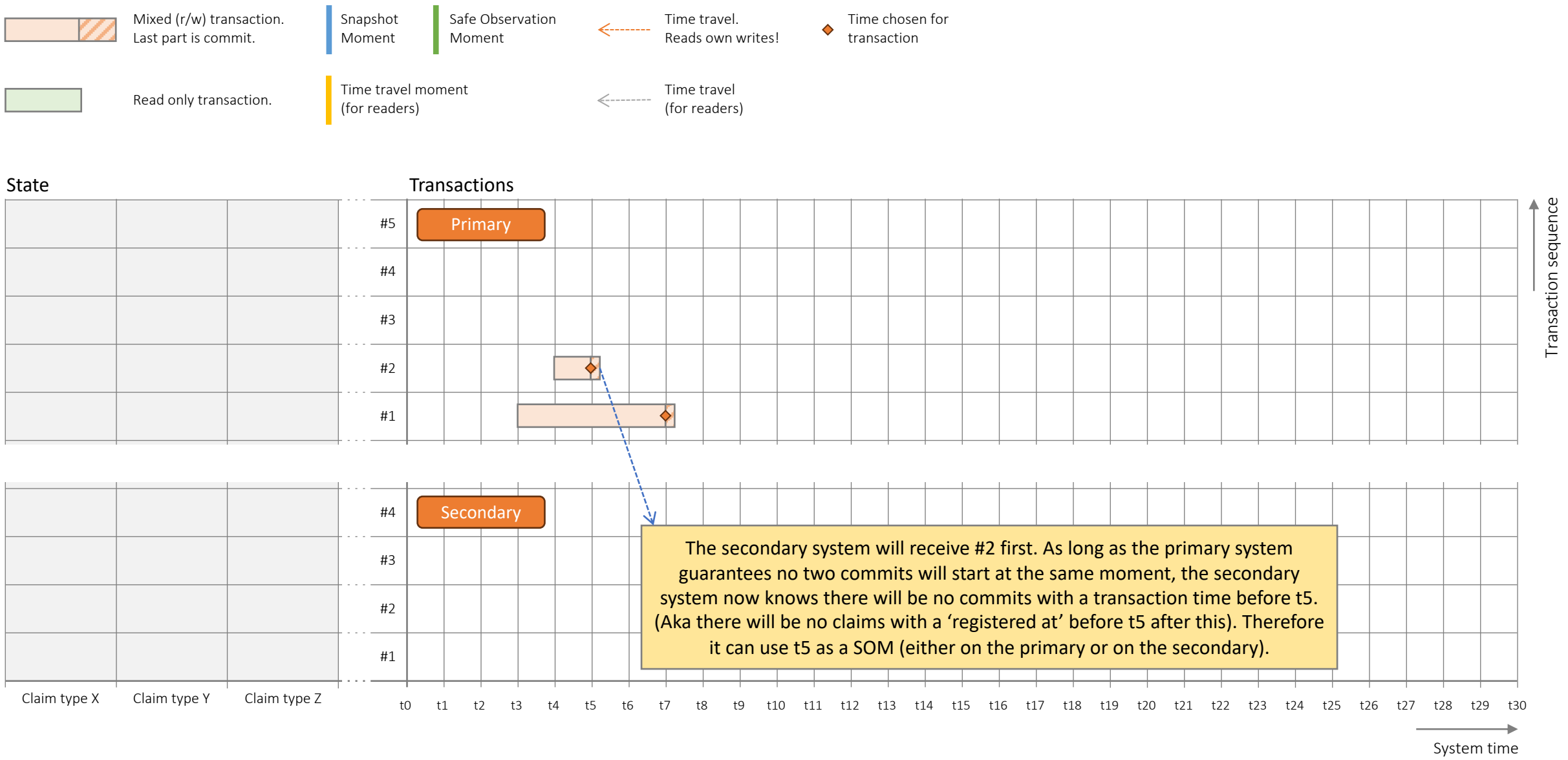
Example: Two mutually dependant writing transactions



Example: Replication based on Transaction Time = Start of Transaction



Example: Replication based on Transaction Time = Start of Commit



Decision

- We use either:
  - A transaction ID, that is unique and increasing (preferred)
  - The start of the commit time.

- For now we also store:
  - Safe Observation Moment (SOM)
  - Start of Transaction
  - Start of Commit

- Starting a transaction should return the SOM to the client.