**Blockchain Stack for First Party DMP, Ad Frameworks**

1)Synchronization of Persona 30-minute files in layer 2 side chain.

Full time series graph of users will be available in layer 2 side chain and Ethereum as well as other blockchain networks - tron, binance synchronization via persona sync gateways (interoperable persona datachains) (Authorisationdatavalidators/ DataOracles for persona transaction synchronization across data chains is also available.

2)Impression/Click Tracker Storage in datachain. (Tracker source can be AdManager, Zedo or any third party Adserver).

Native Adserver with custom tracker support (Trackers can be conventional trackers like impression, click trackers or capturing any custom data points like user input like reaction button, custom text (answer to a simple question))

Data points captured by trackers are ingested in datastores like elasticsearch and then further synchronized to layer 2 data side chain via go/solidity based smart contract transactions.

Local Data transfer files generated can reside in layer 2 data side chain for faster tps as well as other blockchain networks ethereum, tron, binance via persona sync gateways.

3)Datachain view for entities doing persona sync so that user segments - key values can be verified via block explorer. This will ensure KVs synchronized across both entities are not tampered as can be verified via their respective blockchains.

UserID Match tables will need to be prepared for inter publisher first party Id Sync using normal cookie syncing protocol.

**Persona coin / Data Currency**

1. Smart contract is for persona serialization - userId + date --> Persona Json
2. Design of a Data currency/Persona coin currency
3. Cobra Cli for bulk Syncs
4. Persona stored in ES - Fetch by application and this application will issue smart contract transactions, will store data in layer 2 side chain. Protocol buffer with different segments with different data types.
5. KV format - prefix key + suffix to serve as a key, value will be protocol buffer corresponding to persona.
6. Segments are in different domains and have different data types
7. Publisher Analytic Segments
8. Machine Learning Analytic Segments
9. Emotion based Segments
10. Color based Segments
11. IAB segments
12. Topic segments
13. Knowledge graph based segments
14. 1)User knowledge graph based segments
15. 2)Creative knowledge graph based segment
16. Main API
17. Batch burn persona coins
18. Key format - (PublisherID + BatchID, Protocol buffers (Array of Personas))
19. Persona Wallets for Data currency exchange - (PublisherID1 + BatchID, PublisherID2 + BatchID)Two storage formats are required (save latencies, can be utilized by multiple DApps)
20. Payments for Batch Persona Exchange - Can be handled via Coin Wallet Address, Persona Wallet Address)
21. Layer 2 side chain smart contract exist for Coin as well as Persona (Different publishers have account in Persona wallet (Data currency), Coin wallet (Normal currency)/Native Storage (Effective way of storing full analytics data (persona time series graph))
22. Key will be 🡺 publisherId + date + hour + batch Sync Id
23. Multi key with same value KV stores - to be explored
24. Same value key maps to save storage space - to be explored
25. Tamper proof - Persona Wallet
26. BlockExplorer - Fetch data FromPublisher layer 2 side chain using RPC APIs - store it in any custom data store - postgres, elasticsearch with block meta data / block information - block level information - block explorer - Full persona sync transaction view
27. Multiple Applications
28. 1)Cloud storage
29. 2)Blockchain storage
30. Impression/Click Tracker – Storage Impression/Click/User Interaction based Trackers - KVs - Level DB Storage (There will be allegro big cache layer on top of key value store, which will be batch serialised to disk for faster contract transaction handling)
31. Payment wallet (Advertiser wallets) - for secure Impression and clicks based Transaction (CPM/CPC based payments)
32. Custom Adserver with blockchain port
33. Advertiser DApp for secure payment - Analyzes blockchain data - smart contract does transactions accordingly.
34. Publisher wallet (CPM, CPC based payments format)
35. Interoperability - ethereum, tron, binance - (inter blockchain data wallet/currency wallet - interactions via Sync gateway)
36. Persona wallet interoperability using Sync gateways
37. Ensure Data Authenticity for use in data exchange.

**Advantages**

1. **Save lot of bank wire transfer fees**
2. **Gas less layer 2 data side chain transactions**
3. **Less block generation times**
4. **Number of transactions in a block, how many personas one can store in a block (timestamp field - persona protocol buffer) (batch size \* number of transactions)**
5. **Load testing blockchain (tps measurements/persona sync per second measurements, system benchmark, check for throttling)**
6. **Exponential decay, jiiter in synchronizing (App which does transactions, this logic will be incorporated in this application), if batch transactions are throttling the blockchain**

const (

SyncEventTopic = "personacoin:Sync"

ApprovalEventTopic = "personacoin:approval"

)

type (

InitRequest = ctypes.InitRequest

TotalUniquesRequest = ctypes.TotalUniquesRequest

TotalUniquesResponse = ctypes.TotalUniquesResponse

UniquesRequest = ctypes.UniquesRequest

UniquesResponse = ctypes.UniquesResponse

SyncRequest = ctypes.SyncRequest

SyncResponse = ctypes.SyncResponse

SyncEvent = ctypes.SyncEvent

ApproveRequest = ctypes.ApproveRequest

ApproveResponse = ctypes.ApproveResponse

ApprovalEvent = ctypes.ApprovalEvent

SyncFromPublisherRequest = ctypes.SyncFromPublisherRequest

SyncFromPublisherResponse = ctypes.SyncFromPublisherResponse

Initialpublisher = ctypes.Initialpublisher

Warehouse = ctypes.Warehouse

BurnRequest =ctypes.BurnRequest

)

var (

ErrSenderUniquesTooLow = errors.New("sender uniques is too low")

)

var (

warehouseKey = []byte("warehouse")

decimals = 18

)

func publisherKey(addr datachain.Address) []byte {

return util.PrefixKey([]byte("publisher"), addr.Bytes())

}

type PersonaCoin struct {

}

func (c \*PersonaCoin) Meta() (plugin.Meta, error) {

return plugin.Meta{

Name: "coin",

Version: "1.0.0",

}, nil

}

func (c \*PersonaCoin) Init(ctx contract.Context, req \*InitRequest) error {

div := datachain.NewBigUIntFromPublisherInt(10)

div.Exp(div, datachain.NewBigUIntFromPublisherInt(18), nil)

Uniques := datachain.NewBigUIntFromPublisherInt(0)

for \_, initAcct := range req.publishers {

owner := datachain.UnmarshalAddressPB(initAcct.Owner)

unique := datachain.NewBigUIntFromPublisherInt(int64(initAcct.Uniques))

unique.Mul(unique, div)

acct := &publisher{

Owner: owner.MarshalPB(),

Uniques: &types.BigUInt{

Value: \*unique,

},

}

err := ctx.Set(publisherKey(owner), acct)

if err != nil {

return err

}

Uniques.Add(Uniques, &acct.Uniques.Value)

}

warehouse := &Warehouse{

TotalUniques: &types.BigUInt{

Value: \*Uniques,

},

}

return ctx.Set(warehouseKey, econ)

}

**//Burn applicable for persona currency, normal currency**

**//Request FromPublisher / Sync from Authenticated Authority**

func (c \*PersonaCoin) PersonaBurn(ctx contract.Context, req \*BurnRequest) error {

if req.Owner == nil || req.Uniques == nil {

return errors.New("owner or uniques is nil")

}

gatewayAddr, err := ctx.Resolve("datachaincoin-gateway")

if err != nil {

return errUtil.Wrap(err, "failed to burn datachain coin")

}

if ctx.Message().Sender.Compare(gatewayAddr) != 0 {

return errors.New("not authorized to burn datachain coin")

}

return burn(ctx, datachain.UnmarshalAddressPB(req.Owner), &req.Uniques.Value)

}

func burn(ctx contract.Context, FromPublisher datachain.Address, uniquestoBurn \*datachain.BigUInt) error {

publisher, err := loadpublisher(ctx, FromPublisher)

if err != nil {

return err

}

warehouse := &Warehouse{

TotalUniques: &types.BigUInt{Value: \*datachain.NewBigUIntFromPublisherInt(0)},

}

err = ctx.Get(warehouseKey, econ)

if err != nil && err != contract.ErrNotFound {

return err

}

uniques := publisher.Uniques.Value

Uniques := uniq.TotalUniques.Value

uniques.Sub(&uniques, uniquestoBurn)

Uniques.Sub(&Uniques, uniquestoBurn)

publisher.Uniques.Value = uniques

uniq.TotalUniques.Value = Uniques

if err := savepublisher(ctx, publisher); err != nil {

return err

}

// Burn events are Sync events with the empty address as `to`

burnAddress := datachain.RootAddress(ctx.Block().ChainID)

if err := emitSyncEvent(ctx, FromPublisher, burnAddress, uniques); err != nil {

return err

}

return ctx.Set(warehouseKey, warehouse)

}

**//Total Uniques for Data Currency**

func (c \*PersonaCoin) TotalUniques(

ctx contract.StaticContext,

req \*TotalUniquesRequest,

) (\*TotalUniquesResponse, error) {

var warehouse Warehouse

err := ctx.Get(warehouseKey, &warehouse)

if err != nil {

return nil, err

}

return &TotalUniquesResponse{

TotalUniques: warehouse.TotalUniques,

}, nil

}

**//(Total number of userIds, segment wise userIds)**

func (c \*PersonaCoin) Uniques(

ctx contract.StaticContext,

req \*UniquesRequest,

) (\*UniquesResponse, error) {

owner := datachain.UnmarshalAddressPB(req.Owner)

publisheracct, err := loadpublisher(ctx, owner)

if err != nil {

return nil, err

}

return &UniquesResponse{

uniques: publisheracct.Uniques,

}, nil

}

**//(Sync across wallets)**

func (c \*PersonaCoin) Sync(ctx contract.Context, req \*SyncRequest) error {

if ctx.FeatureEnabled(datachainchain.DatacurrencyVersion1\_1Feature, false) {

return c.Sync(ctx, req)

}

return c.legacySync(ctx, req)

}

func (c \*PersonaCoin) Sync(ctx contract.Context, req \*SyncRequest) error {

FromPublisher := ctx.Message().Sender

toPublisher := datachain.UnmarshalAddressPB(req.ToPublisher)

FromPublisherpublisher, err := loadpublisher(ctx, FromPublisher)

if err != nil {

return err

}

uniques := req.Uniques.Value

FromPublisherBalance := FromPublisher.Uniques.Value

if FromPublisherBalance.Cmp(&uniques) < 0 {

return ErrSenderUniquesTooLow

}

FromPublisherUniques.Sub(&FromPublisherBalance, &uniques)

FromPublisher.Uniques.Value = FromPublisherBalance

err = savepublisher(ctx, FromPublisher)

if err != nil {

return err

}

topublisher, err := loadpublisher(ctx, to)

if err != nil {

return err

}

toUniques := topublisher.Uniques.Value

toUniques.Add(&toUniques, &uniques)

topublisher.Uniques.Value = toUniques

err = savepublisher(ctx, topublisher)

if err != nil {

return err

}

return emitSyncEvent(ctx, FromPublisher, to, &uniques)

}

**//Approval for batch size (data currency (bulk user Id (migrate in one transaction)) - if batch size, exceeds a threshold smart contract will reject it**

func (c \*PersonaCoin) Approve(ctx contract.Context, req \*ApproveRequest) error {

owner := ctx.Message().Sender

spender := datachain.UnmarshalAddressPB(req.Spender)

allow, err := loadUniquesAllowance(ctx, owner, spender)

if err != nil {

return err

}

allow.Uniques = req.Uniques

err = saveUniquesAllowance(ctx, allow)

if err != nil {

return err

}

// req.Uniques may be nil at this point, so have to be careful when dereferencing it

var uniques \*datachain.BigUInt

if req.Uniques != nil {

uniques = &req.Uniques.Value

}

return emitApprovalEvent(ctx, owner, spender, uniques)

}

func (c \*PersonaCoin) UniquesAllowance(

ctx contract.StaticContext,

req \*UniquesAllowanceRequest,

) (\*UniquesAllowanceResponse, error) {

publisherowner := datachain.UnmarshalAddressPB(req.PublisherOwner)

publisherspender := datachain.UnmarshalAddressPB(req.PublisherSpender)

allow, err := loadUniquesAllowance(ctx, publisherowner, publisherspender)

if err != nil {

return nil, err

}

return &UniquesAllowanceResponse{

Uniques: allow.Uniques,

}, nil

}

**//To demonstrate Exchange protocols can be updated by datachain chainconfig smart contract**

func (c \*PersonaCoin) SyncFromPublisher(ctx contract.Context, req \*SyncFromPublisherRequest) error {

if ctx.FeatureEnabled(datachain.DataCurrencyFeature, false) {

return c.SyncFromPublisher(ctx, req)

}

return c.legacySyncFromPublisher(ctx, req)

}

func (c \*PersonaCoin) SyncFromPublisher(ctx contract.Context, req \*SyncFromPublisherRequest) error {

spender := ctx.Message().Sender

FromPublisher := datachain.UnmarshalAddressPB(req.FromPublisher)

toPublisher := datachain.UnmarshalAddressPB(req.ToPublisher)

FromPublisher, err := loadpublisher(ctx, FromPublisher)

if err != nil {

return err

}

allow, err := loadUniquesAllowance(ctx, FromPublisher, spender)

if err != nil {

return err

}

allowUniques := allow.Uniques.Value

uniques := req.Uniques.Value

FromPublisherUniques := FromPublisher.Unique.Value

if allowUniques.Cmp(&uniques) < 0 {

return errors.New("uniques is over spender's limit")

}

if FromPublisherUniques.Cmp(&uniques) < 0 {

return ErrSenderUniquesTooLow

}

FromPublisherUniques.Sub(&FromPublisherUniques, &uniques)

FromPublisher.Uniques = FromPublisherUniques)

err = savepublisher(ctx, FromPublisher)

if err != nil {

return err

}

toPublisher, err := loadpublisher(ctx, to)

if err != nil {

return err

}

toUniques := topublisher.Uniques.Value

toUniques.Add(&toUniques, &uniques)

topublisher.Uniques.Value = toUnique

err = savepublisher(ctx, topublisher)

if err != nil {

return err

}

allowUniques.Sub(&allowUniques, &uniques)

allow.Uniques = allowUniques

err = saveUniquesAllowance(ctx, allow)

if err != nil {

return err

}

return emitSyncEvent(ctx, FromPublisher, to, &uniques)

}

func loadpublisher(

ctx contract.StaticContext,

owner datachain.Address,

) (\*publisher, error) {

pubacct := &publisher{

Owner: owner.MarshalPB(),

Uniques: &types.BigUInt{

Value: \*datachain.NewBigUIntFromPublisherInt(0),

},

}

err := ctx.Get(publisherKey(owner), pubacct)

if err != nil && err != contract.ErrNotFound {

return nil, err

}

return pubacct, nil

}

func savepublisher(ctx contract.Context, pubacct \*publisher) error {

owner := datachain.UnmarshalAddressPB(pubacct.Owner)

return ctx.Set(publisherKey(pubacctowner), acct)

}

func loadUniquesAllowance(

ctx contract.StaticContext,

owner, spender datachain.Address,

) (\*UniquesAllowance, error) {

allow := &UniquesAllowance{

Owner: owner.MarshalPB(),

Spender: spender.MarshalPB(),

Uniques: &types.BigUInt{

Value: \*datachain.NewBigUIntFromPublisherInt(0),

},

}

err := ctx.Get(UniquesAllowanceKey(owner, spender), allow)

if err != nil && err != contract.ErrNotFound {

return nil, err

}

return allow, nil

}

func saveUniquesAllowance(ctx contract.Context, allow \*UniquesAllowance) error {

owner := datachain.UnmarshalAddressPB(allow.Owner)

spender := datachain.UnmarshalAddressPB(allow.Spender)

return ctx.Set(UniquesAllowanceKey(owner, spender), allow)

}

var Contract plugin.Contract = contract.MakePluginContract(&PersonaCoin{})

**//For upgrades demonstration - upgrade smart contract/protocol in data chain**

func (c \*PersonaCoin) legacySync(ctx contract.Context, req \*SyncRequest) error {

FromPublisher := ctx.Message().Sender

to := datachain.UnmarshalAddressPB(req.To)

FromPublisher, err := loadpublisher(ctx, FromPublisher)

if err != nil {

return err

}

ToPublisher, err := loadpublisher(ctx, to)

if err != nil {

return err

}

uniques := req.Uniques.Value

FromPublisherUniques := FromPublisher.Uniques.Value

toUniques := topublisher.Uniques.Value

if FromPublisherUniques.Cmp(&uniques) < 0 {

return errors.New("sender uniques is too low")

}

FromPublisherUniques.Sub(&FromPublisherUniques, &uniques)

toUniques.Add(&toUniques, &uniques)

FromPublisher.Uniques.Value = FromPublisherUniques

topublisher.Uniques.Value = toUniques

err = savepublisher(ctx, FromPublisher)

if err != nil {

return err

}

err = savepublisher(ctx, topublisher)

if err != nil {

return err

}

return emitSyncEvent(ctx, FromPublisher, to, &uniques)

}

**//Event driven Architecture**

**//For each bulk persona sync - event is logged in data chain**

**//These events can be tracker in data chain logs or can be subscribed via RPC API**

**// Event Emission corresponding to each bulk Persona Sync**

**// Data currency, coin currency**

**// All the events emitted will be available in Blockchain Logs for tracking purpose.**

func emitSyncEvent(ctx contract.Context, FromPublisher, spender datachain.Address, uniques \*datachain.BigUInt) error {

var Uniques \*types.BigUInt

if uniques == nil {

uniques = datachain.BigZeroPB()

} else {

uniques = &types.BigUInt{Value: \*uniques}

}

marshalled, err := proto.Marshal(&SyncEvent{

FromPublisher: FromPublisher.MarshalPB(),

ToPublisher: spender.MarshalPB(),

Uniques : uniques,

})

if err != nil {

return err

}

ctx.EmitTopics(marshalled, SyncEventTopic)

return nil

}

func emitApprovalEvent(ctx contract.Context, FromPublisher, to datachain.Address, uniques \*datachain.BigUInt) error {

var uniques \*types.BigUInt

if uniques == nil {

uniques = datachain.BigZeroPB()

} else {

uniques = &types.BigUInt{Value: \*uniques}

}

marshalled, err := proto.Marshal(&ApprovalEvent{

FromPublisher: FromPublisher.MarshalPB(),

Spender: toPublisher.MarshalPB(),

Uniques: uniques,

})

if err != nil {

return err

}

ctx.EmitTopics(marshalled, ApprovalEventTopic)

return nil

}

**Tendermint Integration**

**func (a \*Application) BeginBlock(req abci.RequestBeginBlock) abci.ResponseBeginBlock {**

**block := req.Header**

**//Blockchain consensus sanity check**

**if block.Height != a.height() {**

**panic(fmt.Sprintf("app height %d doesn't match BeginBlock height %d", a.height(), block.Height))**

**}**

**a.curBlockHeader = block**

**a.curBlockHash = req.Hash**

**if a.CreateContractUpkeepHandler != nil {**

**upkeepStoreTx := store.WrapAtomic(a.Store).BeginTx()**

**upkeepState := NewStoreState(**

**context.Background(),**

**upkeepStoreTx,**

**a.curBlockHeader,**

**a.curBlockHash,**

**a.GetValidatorSet,**

**)**

**contractUpkeepHandler, err := a.CreateContractUpkeepHandler(upkeepState)**

**if err != nil {**

**panic(err)**

**}**

**if contractUpkeepHandler != nil {**

**if err := contractUpkeepHandler.Upkeep(); err != nil {**

**panic(err)**

**}**

**upkeepStoreTx.Commit()**

**}**

**}**

**storeTx := store.WrapAtomic(a.Store).BeginTx()**

**state := NewStoreState(**

**context.Background(),**

**storeTx,**

**a.curBlockHeader,**

**nil,**

**a.GetValidatorSet,**

**)**

**//Validator elections are held at the start of the block**

**validatorManager, err := a.CreateValidatorManager(state)**

**if err != registry.ErrNotFound {**

**if err != nil {**

**panic(err)**

**}**

**err =datavalidatorManager.BeginBlock(req, a.height())**

**if err != nil {**

**panic(err)**

**}**

**}**

**//Enable Features**

**chainConfigManager, err := a.CreateChainConfigManager(state)**

**if err != nil {**

**panic(err)**

**}**

**if chainConfigManager != nil {**

**if err := chainConfigManager.EnableFeatures(a.height()); err != nil {**

**panic(err)**

**}**

**}**

**storeTx.Commit()**

**return abci.ResponseBeginBlock{}**

**}**

**func (a \*Application) EndBlock(req abci.RequestEndBlock) abci.ResponseEndBlock {**

**if req.Height != a.height() {**

**panic(fmt.Sprintf("app height %d doesn't match EndBlock height %d", a.height(), req.Height))**

**}**

**storeTx := store.WrapAtomic(a.Store).BeginTx()**

**state := NewStoreState(**

**context.Background(),**

**storeTx,**

**a.curBlockHeader,**

**nil,**

**a.GetValidatorSet,**

**)**

**receiptHandler, err := a.ReceiptHandlerProvider.StoreAt(a.height(), state.FeatureEnabled(EvmTxReceiptsVersion2Feature, false))**

**if err != nil {**

**panic(err)**

**}**

**if err := receiptHandler.CommitBlock(state, a.height()); err != nil {**

**storeTx.Rollback()**

**log.Error(fmt.Sprintf("aborted committing block receipts, %v", err.Error()))**

**} else {**

**storeTx.Commit()**

**}**

**storeTx = store.WrapAtomic(a.Store).BeginTx()**

**state = NewStoreState(**

**context.Background(),**

**storeTx,**

**a.curBlockHeader,**

**nil,**

**a.GetValidatorSet,**

**)**

**validatorManager, err := a.CreateValidatorManager(state)**

**if err != registry.ErrNotFound {**

**if err != nil {**

**panic(err)**

**}**

**t2 := time.Now()**

**validators, err :=datavalidatorManager.EndBlock(req)**

**diffsecs := time.Since(t2).Seconds()**

**validatorFuncLatency.Observe(diffsecs)**

**log.Info(fmt.Sprintf("validator manager took %f seconds-----\n", diffsecs))**

**if err != nil {**

**panic(err)**

**}**

**storeTx.Commit()**

**return abci.ResponseEndBlock{**

**ValidatorUpdates:datavalidators,**

**}**

**}**

**return abci.ResponseEndBlock{**

**ValidatorUpdates: []abci.ValidatorUpdate{},**

**}**

**}**

**//Check tx - for a transaction dry run - transaction execution is normal (no panic or anomalies)**

**func (a \*Application) CheckTx(txBytes []byte) abci.ResponseCheckTx {**

**ok := abci.ResponseCheckTx{Code: abci.CodeTypeOK}**

**var err error**

**defer func(begin time.Time) {**

**lvs := []string{"method", "CheckTx", "error", fmt.Sprint(err != nil)}**

**checkTxLatency.With(lvs...).Observe(time.Since(begin).Seconds())**

**}(time.Now())**

**// If the chain is configured not to generate empty blocks then CheckTx may be called before**

**// BeginBlock when the application restarts, which means that both curBlockHeader and**

**// lastBlockHeader will be default initialized. Instead of invoking a contract method with**

**// a vastly innacurate block header simply skip invoking the contract. This has the minor**

**// disadvantage of letting an potentially invalid tx propagate to other nodes, but this should**

**// only happen on node restarts, and only if the node doesn't receive any txs from it's peers**

**// before a client sends it a tx.**

**if a.curBlockHeader.Height == 0 {**

**return ok**

**}**

**\_, err = a.processTx(txBytes, true)**

**if err != nil {**

**log.Error(fmt.Sprintf("CheckTx: %s", err.Error()))**

**return abci.ResponseCheckTx{Code: 1, Log: err.Error()}**

**}**

**return ok**

**}**

**func (a \*Application) DeliverTx(txBytes []byte) abci.ResponseDeliverTx {**

**var err error**

**isEvmTx := false**

**defer func(begin time.Time) {**

**lvs := []string{**

**"method", "DeliverTx",**

**"error", fmt.Sprint(err != nil),**

**"evm", fmt.Sprintf("%t", isEvmTx),**

**}**

**requestCount.With(lvs[:4]...).Add(1)**

**deliverTxLatency.With(lvs...).Observe(time.Since(begin).Seconds())**

**}(time.Now())**

**r, err := a.processTx(txBytes, false)**

**if err != nil {**

**log.Error(fmt.Sprintf("DeliverTx: %s", err.Error()))**

**return abci.ResponseDeliverTx{Code: 1, Log: err.Error()}**

**}**

**if r.Info == utils.CallEVM || r.Info == utils.DeployEvm {**

**isEvmTx = true**

**}**

**return abci.ResponseDeliverTx{Code: abci.CodeTypeOK, Data: r.Data, Tags: r.Tags, Info: r.Info}**

**}**

**func (a \*Application) processTx(txBytes []byte, isCheckTx bool) (TxHandlerResult, error) {**

**var err error**

**// we should be keeping this across multiple checktx, and only rolling back after they all complete**

**// for now the nonce will have a special cache that it rolls back each block**

**storeTx := store.WrapAtomic(a.Store).BeginTx()**

**state := NewStoreState(**

**context.Background(),**

**storeTx,**

**a.curBlockHeader,**

**a.curBlockHash,**

**a.GetValidatorSet,**

**)**

**receiptHandler, err := a.ReceiptHandlerProvider.StoreAt(a.height(), state.FeatureEnabled(EvmTxReceiptsVersion2Feature, false))**

**if err != nil {**

**panic(err)**

**}**

**r, err := a.TxHandler.ProcessTx(state, txBytes, isCheckTx)**

**if err != nil {**

**storeTx.Rollback()**

**//save receipt & hash of failed EVM tx to node-local persistent cache (not app state)**

**receiptHandler.DiscardCurrentReceipt()**

**return r, err**

**}**

**if !isCheckTx {**

**if r.Info == utils.CallEVM || r.Info == utils.DeployEvm {**

**err := a.EventHandler.LegacyEthSubscriptionSet().EmitTxEvent(r.Data, r.Info)**

**if err != nil {**

**log.Error("Emit Tx Event error", "err", err)**

**}**

**reader, err := a.ReceiptHandlerProvider.ReaderAt(state.Block().Height, state.FeatureEnabled(EvmTxReceiptsVersion2Feature, false))**

**if err != nil {**

**log.Error("failed to load receipt", "height", state.Block().Height, "err", err)**

**} else {**

**if reader.GetCurrentReceipt() != nil {**

**if err = a.EventHandler.EthSubscriptionSet().EmitTxEvent(reader.GetCurrentReceipt().TxHash); err != nil {**

**log.Error("failed to load receipt", "err", err)**

**}**

**}**

**}**

**receiptHandler.CommitCurrentReceipt()**

**}**

**storeTx.Commit()**

**}**

**return r, nil**

**}**

**// Commit commits the current block**

**func (a \*Application) Commit() abci.ResponseCommit {**

**var err error**

**defer func(begin time.Time) {**

**lvs := []string{"method", "Commit", "error", fmt.Sprint(err != nil)}**

**committedBlockCount.With(lvs...).Add(1)**

**commitBlockLatency.With(lvs...).Observe(time.Since(begin).Seconds())**

**log.Info(fmt.Sprintf("commit took %f seconds-----\n", time.Since(begin).Seconds())) //todo we can remove these once performance comes back to normal state**

**}(time.Now())**

**appHash, \_, err := a.Store.SaveVersion()**

**if err != nil {**

**panic(err)**

**}**

**height := a.curBlockHeader.GetHeight()**

**//Event emission is done in a go routine at the time of block commit - events are stored in a stash which is emptied at the time of block commit**

**go func(height int64, blockHeader abci.Header) {**

**if err := a.EventHandler.EmitBlockTx(uint64(height), blockHeader.Time); err != nil {**

**log.Error("Emit Block Event error", "err", err)**

**}**

**if err := a.EventHandler.LegacyEthSubscriptionSet().EmitBlockEvent(blockHeader); err != nil {**

**log.Error("Emit Block Event error", "err", err)**

**}**

**if err := a.EventHandler.EthSubscriptionSet().EmitBlockEvent(blockHeader); err != nil {**

**log.Error("Emit Block Event error", "err", err)**

**}**

**}(height, a.curBlockHeader)**

**a.lastBlockHeader = a.curBlockHeader**

**if err := a.Store.Prune(); err != nil {**

**log.Error("failed to prune app.db", "err", err)**

**}**

**if a.BlockIndexStore != nil {**

**a.BlockIndexStore.SetBlockHashAtHeight(uint64(height), a.curBlockHash)**

**}**

**return abci.ResponseCommit{**

**Data: appHash,**

**}**

**}**

**func (a \*Application) Query(req abci.RequestQuery) abci.ResponseQuery {**

**if a.QueryHandler == nil {**

**return abci.ResponseQuery{Code: 1, Log: "not implemented"}**

**}**

**result, err := a.QueryHandler.Handle(a.ReadOnlyState(), req.Path, req.Data)**

**if err != nil {**

**return abci.ResponseQuery{Code: 1, Log: err.Error()}**

**}**

**return abci.ResponseQuery{Code: abci.CodeTypeOK, Value: result}**

**}**

**func (a \*Application) height() int64 {**

**return a.Store.Version() + 1**

**}**

**func (a \*Application) ReadOnlyState() State {**

**// the store snapshot should be created atomically**

**return NewStoreStateSnapshot(**

**nil,**

**a.Store.GetSnapshot(),**

**a.lastBlockHeader,**

**nil,**

**a.GetValidatorSet,**

**)**

**}**

**Tendermint Tools**

func newConvertBlockStoreCommand() \*cobra.Command {

cmdFlags.StringSliceVarP(&privValidatorFiles, "validators", "v", []string{}, "List of priv\_validator.json files")

cmdFlags.Int64VarP(&startHeight, "from", "f", int64(1), "**Block height to start conversion from**")

}

func newBlockStoreInfoCommand() \*cobra.Command {

#Print info about block & state store

cmdFlags.IntVar(&blockHeight, "height", 1, "**Specify height of block to view**")

}

func newAppStoreInfoCommand() \*cobra.Command {

"Print info about app store",

cmdFlags.StringVar(&rootHash, "root-hash", “”, "**Find an app store version matching the given root node hash**")

cmdFlags.StringVar(&nodeHash, "node-hash", "", "**Find a node matching the given hash**")

return cmd

}

func newConvertIAVLStoreCommand() \*cobra.Command {

"iavl-store",

"Convert IAVL store (e.g. app.db)",

fmt.Printf("Saved IAVLStore version %d, hash %x, %d keys\n", ver, hash, numKeys”)

cmdFlags.StringVarP(&dbName, "name", "n", "app", "**Name of IAVL store (.db suffix will be appended automatically**)")

}

func newMigrateNodeCommand() \*cobra.Command {

**"migrate-node <socket-path> <path/to/old/node> <path/to/new/node>",**

**"Migrate chain node to TM (Given version) (both chaindata and app.db will be converted)",**

**// check app.db version matches latest block height in block store**

**// Migration complete and finalized, migrated node should now be operational**

**cmdFlags.Int64Var(&startHeight, "start", 0, "Block height at which migration should start")**

**cmdFlags.Int64Var(&endHeight, "end", 0, "Block height at which migration should end")**

cmdFlags.StringSliceVarP(&privValidatorFiles, "validators", "v", []string{}, "List of priv\_validator.json files")

cmdFlags.BoolVar(&keepOldBlockTime, "keep-old-block-time", false, "Keep old block timestamps in migrated blocks")

cmdFlags.BoolVar(&signLastSeenCommit, "sign-last-seen-commit", false, "Sign the last seen commit, should only be done on the final migration")

}

func newMigrateValidatorsToolConfigCommand() \*cobra.Command {

Use: "migrate-cluster-runner",

Short: "Copy & adjustdatavalidators-tool runner config to a new cluster",

convert.ConvertValidatorsToolConfig(oldCfgPath, newCfgPath, datachainBinPath, newClusterPath)

cmdFlags.StringVar(&datachainBinPath, "datachain", "", "Path to datachain binary to use to run the cluster nodes")

cmdFlags.StringVar(&newClusterPath, "cluster", "", "Path to the new cluster base directory")

}

func newReplayCommand() \*cobra.Command {

Use: "replay",

Short: **"Starts a non-validator node that attempts to replay blocks from an existing cluster.",**

cmdFlags.StringVar(&startBuild, "start-build", "", "chain build number to start replaying blocks with.")

cmdFlags.BoolVar(&reset, "reset", false, "Wipe chaindata/data & app.db, and start replay from genesis.")

cmdFlags.StringSliceVar(&peers, "peers", nil, "Comma separated list of peers to connect to.")

}

func newRollbackCommand() \*cobra.Command {

"rollback",

"Rolls back the various stores",

}

func newRollbackAppStoreCommand() \*cobra.Command {

"app <node-dir>",

**"Rolls back app.db by the specified number of blocks, or to a specific version."**

appStore := legacy.NewIAVLStore("app", args[0])

"Can't rollback app.db %d versions, there are only %d versions", numVers, curVer

"Attempting app.db rollback from version %d to %d\n", curVer, curVer-numVers

“app.db rollback failed:"

"app.db now at version"

“Can't rollback app.db %d versions, there are only %d versions"

"Attempting app.db rollback from version %d to %d\n"

"app.db rollback failed:"

"app.db now at version”

cmdFlags.Int64Var(&height, "height", 0, "Block height to rollback to, specify zero to rollback one block.")

cmdFlags.Int64Var(&numBlocks, "num-blocks", 0, "Number of blocks to rollback, overriden by height flag.")

}

func newRollbackAllCommand() \*cobra.Command {

"all <node-dir>

**"Rolls back app.db & TM dbs to the specified height."**

cmdFlags.Int64Var(&height, "height", 0, "Block height to rollback to, specify zero to rollback one block.")

cmdFlags.BoolVar(&rollbackBlockStore, "rollback\_block\_store", false, "Whether to rollback block store or not")

}

func newDiffCommand() \*cobra.Command {

"diff <old-chaindata> <new-chaindata>",

"Checks for differences between TM dbs",

convert.DiffBlockStore(args[0], args[1], startHeight, endHeight)

}

**#PPROF CPU Profiling**

if os.Getenv("CPUPROFILE") != "" {

f, err := os.Create(os.Getenv("CPUPROFILE"))

if err != nil {

log.Fatal(err)

}

pprof.StartCPUProfile(f)

defer func() {

pprof.StopCPUProfile()

f.Close()

fmt.Printf("profile: cpu profiling disabled\n")

}()

}

**//Start pprof server**

go func() {

log.Println(http.ListenAndServe("localhost:6060", nil))

}()

rootCmd := &cobra.Command{

Use: "tmtools",

Short: "Tendermint blockchain tools",

}

**Data Structures being used -**

**Blockstore**

**//Fetches block information from levelDBs generated by tendermint**

type BlockStore interface {

**// GetBlockByHeight retrieves block info at the specified height,**

**// specify nil to retrieve the latest block info.**

GetBlockByHeight(height \*int64) (\*ctypes.ResultBlock, error)

**// GetBlockRangeByHeight retrieves block info at the specified height range,**

**// specify nil to retrieve the latest block info.**

GetBlockRangeByHeight(minHeight, maxHeight int64) (\*ctypes.ResultBlockchainInfo, error)

**// GetBlockResults retrieves the results of the txs committed to the block at the specified height,**

**// specify nil to retrieve results from the latest block.**

GetBlockResults(height \*int64) (\*ctypes.ResultBlockResults, error)

**// Get Transaction Results from Tendermint Tx Hash**

GetTxResult(txHash []byte) (\*ctypes.ResultTx, error)

}

**type TwoQueueBlockStoreCache struct {**

CachedBlockStore BlockStore

**// value of Add() is different, should just build a small adapter for this so we can just**

**// have one interface for both caching algos and then we won't need to reimplement the**

**// block store cache twice, and write tests twice.**

TwoQueueCache \*lru.TwoQueueCache

}

func NewTwoQueueBlockStoreCache(size int64, blockstore BlockStore) (\*TwoQueueBlockStoreCache, error) {

var err error

twoQueueCacheBlockStore := &TwoQueueBlockStoreCache{}

twoQueueCacheBlockStore.CachedBlockStore = blockstore

twoQueueCacheBlockStore.TwoQueueCache, err = lru.New2Q(int(size))

if err != nil {

return nil, err

}

return twoQueueCacheBlockStore, nil

}

**type LRUBlockStoreCache struct** {

CachedBlockStore BlockStore

Cache \*lru.Cache

}

func NewLRUBlockStoreCache(size int64, blockstore BlockStore) (\*LRUBlockStoreCache, error) {

var err error

lruCacheBlockStore := &LRUBlockStoreCache{}

lruCacheBlockStore.CachedBlockStore = blockstore

lruCacheBlockStore.Cache, err = lru.New(int(size))

if err != nil {

return nil, err

}

return lruCacheBlockStore, nil

}

**Caching Store**

type CachingStoreConfig struct {

CachingEnabled bool

**// CachingEnabled may be ignored in some configurations, this will force enable the caching**

**// store in those cases.**

**// Number of cache shards, value must be a power of two**

Shards int

**// Time after we need to evict the key**

EvictionTimeInSeconds int64

**// interval at which clean up of expired keys will occur**

CleaningIntervalInSeconds int64

**// Total size of cache would be: MaxKeys\*MaxSizeOfValueInBytes**

MaxKeys int

MaxSizeOfValueInBytes int

LogLevel string

LogDestination string

}

func (c \*CachingStore) Get(key []byte) []byte {

}

func (c \*CachingStore) Has(key []byte) bool {

}

func (c \*CachingStore) Set(key []byte, value []byte) {

}

func (c \*CachingStore) Delete(key []byte) {

}

func (c \*CachingStore) Range(prefix []byte) plugin.RangeData {

}

func (c \*CachingStore) Hash() []byte {

}

func (c \*CachingStore) Version() int64 {

}

func (c \*CachingStore) SaveVersion() ([]byte, int64, error) {

}

func (c \*CachingStore) Prune() error {

return nil

}

func (c \*CachingStore) GetSnapshot() Snapshot {

}

**EVM Store**

**// EvmStore persists EVM state to a DB.**

type EvmStore struct {

evmDB db.DBWrapper

cache map[string]cacheItem

rootHash []byte

lastSavedRoot []byte

rootCache \*lru.Cache

version int64

}

func evmRootKey(blockHeight int64) []byte {

}

func getVersionFromEvmRootKey(key []byte) (int64, error) {

}

**// EvmStore persists EVM state to a DB.**

type EvmStore struct {

evmDB db.DBWrapper

cache map[string]cacheItem

rootHash []byte

lastSavedRoot []byte

rootCache \*lru.Cache

version int64

}

**// NewEvmStore returns a new instance of the store backed by the given DB.**

func NewEvmStore(evmDB db.DBWrapper, numCachedRoots int) \*EvmStore {

}

func (s \*EvmStore) setCache(key, val []byte, deleted bool) {

}

**// Range iterates in-order over the keys in the store prefixed by the given prefix.**

**// this function, only place it's called is from MultiWriterAppStore.Range but only when**

**// iterating over the "vm"**

**//This version of EvmStore supports Range(nil)**

func (s \*EvmStore) Range(prefix []byte) plugin.RangeData {

**// Update range cache with data in cache**

**// Make Range return root hash (vmvmroot) from EvmStore.rootHash**

**// Sorting makes RangeData deterministic**

}

func (s \*EvmStore) Has(key []byte) bool {

**// EvmStore always has Patricia root**

}

func (s \*EvmStore) Get(key []byte) []byte {

}

func (s \*EvmStore) Delete(key []byte) {

}

func (s \*EvmStore) Set(key, val []byte) {

}

func (s \*EvmStore) Commit(version int64) []byte {

**// save Patricia root of EVM state only if it changes**

}

func (s \*EvmStore) LoadVersion(targetVersion int64) error {

**// find the last saved root**

**// nil root indicates that latest saved root below target version is not found**

}

func (s \*EvmStore) Version() ([]byte, int64) {

}

func (s \*EvmStore) getLastSavedRoot(targetVersion int64) ([]byte, int64) {

}

func (s \*EvmStore) GetSnapshot(version int64) db.Snapshot {

**// Expect cache to be almost 100% hit since cache miss yields extremely poor performance**

}

func NewEvmStoreSnapshot(snapshot db.Snapshot, rootHash []byte) \*EvmStoreSnapshot {

}

type EvmStoreSnapshot struct {

db.Snapshot

rootHash []byte

}

func (s \*EvmStoreSnapshot) Get(key []byte) []byte {

}

func (s \*EvmStoreSnapshot) Has(key []byte) bool {

**// snapshot always has a root hash**

**// nil or empty root hash is considered valid root hash**

}

func remove(keys []string, key string) []string {

}

IAVL Store

type IAVLStore struct {

tree \*iavl.MutableTree

**maxVersions int64 // maximum number of versions to keep when pruning**

**flushInterval int64 // how often we persist to disk**

}

func (s \*IAVLStore) Delete(key []byte) {

}

func (s \*IAVLStore) Set(key, val []byte) {

}

func (s \*IAVLStore) Has(key []byte) bool {

}

func (s \*IAVLStore) Get(key []byte) []byte {

}

**// Returns the bytes that mark the end of the key range for the given prefix.**

func prefixRangeEnd(prefix []byte) []byte {

}

func (s \*IAVLStore) Range(prefix []byte) plugin.RangeData {

**// Tree range gives all keys that has prefix but it does not check zero byte**

**// after the prefix. So we have to check zero byte after prefix using util.HasPrefix**

**// Skip this key as it does not have the prefix**

}

func (s \*IAVLStore) Hash() []byte {

}

func (s \*IAVLStore) Version() int64 {

}

func (s \*IAVLStore) SaveVersion() ([]byte, int64, error) {

**//Every X versions we should persist to disk**

**//Flushing mem to disk at version**

**//failed to save tree version**

}

func (s \*IAVLStore) Prune() error {

}

func (s \*IAVLStore) GetSnapshot() Snapshot {

}

**// NewIAVLStore creates a new IAVLStore.**

**// maxVersions can be used to specify how many versions should be retained, if set to zero then**

**// old versions will never been deleted.**

**// targetVersion can be used to load any previously saved version of the store, if set to zero then**

**// the last version that was saved will be loaded.**

func NewIAVLStore(db dbm.DB, maxVersions, targetVersion, flushInterval int64) (\*IAVLStore, error) {

// always keep at least 2 of the last versions

if (maxVersions != 0) && (maxVersions < 2) {

maxVersions = 2

}

return &IAVLStore{

tree: tree,

maxVersions: maxVersions,

flushInterval: flushInterval,

}, nil

}

type iavlStoreSnapshot struct {

}

func (s \*iavlStoreSnapshot) Release() {

}

**LogStore**

**LogStore layer on top of existing KV store to generate a log of all store operations, can be used for debugging and turned off to reduce latencies and in production**

type LogStore struct {

store VersionedKVStore

logger log.Logger

params LogParams

}

func NewLogStore(store VersionedKVStore) (ls \*LogStore, err error) {

ls.params = LogParams{

LogFilename: "app-store.log",

LogFlags: 0,

LogVersion: false,

LogDelete: true,

LogSetKey: true,

LogSetValue: false,

LogSetSize: true,

LogGet: false,

LogRange: false,

LogHas: false,

LogSaveVersion: false,

LogHash: false,

}

}

func (s \*LogStore) Delete(key []byte) {

if s.params.LogDelete {

s.logger.Println("Delete key: ", string(key))

}

s.store.Delete(key)

}

func (s \*LogStore) Set(key, val []byte) {

if s.params.LogSetKey {

s.logger.Println("Set key: ", string(key))

}

if s.params.LogSetValue {

s.logger.Println("Set Value: ", string(val))

}

if s.params.LogSetSize {

s.logger.Println("Set Size: ", len(val))

}

s.store.Set(key, val)

}

func (s \*LogStore) Has(key []byte) bool {

if s.params.LogHas {

s.logger.Println("Has key: ", string(key))

}

return s.store.Has(key)

}

func (s \*LogStore) Range(prefix []byte) plugin.RangeData {

val := s.store.Range(prefix)

if s.params.LogRange {

s.logger.Println("Range prefix: ", string(prefix), " val: ", val)

}

return val

}

func (s \*LogStore) Get(key []byte) []byte {

val := s.store.Get(key)

if s.params.LogGet {

s.logger.Println("Get key: ", string(key), " val: ", val)

}

return val

}

func (s \*LogStore) Hash() []byte {

hash := s.store.Hash()

if s.params.LogHash {

s.logger.Println("Hash ", hash)

}

return hash

}

func (s \*LogStore) Version() int64 {

version := s.store.Version()

if s.params.LogVersion {

s.logger.Println("Version ", version)

}

return version

}

func (s \*LogStore) SaveVersion() ([]byte, int64, error) {

vByte, vInt, err := s.store.SaveVersion()

if s.params.LogSaveVersion {

s.logger.Println("SaveVersion", string(vByte), " int ", vInt, " err ", err)

}

return vByte, vInt, err

}

func (s \*LogStore) Prune() error {

return s.store.Prune()

}

func (s \*LogStore) GetSnapshot() Snapshot {

return s.store.GetSnapshot()

}

Memstore

type MemStore struct {

store map[string][]byte

}

func NewMemStore() \*MemStore {

}

func (m \*MemStore) Range(prefix []byte) plugin.RangeData {

}

// Get returns nil iff key doesn't exist. Panics on nil key.

func (m \*MemStore) Get(key []byte) []byte {

}

// Has checks if a key exists.

func (m \*MemStore) Has(key []byte) bool {

}

// Set sets the key. Panics on nil key.

func (m \*MemStore) Set(key, value []byte) {

}

// Delete deletes the key. Panics on nil key.

func (m \*MemStore) Delete(key []byte) {

}

func (m \*MemStore) Hash() []byte {

}

func (m \*MemStore) Version() int64 {

}

func (m \*MemStore) SaveVersion() ([]byte, int64, error) {

}

func (m \*MemStore) Prune() error {

}

func (m \*MemStore) GetSnapshot() Snapshot {

}

**MultiWriter App Store**

**Caching store is wrapped on top of appStore, evmStore and transactions are batch serialised to disk store (levelDB, badgerDB, boltDB (In built LRU cache is not sufficient for the volume, latencies, functionality required, allegro big cache is layered on top to meet latencies, volume and serialization requirements**

**Data chain can deploy both solidity based persona smart contracts as well as go based persona smart contracts**

type MultiWriterAppStore struct {

appStore \*IAVLStore

evmStore \*EvmStore

lastSavedTree unsafe.Pointer // \*iavl.ImmutableTree

onlySaveEvmStateToEvmStore bool

}

// NewMultiWriterAppStore creates a new NewMultiWriterAppStore.

func NewMultiWriterAppStore(appStore \*IAVLStore, evmStore \*EvmStore, saveEVMStateToIAVL bool) (\*MultiWriterAppStore, error) {

// if root is nil, this is the first run after migration, so get evmroot from vmvmroot

if !bytes.Equal(appStoreEvmRoot, evmStoreEvmRoot) {

return nil, fmt.Errorf("EVM roots mismatch, evm.db(%d): %X, app.db(%d): %X",

version, evmStoreEvmRoot, appStore.Version(), appStoreEvmRoot)

}

// feature flag overrides SaveEVMStateToIAVL

}

func (s \*MultiWriterAppStore) Delete(key []byte) {

}

func (s \*MultiWriterAppStore) Set(key, val []byte) {

}

func (s \*MultiWriterAppStore) Has(key []byte) bool {

}

func (s \*MultiWriterAppStore) Get(key []byte) []byte {

}

// Range iterates in-order over the keys in the store prefixed by the given prefix.

func (s \*MultiWriterAppStore) Range(prefix []byte) plugin.RangeData {

}

func (s \*MultiWriterAppStore) Hash() []byte {

}

func (s \*MultiWriterAppStore) Version() int64 {

}

func (s \*MultiWriterAppStore) SaveVersion() ([]byte, int64, error) {

// Tie up Patricia tree with IAVL tree.

// Do this after the feature flag is enabled so that we can detect

// inconsistency in evm.db across the cluster

// AppStore 3.1 write EVM root to app.db only if it changes

}

func (s \*MultiWriterAppStore) setLastSavedTreeToVersion(version int64) error {

"failed to load immutable tree for version"

}

func (s \*MultiWriterAppStore) Prune() error {

}

func (s \*MultiWriterAppStore) GetSnapshot() Snapshot {

}

type multiWriterStoreSnapshot struct {

evmDbSnapshot db.Snapshot

appStoreTree \*iavl.ImmutableTree

}

func newMultiWriterStoreSnapshot(evmDbSnapshot db.Snapshot, appStoreTree \*iavl.ImmutableTree) \*multiWriterStoreSnapshot {

}

func (s \*multiWriterStoreSnapshot) Release() {

}

func (s \*multiWriterStoreSnapshot) Has(key []byte) bool {

}

func (s \*multiWriterStoreSnapshot) Get(key []byte) []byte {

}

// Range iterates in-order over the keys in the store prefixed by the given prefix.

func (s \*multiWriterStoreSnapshot) Range(prefix []byte) plugin.RangeData {

errors.New("Range over nil prefix not implemented")

// Otherwise iterate over the IAVL tree

// Tree range gives all keys that has prefix but it does not check zero byte

// after the prefix. So we have to check zero byte after prefix using util.HasPrefix

}

**Pruning IAVL Store**

type PruningIAVLStoreConfig struct {

MaxVersions int64 // maximum number of versions to keep when pruning

BatchSize int64 // maximum number of versions to delete in each cycle

FlushInterval int64 // number of versions before flushing to disk

Interval time.Duration

Logger \*datachain.Logger

}

// PruningIAVLStore is a specialized IAVLStore that has a background thread that periodically prunes

// old versions. It should only be used to prune old clusters, on new clusters nodes will delete

// a version each time they save a new one, so the background thread, and all the extra locking

// is unnecessary.

type PruningIAVLStore struct {

store \*IAVLStore

mutex \*sync.RWMutex

oldestVer int64

maxVersions int64

batchSize int64

batchCount uint64

logger \*datachain.Logger

}

// NewPruningIAVLStore creates a new PruningIAVLStore.

// maxVersions can be used to specify how many versions should be retained, if set to zero then

// old versions will never been deleted.

//Race conditions preventions

//Race conditions detected in modules from go race detector reports

func NewPruningIAVLStore(db dbm.DB, cfg PruningIAVLStoreConfig) (\*PruningIAVLStore, error) {

// always keep at least 2 of the latest versions

}

func (s \*PruningIAVLStore) Delete(key []byte) {

s.mutex.Lock()

defer s.mutex.Unlock()

s.store.Delete(key)

}

func (s \*PruningIAVLStore) Set(key, val []byte) {

s.mutex.Lock()

defer s.mutex.Unlock()

s.store.Set(key, val)

}

func (s \*PruningIAVLStore) Has(key []byte) bool {

s.mutex.RLock()

defer s.mutex.RUnlock()

return s.store.Has(key)

}

func (s \*PruningIAVLStore) Get(key []byte) []byte {

s.mutex.RLock()

defer s.mutex.RUnlock()

return s.store.Get(key)

}

func (s \*PruningIAVLStore) Range(prefix []byte) plugin.RangeData {

s.mutex.RLock()

defer s.mutex.RUnlock()

return s.store.Range(prefix)

}

func (s \*PruningIAVLStore) Hash() []byte {

s.mutex.Lock()

defer s.mutex.Unlock()

return s.store.Hash()

}

func (s \*PruningIAVLStore) Version() int64 {

s.mutex.RLock()

defer s.mutex.RUnlock()

return s.store.Version()

}

func (s \*PruningIAVLStore) SaveVersion() ([]byte, int64, error) {

s.mutex.Lock()

defer s.mutex.Unlock()

hash, ver, err := s.store.SaveVersion()

if err == nil && s.oldestVer == 0 {

s.oldestVer = ver

}

return hash, ver, err

}

func (s \*PruningIAVLStore) Prune() error {

// pruning is done in the goroutine, so do nothing here

return nil

}

func (s \*PruningIAVLStore) GetSnapshot() Snapshot {

// This isn't an actual snapshot obviously, and never will be, but lets pretend...

return &pruningIAVLStoreSnapshot{

PruningIAVLStore: s,

}

}

func (s \*PruningIAVLStore) prune() error {

s.mutex.Lock()

defer s.mutex.Unlock()

var err error

defer func(begin time.Time) {

lvs := []string{"error", fmt.Sprint(err != nil)}

pruneDuration.With(lvs...).Observe(time.Since(begin).Seconds())

}(time.Now())

latestVer := s.store.Version()

endVer := latestVer - s.maxVersions

if (s.oldestVer == 0) || (s.oldestVer > endVer) {

return nil // nothing to prune yet

}

if (endVer - s.oldestVer) > s.batchSize {

endVer = s.oldestVer + s.batchSize

}

if endVer > (latestVer - 2) {

endVer = latestVer - 2

}

for i := s.oldestVer; i <= endVer; i++ {

if s.store.tree.VersionExists(i) {

if err = s.deleteVersion(i); err != nil {

return errors.Wrapf(err, "failed to delete tree version %d", i)

}

}

s.oldestVer++

}

s.batchCount++

return nil

}

func (s \*PruningIAVLStore) deleteVersion(ver int64) error {

var err error

defer func(begin time.Time) {

lvs := []string{"error", fmt.Sprint(err != nil)}

deleteVersionDuration.With(lvs...).Observe(time.Since(begin).Seconds())

}(time.Now())

err = s.store.tree.DeleteVersion(ver)

return err

}

// runWithRecovery should run in a goroutine, it will ensure the given function keeps on running in

// a goroutine as long as it doesn't panic due to a runtime error.

//Healing goroutines with hearbeat architecture to detect if goroutine is alive and restart if go routine is dead

//Retry Algorithm to restart go routines – using exponential decay and jitters

func (s \*PruningIAVLStore) runWithRecovery(run func()) {

defer func() {

if r := recover(); r != nil {

s.logger.Error("Recovered from panic in PruningIAVLStore goroutine", "r", r)

// Unless it's a runtime error restart the goroutine

if \_, ok := r.(runtime.Error); !ok {

time.Sleep(30 \* time.Second)

s.logger.Info("Restarting PruningIAVLStore goroutine...\n")

go s.runWithRecovery(run)

}

}

}()

run()

}

// loopWithInterval will execute the step function in an endless loop, sleeping for the specified

// interval at the end of each loop iteration.

func (s \*PruningIAVLStore) loopWithInterval(step func() error, interval time.Duration) {

for {

if err := step(); err != nil {

s.logger.Error("PruneIAVLStore encountered an error", "err", err)

}

time.Sleep(interval)

}

}

type pruningIAVLStoreSnapshot struct {

\*PruningIAVLStore

}

func (s \*pruningIAVLStoreSnapshot) Release() {

}

**Versioned Caching Store**

// KeyVersionTable keeps versions of a cached key

type KeyVersionTable map[int64]bool

func versionedKey(key string, version int64) string {

}

func unversionedKey(key string) (string, int64, error) {

}

type versionedBigCache struct {

cache \*bigcache.BigCache

cacheLogger \*datachain.Logger

keyTableMutex sync.RWMutex

keyTable map[string]KeyVersionTable

}

func newVersionedBigCache(config \*CachingStoreConfig, cacheLogger \*datachain.Logger) (\*versionedBigCache, error) {

// when a key get evicted from BigCache, KeyVersionTable and KeyTable must be updated

}

func (c \*versionedBigCache) onRemove(key string, entry []byte){

}

func (c \*versionedBigCache) Delete(key []byte, version int64) error {

// delete data in cache if it does exist

c.cache.Delete(string(versionedKey))

// add key to inidicate that this is the latest version but

// the data has been deleted

}

func (c \*versionedBigCache) Set(key, val []byte, version int64) error {

versionedKey := versionedKey(string(key), version)

c.addKeyVersion(key, version)

}

func (c \*versionedBigCache) Get(key []byte, version int64) ([]byte, error) {

latestVersion := c.getKeyVersion(key, version)

versionedKey := versionedKey(string(key), latestVersion)

return c.cache.Get(string(versionedKey))

}

// getKeyVersion returns the latest version number (limited by version argument) of a particular key

func (c \*versionedBigCache) getKeyVersion(key []byte, version int64) int64 {

}

// addKeyVersion adds version number of a key to KeyVersionTable

func (c \*versionedBigCache) addKeyVersion(key []byte, version int64) {

}

// versionedCachingStore wraps a write-through cache around a VersionedKVStore.

// It is compatible with MultiWriterAppStore only.

type versionedCachingStore struct {

VersionedKVStore

cache \*versionedBigCache

version int64

logger \*datachain.Logger

}

// NewVersionedCachingStore wraps the source VersionedKVStore in a cache.

func NewVersionedCachingStore(

source VersionedKVStore, config \*CachingStoreConfig, version int64,

) (VersionedKVStore, error) {

}

func (c \*versionedCachingStore) Delete(key []byte) {

}

func (c \*versionedCachingStore) Set(key, val []byte) {

}

func (c \*versionedCachingStore) SaveVersion() ([]byte, int64, error) {)

if err == nil {

// Cache version is always 1 block ahead of KV store version, that way when

// GetSnapshot() is called it won't return the current unpersisted state of the cache,

// but rather the last persisted version.

c.version = version + 1

}

}

func (c \*versionedCachingStore) GetSnapshot() Snapshot {

}

// CachingStoreSnapshot is a read-only CachingStore with specified version

type versionedCachingStoreSnapshot struct {

Snapshot

cache \*versionedBigCache

version int64

logger \*datachain.Logger

}

func newVersionedCachingStoreSnapshot(snapshot Snapshot, cache \*versionedBigCache{

}

func (c \*versionedCachingStoreSnapshot) Delete(key []byte) {

panic("[versionedCachingStoreSnapshot] Delete() not implemented")

}

func (c \*versionedCachingStoreSnapshot) Set(key, val []byte) {

panic("[versionedCachingStoreSnapshot] Set() not implemented")

}

func (c \*versionedCachingStoreSnapshot) Has(key []byte) bool {

}

func (c \*versionedCachingStoreSnapshot) Get(key []byte) []byte {

}

func (c \*versionedCachingStoreSnapshot) SaveVersion() ([]byte, int64, error) {

return nil, 0, errors.New("[VersionedCachingStoreSnapshot] SaveVersion() not implemented")

}

func (c \*versionedCachingStoreSnapshot) Prune() error {

return errors.New("[VersionedCachingStoreSnapshot] Prune() not implemented")

}

func (c \*versionedCachingStoreSnapshot) Release() {

c.Snapshot.Release()

}

**Persona stores**

// KVReader interface for reading data out of a Persona store

type KVReader interface {

// Get returns nil iff key doesn't exist. Panics on nil key.

Get(key []byte) []byte

// Range returns a range of keys

Range(prefix []byte) plugin.RangeData

// Has checks if a key exists.

Has(key []byte) bool

}

type KVWriter interface {

// Set sets the key. Panics on nil key.

Set(key, value []byte)

// Delete deletes the key. Panics on nil key.

Delete(key []byte)

}

type KVStore interface {

KVReader

KVWriter

}

type KVStoreTx interface {

KVStore

Commit()

Rollback()

}

type AtomicKVStore interface {

KVStore

BeginTx() KVStoreTx

}

type Snapshot interface {

KVReader

Release()

}

type VersionedKVStore interface {

KVStore

Hash() []byte

Version() int64

SaveVersion() ([]byte, int64, error)

// Delete old version of the store

Prune() error

GetSnapshot() Snapshot

}

type cacheItem struct {

Value []byte

Deleted bool

}

type txAction int

const (

txSet txAction = iota

txDelete

)

type tempTx struct {

Action txAction

Key, Value []byte

}

// cacheTx is a simple write-back cache

type cacheTx struct {

store KVStore

cache map[string]cacheItem

// tmpTxs preserves the order of set and delete actions

tmpTxs []tempTx

**//In case of Rollback - tmpTxs []tempTx this array is reset and all transactions stored in transaction array is flushed (Persona Sync Error)**

**//In case of commit - tmpTxs []tempTx this array is read and transactions are executed in order as stored in array** **(GET,SET,DELETE)**

}

func newCacheTx(store KVStore) \*cacheTx {

c := &cacheTx{

store: store,

}

c.Rollback()

return c

}

func (c \*cacheTx) addAction(action txAction, key, value []byte) {

c.tmpTxs = append(c.tmpTxs, tempTx{

Action: action,

Key: key,

Value: value,

})

}

func (c \*cacheTx) setCache(key, val []byte, deleted bool) {

c.cache[string(key)] = cacheItem{

Value: val,

Deleted: deleted,

}

}

func (c \*cacheTx) Delete(key []byte) {

c.addAction(txDelete, key, nil)

c.setCache(key, nil, true)

}

func (c \*cacheTx) Set(key, val []byte) {

c.addAction(txSet, key, val)

c.setCache(key, val, false)

}

func (c \*cacheTx) Range(prefix []byte) plugin.RangeData {

return c.store.Range(prefix)

}

func (c \*cacheTx) Has(key []byte) bool {

if item, ok := c.cache[string(key)]; ok {

return !item.Deleted

}

return c.store.Has(key)

}

func (c \*cacheTx) Get(key []byte) []byte {

if item, ok := c.cache[string(key)]; ok {

return item.Value

}

return c.store.Get(key)

}

func (c \*cacheTx) Commit() {

for \_, tx := range c.tmpTxs {

if tx.Action == txSet {

c.store.Set(tx.Key, tx.Value)

} else if tx.Action == txDelete {

c.store.Delete(tx.Key)

} else {

panic("invalid cacheTx action type")

}

}

}

func (c \*cacheTx) Rollback() {

c.tmpTxs = make([]tempTx, 0)

c.cache = make(map[string]cacheItem)

}

type atomicWrapStore struct {

KVStore

}

func (a \*atomicWrapStore) BeginTx() KVStoreTx {

return newCacheTx(a)

}

func WrapAtomic(store KVStore) AtomicKVStore {

return &atomicWrapStore{

KVStore: store,

}

}

//For use of prefixes in front of keys for different operations

//prefixes can be - evm, plugin, whitelist

type prefixReader struct {

prefix []byte

reader KVReader

}

func (r \*prefixReader) Range(prefix []byte) plugin.RangeData {

return r.reader.Range(util.PrefixKey(r.prefix, prefix))

}

func (r \*prefixReader) Get(key []byte) []byte {

return r.reader.Get(util.PrefixKey(r.prefix, key))

}

func (r \*prefixReader) Has(key []byte) bool {

return r.reader.Has(util.PrefixKey(r.prefix, key))

}

func PrefixKVReader(prefix []byte, reader KVReader) KVReader {

return &prefixReader{

prefix: prefix,

reader: reader,

}

}

type prefixWriter struct {

prefix []byte

writer KVWriter

}

func (w \*prefixWriter) Set(key, val []byte) {

w.writer.Set(util.PrefixKey(w.prefix, key), val)

}

func (w \*prefixWriter) Delete(key []byte) {

w.writer.Delete(util.PrefixKey(w.prefix, key))

}

func PrefixKVWriter(prefix []byte, writer KVWriter) KVWriter {

return &prefixWriter{

prefix: prefix,

writer: writer,

}

}

type prefixStore struct {

prefixReader

prefixWriter

}

func PrefixKVStore(prefix []byte, store KVStore) KVStore {

return &prefixStore{

prefixReader{

prefix: prefix,

reader: store,

},

prefixWriter{

prefix: prefix,

writer: store,

},

}

}

**Different Type of Operations Supported**

**1)Read**

**2)Write**

**3)Migrate**

**There is a separate handler for each of type of operation.**

**Virtual Machines**

**Features**

Solidity contract deployment

Go contract deployment

i)Go contract based virtual machine

ii)Ethereum (Solidity) contracts based virtual machine

**Go contracts state (Order of transactions (KVs)) – state saved to app.db**

**Ethereum (solidity) contract state (Order of Transactions (KVs)) - EVM state saved to evm.db**

**Chainconfig**

chain-cfg list-features

chain-cfg add-feature hardfork multichain --build 0 --no-auto-enable -k {{index $.NodePrivKeyPathList 0}}

chain-cfg add-feature hardfork --build 0 -k {{index $.NodePrivKeyPathList 2}}

chain-cfg add-feature hardfork --build 0 -k {{index $.NodePrivKeyPathList 3}}

chain-cfg add-feature hardfork --build 0 -k {{index $.NodePrivKeyPathList 0}}

chain-cfg enable-feature hardfork -k {{index $.NodePrivKeyPathList 0}}

chain-cfg enable-feature hardfork -k {{index $.NodePrivKeyPathList 1}}

chain-cfg get-feature hardfork

chain-cfg enable-feature hardfork multichain -k {{index $.NodePrivKeyPathList 2}}

chain-cfg get-feature hardfork

chain-cfg add-feature newfeature --build=0 --no-auto-enable -k {{index $.NodePrivKeyPathList 0}}

chain-cfg add-feature chaincfg:v1.2 --build 0 -k {{index $.NodePrivKeyPathList 0}}

chain-cfg list-features

chain-cfg enable-feature chaincfg:v1.2 -k {{index $.NodePrivKeyPathList 0}}

chain-cfg enable-feature chaincfg:v1.2 -k {{index $.NodePrivKeyPathList 1}}

chain-cfg enable-feature chaincfg:v1.2 -k {{index $.NodePrivKeyPathList 2}}

chain-cfg enable-feature chaincfg:v1.2 -k {{index $.NodePrivKeyPathList 3}}

chain-cfg list-features

chain-cfg get-feature hardfork

chain-cfg get-params

chain-cfg set-params -k {{index $.NodePrivKeyPathList 0}} --vote-threshold 50 --block-confirmations 1000

chain-cfg get-params

chain-cfg set-params -k {{index $.NodePrivKeyPathList 0}} --vote-threshold 100 --block-confirmations 1

chain-cfg get-params

chain-cfg get-feature hardfork

chain-cfg enable-feature hardfork -k {{index $.NodePrivKeyPathList 3}}

chain-cfg get-feature hardfork

chain-cfg add-feature feature1 feature2 --build 567 --no-auto-enable -k {{index $.NodePrivKeyPathList 0}}

chain-cfg get-feature feature1

chain-cfg get-feature feature2

chain-cfg set-datavalidator-info --build 578 -k {{index $.AccountPrivKeyPathList 0}}

chain-cfg set-datavalidator-info --build 578 -k {{index $.NodePrivKeyPathList 0}}

chain-cfg get-datavalidator-info {{index $.NodeAddressList 0}}

chain-cfg set-datavalidator-info --build 50 -k {{index $.NodePrivKeyPathList 1}}

chain-cfg set-datavalidator-info --build 50 -k {{index $.NodePrivKeyPathList 2}}

chain-cfg set-datavalidator-info --build 50 -k {{index $.NodePrivKeyPathList 3}}

**DPOS contract**

coin approve dposV3 1250000 -k {{index $.NodePrivKeyPathList 0}}

dpos3 register-candidate {{index $.NodePubKeyList 0}} 100 3 -k {{index $.NodePrivKeyPathList 0}} --name DATACHAIN\_0

coin approve dposV3 1250000 -k {{index $.NodePrivKeyPathList 1}}

dpos3 register-candidate {{index $.NodePubKeyList 1}} 100 3 -k {{index $.NodePrivKeyPathList 1}} --name DATACHAIN\_1

coin approve dposV3 1250000 -k {{index $.NodePrivKeyPathList 2}}

dpos3 register-candidate {{index $.NodePubKeyList 2}} 100 3 -k {{index $.NodePrivKeyPathList 2}} --name DATACHAIN\_2

dpos3 list-candidates

chain-cfg list-datavalidators

**Coin Contract**

**Side chain cryptocurrency token code - ERC20 Token port in GoLang**

**(Aajtak.IndiaToday Custom cryptocurrency coin)**

coin balance {{index $.AccountAddressList 0}}

coin balance {{index $.AccountAddressList 1}}

coin balance {{index $.AccountAddressList 2}}

coin transfer {{index $.AccountAddressList 2}} 20000000 -k {{index $.AccountPrivKeyPathList 2}}

coin balance {{index $.AccountAddressList 1}}

coin balance {{index $.AccountAddressList 2}}

**DPOS Contract**

**Consensus Algorithm for Persona Transactions data chain**

dpos3 list-datavalidators

coin approve dposV3 1250000 -k {{index $.NodePrivKeyPathList 1}}

**# Note: Node1 registers with a 10% maximum referral fee**

dpos3 register-candidate {{index $.NodePubKeyList 1}} 100 0 1000 --name numero-uno --website one.com --description the-number-one-validator-in-the-world -k {{index $.NodePrivKeyPathList 1}}

coin approve dposV3 1250000 -k {{index $.NodePrivKeyPathList 2}}

dpos3 register-candidate {{index $.NodePubKeyList 2}} 50 0 600 --name numero-dos -k {{index $.NodePrivKeyPathList 2}}

dpos3 change-fee 100 -k {{index $.NodePrivKeyPathList 2}}

dpos3 set-min-candidate-fee 900 -k {{index $.NodePrivKeyPathList 0}}

dpos3 change-fee 200 -k {{index $.NodePrivKeyPathList 2}}

dpos3 list-candidates

coin approve dposV3 10 -k {{index $.NodePrivKeyPathList 1}}

coin approve dposV3 20 -k {{index $.NodePrivKeyPathList 2}}

dpos3 delegate {{index $.NodeAddressList 1}} 20 -k {{index $.NodePrivKeyPathList 2}}

coin approve dposV3 1250000 -k {{index $.NodePrivKeyPathList 3}}

coin approve dposV3 200000 -k {{index $.AccountPrivKeyPathList 0}}

dpos3 delegate {{index $.NodeAddressList 2}} 200000 -k {{index $.AccountPrivKeyPathList 0}}

dpos3 register-candidate {{index $.NodePubKeyList 3}} 1200 -k {{index $.NodePrivKeyPathList 3}}

dpos3 update-candidate-info updated-name updated-description update.com 1000 -k {{index $.NodePrivKeyPathList 1}}

dpos3 list-candidates

dpos3 list-datavalidators

coin approve dposV3 20 -k {{index $.NodePrivKeyPathList 3}}

dpos3 delegate {{index $.NodeAddressList 2}} 20 -k {{index $.NodePrivKeyPathList 3}}

dpos3 delegate {{index $.NodeAddressList 3}} 10 -k {{index $.NodePrivKeyPathList 1}}

dpos3 list-datavalidators

dpos3 check-delegation {{index $.NodeAddressList 2}} {{index $.NodeAddressList 3}}

dpos3 check-delegation {{index $.NodeAddressList 1}} {{index $.NodeAddressList 2}}

dpos3 check-delegation {{index $.NodeAddressList 3}} {{index $.NodeAddressList 1}}

dpos3 list-datavalidators

coin balance {{index $.NodeAddressList 2}}

dpos3 unbond {{index $.NodeAddressList 1}} 15 1 -k {{index $.NodePrivKeyPathList 2}}

coin balance {{index $.NodeAddressList 2}}

**# After node2 unbonded from node1, node1 should be removed from datavalidators list (num-validators = 2) and be replaced with node3 which only had a delegation 1/2 as large as 1 & 2.**

dpos3 list-datavalidators

coin approve dposV3 20 -k {{index $.NodePrivKeyPathList 2}}

dpos3 check-delegation {{index $.NodeAddressList 1}} {{index $.NodeAddressList 2}}

dpos3 delegate {{index $.NodeAddressList 1}} 20 -k {{index $.NodePrivKeyPathList 2}}

dpos3 check-delegation {{index $.NodeAddressList 1}} {{index $.NodeAddressList 2}}

dpos3 list-datavalidators

dpos3 unbond {{index $.NodeAddressList 1}} 15 2 -k {{index $.NodePrivKeyPathList 2}}

dpos3 check-delegation {{index $.NodeAddressList 1}} {{index $.NodeAddressList 2}}

dpos3 unbond {{index $.NodeAddressList 1}} 5 1 -k {{index $.NodePrivKeyPathList 2}}

dpos3 check-delegation {{index $.NodeAddressList 1}} {{index $.NodeAddressList 2}}

**# Redelegation testing**

dpos3 redelegate {{index $.NodeAddressList 1}} {{index $.NodeAddressList 2}} 1 -k {{index $.NodePrivKeyPathList 3}}

dpos3 check-delegation {{index $.NodeAddressList 1}} {{index $.NodeAddressList 3}}

dpos3 list-delegations {{index $.NodeAddressList 1}}

dpos3 list-all-delegations

dpos3 check-all-delegations {{index $.NodeAddressList 1}}

dpos3 register-referrer acct1 {{index $.AccountAddressList 1}} -k {{index $.NodePrivKeyPathList 0}}

dpos3 register-referrer acct2 {{index $.AccountAddressList 2}} -k {{index $.NodePrivKeyPathList 0}}

coin approve dposV3 40 -k {{index $.NodePrivKeyPathList 2}}

dpos3 list-candidates

dpos3 delegate {{index $.NodeAddressList 1}} 20 0 acct1 -k {{index $.NodePrivKeyPathList 2}}

dpos3 delegate {{index $.NodeAddressList 1}} 20 0 acct2 -k {{index $.NodePrivKeyPathList 2}}

**# referrer acct1 should have a non-negative reward distribution**

dpos3 check-all-delegations {{index $.AccountAddressList 1}}

**# referrer acct2 should have a non-negative reward distribution**

dpos3 check-all-delegations {{index $.AccountAddressList 2}}

dpos3 set-downtime-period 1 -k {{index $.NodePrivKeyPathList 0}}

dpos3 enable-datavalidator-jailing true -k {{index $.NodePrivKeyPathList 0}}

coin approve dposV3 1250000 -k {{index $.NodePrivKeyPathList 0}}

dpos3 register-candidate {{index $.NodePubKeyList 0}} 100 -k {{index $.NodePrivKeyPathList 0}}

coin approve dposV3 1250000 -k {{index $.NodePrivKeyPathList 1}}

dpos3 register-candidate {{index $.NodePubKeyList 1}} 100 -k {{index $.NodePrivKeyPathList 1}}

coin approve dposV3 1250000 -k {{index $.NodePrivKeyPathList 2}}

dpos3 register-candidate {{index $.NodePubKeyList 2}} 100 --name numero\_dos -k {{index $.NodePrivKeyPathList 2}}

coin approve dposV3 1250000 -k {{index $.NodePrivKeyPathList 3}}

dpos3 register-candidate {{index $.NodePubKeyList 3}} 100 -k {{index $.NodePrivKeyPathList 3}}

dpos3 list-candidates

coin approve dposV3 222 -k {{index $.NodePrivKeyPathList 2}}

dpos3 delegate {{index $.NodeAddressList 1}} 222 0 -k {{index $.NodePrivKeyPathList 2}}

dpos3 check-all-delegations {{index $.NodeAddressList 2}}"

**# kill node 1 for 15s, it should be jailed after going offline for 15s**

# **add some delay so that downtime periods of node 1 are cleared out**

dpos3 downtime-record {{index $.NodeAddressList 1}} -u {{index $.NodeProxyAppAddressList 2}}

# **wait for block height to increase at least 5 blocks**

**# wait\_for\_block\_height\_to\_increase 0 5**

**# check downtime periods, it should be cleared out after 5 blocks**

dpos3 downtime-record {{index $.NodeAddressList 1}} -u {{index $.NodeProxyAppAddressList 2}}

**# node 1 should be jailed as it is offline for 15s**

dpos3 list-candidates -u {{index $.NodeProxyAppAddressList 2}}

**# check downtime periods before unjailing thedatavalidator**

dpos3 downtime-record {{index $.NodeAddressList 1}} -u {{index $.NodeProxyAppAddressList 2}}

dpos3 unjail-datavalidator -k {{index $.NodePrivKeyPathList 1}} -u {{index $.NodeProxyAppAddressList 2}}

**# wait for block height to increase at least 2 blocks**

**# wait\_for\_block\_height\_to\_increase 0 2**

dpos3 list-candidates -u {{index $.NodeProxyAppAddressList 2}}

**Transfer Gateway**

**Application – Synchronizing data currency as well as normal currency across chains**

**Interoperable Blockchain**

func (gw \*PersonaGateway) GetDataValidatorAuthStrategy(ctx contract.StaticContext, req \*GetDataValidatorAuthStrategyRequest) (\*GetDataValidatorAuthStrategyResponse, error) {

func (gw \*PersonaGateway) GetTrustedDataValidators(ctx contract.StaticContext, req \*TrustedDataValidatorsRequest) (\*TrustedDataValidatorsResponse, error) {

func (gw \*PersonaGateway) UpdateTrustedDataValidators(ctx contract.Context, req \*UpdateTrustedDataValidatorsRequest) error {

func (gw \*PersonaGateway) UpdateDataValidatorsAuthStrategy(ctx contract.Context, req \*UpdateDataValidatorsAuthStrategyRequest) error {

func (gw \*PersonaGateway) AddDataOracle(ctx contract.Context, req \*AddDataOracleRequest) error {

func (gw \*PersonaGateway) RemoveDataOracle(ctx contract.Context, req \*RemoveDataOracleRequest) error {

func (gw \*PersonaGateway) ReplaceDataOwner(ctx contract.Context, req \*AddDataOracleRequest) error {

func removeDataOracle(ctx contract.Context, dataoracleAddr datachain.Address) error {

func (gw \*PersonaGateway) GetDataOracles(ctx contract.StaticContext, req \*GetDataOraclesRequest) (\*GetDataOraclesResponse, error) {

**// ProcessPersonaDepositEventByTxHash tries to submit Sync events by tx hash**

**// This method expects that PGCheckTxHashFeature is enabled on chain**

func (gw \*PersonaGateway) ProcessDepositEventByTxHash(ctx contract.Context, req \*ProcessEventBatchRequest) error {

func (gw \*PersonaGateway) ProcessEventBatch(ctx contract.Context, req \*ProcessEventBatchRequest) error {

func (gw \*PersonaGateway) handleDeposit(ctx contract.Context, ev \*MainnetEvent, checkTxHash bool) error {

func (gw \*PersonaGateway) GetState(ctx contract.StaticContext, req \*PersonaGatewayStateRequest) (\*PersonaGatewayStateResponse, error) {

**// WithdrawPersonaToken will attempt to transfer Persona token to the Gateway contract,**

**// if the Persona Sync is successful the contract will create a receipt than can be used by the**

**// depositor to reclaim ownership of the Persona token through the Mainnet Gateway contract.**

**// Currently an entity must complete each Persona withdrawal by reclaiming ownership on Mainnet**

**// before it can make another one withdrawal (even if the Persona tokens originate from different contracts).**

func (gw \*PersonaGateway) WithdrawPersonaToken(ctx contract.Context, req \*WithdrawPersonaTokenRequest) error {

**// WithdrawETHPersona will attempt to transfer ETHPersona to the Gateway contract,**

**// if it's successful it will store a receipt than can be used by the persona depositor to reclaim ownership**

**// of the Persona ETH through the Mainnet Persona Gateway contract.**

**// NOTE: Currently an entity must complete each persona withdrawal by reclaiming ownership on Mainnet**

**// before it can make another persona withdrawal (even if the persona tokens/ETH originate from different**

**// ETH persona contracts).**

func (gw \*PersonaGateway) WithdrawETHPersona(ctx contract.Context, req \*WithdrawETHPersonaRequest) error {

**// SubmitDepositTxHash is called by a user to submit the Ethereum hash of a Persona token deposit to the**

**// Ethereum Persona Sync Gateway contract. Later the Data Oracle will verify the tx hash and forward the deposit event**

**// to the DataChain Gateway contract.**

**// The user must have a mapping between their DataChain address and their ETH address before they**

**// can submit a tx hash.**

func (gw \*PersonaGateway) SubmitPersonaDepositTxHash(ctx contract.Context, req \*SubmitPersonaDepositTxHashRequest) error {

**// ClearInvalidPersonaDepositTxHash is an DataOracle**

func (gw \*PersonaGateway) ClearInvalidPersonaDepositTxHash(ctx contract.Context, req \*ClearInvalidPersonaDepositTxHashRequest) error {

}

**// UnprocessedPersonaDepositTxHashes returns deposit tx hashes that haven’t been processed yet by the DataOracle.**

func (gw \*PersonaGateway) UnprocessedPersonaDepositTxHashes(

ctx contract.StaticContext, req \*UnprocessedPersonaDepositTxHashesRequest,

)

func clearPersonaDepositTxHashIfExists(ctx contract.Context, ownerAddress datachain.Address) error {

**// PersonaWithdrawalReceipt will return the receipt generated by the last successful call to Persona Withdrawl**

**// The receipt can be used to reclaim ownership of the token through the Mainnet Gateway.**

func (gw \*PersonaGateway) WithdrawalReceipt(ctx contract.StaticContext, req \*WithdrawalReceiptRequest) (\*WithdrawalReceiptResponse, error) {

**// assume the caller is the owner if the request doesn't specify one**

**// ConfirmWithdrawalReceipt will attempt to set the Data Oracle signature on an existing withdrawal**

**// receipt. This method is only allowed to be invoked by Data Oracles with withdrawal signing permission,**

**// and only one Data Oracle will ever be able to successfully set the signature for any particular**

**// receipt, all other attempts will error out.**

func (gw \*PersonaGateway) ConfirmPersonaWithdrawalReceipt(ctx contract.Context, req \*ConfirmPersonaWithdrawalReceiptRequest) error {

**// (added as a separate method to not break consensus - backwards compatibility)**

**// ConfirmPersonaWithdrawalReceiptV2 will attempt to set the Data Oracle signature on an existing withdrawal**

**// receipt. This method is allowed to be invoked by any Datadatavalidator ,**

**// and only one Data datavalidator will ever be able to successfully set the signature for any particular**

**// persona receipt, all other attempts will error out.**

func (gw \*PersonaGateway) ConfirmPersonaWithdrawalReceiptV2(ctx contract.Context, req \*ConfirmPersonaWithdrawalReceiptRequestV2) error {

func (gw \*PersonaGateway) calculateHashFromReceipt(mainnetPersonaGatewayAddr \*types.Address, receipt \*tgtypes.PersonaGatewayWithdrawalReceipt) []byte {

func (gw \*PersonaGateway) calculateHashFromReceiptV2(mainnetPersonaGatewayAddr \*types.Address, receipt \*syncpgtypes.PersonaGatewayWithdrawalReceipt) []byte {

func (gw \*PersonaGateway) doConfirmWithdrawalPersonaReceiptV2(ctx contract.Context, account \*LocalAccount, dataoracleSignature []byte) error {

func (gw \*PersonaGateway) doConfirmWithdrawalPersonaReceipt(ctx contract.Context, req \*ConfirmWithdrawalPersonaReceiptRequest) error {

**// PendingPersonaWithdrawals will return the Persona token owner & Persona withdrawal hash for all pending persona withdrawals.**

**// The Data Oracle will call this method periodically and sign all the retrieved hashes.**

func (gw \*PersonaGateway) PendingPersonaWithdrawalsV2(ctx contract.StaticContext, req \*PendingPersonaWithdrawalsRequest) (\*PendingPersonaWithdrawalsResponse, error) {

**// PendingPersonaWithdrawals will return the persona token owner & persona withdrawal hash for all pending persona withdrawals.**

**// The Oracle will call this method periodically and sign all the retrieved hashes.**

func (gw \*PersonaGateway) PendingWithdrawals(ctx contract.StaticContext, req \*PendingPersonaWithdrawalsRequest) (\*PendingPersonaWithdrawalsResponse, error) {

**// GetPersonaWithdrawlsWithStatus will return the withdrawal summary filtered with TxStatus**

func (gw \*PersonaGateway) GetPersonaWithdrawalsWithStatus(ctx contract.StaticContext, req \*PendingPersonaWithdrawalsRequest) (\*PendingPersonaWithdrawalsResponse, error) {

**// ReclaimPersonaDepositorTokens will attempt to transfer any persona tokens that the caller may have deposited**

**// into the Mainnet Gateway but hasn't yet received from the Data Gateway because of a missing**

**// identity or contract mapping.**

func (gw \*PersonaGateway) ReclaimPersonaDepositorTokens(ctx contract.Context, req \*ReclaimPersonaDepositorTokensRequest) error

**// Assume the caller is trying to reclaim their own persona tokens if persona depositors are not specified**

func (gw \*PersonaGateway) GetUnclaimedPersonaTokens(ctx contract.StaticContext, req \*GetUnclaimedPersonaTokensRequest) (\*GetUnclaimedPersonaTokensResponse, error) {

func (gw \*PersonaGateway) GetUnclaimedContractPersonaTokens(

ctx contract.StaticContext, req \*GetUnclaimedContractPersonaTokensRequest,

)

**// ReclaimContractPersonaTokens will attempt to transfer tokens that originated from the specified Mainnet**

**// contract, and that have been deposited to the Mainnet Gateway, but haven't yet been received by**

**// the depositors on the DataChain because of a missing identity or contract mapping. This function**

**// can only be called by the creator of the specified Persona token contract, or the Persona Gateway contract owner**.

func (gw \*PersonaGateway) ReclaimContractPersonaTokens(ctx contract.Context, req \*ReclaimContractPersonaTokensRequest) error {

func reclaimDepositorPersonaTokens(ctx contract.Context, ownerAddr datachain.Address) error {

func reclaimPersonaDepositorTokensForContract(

ctx contract.Context, ownerAddr, persona tokenAddr datachain.Address, unclaimedToken \*UnclaimedToken,

) error {

**// Performs basic validation to ensure all required persona deposit fields are set.**

func validatePersonaTokenDeposit(deposit \*MainnetPersonaTokenDeposited) error {

**// When a Person token is deposited to the Mainnet Gateway mint it on the DataChain if it doesn't exist**

**// yet, and transfer it to the owner's Datachain address.**

func transferPersonaTokenDeposit(

func storeUnclaimedPersonaToken(ctx contract.Context, deposit \*MainnetPersonaTokenDeposited) error {

**// When a Persona token is withdrawn from the Mainnet Gateway find the corresponding withdrawal receipt**

**// and remove it from the owner's account, once the receipt is removed the owner will be able to**

**// initiate another withdrawal to Mainnet.**

func completePersonaTokenWithdraw(ctx contract.Context, state \*GatewayState, withdrawal \*MainnetPersonaTokenWithdrawn, gwType PersonaGatewayType) error {

func loadState(ctx contract.StaticContext) (\*PersonaGatewayState, error) {

func saveState(ctx contract.Context, state \*PersonaGatewayState) error {

**// Returns the address of the DataChain account or contract that corresponds to the given Ethereum address**

func resolveToDataAddr(ctx contract.StaticContext, mapperAddr, ethAddr datachain.Address) (datachain.Address, error) {

**// Returns the address of the Ethereum account or contract that corresponds to the given DataChain address**

func resolveToEthPersonaAddr(ctx contract.StaticContext, mapperAddr, datachainAddr datachain.Address) (datachain.Address, error) {

func loadLocalAccount(ctx contract.StaticContext, owner datachain.Address) (\*LocalAccount, error) {

func saveLocalAccount(ctx contract.Context, acct \*LocalAccount) error {

func loadForeignAccount(ctx contract.StaticContext, owner datachain.Address) (\*ForeignAccount, error) {

func saveForeignAccount(ctx contract.Context, acct \*ForeignAccount) error {

func loadPersonaDepositTxHash(ctx contract.StaticContext, owner datachain.Address) ([]byte, error) {

func deletePersonaDepositTxHash(ctx contract.Context, owner datachain.Address) {

func savePersonaDepositTxHash(ctx contract.Context, owner datachain.Address, txHash []byte) error {

func loadExtendedState(ctx contract.StaticContext) (\*ExtendedState, error) {

func saveExtendedState(ctx contract.Context, extState \*ExtendedState) error {

func addPersonaDepositTxHashSubmitter(ctx contract.StaticContext, extState \*ExtendedState, owner datachain.Address) error {

func removePersonaDepositTxHashSubmitter(ctx contract.StaticContext, extState \*ExtendedState, owner datachain.Address) error {

func addPersonaTokenWithdrawer(ctx contract.StaticContext, state \*PersonaGatewayState, owner datachain.Address) error {

func removePersonaTokenWithdrawer(ctx contract.StaticContext, state \*PersonaGatewayState, owner datachain.Address) error {

func addDataOracle(ctx contract.Context, dataoracleAddr datachain.Address) error {

var Contract plugin.Contract = contract.MakePluginContract(&PersonaGateway{

Type: EthereumPersonaGateway,

})

var datachainContract plugin.Contract = contract.MakePluginContract(&PersonaGateway{

})

var TronContract plugin.Contract = contract.MakePluginContract(&PersonaGateway{

Type: TronPersonaGateway,

})

var BinanceContract plugin.Contract = contract.MakePluginContract(&PersonaGateway{

Type: BinancePersonaGateway,

})

func hasSeenTxHash(ctx contract.StaticContext, txHash []byte) bool {

return ctx.Has(seenTxHashKey(txHash))

}

func saveSeenTxHash(ctx contract.Context, txHash []byte, tokenKind TokenKind) error {

func emitReclaimError(ctx contract.Context, errorMessage string, ownerAddress datachain.Address) error {

func emitWithdrawETHPersonaError(ctx contract.Context, errorMessage string, request \*tgtypes.TransferGatewayWithdrawETHPersonaRequest) error {

func emitWithdrawPersonaTokenError(ctx contract.Context, errorMessage string, request \*tgtypes.TransferGatewayWithdrawPersonaTokenRequest) error {

func emitWithdrawSidechainPersonaTokenError(ctx contract.Context, errorMessage string, request \*tgtypes.TransferGatewayWithdrawPersonaTokenSidechainRequest) error {

func emitResubmitWithdrawalPersonaError(ctx contract.Context, errorMessage string, receipt \*tgtypes.TransferGatewayWithdrawalPersonaReceipt) error {

func getMappedEthPersonaAddress(ctx contract.StaticContext, trustedDataValidators []\*types.Address) ([]common.Address, error) {

**// Returns all unclaimed persona tokens for an publisher account**

func unclaimedPersonaTokensByOwner(ctx contract.StaticContext, ownerAddr datachain.Address) ([]\*UnclaimedPersonaToken, error) {}

**// Returns all unclaimed Persona tokens for a token contract**

func unclaimedPersonaTokenDepositorsByContract(ctx contract.StaticContext, personatokenAddr datachain.Address) ([]datachain.Address, error) { }

func getCurrentDataValidators(ctx contract.StaticContext) ([]\*types.Address, []\*big.Int, \*big.Int, error) {}

func isPersonaTokenKindAllowed(gwType GatewayType, tokenKind TokenKind) bool {

**// UpdatePersonaWithdrawalReceipt updates the status of a pending withdrawal from the Binance Persona gateway.**

**// The status of a pending persona withdrawal can either change to confirmed or rejected. This function**

**// can only be called by the Data Oracle.**

func (gw \*PersonaGateway) UpdateWithdrawalPersonaReceipt(ctx contract.Context, req \*ConfirmPersonaWithdrawalReceiptRequest) error {

}

func filterPersonaWithdrawalsByTxStatus(ctx contract.StaticContext, status TxStatus) ([]\*PendingPersonaWithdrawalSummary, error) {

state, err := loadState(ctx)

**// ResubmitPersonaWithdrawal resubmits a previously rejected token withdrawal, this is currently only**

**// supported by the Binance Persona gateway. Only the original withdrawer can resubmit a reject withdrawal.**

func (gw \*PersonaGateway) ResubmitPersonaWithdrawal(ctx contract.Context, req \*ResubmitPersonaWithdrawalRequest) error {

**// SetPersonaSyncFee sets the fee that should be charged by the gateway per persona withdrawal.**

func (gw \*PersonaGateway) SetPersonaSyncFee(ctx contract.Context, req \*UpdateBinanceSyncFeeRequest) error {

**Transferring a Persona coin from layer 2 data side chain to Ethereum chain**

**via Persona Sync Gateway**

Persona Coin Contract

(Layer 2 data side chain)

Persona Coin Contract

(Ethereum| Tron | Binance)

Data Oracle

Persona Sync Gateway Contract

(Ethereum | Tron | Binance)

Persona Sync Gateway Contract

( Layer 2 data side chain)

**Graphical user interface

Description automatically generated with medium confidence**

**Evolution of Datachain – (Module wise)**

**1)New features developed in data chain in order as updated by chainconfig smart contract**

**Evolution of datachain.**

**Enables support for Binance contract mappings in the Binance Persona Gateway contract**

* + **SGBinanceContractMappingFeature = "sg:binance-cm"**

**Enables support for mapping DataChain accounts to Binance accounts**

* + **AddressMapperVersion1\_1 = "addrmapper:v1.1"**

**Enables processing of txs via MultiChainSignatureTxMiddleware, there's a feature flag per allowed chain ID, e.g. auth:sigtx:default, auth:sigtx:eth**

* + **AuthSigTxFeaturePrefix = "auth:sigtx:"**

**Enables stricter chain-specific signature verification in MultiChainSignatureTxMiddleware**

* + **MultiChainSigTxMiddlewareVersion1\_1 = "mw:mulcsigtx:v1.1"**

**Enables DPOS v3**

* **DPOSVersion3Feature = "dpos:v3"**

**Enables precise rewards calculations in DPOSv3**

* + **DPOSVersion3\_1 = "dpos:v3.1"**

**Enables slashing metrics**

* + **DPOSVersion3\_2 = "dpos:v3.2"**

**Enables jailing offline data validators**

* + **DPOSVersion3\_3 = "dpos:v3.3"**

**Enables both downtime slashing and a parameter flag to toggle jailing offline data validators on/off**

* + **DPOSVersion3\_4 = "dpos:v3.4"**

**Fixes prefixing of referrer keys so that List Referrers method works**

* + **DPOSVersion3\_5 = "dpos:v3.5"**

**Enables rewards to be distributed even when a delegator owns less than 0.01% of the data validator's stake**

**Also makes whitelists give bonuses correctly if whitelist locktime tier is set to be 0-3 (else defaults to 5%)**

* + **DPOSVersion2\_1 = "dpos:v2.1"**

**Enables EVM tx receipts storage in separate DB.**

* + **EvmTxReceiptsVersion2Feature = "receipts:v2"**

**Enables deployer whitelist middleware that only allows whitelisted accounts to**

**deploy contracts & run migrations.**

* + **DeployerWhitelistFeature = "mw:deploy-wl"**

**Enables post commit middleware for user-deployer-whitelist**

* + **UserDeployerWhitelistFeature = "mw:userdeploy-wl"**

**Enables block range & max txs fields in tier info stored in User Deployer Whitelist contract**

* + **UserDeployerWhitelistVersion1\_1Feature = "userdeploy-wl:v1.1"**

**Makes UserDeployerWhitelist.RemoveUserDeployer mark deployer accounts as inactive instead of deleting them.**

* + **UserDeployerWhitelistVersion1\_2Feature = "userdeploy-wl:v1.2"**

**Enables processing of MigrationTx.**

* + **MigrationTxFeature = "tx:migration"**

**Enables specific migrations, each migration has an ID that's prefixed by this string.**

* + **Migration Feature Prefix = "migration:"**

**Enables usage of ctx.Validators() in ChainConfig contract.**

* + **ChainCfgVersion1\_1 = "chaincfg:v1.1"**

**Enables data validator build number tracking via the ChainConfig contract.**

* + **ChainCfgVersion1\_2 = "chaincfg:v1.2"**

**Forces the MultiWriterAppStore to write EVM state only to evm.db, otherwise it'll write EVM state to both evm.db & app.db.**

* + **EvmDBFeature = "db:evm"**

**Enables PersonaCoin v1.1 contract (also applies to ETHPersonaCoin)**

* + **PersonaCoinVersion1\_1Feature = "personacoin:v1.1**

**Enables PersonaCoin v1.2 to validate fields in request of PersonaCoin and ETH Persona Coin contract**

* + **CoinVersion1\_2Feature = "personacoin:v1.2"**

**Force ReceiptHandler to write datachainFilter and EVM TxHash only to receipts\_db, otherwise it'll**

* + **write datachainFilter and EVM TxHash to both receipts\_db & app.db.**
  + **This feature has been deprecated along with legacy code.**
  + **AuxEvmDBFeature = "db:auxevm"**
  + **Force MultiWriterAppStore to write EVM root to app.db only if the root changes**
  + **AppStoreVersion3\_1 = "appstore:v3.1"**

**Enable option to allow checking the registry error**

* + **DeployTxVersion1\_1Feature = "deploytx:v1.1"**

**Restrict the value of call & deploy txs to non-negative uniquess**

* + **CheckTxValueFeature = "tx:check-value"**