Design considerations permanodes by Olaf van Wijk

This document will make a few assumptions based on my own personal needs for a permanode but will consider other use-cases as well and can evolve if other use-cases are added.

The permanode problem

IOTA as a distributed ledger uses the Tangle as a consensus structure allowing for no theoretical limit in the amount of transactions that it can process. Since each transaction validates two other transactions a structure of a direct acyclic graph (DAG) is created and this allows for parallel consensus giving the Tangle it's scaling capabilities. However as more transactions are processed the amount of storage **and memory** required increases dramatically in the current implementation. IOTA therefore uses a mechanism called snapshots to periodically reduce the storage and memory requirements. To prevent misunderstandings there I will make some statements about snapshots:

- Snapshots are generated by taking milestone X and obtain the balances from all addresses referenced under that milestone.
- About the process: https://blog.iota.org/the-april-29-2018-iota-snapshot-and-iri-1-4-2-4-behind-the-scenes-7e034babcd44
- The current way of snapshots is an intermediate solution as stated in the article.
- Snapshots act as a new Genesis block. There is NO relation whatsoever anymore, AT ALL, to the previous tangle. The output of the previous tangle in the form of address balances is used to bootstrap the new tangle.
- For visualization: this means that the first milestone will reference the Genesis transaction twice for its validation process, because it is the only transaction in existence so tip selection will select it twice.
- Old tangles can therefore be safely stored as blobs of data. (maybe not completely if you want to also store unrefenced zero value data)

However, what is inside an old tangle stays valuable and data structures that don't use the DAG structure can exist cross snapshot. Like MAM channels, however a MAM root that lived in the snapshot will no-longer be accessible in the new tangle.
The permanode or archive should make all of the history of the tangle available.
The permanode should solve the **storage** from problem of IRI, NOT the **memory** problem.

Use-cases

1. When the live tangle doesn't have what you need request a perma-node.
2. Do analytics
3. Graph traversal
4. Use the perma node as a normal node.

Each of these use-cases have their own implications. (Please add any use-case but dumb them down to their base function)

Archiving vs Analytics.

When perma-nodes are discussed I read a lot about how people want to do graph traversals and analytics. These are fundamentally different problems compared to simple store and fetch. In my experience as distributed systems architect you build a system per purpose. Meaning archiving and analytics are two different systems. Yes to do analytics you need a proper source that you can then transform to the structure required for the analytics you want to do. The source/archive is a prerequisite for analytics.
To me we should focus on archiving first.

Fun NOTE: Fun thing about DAG and graphs, the real question for analytics is: which graph is actually interesting? Because even though the validation data structure is a DAG.... there is another one that is exactly the same as blockchain. That is how funds transfer between addresses. The tools used to analyse bitcoin could be used to analyse the tangle as well. Because the internal datastructure of blockchain is a linked list doesn't mean their functional structure (movement of funds) is not a graph, because it is.

Accessibility

Good API's are important but so is portability to already existing tools. To me that means we already have an API definition if we want to be compliant with the existing tools. However not all functions from the API need to be implemented mainly because most of them relate to the live tangle.
From the API docs: https://iota.readme.io/reference

- `getNodeInfo`: Yes -> could provide info that it is a permanode
- `getNeighbors`: Maybe -> if they form their own network
- `addNeighbors`: Maybe
- `removeNeighbors`: Maybe
- `findTransactions`: Yes -> this is an index search function.
- `getTrytes`: Yes -> this is that actual data fetch function
- `getInclusionStates`: No -> this is something live nodes can do
- `getBalances`: No -> irrelevant
- `getTransactionsToApprove`: No -> this is the reason for iri to be memory constrained. Besides that we don't want to be a live node as well.
- `attachToTangle`: No -> perma's deal with storage, they need their resources for that.
- `interuptAttachToTangle`: No

- `broadcastTransactions`: No -> if live just consume
- `storeTransactions`: No -> just consume

This leads to an initial very simple API. Namely `findTransactions` and `getTrytes`. If we have those with access to the entire history of the tangle I believe we are solving most of the use-cases already and fill conditions for many future improvements. And by taking over their specification the API becomes plug and play with already existing IOTA libs.

Discussion point: Should a perma-node/archive contain live data?

Obviously functionally this could be extremely handy, but we already have a system that contains the live and active Tangle. The reason I bring this is up is the following: "If perma-nodes exist and also contain all live data, why bother using live nodes at all?". This could potentially bring a shit-ton of traffic to permanodes while it could be perfectly dealt with by the live network.

When designing a system that can potentially store terabytes of data and contain millions of records that are exposed to the public... these are very important technical considerations. If we had a functional manager he would say: yes this is important! But we have to look if what we want can be partly achieved by functionality we already have in systems that already run and later can be combined in a client side library instead of a server side API. This to make a solid separation of concerns and design a system that is good in 1 thing instead of creating a system that is average in a few things and will eventually fail when it really needs to scale.

Don't get me wrong here, in the future I think an API could itself decide if it should query a live node or when to use the archive.

But the current problem is not retrieving live data, but historical data. This means that perma-nodes can be relatively dumb static data providers that only need to be updated once every snapshot.

Initial data distribution

An important part of setting up a new archive is initial data retrieval of all snapshots. For the sake of argument lets assume that there is a consensus based list of all snapshots in existence and the HASH to validate that they are correct.

What could be done is use a content addressed system like Torrents or IPFS to distribute snapshots initially and have all permanodes seed the snapshots so everyone can retrieve the raw data. (Think this is an important function of the archive as well, cold storage). Once a snapshot is received it can be loaded into a datastore for accessibility. Does this mean data duplication? Hell yes, source data distribution and making it available for an API to use are two different pieces of functionality. The purpose of an archive is to archive the data, that means all data. Better get some cheap spinning disks, or wait for DNA storage lol.

This will be one of the first things we will have to agree on. How do we distribute the initial data that can be loaded into the a datastore of choice. What format will it be in etc, is it format from the live data (rocksDB) or will part of the permanodes transform the original data into a more efficient datastore that is meant to load sequentially as a means to create insert statements and can be easily compressed?

IOTA transactions data structure

The data structure IOTA uses is extremely simple at its base.

HASH > references > Transaction data.

A KEY and a VALUE.

Everything else, addresses, bundles, tags etc are all inside a transaction.
However if we look at the `findTransactions` documentation we can see we can use 'Addresses', 'Bundles', 'Tags' and 'Approvees'. These eventually all converge to transaction hashes. This means findTransactions is an index search and not a read operation. IRI however keeps the entire Tangle in memory so there is no direct difference. For an Archive we will not be able to do that.

Discussion point: what will be the main entry point for fetches?

Because transactions are setup as key-values it doesn't necessarily mean it is the will be the most performant for the most common use-case. For pure fetches then yes a key-value structure will be fastest. But if 90% of all reads will be based on addresses we could consider duplicating some data or restructure our data is such a way that it will accommodate the most common use-case.
For my personal use-case and IOTA-Pay I will always query based on an address, so does MAM for example. Question is, is that really the most common use case? I actually think it is but assumptions are the base of all fuck-ups so...
To conform to the findTransactions API however we will be best off using a key-value store and index the different values. Nice part here is even that the fetch with `getTrytes` could be performed in parallel to other nodes as well. Doing client-side load balancing.

Data storage

Since the Tangle might grow to terabytes we will need to scale our storage component. There are plenty of good solutions for scaling if we just look at the CAP theorem. Because the most difficult component of scaling any data source is its data consistency. Since we archive and what we archive is inherently designed to handle inconsistencies we have plenty of choice. Because Consistency is not that important we can fully use the Availability and Partition Tolerant parts (for storage!).
Basically these systems will sacrifice something we might need for consistency:

- Any relational database system, MySQL, Postgress, Oracle, SqlServer etc.
- MongoDB

- Neo4j

Since relational databases always fall in the Consistency category we need to look at 3 main other database types:

- Wide column databases
- Key value stores
- Graph databases
- (Time series DBs are not relevant)

Wide column stores take a key and are adding information to this reference. They are just giant distributed map of maps. An example is for examples likes on a facebook post. The facebook post(key) appends likes (timestamp + userid).
The idea behind wide column databases is that you want all information from a column at once to do aggregation and analytics. De-normalization, data replication and adding metadata is encouraged. Where in a RMDBS you have a column IP address = 192.168.1.1, with wide columns you have a column named '192.168.1.1' with value 'key'.
They tend to scale very well because everything is deterministic and therefore from your db topology the client can calculate based on its query what server it actually needs to query. The first open source implementation was cassandra based on the BigTable paper from google.
(There are not that many use-cases you really need such a database, that is why there are so few actual implementations. DataStax (the for profit company making Cassandra) did a very good job selling it otherwise.)

- Cassandra
- Scylla
- Hbase

Arguments around wide column databases
It is true wide column databases scale very well but the data structure we have actually has very few columns and doesn't aggregate data. When the main query would be an address this might be a good option if we step off the idea of conforming to `findTransactions`. However we should consider that wide-columns do promote heavy data duplication. For each query type you duplicate the data. So if you want to query on address & tx you store it twice: if you want tags (thing this is always better to be a filter but that is a discussion for later) & Approvees you will store the data 4 times.

Duplication could also be done by just address + txs
Speed vs Space efficiency

Top choice for me: Cassandra, even though it really is a pain in the ass to maintain. It is not as easy and stable as they make you believe, unless you pay a 10.000 usd JVM per machine.

Key value stores on the other hand are quite often very simple data structures that can be build upon. The idea is very simple. You have a key, you go to you file and fetch it. A more popular version is Document databases that tend to add JSON compatibility and dynamic schema detection etc. But in the end they are key value stores as well.

Because the data consists of separate entries it becomes very easy to scale. Because the simplicity, KV stores tend to build other models on top of their KV stores, like graphDBs and time series databases.

However the difficult part for KV stores is search, where a wide-column database solves this by duplication key value stores tend to do this with in memory indexes or incorporate other separate indexing engines like Solr or ElasticSearch

- Redis
- MongoDB (Consistency DB, won't use)
- Dynamo
- Couchbase
- Riak

Arguments around key-value stores
The structure of the data is a key and a value so it would fit a Key Value store on first sight. However I think that indeed for storage it might be most efficient. However I am not so sure about efficient indexing. The question here is: how sparse is the data really? If an address index on average directs to 3 transactions it means we need shitload of memory because the amount of addresses is enormous. With external indexing engines like ElasticSearch you might be able to scale the index separately, but that means extra complexity.

My top choice: Redis for being most common, personally however I have good experiences with Riak (and I love Erlang).

Graph database are for our use cases a simple no-go. Even though it is a DAG you will never traverse the Graph unless you want to do analytics. Then as discussed before, what graph do you really want to query? The value graph or transaction graph?
There is no good reason I can think of to use a graphDB for an Archive.

Storage Engines.

Wanted to do a piece on Log Structured Merge trees vs B+ Tree storage engines like RocksDB vs LMDB or something like but maybe later and goes maybe a bit in too much detail for this iteration.

The REST api.

Actually I do not care to much about where this is made in. As long it is stateless so it is easy to scale and spin up multiple instances.

My conclusion

I think for the very first iteration we should not try to load live data into a database but focus on getting the old snapshots exposed as the `findTransactions` and `getTrytes` because that will expose functionality the current IRI does not have.
To create an API that just executes the queries to the database, can be extremely lightweight. And with an easily replacable database interface.

A key-value database with build-in indexing engine (have to do some more in-depth research for Redis, I know they do provide ES en Solr integration etc but would be easy if it can be managed through the same interface.)

ADDED: I did some more research into Redis and I would not pick it for this project because it focuses on the Consistency part of the CAP theorem. It is mostly to be used as an in-memory storage system. My top pick would be RiakKV as database, everything we need is available in the non-enterprise version. With a LevelDB backend opposed to a Bitcask(default) backend because the key-space a perma-node will have is truly enormous.

Get some torrents up so we can download all snapshots (maybe they are already there)? This will be the very simple and easy version of an archive after all.

PURE MD

Design considerations permanodes by Olaf van Wijk

This document will make a few assumptions based on my own personal needs for a permanode but will consider other use-cases as well and can evolve if other use-cases are added.

The permanode problem

IOTA as a distributed ledger uses the Tangle as a consensus structure allowing for no theoretical limit in the amount of transactions that it can process. Since each transaction validates two other transactions a structure of a direct acyclic graph (DAG) is created and this allows for parallel consensus giving the Tangle it's scaling capabilities. However as more transactions are processed the amount of storage **and memory** required increases dramatically in the current implementation.

IOTA therefore uses a mechanism called snapshots to periodically reduce the storage and memory requirements. To prevent misunderstandings there I will make some statements about snapshots:

- Snapshots are generated by taking milestone X and obtain the balances from all addresses referenced under that milestone.
- About the process: https://blog.iota.org/the-april-29-2018-iota-snapshot-and-iri-1-4-2-4-behind-the-scenes-7e034babcd44
- The current way of snapshots is an intermediate solution as stated in the article.
- Snapshots act as a new Genesis block. There is NO relation whatsoever anymore, AT ALL, to the previous tangle. The output of the previous tangle in the form of address balances is used to bootstrap the new tangle.
- For visualization: this means that the first milestone will reference the Genesis transaction twice for its validation process, because it is the only transaction in existence so tip selection will select it twice.
- Old tangles can therefore be safely stored as blobs of data. (maybe not completely if you want to also store unrefenced zero value data)

However, what is inside an old tangle stays valuable and data structures that don't use the DAG structure can exist cross snapshot. Like MAM channels, however a MAM root that lived in the snapshot will no-longer be accessible in the new tangle.

The permanode or archive should make all of the history of the tangle available.

The permanode should solve the **storage** from problem of IRI, NOT the **memory** problem.

Use-cases

1. When the live tangle doesn't have what you need request a perma-node.
2. Do analytics
3. Graph traversal
4. Use the perma node as a normal node.

Each of these use-cases have their own implications.
(Please add any use-case but dumb them down to their base function)

Archiving vs Analytics.

When perma-nodes are discussed I read a lot about how people want to do graph traversals and analytics. These are fundamentally different problems compared to simple store and fetch. In my experience as distributed systems architect you build a system per purpose. Meaning archiving and analytics are two different systems. Yes to do analytics you need a proper source that you can then transform to the structure required for the analytics you want to do. The source/archive is a prerequisite for analytics.

To me we should focus on archiving first.

Fun NOTE: Fun thing about DAG and graphs, the real question for analytics is: which graph is actually interesting? Because even though the validation data structure is a DAG.... there is another one that is exactly the same as blockchain. That is how funds transfer between addresses. The tools used to analyse bitcoin could be used to analyse the tangle as well. Because the internal datastructure of blockchain is a linked list doesn't mean their functional structure (movement of funds) is not a graph, because it is.

Accessibility

Good API's are important but so is portability to already existing tools. To me that means we already have an API definition if we want to be compliant with the existing tools. However not all functions from the API need to be implemented mainly because most of them relate to the live tangle.

From the API docs: https://iota.readme.io/reference

- `getNodeInfo`: Yes -> could provide info that it is a permanode
- `getNeighbors`: Maybe -> if they form their own network
- `addNeighbors`: Maybe
- `removeNeighbors`: Maybe

- `findTransactions`: Yes -> this is an index search function.
- `getTrytes`: Yes -> this is that actual data fetch function
- `getInclusionStates`: No -> this is something live nodes can do
- `getBalances`: No -> irrelevant
- `getTransactionsToApprove`: No -> this is the reason for iri to be memory constrainted. Besides that we don't want to be a live node as well.
- `attachToTangle`: No -> perma's deal with storage, they need their resources for that.
- `interuptAttachToTangle`: No
- `broadcastTransactions`: No -> if live just consume
- `storeTransactions`: No -> just consume

This leads to an initial very simple API. Namely `findTransactions` and `getTrytes`. If we have those with access to the entire history of the tangle I believe we are solving most of the use-cases already and fill conditions for many future improvements. And by taking over their specification the API becomes plug and play with already existing IOTA libs.

Discussion point: Should a perma-node/archive contain live data?

Obviously functionally this could be extremely handy, but we already have a system that contains the live and active Tangle. The reason I bring this is up is the following: "If perma-nodes exist and also contain all live data, why bother using live nodes at all?". This could potentially bring a shit-ton of traffic to permanodes while it could be perfectly dealt with by the live network.

When designing a system that can potentially store terabytes of data and contain millions of records that are exposed to the public... these are very important technical considerations. If we had a functional manager he would say: yes this is important! But we have to look if what we want can be partly achieved by functionality we already have in systems that already run and later can be combined in a client side library instead of a server side API. This to make a solid separation of concerns and design a system that is good in 1 thing instead of creating a system that is average in a few things and will eventually fail when it really needs to scale.

Don't get me wrong here, in the future I think an API could itself decide if it should query a live node or when to use the archive.

But the current problem is not retrieving live data, but historical data. This means that perma-nodes can be relatively dumb static data providers that only need to be updated once every snapshot.

Initial data distribution

An important part of setting up a new archive is initial data retrieval of all snapshots. For the

sake of argument lets assume that there is a consensus based list of all snapshots in existence and the HASH to validate that they are correct.

What could be done is use a content addressed system like Torrents or IPFS to distribute snapshots initially and have all permanodes seed the snapshots so everyone can retrieve the raw data. (Think this is an important function of the archive as well, cold storage). Once a snapshot is received it can be loaded into a datastore for accessibility. Does this mean data duplication? Hell yes, source data distribution and making it available for an API to use are two different pieces of functionality. The purpose of an archive is to archive the data, that means all data. Better get some cheap spinning disks, or wait for DNA storage lol.

This will be one of the first things we will have to agree on. How do we distribute the initial data that can be loaded into the a datastore of choice. What format will it be in etc, is it format from the live data (rocksDB) or will part of the permanodes transform the original data into a more efficient datastore that is meant to load sequentially as a means to create insert statements and can be easily compressed?

IOTA transactions data structure

The data structure IOTA uses is extremely simple at its base.

HASH > references > Transaction data.

A KEY and a VALUE.

Everything else, addresses, bundles, tags etc are all inside a transaction.

However if we look at the `findTransactions` documentation we can see we can use 'Addresses', 'Bundles', 'Tags' and 'Approvees'. These eventually all converge to transaction hashes. This means findTransactions is an index search and not a read operation. IRI however keeps the entire Tangle in memory so there is no direct difference. For an Archive we will not be able to do that.

Discussion point: what will be the main entry point for fetches?

Because transactions are setup as key-values it doesn't necessarily mean it is the will be the most performant for the most common use-case. For pure fetches then yes a key-value structure will be fastest. But if 90% of all reads will be based on addresses we could consider duplicating some data or restructure our data is such a way that it will accomodate the most common use-case.

For my personal use-case and IOTA-Pay I will always query based on an address, so does

MAM for example. Question is, is that really the most common use case? I actually think it is but assumptions are the base of all fuck-ups so...

To conform to the findTransactions API however we will be best off using a key-value store and index the different values. Nice part here is even that the fetch with `getTrytes` could be performed in parallel to other nodes as well. Doing client-side load balancing.

Data storage

Since the Tangle might grow to terabytes we will need to scale our storage component. There are plenty of good solutions for scaling if we just look at the CAP theorum. Because the most difficult component of scaling any data source is its data consistency. Since we archive and what we archive is inherently designed to handle inconsistencies we have plenty of choice. Because Consistency is not that important we can fully use the Availability and Partition Tolerant parts (for storage!).

Basically these systems will sacrifice something we might need for consistency:
- Any relational database system, MySQL, Postgress, Oracle, SqlServer etc.
- MongoDB
- Neo4j

Since relational databases always fall in the Consistency category we need to look at 3 main other database types:

- Wide column databases
- Key value stores
- Graph databases
- (Time series DBs are not relevant)

Wide column stores take a key and are adding information to this reference. They are just giant distributed map of maps. An example is for examples likes on a facebook post. The facebook post(key) appends likes (timestamp + userid).

The idea behind wide column databases is that you want all information from a column at once to do aggregation and analytics. De-normalization, data replication and adding meta data is encouraged. Where in a RMDBS you have a column IP address = 192.168.1.1, with wide columns you have a column named '192.168.1.1' with value 'key'.

They tend to scale very well because everything is deterministic and therefore from your db topology the client can calculate based on its query what server it actually needs to query. The first open source implementation was cassandra based on the BigTable paper from google.

(There are not that many use-cases you really need such a database, that is why there are so few actual implementations. DataStax (the for profit company making Cassandra) did a very good job selling it otherwise.)

- Cassandra
- Scylla
- Hbase

Arguments around wide column databases

It is true wide column data bases scale very well but the data structure we have actually has very few columns and doesn't aggregate data. When the main query would be an address this might be a good option if we step off the idea of conforming to `findTransactions`. However we should consider that wide-columns do promote heavy data duplication. For each query type you duplicate the data. So if you want to query on address & tx you store it twice: if you want tags (thing this is always better to be a filter but that is a discussion for later) & Approvees you will store the data 4 times.

Duplication could also be done by just address + txs

Speed vs Space efficiency

Top choice for me: Cassandra, even though it really is a pain in the ass to maintain. It is not as easy and stable as they make you believe, unless you pay a 10.000 usd JVM per machine.

key value stores on the other hand are quite often very simple data structures that can be build upon. The idea is very simple. You have a key, you go to you file and fetch it. A more popular version is Document databases that tend to add JSON compatibility and dynamic schema detection etc. But in the end they are key value stores as well.

Because the data consists of separate entries it becomes very easy to scale. Because the simplicity, KV stores tend to build other models on top of their KV stores, like graphDBs and time series databases.

However the difficult part for KV stores is search, where a wide-column database solves this by duplication key value stores tend to do this with in memory indexes or incorporate other separate indexing engines like Solr or ElasticSearch

- Redis
- MongoDB (Consistency DB, wont use)

- Dynamo
- Riak

Arguments around key-value stores

The structure of the data is a key and a value so it would fit a Key Value store on first sight. However I think that indeed for storage it might be most efficient. However I am not so sure about efficient indexing. The question here is: how sparse is the data really? If an address index on average directs to 3 transactions it means we need shitload of memory because the amount of addresses is enormous. With external indexing engines like ElastichSearch you might be able to scale the index separately, but that means extra complexity.

My top choice: Redis for being most common, personally however I have good experiences with Riak (and I love Erlang)

Graph database are for our use cases a simple no-go. Even though it is a DAG you will never traverse the Graph unless you want to do analytics. Then as discussed before, what graph do you really want to query? The value graph or transaction graph?

There is no good reason I can think of to use a graphDB for an Archive.

Storage Engines.

Wanted to do a piece on Log Structured Merge trees vs B+ Tree storage enginges like RocksDB vs LMDB or something like but maybe later and goes maybe a bit in too much detail for this iteration.

The REST api.

Actually I do not care to much about where this is made in. As long it is stateless so it is easy to scale and spin up multiple instances.

My conclusion

I think for the very first iteration we should not try to load live data into a database but focus on getting the old snapshots exposed as the `findTransactions` and `getTrytes` because that will expose functionality the current IRI does not have.

To create an API that just executes the queries to the database, can be extremely lightweight. And with an easily replacable database interface.

A key-value database with build-in indexing engine (have to do some more in-depth research for Redis, I know they do provide ES en Solr integration etc but would be easy if it can be managed through the same interface.)

Get some torrents up so we can download all snapshots (maybe they are already there)? This will be the very simple and easy version of an archive after all.