EVERSTORE

The Distributed Database Protocol

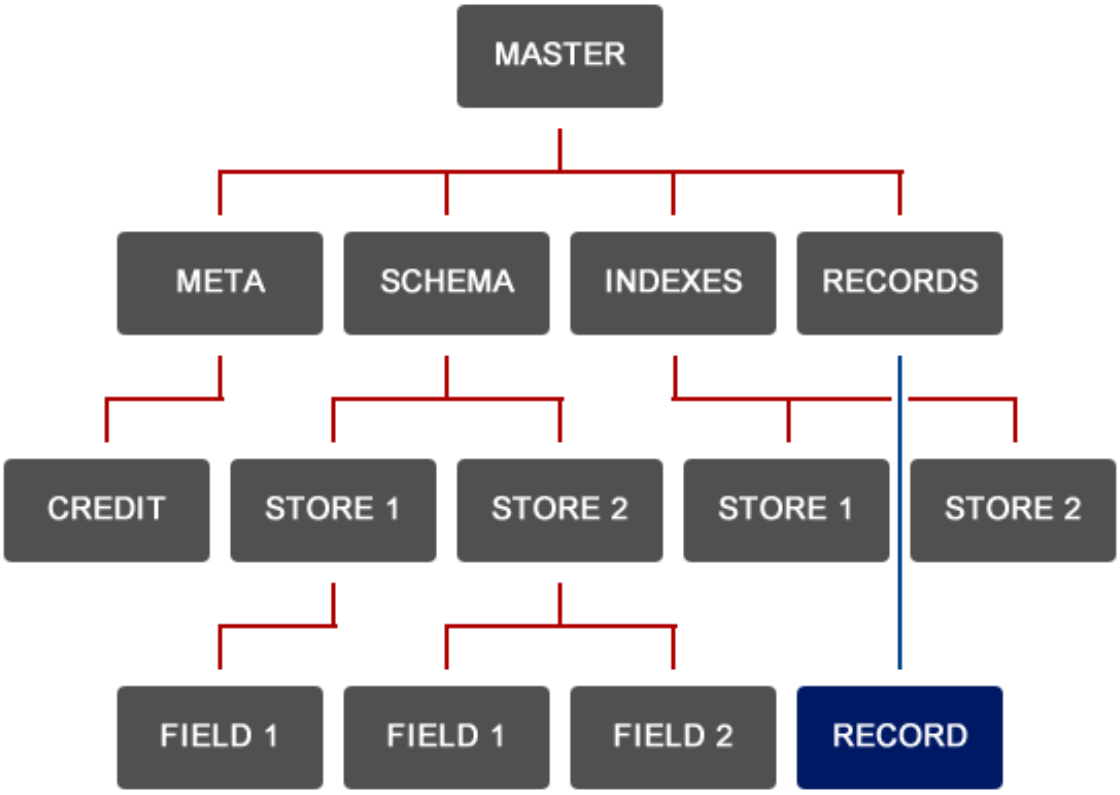
http://everstore.io

Last updated on August 16th, 2017

Table of Contents

Abstract

Since the advent of Bitcoin[1] the ecosystem has been lacking a solution to store arbitrary forms of data within the underlying blockchain itself. Most approaches are limited to the storing of anchored data-hashes such as those provided by Factom[2] and Tierion[3], with the actual data then stored off-chain. By employing a distributed database as a protocol, entities are able to read or write structured data directly into or out of a compatible blockchain - removing the need to store any of that information within their own siloed databases. Applications could operate freely without the need to rely on a centralised version of the truth; confident that the data they have received is from an immutable source. The ability to store complete data schemas and their corresponding data is achieved by structuring the limited storage space available from within the OP_RETURN[4] functionality of outgoing transactions across multiple nodes from within a hierarchical tree of related addresses. These transactions are sent from specifically derived HD (Hierarchical Deterministic) BIP32[5] addresses - where each branch of derived addresses represents a certain series of actions and (or) defining properties.



(Figure A - Abstract View of Everstore Node-Map)

Introduction

Bitcoin, blockchain technology, and other new forms of distributed ledgers have forced a major rethink of how organisations and applications can share data, track digital assets, and perform transactions across various networks. While the idea of blockchains is relatively new and still evolving, the numerous computer science methods that have been merged together in order to create them are not. From the hashes and various encryption algorithms that were created in the 1970s, through to the more recent peer-to-peer inclusions that were popularised in the 1990s by Napster; blockchains are built upon solid foundations that combine several well proven technologies.

The same can be said for Everstore; in that its development is merely a merger of numerous established blockchain standards that have been repurposed in order to expand the capability of the underlying blockchain. By introducing new features as protocols that can work across multiple blockchains as a layer of added functionality, rather than building this functionality directly into the base protocol and it's mining process - such as Ethereum[6] has done; we reaffirm permissionless innovation. We believe that the mining process should be as simple as possible and that with fewer moving parts, there is far less chances of running into problems. When it comes to maintaining an immutable record of data it is nice to have as few problems as possible.

The main purpose for developing Everstore was to allow distributed applications to be dynamically available from any static environment, without the need for ongoing hosting providers and their associated costs. In doing so, we have been able to add previously unavailable database-like characteristics and functionality to blockchains that do not support and need not support them natively. By using an established and secure blockchain as an actual storage engine, databases can efficiently and transparently do entirely new things, such as providing new "public" fields that allow for values to be voted on as well as knowing that all values are derived from an immutable event-store.

If you are able to imagine a distributed, freely available, always-on version of Meteor[7] that does not require registration, credit cards or any other form of direct payment, you are almost able to understand the potential of a distributed database as a protocol. Free of censorship and vendor lock-in, data can be shared between entities to create new opportunities within multiple industries. From governance to gaming, self-managed identities and interoperable medical-records, deeds, titles and ownership; it's hard to imagine a world without distributed data-management once you catch that first glimpse.

Current Requirements

Everstore is blockchain-agnostic, with the only requirements being the ability to add at least 38 bytes of arbitrary data within a distributed peer-to-peer network transaction and the ability to generate network addresses using the HD (BIP32) methodology. Everstore has already been successfully tested on both the Bitcoin and Dogecoin blockchains, as well as their corresponding testnets. Data can be encoded directly into multiple blockchains at the same time or switched from one to another at any future point, which prevents the risk of locking data into one particular source or vendor, while also enabling for new forms of autonomous distributed backups.

OP_RETURNs

Bitcoin and other similar cryptocurrencies use a FORTH-like scripting language for handling transactions. This list of stacked instructions, known as the script, describes to the network the conditions with which the next person wanting to spend any of the coins from the transaction are able to access them. In the majority of transactions, these instructions are simple and allow anyone with the private key to access the coins. However, there are over 50 operational codes that can be used within the script to cater for a wide range of use cases. One of these codes is known as an OP_RETURN, which is a standardized way to add extra data to a zero-value output that is attached to a valid transaction. An OP_RETURN on the Bitcoin blockchain can include up to 38 bytes of arbitrary data, whereas the Dogecoin blockchain allows for up to 78 bytes.

HD (BIP32) Key-Trees

The Bitcoin reference client uses random seeds in order to generate new cryptographic private keys for each address and then stores each of those keys locally. However, in 2012, BIP32 introduced a new optional standardized method for generating key-tree chains in a predictable way - all of which could be derived from a single master key. Hierarchical Deterministic Wallets are capable of creating vast numbers of cryptographic keys, and because they can be recreated in different environments by using the same base seed and mathematical inputs, there is no need to save each and every key. Given the master public key, other entities are also able to follow the HD (BIP32) specifications and independently calculate descending public keys without needing to know the secret seed originally used to generate the master keys.

Defining an Instance

In order to broadcast to the blockchain that a particular master key is being used as an Everstore instance, we must first relay the desired name of the instance to the network. This can be done by following the relevant route from the node-map below to find the correct address from which to commit a valid meta-entry.

Routes

Routes are the mathematical paths that are used by the HD (BIP32) algorithm to reach individual nodes from within the key-tree and always assumes that the starting point is the master key. In order to reach the meta node we would use a route of 1, whereas reaching the thirty-eighth field from within the six thousand and thirty-second datastore schema would use a route of 2 | 6032 | 38.

Node Map

The node-map defined by the Everstore protocol is a way to map the different branches of the master key-tree to their corresponding uses. If each Everstore instance uses the same node-map, other independent applications can retrieve data from the blockchains whilst only needing to know the master public key. For added obfustication, applications could elect to define their own unique node-maps, which would prevent anyone with any of the keys to be able to recreate the data without also knowing the correct node-map. The current default Everstore node-map can be introduced as follows:

Use	**Route**
Meta Node	1
Credit Node	1 | 1
Schema-Tree	2
Index-Tree	3
Record-Tree	4
Accumulation-Tree	5
Public-Tree	6

Meta Node

The meta node is the point from where transactions containing information related to the instance should be sent from - such as the name of the instance. Data contained within the meta node transactions should be added to the blockchain with an "**es**" checksum and then piped with the name and value of the attributes added.

In the case of first defining the Everstore instance with the name "**Testing Everstore**" the data added to the OP_RETURN would be as follows (in-turn utilizing 25 bytes):

Route 1 OP_RETURN* = **es|name|Testing Everstore**

**Please note that future piping options may have spaces, but only so they are easier to view.*

FYI: | This | is | an | example | of | piping

Credit Node

The credit node is the point from where the funds for each transaction are taken, sent and returned (when change is required). In future versions, different descendants of the credit node could be used for different purposes and (or) different users.

Schema-Tree

The schema-tree is the point from where an Everstore instance is able to define the properties of the required datastores and their corresponding fields. The first level of descendants is used to define the datastores themselves, whereas each of their descendants is then used to define the fields for that particular datastore.

Index-Tree

The index-tree is the point from where the primary data-key (field 0) for new records is first registered in order to provide a unique transaction ID that can then be converted into a unique tree-route for that particular record, which is known as a transaction-route.

Record-Tree

The record-tree is the point from where transaction-routes are traversed in order to retrieve the individual record-node - with each of its descendants representing the fields belonging to that record as defined by the schema.

Accumulation-Tree

The accumulation-tree is the point from where the accumulated field-types (total, count and media) are stored. In each of these cases, the initial value (V) of the relevant field (F) from the associated datastore (D) act as integer based pointers to reach the true value of the field, which is calculated by first reaching the relevant accumulation-node:

Use	**Route**
Relevant Total Node	5 | 1 | D | F | V
Relevant Count Node	5 | 2 | D | F | V
Relevant Media Node	5 | 3 | D | F | V

This allows the initial value to act as a version number to manually redirect references to the relevant data when wishing to pseudo-update or pseudo-delete accumulated data. More detailed instructions on reading and writing values to accumulated field type nodes can be seen within the Everstore public repository's technical documentation.

Public-Tree

The public-tree is the point from where the public field type is stored, with each of its options (unique, count, total and all) each receiving its own sub-tree - where the initial value (V) of the relevant field (F) from the associated datastore (D) act as integer based pointers to reach the true value of the field, which is calculated by first reaching the relevant public-node using the advanced node-map below:

Use	**Route**
Relevant Public-Unique Node	6 | 1 | D | F | V
Relevant Public-Count Node	6 | 2 | D | F | V
Relevant Public-Total Node	6 | 3 | D | F | V
Relevant Public-All Node	6 | 4 | D | F | V

More detailed instructions on reading and writing values to public field type nodes can be seen within the Everstore public repository's technical documentation.

Events

There are two phases to each Everstore data-transaction. First we must prepare the relevant nodes so that they have the necessary funds. Once this has been done, we can then commit the data piece by piece from each of the prepared nodes.

Preparing Nodes

Since preparing the nodes does not require an OP_RETURN, we are able to prepare multiple nodes within a single blockchain transaction. We typically prepare individual nodes by sending them twice the minimum network mining fee, which is enough to then send back the minimum network mining fee whilst also paying the network the minimum mining fee required for relaying that transaction to the network. It is also worth clarifying that payment of the minimum network mining fee is also required in order to prepare the nodes, which is why data-transactions are batched and processed within two phases.

Committing Data

Once all of the nodes for a particular data-transaction-batch have been prepared, the individual commits can begin to be processed. Each commit should contain only a single OP_RETURN, but it can be placed in one of three ways:

1. **Primary OP_Return Placement** (the first output in the transaction)
2. **Secondary OP_Return Placement** (by default it is the second and final output)
3. **Split OP_Return Placement** (by having it as the second output but also having a third valid output, which is the circumstance under which a node must be prepared with three times the minimum network mining fee rather than just twice)

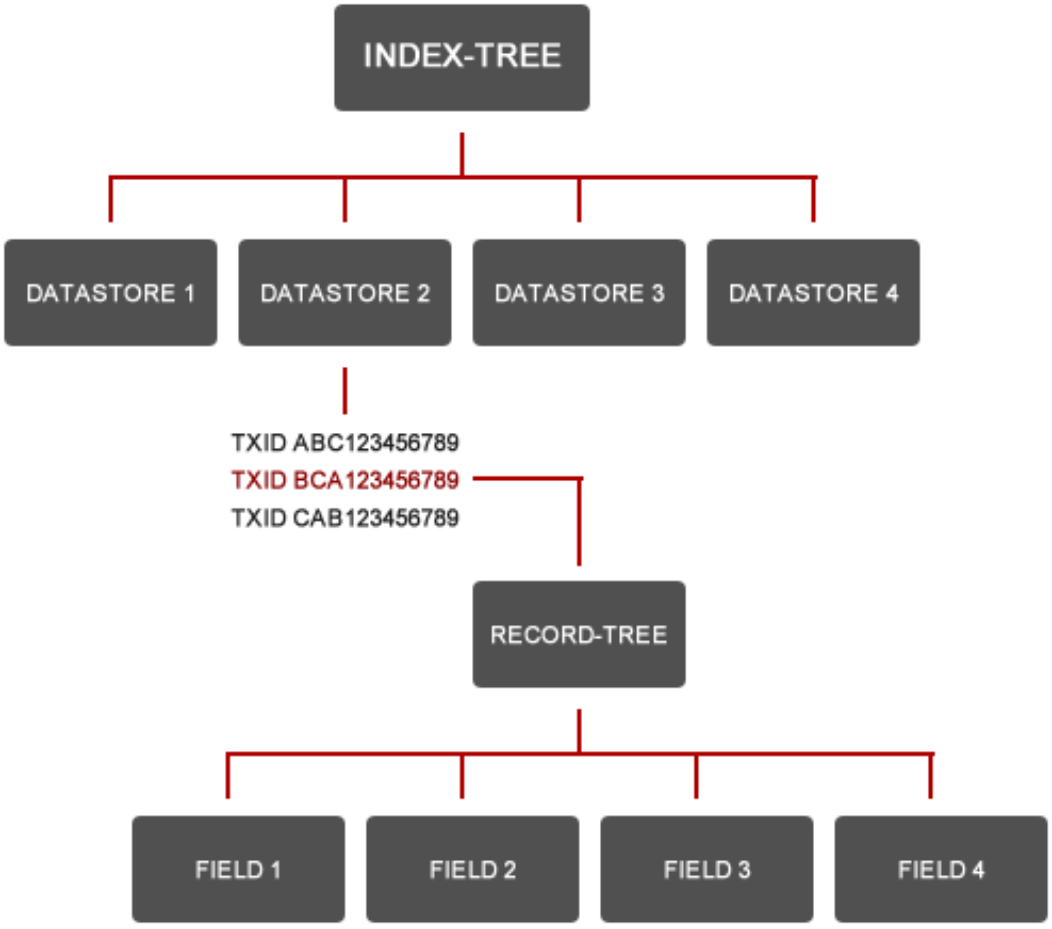
The data stored within the OP_RETURN can have one of three levels of transparency:

1. **Public** (default setting - anyone viewing the transaction is able to see the data)
2. **Secure** (anyone with access to the private key to that node or its hash can view)
3. **Private** (only those with access to the master private key or its hash can view)

Non-public encoding uses AES encryption with a hash of the relevant key as its secret.

Reading Data

When reading data from a node - the latest transaction typically represents the current value or requested property, with previous transactions representing an audit trail of changes. An exception to this method is when viewing transactions that are derived from the index-tree, where each outgoing transaction from its descendant datastore nodes represent individual records, as this allows for greater numbers of indexed fields to be retrieved as quickly as possible and from as few places as possible.



(Figure B - Transaction-Route for Finding Records)

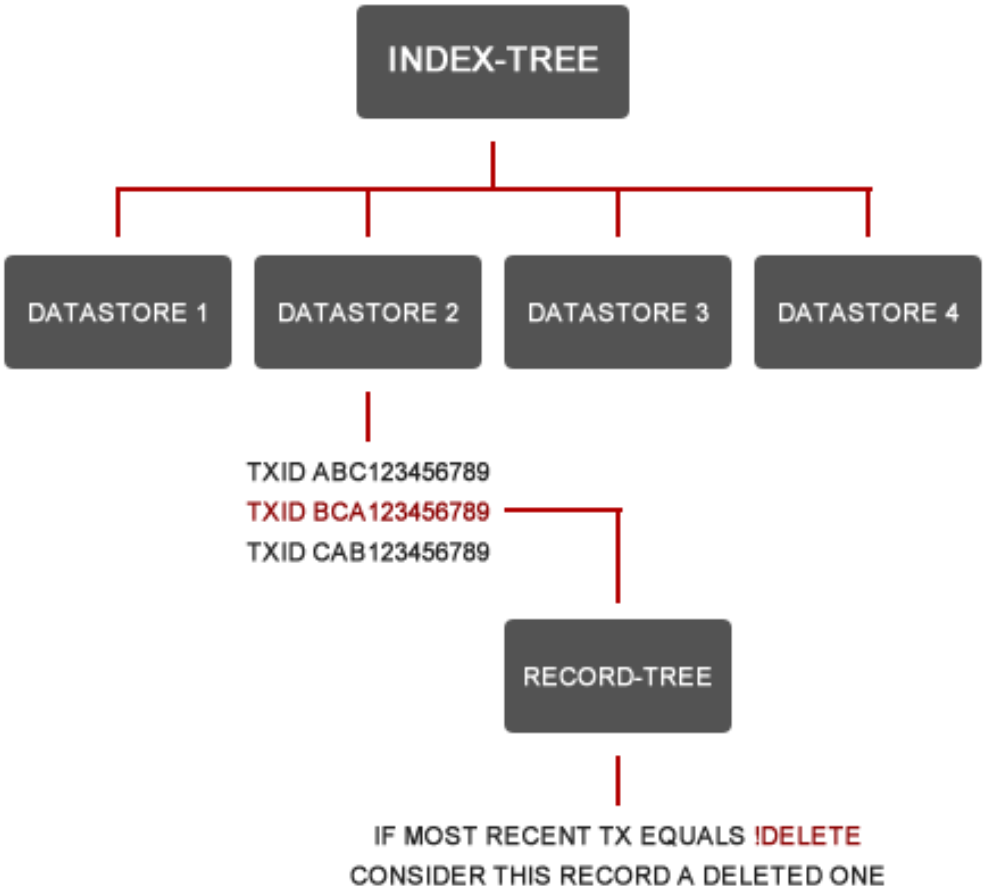
Updating Data

Since data cannot be removed from a blockchain, updating data in Everstore is known as a pseudo-operation, which typically involves sending a new transaction from the relevant node with the new data attached.

Deleting Data

Since data cannot be removed from a blockchain, deleting data in Everstore is known as a pseudo-operation, which typically involves sending a new transaction from the relevant node with a specific hashbang command (**!delete**) attached. This also allows data to be undeleted by another pseudo-update containing non-hashbanged content.

An exception to registering deleted information is in regards to records, where each outgoing transaction represents an individual record as opposed to the latest value, in which case we must first check that the corresponding record-node does not have any of its own outgoing transactions. From this point, each new outgoing transaction should represent the latest value of the associated indexed field.



(Figure C - Abstract View of Pseudo-Deleting an Everstore Record)

Datastores

A datastore is the Everstore equivalent of a database table (also known as collections). Each Everstore instance can hold up to 999,999,9999 datastores, each of which can have a distinct set of up to 999,999,999 fields and an unlimited number of records, all of which can have every action taken upon them linked to a single master key.

A datastore can be created with the following properties:

1. Name / Label
2. Transparency (public, secure or private - as mentioned previously)
3. Fields

If we wish to establish our third datastore as one that is not public, we must pipe the transparency option into the value used to store the name, such as:

OP_RETURN from Node at Route 2 | 3 = **My Evestore Name|private**

Electing to define a secure or private datastore should instruct applications to force all other information pertaining to that datastore to be encrypted by the relevant hash.

Fields

Each field can be defined with the following properties:

1. Name / Label
2. Type (defaults to string)
3. Default (defaults to null)
4. Priority (defaults to optional)
5. Transparency (defaults to public)

Available priorities for fields include:

1. **Optional** (default behaviour also obtained by not defining the priority)
2. **Required** (meaning a record should not be committed if the field has no value)
3. **Unique** (meaning a record should not be committed if another record in the same datastore has the same value as the value of the field being saved)

Everstore currently supports the following 20 field-types, which have been specifically introduced for running dynamic serverless and distributed web-based applications.

Type	**Piped Options**	**Default Option**	**Fixed Choices**
Index	type	auto	auto, md5, string
Auto	type	time	time, MD5, increment
String*			
Integer*	type	standard	standard, small, big
Float*	decimals		
Boolean*			
Datetime*	format	dd/mm/yy	all valid formats
Email	format|text	string	format = string, HTML
URL	format|before|after|text	http://	format = string, HTML
Array*			
Object			
Enumeration			
MD5*			
JSON			
Geolocation			
Public	type|format	all	all, total, count | string, float, int
Total	type	int	int, float, string
Count	type	public	int, float, string
Media	header	image/jpeg	all valid headers
Lookup	datastore|field		

More detailed information regarding the field-types and how different client-applications should interpret the information is available within the relevant technical documentation.

However, since the current node-map only uses 6 out of its 999,999,999 available first-level slots, Everstore has much room for future growth and feature-set expansion.

Field properties are dispersed by the OP_RETURN placement and piped as follows:

1. Secondary Placement = **name**
2. Primary Placement = **type** | **priority** | **transparency** | **options**
3. Split Placement = **default**

For example; in order to register an integer field called “The Answer to Life” with a default value of 42 as a required field that should be encoded securely, the relevant node would need to commit three individual transactions as follows:

- **1st TX** = **The Answer to Life** (with a secondary OP_RETURN placement)
- **2nd TX** = **integer|required|secure** (with a primary OP_RETURN placement)
- **3rd TX** = **42** (with a split OP_RETURN placement)

This approach allows for all field information to be available from the transactions associated with a single node rather than needing to gather transactions from five different nodes. Defining this new field within the embedded schema in a single batch would have an associated cost of four times the minimum network mining fee. One for initially preparing the node and three for the individual transactions to define the field.

Advanced Field Types

Those fields listed above with asterisks are common data-types[8] as defined by the majority of databases, whereas those listed below may require further clarification or specific explanations as to how and why Everstore uses them:

Index Fields (Field 0)

In the current version, only one field within any record can be indexed and it is currently fixed to field 0, which by default is an auto-generated hash based upon the time the field value was generated. This could be changed to an MD5 type and then an email address could for example get hashed to become the new ID, which would then be used by the application to more easily allow individuals to look-up their own specific records. It is important that each of the values for this field remains unique to the datastore it is in, which is a property that must be enforced by the application.

Auto Fields

Auto fields can create automatic values based on either the time, an MD5 of the time or and incremented value based upon the number of outgoing transactions that are already indexed within the relevant datastore node of the index-tree.

Email Fields

A simple string-based field that should be verified as an email address and can be rendered by the application as either a string or an actual HTML link with text that is different to the address by using piped options.

My Email = **me@email.com|My Email**

This HTML text can also be initiated as a default to all records when set from the field options as follows: **email ||| html | My Email**

URL Fields

A simple string-based field with similar rendering options to the email field, which has been designed to help save potential bytes by being able to set specific start or end points to the URL when defining the field options as follows:

Example Field Options	**Stored Value**	**Output**
http://neuroware.io/img/ | .png	mark-smalley	http://neuroware.io/img/mark-smalley.png
http:// | /robots.txt | Robots	any-website.com	Robots

Object Fields

Objects are similar to arrays but include defined properties and are often used to group similar items, such as social media profile usernames in the example below:

Field Setup	**Stored Value**	**Output**

TX 1 = Social Profiles
 	
TX 2 =
object ||| fb | twitter | github
	mark.smalley | m_smalley | msmalley	var social_profiles = {
		fb: “mark.smalley”,
		twitter: “m_smalley”,
		github: “msmalley”
		}

Enumeration Fields

Enumeration fields allow fields to have a pre-defined set of values that are accepted and are often expressed as select boxes when rendered by an HTML application. They can be defined at a schema level in the same way objects are defined, with the only difference being that only one of the properties can be selected, rather than requiring values for each of the properties.

Geolocation Fields

A simple string-based field that allows applications to convert a string of comma separated numbers into a relevant GeoJSON object - where one point equals a coordinate, multiple points not connected are converted into lines and multiple points with a connection are represented as polygons.

Public Fields

Public fields are a unique concept specific to Everstore, where instead of taking the values of outgoing transactions, which is done to ensure that the properly authorised entity is updating or creating data - we instead allow the values of incoming transactions to define our values, in-turn allowing others to essentially vote for the value. There are three types of public fields, each with three formats that allow for multiple use-cases:

Type	**Format**	**Example Use**
All	String	Comments (default setting)
Total	String	Long String (beyond byte limit)
Count	String	Reactions
All	Float	Donations
Total	Float	Goal
Count	Float	Popularity
All	Integer	Stats
Total	Integer	Preference
Count	Integer	Votes

JSON Fields

A simple string-based field that allows applications to convert valid JSON into objects.

Total Fields

The total-field is similar to the public total-field but can instead only be updated using the traditional outgoing transaction method used by non-public fields. When used in combination with a string setting, it becomes possible to bypass the 38 to 78 byte limit.

Count Fields

The count-field is similar to the public count-field but can instead only be updated using the traditional outgoing transaction method used by non-public fields.

Media Fields

Media-fields are specifically designed for holding low-grade basic images and (or) icons, where the base64 source for the file is split between multiple transactions.

Lookup Fields

Lookup fields are often presented as select boxes that show values derived from another field and (or) datastore and are first defined by storing a references to the required datastore and the field with which to generate the list of values to choose from.

Records

It is the Everstore node-map that instructs the system as to the descendant address that should be used to represent each of the individual database elements. With this defined, we can start sending-out transactions from the relevant addresses in order to give us those opportunities to add our arbitrary data into the available OP_Returns.

However, an Everstore record itself once re-composed by the client is actually just a virtual construct that is assembled from a series of individual transactions that are mapped to the application schema through assigned routes to sub-trees that hold the entire record in its descending nodes. As an example, where we wish to discover the embedded application schema and get the most recent record from the final datastore of the instance (often the manually constructed relationship for that particular application's use-case), we would perform several sequences of events.

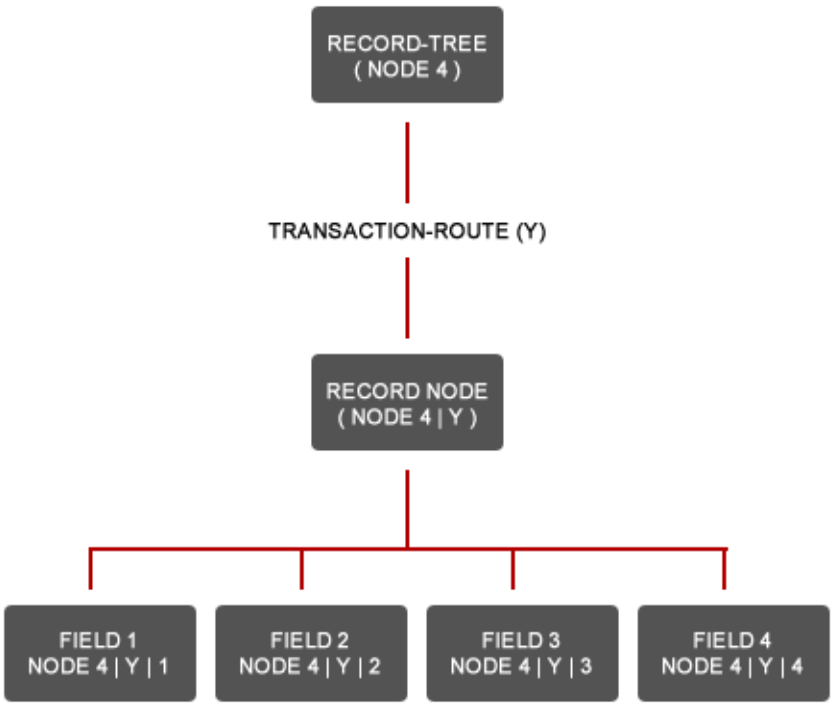
To first establish the start of a schema, we would need to search for datastores:

1. Check if the first datastore schema exists (transactions from node 2 | 1)
2. Check if more than one datastore exists (transactions from node 2 | 2 onwards)
3. Repeat until finding no more datastores

We would then need to discover the fields for each of the datastores:

1. Check if datastore has a field (transactions from node 2 | 1 | 1)
2. Check if datastore has more fields (transactions from node 2 | 1 | 2 onwards)
3. Repeat until finding no more fields for each datastore

Using the information gathered from all of the nodes within the schema-tree, we are then able to construct a working schema object within the application, which we now want to use in order to query the last datastore (X) to discover the most recent entry by switching from the schema-tree to the index-tree and checking the most recent outgoing transaction associated with the node at 3 | X. We would not only discover the value of the indexed field associated with that record (field 0), but it would also provide the necessary transaction ID needed to generate the transaction-route (Y). Using the transaction-route, we can switch to the record-tree and traverse its fields as follows:



(Figure D - Transaction-Route for Finding Fields)

Relationships

The current version of Everstore requires the manual creation of relationships as individual datastores that are able to link various datastores together using lookup fields. If we were to take a simple example use-case where we want to register people that are participating in specific events at different times, we might choose to use the following schema to manage our data:

1. People Datastore
    - Email (md5 index type)
    - Name (string)
2. Event Datastore
    - Type of Event (string index type)
3. Events Datastore
    - ID (auto index type)
    - Time (datetime)
    - Person (lookup linked to name from people datastore)
    - Event (lookup linked to type of event from event datastore)

The third and final datastore in this example (Events) acts as our relationship, linking the name of people and types of events that they have undergone together into one datastore, which can then be made available by the application as a series of objects.

This is a similar yet simplified explanation of the example that is used and included within the public repository under Example Everstore Event Schema, which was used to power our example applications.

In future versions this same concept could be applied to individual instances by abstracting the schema with one additional layer for the instances prior to then defining the datastores for each instance. This would effectively allow a single master key to contain up to 999,999,999 databases, allowing for any of those database to lookup any information it had permission to from any of the other databases.

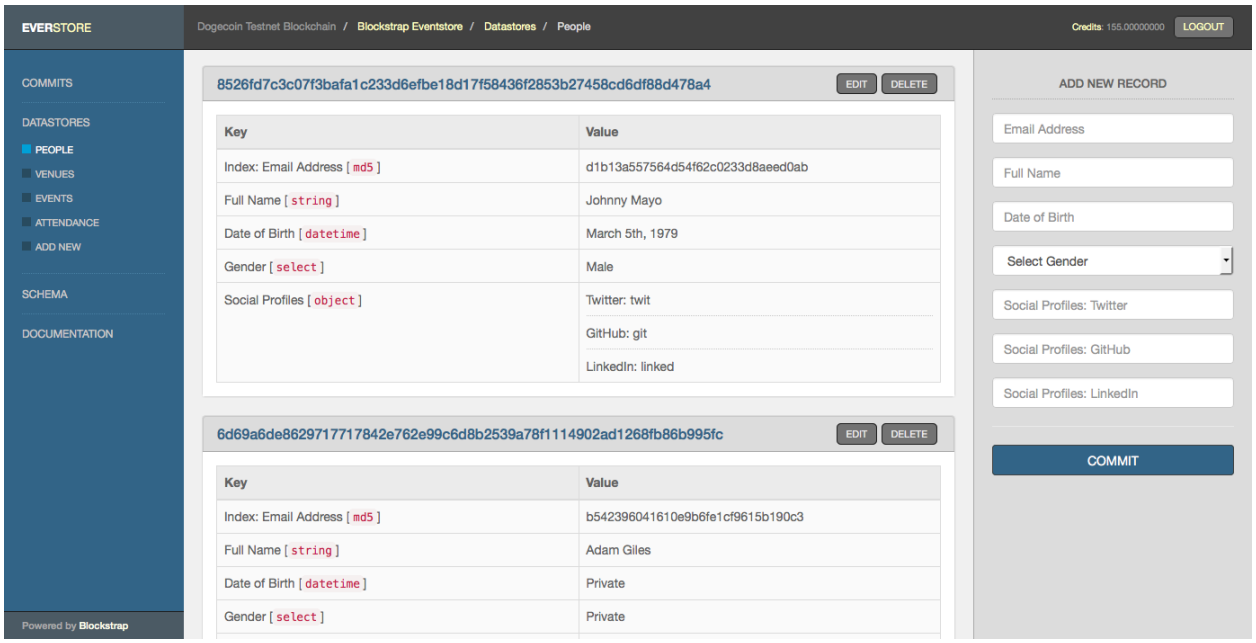
Example Applications

The first example application that we built for Everstore was a browser-based serverless interface for initiating and managing any Everstore instance, which can be used from any device with an internet connection that supports HTML5 localStorage (for caching).

Powered by the open-source Blockstrap framework, this interface can support up to eight different blockchains whilst also allowing the communication with the blockchains to be conducted via a limitless choice of web-accessible APIs.

Serverless Interfaces

More information regarding how these serverless interfaces are constructed is available from within the project's repository. The repository is currently private and available to select partners, but will be made available to all and open-sourced soon.



(Figure E - Screenshot of People Datastore from Everstore Interface)

Embedded Schemas

One of the features of the serverless interface is the ability to export the embedded schema of an instance as a Javascript variable that could be pasted directly into an external application as a way of skipping the need to discover the schema.

Example Everstore Event Schema

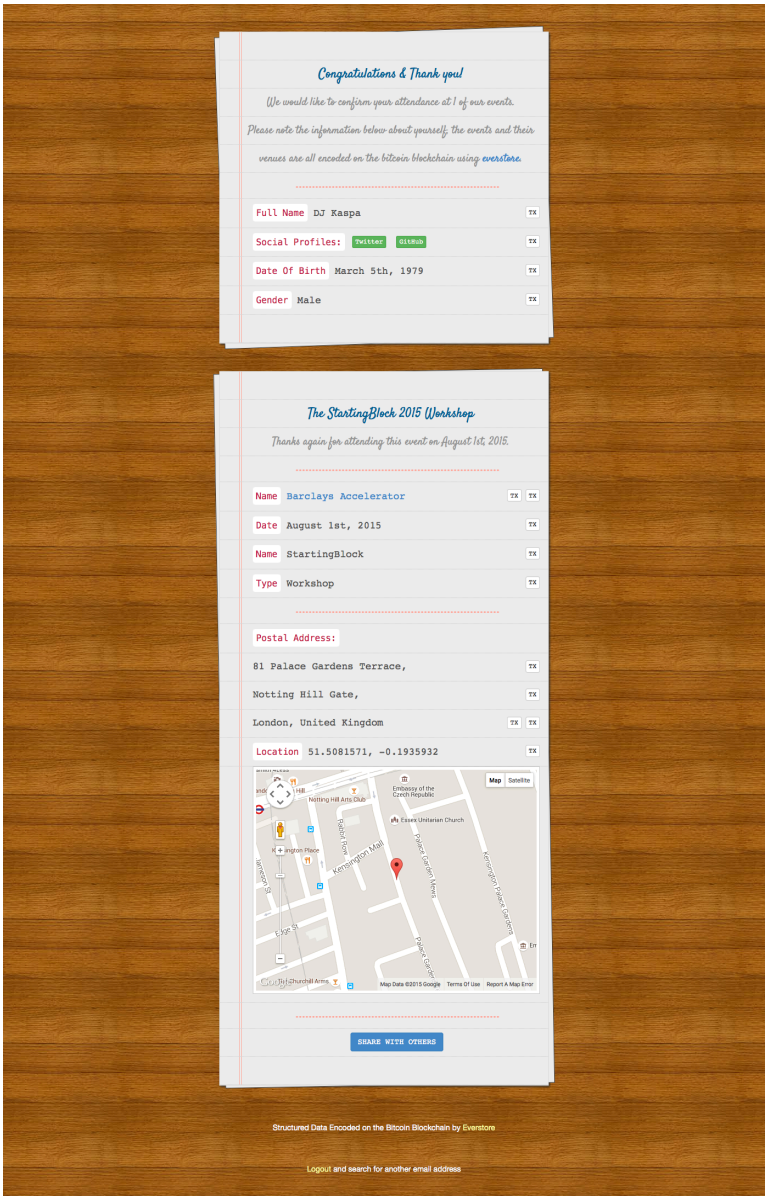
By re-creating the schema below into your own Everstore instance, you will then be able to reuse the supplied Attendance theme for your own records.

1. People Datastore
    - Field 0 = Email Address (md5 index type)
    - Field 1 = Full Name (string)
    - Field 2 = Date of Birth (datetime)
    - Field 3 = Gender (enumeration)
    - Field 4 = Social Profiles (object)
2. Venues Datastore
    - Field 0 = Name (string index type)
    - Field 1 = Website (url)
    - Field 2 = Country (string)
    - Field 3 = City (string)
    - Field 4 = Address (total with string type)
    - Field 5 = Location (geolocation)
3. Events Datastore
    - Field 0 = Name (string index type)
    - Field 1 = Type (enumeration)
4. Attendance Datastore
    - Field 0 = ID (auto index type)
    - Field 1 = Date (datetime)
    - Field 2 = Person (lookup linked to full name from people datastore)
    - Field 3 = Venue (lookup linked to name from venue datastore)
    - Field 4 = Event (lookup linked to name from events datastore)

You could expand upon any of this information as required, but in order to use the included sample integration application (the Attendance Theme), you must set the minimum field and datastore values as listed above.

External Integration

The second application that was developed for Everstore was the Attendance theme, which when used with the “Example Everstore Event Schema” would allow attendees to independently verify their attendance of specific events. It acts as a showcase for how a standalone third-party web-based application could access Everstore information without needing to run its own instance of Everstore or any server-side components.



(Figure F - View of Verified Attendance)

Limitations and Caveats

Although there's no benchmark yet, we can confidently say that Everstore is likely to be the world's slowest database. On the plus side, it's also the world's only database that can be proven immutable and is secured by thousands of globally distributed redundant nodes that are constantly running and at no additional cost to the owner of the data. By introducing caching layers at the application level the speeds can be vastly improved. The example applications use localStorage to store data in the browser on the client, which improves load times on returning visits after having stored all the data needed locally. Traditional NoSQL databases could also be used as a smart-caching layer, but in all cases; the trade-off between immutability and speed is prevalent.

It is worth noting that the underlying networks have their own limitations and problems. In the case of Bitcoin, it's network can only handle 7 transactions per second, which means that with its philosophy, the more money spent on transaction fees, the higher the chance of it being accepted by the network as a valid transaction and included within the next block. If we were to set a value of US$3,800 per Bitcoin, we can determine that it would cost up to US$10,000 to store each 1 MB of data if we were to pay BTC 0.0001 per transaction and were limited to 38 bytes within a single transaction:

Byte Limit	38		TXs for 1 MB	26,315.79
Bytes in 1 MB	1000000		Cost per TX (US$)	0.38
US$ per 1 BTC	3800		Cost per MB	10000

This is expensive compared to Amazon[9], which costs US$0.03 per annum to store 1MB, versus US$10,000 using Everstore and the Bitcoin blockchain (at US$3,800 per Bitcoin). However with Everstore, that cost is a one-time fee and the data is available for as long as the network that it resides on is functioning.

It is also worth noting that the Bitcoin ecosystem currently has a long-standing unresolved issue known as the Block Size Debate[10] that questions the 1MB block-size limits in place since first being established eight years ago. Any project that is utilizing OP_Returns for arbitrary data-storage also runs the risk of causing what could be perceived as bloat upon the network. There are mass economic implications, not just for adopting or refusing to adopt new block-sizes, or for those miners that are adopting the alternative node implementations, but more importantly; everyone is being forced to decide upon a definitive use-case for this distributed storage engine that is the network.

Blockchain's CAP Theorem

If you need the ultimate partition tolerance and a higher availability than AWS for the data that you are storing then something such as Everstore and the use of immutable storage-engines might be for you. If you're looking for speed, consistency, and general efficiency; you've definitely come to the wrong place. Different databases are good at different things. Some are good at mass amounts of unstructured data, where others are more suitable for reports or some for graphs. It's fair to say that storing data in an immutable open network is not the right place for tracking the number of likes for photos we share of our pets wearing sunglasses, but when we are talking about tracking births, deaths and marriages, or where the country's budget got spent - why would we not be insisting upon it being in an immutable data-storage engine available for all to see?

Economic Implications & Conclusions

The standardized method for inserting arbitrary data upon the majority of blockchains is through the use of OP_Returns, which are currently limited to a single OP_Return per transaction. This is somewhat strange, as a transaction without OP_Returns can have as many inputs and outputs as you like, so long as you pay for the KB space used by the script to relay that transaction to the network. It is the fact that a transaction cannot include multiple OP_Returns that increases the blockchain bloat, not the data itself.

We propose that data should be neutral and that we should allow the network to agree that there can and should be more than one type of block doing more than one thing. The trouble with distributed protocols is that different people with different views want to to do different things, and right now, everyone thinks it can only be doing just one thing. Taking a step back, we should be able to see the accomplishment of developing a freely available, open and immutable data-storage engine. If we can agree that is what it really is and then let the politics play-out at an auxiliary protocol level, where heated opinions can remain entirely focused on specific use-cases; the world would be a better place.

In summary, the significance of being able to freely store structured immutable data within the world's most powerful open networks has implications that reach far beyond the mere size or frequency of blocks. Questions have many perspectives and although the blockchain ecosystem is evolving at a rapidly increasing pace as more money and talent is invested, much of that time and resource is unfortunately being spent creating new distributed silos or developing centralized solutions to distributed problems that only work within a single closed network.

The importance of blockchain-agnosticism cannot be stressed enough.

Until there are a selection of networks as computationally secure as the current Bitcoin blockchain that are also allowing arbitrary data encoding, or someone is able to use our starting point of combining HD (BIP32) hierarchy with OP_Returns we have little choice. For now, to ensure that we are able to immutably store whatever data we choose on the blockchain of our own choice; we are instead forced to use something as complicated to understand and process as Everstore. For more information on this, please read about our BitDB Project - http://neuroware.io/blog/introducing-bitdb-the-complimentary-blockchain-agnositic-node/

References

The following links are provided as further information to their relevant references:

[1] Satoshi Nakamoto, *Bitcoin: "A Peer-to-Peer Electronic Cash System",* 2008
https://bitcoin.org/bitcoin.pdf

[2] Paul Snow, Brian Deery, Jack Lu, David Johnston, Peter Kirby, *"Factom: Business Processes Secured by Immutable Audit Trails on the Blockchain"*, 2014
https://github.com/FactomProject/FactomDocs/raw/master/Factom_Whitepaper.pdf

[3] Wayne Vaughan, Shawn Wilkinson, Jason Bukowski, Chainpoint (by Tierion), "*A scalable protocol for recording data in the blockchain and generating receipts*"
https://tierion.com/chainpoint

[4] OP_RETURNs (https://en.bitcoin.it/wiki/Script)

[5] Peter Wuille, *"BIP0032 - Hierarchical Deterministic Wallets",* 2012
https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki

[6] Ethereum, *"Next-Generation Smart Contract and Decentralized Application Platform*":
https://github.com/ethereum/wiki/wiki/White-Paper

[7] Meteor, *"Full-Stack Javascript Application Platform"* - https://www.meteor.com/

[8] Common Data Types (https://en.wikipedia.org/wiki/Data_type)

[9] Amazon EC2 Pricing - https://aws.amazon.com/ec2/pricing/

[10] Block Size Debate, Many Views:
https://google.com/search?q=block+sizedebate&oe=utf-8#q=block+size+debate