OASIS DECENTRALIZED DATABASE

Database Management System

Technical description

Anton Filatov[1], Dmitry Kochin[2]

August, 2017 - March, 2019

[1] https://www.linkedin.com/in/antonfilatov
[2] https://www.linkedin.com/in/kochin

Version 0.1.1-DRAFT

Table of contents

Introduction

Oasis.DB is an open-source, public, blockchain-based, distributed non relational database management system.

Oasis.DDB is a fully decentralized database. This means that it has neither a single point of failure nor a single point of control. That is all nodes of Oasis.DDB are equal, and the introduction to the network of a new Oasis.DDB node is available to everyone. As a result, Oasis.DDB is a cloud-based database running on servers provided by community members. Participants at the same time are heterogeneous and do not belong to a single organization, which is why there is no single point of control in Oasis.DDB.

Full decentralization imposes on Oasis.DDB requirements for resistance to malicious behavior of its individual participants. That is, the participant while providing the node to the Oasis.DDB network, should not be able to disrupt its operation, even if it modifies the node software for harm.

Below are the principles of operation of the Oasis.DDB nodes. The main attention is paid to the description of the nodes interaction. Low-level data storage on individual nodes is not considered because it is assumed that this will use one of the existing database implementations. For a high level overview of Oasis.DDB and its concept please refer to the white paper https://oasisddb.com/files/white-paper-en.pdf.

The fact that Oasis.DDB is a non-relational database demonstrates some features of its operation. In particular, the inability to use the principles of **ACID** (**A**tomicity, **C**onsistency, **I**solation, **D**urability). **BASE** semantics (**B**asically **A**vailable, **S**oft state, **E**ventual consistency) is used instead of them, allowing to choose the trade-off between data consistency, availability and partition tolerance of the distributed system during its operation. More details about solving these problems and features of operation are described in the section on system features.

The system entities

Subjects of the system

The main subjects of the system are:

- Node - a node is a member of the system that stores and transmits data provided by users for a reward. The exact way to store data does not matter, the main fact is the ability to work in accordance with the protocol. The node receives a reward from the user for the work performed.
- Coordinator - a coordinator is a member of the database network that accept client requests, check the format of input data, transfer it to the nodes, and combine the results to send the response to the user. It can be represented by a node or a client. The coordinator receives a reward for the performed work from the nodes. If the user's client acts as a coordinator, the reward will be returned to the user after the nodes pay off their cheques.
- User - a user provides data to be stored in the decentral database and requests data for processing. The user rewards the nodes for the work performed.
- Client - a client is a tool (software component or application) that acts on behalf of the user, which formats the provided data for storage and processing, and performs financial settlements with the nodes.

System's Objects

- Tablespace - a tablespace is a collection of table descriptions and rules for accessing tables. Creating and managing tablespaces is implemented using smart-contracts.
- Table - a table is a collection of fields describing the data storage format, constraints, indexes for search and records containing user data. Access to the table is managed by smart-contracts related to the tablespace.
- Field - a field is a description of a data storage method, and an identifier for data access. Field management is performed for each table using smart-contracts
- Record - a record is a set of data transferred by a user to the database for storage and processing. The data structure must correspond to the fields in the table where they will be stored, and meet the requirements of table constraints.

- Index - an index can be of type primary (clustered), used to determine the node on which the record is stored, and non-clustered, used to search for data and work restrictions. Management of non-clustered indexes is performed using smart-contracts. A primary index can be assigned only once for each table and cannot be changed after creation.
- Constraint - a constraint is a condition for checking data. Data that does not satisfying one or more restrictions cannot be accepted by nodes.
- Request - a request is a data management tool. Requests can be performed to query or modify stored data.
    - Access request - allows you to search and retrieve data.
    - Modification request - allows you to add, modify and delete data.
- Cheque - a cheque is the main tool for financial settlements between the subjects of the system. There are several types of cheques, depending on the operations they are associated with. Cheques are eventually redeemed through a blockchain.

The rest of this document will provide more details about each subject and object of the system. Definitions of components will allow the reader to understand similarities and differences between Oasis.DDB and typical relational and nonrelational database management systems..

Node

A node is a main building element of the system. Nodes store data, perform search and verification for compliance with the constraints, and can also act as coordinators.

Coordinator

A coordinator is a role that a specific node or a client take over during the transmission of a request to the database system. Coordinators are the entities in charge of transport operations in the system. They are responsible for coordinating client requests and node responses and merging search results. They also perform data reconciliation and resolve conflicts when they encounter inconsistencies. In this case, the coordinators receive a reward from the nodes for successfully eliminating

the inconsistencies (for details, see the section on healing) and the successful transfer of data from the nodes to the client and from the client to the nodes. Also, the coordinators aggregate the responses from the nodes into a common result.

User

A user is a person on whose behalf the Client is working. The user provides data and manages the client's functions for making requests to Oasis.DDB, and makes financial settlements with nodes. Users are identified by their public keys, which are used to sign cheques, requests and data transferred to the nodes.

Client

A client is a tool to interface with the database. It can be a program written for a specific operating system, a library for a specific programming language, or any other solution that allows you to communicate with the Oasis.DDB system, according to the protocol. Like a node, it can act as a coordinator if needed.

Tablespace

A tablespace is a logical grouping of tables and table management rights. Usually one tablespace is allocated for each application. It groups the tables that hold the application data entities. Tablespaces allows you to define the operations that can be performed on tables by application users depending on their roles. Creating and managing tablespaces is performed on a blockchain using a smart-contract. All information about the rights to manage the structure of the database (creating and deleting tables, changing fields) is also stored in the tablespace in the blockchain.

Table

Tables directly stores user data organized into records. The mechanism of record storage may differ on different nodes, but should always provide the ability to execute queries according to the Oasis.DDB protocol. Each table should be assigned a primary (clustered) index on which the data will be distributed among the nodes of the network. Access rights to the table, as well as access to the tablespaces, are stored in the blockchain. Unlike the rights of a tablespace, they define the rights to create and modify data. Access to data search and retrieval is granted to all participants in Oasis.DDB, therefore it is recommended to use encryption to restrict access to vulnerable data, but search capabilities will be limited in this case.

Replication and partitioning levels

A table has parameters for replication and partitioning. They indicate how many replicas will exist for the given table, and how many parts it will be divided into accordingly. The maximum number of nodes serving this table can be calculated by multiplying these parameters. And the minimum number of nodes is the highest value among the level of replication and the level of partitioning, so that each node stores different ranges of different replicas. For example, to maintain a table with a replication level of 9 and a partitioning level of 100, you need at least 100 and not more than 900 nodes.

Field

The structure of a table is composed of fields and indexes. Each field has a unique name that allows it to be identified within a table, and a format that describes what type of data can be stored in this field. Information about the fields, as well as other information about the structure is stored in the blockchain, and is managed through smart-contracts.

Field format

The field format is defined by the datatype and size. The datatype characterizes the information stored in the field, and the possible ways of processing it. Filters and operations allowed to access request depend on field's data type. The size of the

field, relative to the type, imposes basic restrictions on the amount of data to be stored. A field cannot contain an amount of data that is bigger than its size.

Timestamp

All timestamps are defined as Unix epoch, that is the number of seconds elapsed since January 1st, 1970 UTC. Since the time synchronization is non-trivial in a decentralized system, timestamps in Oasis.DDB are synchronized with the blockchain (the timestamp of the last block). Such labels have a low resolution (about 15 seconds in the case of Ethereum), however that is sufficient for Oasis.DDB purposes.

Record

Records contain the data stored in Oasis.DDB tables. They consist of collections of items. Each item corresponds to a field in the target table and its type and size are defined by the target table's field definition. A Oasis.DDB client composes the single records according to the information provided and signed by the user. The record cannot contain fields that do not belong to the table specified in the record, and must refer to existing fields, tables, and tablespaces. Each record has a timestamp.

Version counter

Each record has a sequential version counter that cannot be modified directly. It controls modification concurrency and the sequence of changes. Each request to add a record sets the version counter to one (if the entry is added in place of the deleted one, the version counter continues with the version following the version of the deleted entry), and subsequent changes increment its value by one. Thus, this counter allows to easily identify and resolve conflicts of competitive changes in the record during the operation.

Record signature

Each record must have a header signed by the user who created this record. The signature of the header includes a hash of all the fields of the record, a timestamp and a version counter.

Index

Indexes are registries of records, allowing them to be found in the system. They can be built on one or several fields, and there are three types of them: clustered index, non-clustered (search) index, and local index.

Clustered index

The clustered (primary) index is the global unique identifier of the record. It is used to define the nodes on which the record will be stored, and from which the data will be retrieved. For each table, there can be only one clustered index. In Oasis.DDB, clustered indexes are implemented using hashes. The entire range of possible values of the hash function is distributed at once to the nodes that can serve it. A clustered index can be composite and can include several fields of primitive and compound types, excluding collection types.

Non-clustered (external) index

Non-clustered indexes are intended to speed up access to data and to organize certain types of restrictions. Non-clustered indexes, like primary ones, are global and distributed to nodes regardless of the records on which they are built. Such indexes, as well as tables, have replication and partitioning level parameters that indicates the number of nodes to which the index will be replicated and how many parts it will be divided into. Acceleration of access occurs due to the reduction in the number of requests to nodes in most cases. If there are indexes, the query conditions are first checked on the nodes storing the index, then the query goes only to the nodes that were found in the index. It should also be noted that if such indexes are present, the data addition and modification speed may decrease, because it will be necessary to maintain (update) all indexes in which the mutable fields participate.

Local (internal) index

Local indexes are used within the nodes that stores data. The local index is a kind of non-clustered index, but is distributed locally, along with the data. As a result, each node maintains its own local index, which is used for searches within the node. Local indexes can be convenient for sorting the results of a search query, or used as global non-clustered indexes without partitioning. It may be useful on small tables with a solid clustered index (the entire range of cluster index values, and thereby all records, are stored within the same node).

Constraints

Constraints in Oasis.DDB are understood as conditions that a field value must satisfy for a record to be accepted by the nodes. Constraints can be set on a specific field, or the entire table. Several different constraints can be applied to a record. The results of multiple constraints, set for fields and tables, will be conjunctively combined (logical multiplication) into one final result. Constraints are checked by nodes, and are monitored by a coordinator and can be of static, dynamic and "smart" type.

Static constraint

Static constraint, or field constraint, are conditions that a value in a record in a particular field should match. In case of a static constraint you can only use constants. The check is only applied to an incoming new value when this is set in a specific field of a new (inserted) or existing (updated) record. Records in which at least one field do not satisfy the conditions of static constraints are rejected by the nodes.

Dynamic constraint

Dynamic constraints are conditions that must be met by a combination of fields in a single record. An expression contained in a dynamic constraint can reference values contained in both the new and the existing record (if data is being modified), similarly to triggers in relational databases. Built-in functions that work with all table entries, such as checking the uniqueness of the value, can also be used. Dynamic

constraints allow convenient consistency controls typically provided by relational database management systems, but not by most non-relational ones. Dynamic constraints can be checked both by the nodes and the coordinator, indexes will be used to check the conditions if possible.

Smart constraint

Smart constraints are record checks based on smart-contract functions. Like in a dynamic constraint, arguments of the function of the smart-contract can originate from values present in new records, existing ones (in case of updates), and built-in functions. Smart-contract function calls can perform only read-only operations on the smart-contract data, and can only return one Boolean result.

Request

A request is a collection of information provided and signed by a user, containing either an appropriately structured record (raw data in case of data manipulation requests) or a search (query) statement and, if necessary, cheques for payment. The request is sent by the client, through the coordinator, to the nodes for receiving, adding and modifying data. Requests allow to perform only data operations, the database structure can be changed only through the blockchain. Each request, as well as a record, should be signed by a user. Each request is executed according to the specified level of consistency, which is defined by the logic of the client, or directly by the user.

Data access request

Requests to access data are used to search and retrieve data stored on nodes. The data access request contains search conditions and payment checks for data retrieval. When this query is executed, the coordinator parses the search conditions by checking whether indexes can be used to reduce the number of polled nodes. The request is then translated to the nodes on which data is stored (limited by index search results). The nodes, after performing the necessary operations with the data (search, filtering, sorting), return them to the client through the coordinator, in exchange for payment cheques for reading the data.

Data version request

Data version requests are lightweight and allow you to get the current version of the entry that you want to add, modify, or delete. It contains a set of primary key values and cheques for payment of query execution. The answer to this request is the version of the record for each primary key value in the query.

Data manipulation request

This type of request adds or modifies existing data in the Oasis.DDB database. Each data manipulation request provides a record including the new version of the record (if the record is created for the first time, then the version of the record will be equal to one, if the entry is inserted in place of a deleted one, the version of the record will be the next version of the deleted entry), the timestamp and the cheques to pay for the storage of the record if necessary (for the a new record storage cheques are required). The coordinator passes the data modification requests to the target nodes (responsible for the range of the clustered index in which the record is located) and receives from them the confirmation of storing the record (necessary to resolve the concurrent record racing problem) and the reward for the operation.

Request for adding data

The data adding request contains a record that you want to add. The request to add data will be accepted only if the entry with the specified values of the primary key is missing or was deleted, otherwise the request will be rejected by the node.

Request for data modification

Data modification request contains a record consisting of fields with new values that will replace the stored ones, primary key, timestamp and the version counter increased by one. The record in the request is signed using hashes of the values of all fields (returned in the access request along with the current version of the record), but contains only the changed data.

Request for data deletion

Data deletion request contains a record with the version number that come next to the version of the record to be deleted. The data in such a record is not

transmitted, and the signature is formed only by the version field and the timestamp. Deleted record are not returned anymore in response to data access requests.

Request consistency level

Consistency levels allow you to manage the **CAP** (**C**onsistency, **A**vailability, **P**artition tolerance) parameters dynamically while the system is running. Each client chooses which of the parameters to give preference to, depending on the function being performed. There are several levels of consistency for access requests and data modifications:

- **A**LL - the request must accept all the related nodes. The slowest and most consistent query. When you return a query result with this level of consistency, you can be sure that the record will be retrieved at any level of consistency in subsequent access requests.
- **Q**UORUM - request must be accepted by a quorum of nodes related to it. A quorum is the minimum number of nodes to achieve a majority (ie, more than half the nodes for each record). For example, for a table with a partitioning level of 2, and a replication level of 3, in the worst case (when the primary key of the desired record is not known in advance), 4 nodes will be polled (2 nodes per band). If the primary key is known, only the nodes servicing the range into which this key is included will be polled, that is, only one node will be polled for the larger half of the replicas - in this case only 2 nodes. This ensures that the information is stored on most nodes. If there is a discrepancy in the data before the return of the response, an automatic elimination of the inconsistency (healing) will be performed. This ensures that the returned record will propagate to the remaining nodes and will not be overwritten by another version. Thus, this is the second most consistent level of the query. When you return a query result with this consistency level, you can ensure that the record is read at the level of the access request consistency above or equal to the quorum level. By sacrificing availability, the user increases the speed of response and partitioning tolerance.
- **O**NE - the request must be accepted at least by one node. If this is a data manipulation request, then at least one node serving the primary key range and one node per index must accept it. If this is a data access request, then it

must be accepted by one node for each part of the primary key range, or an index node and a data node found in the index. In other words, the data access response is returned after polling only one replica of the data. It is the most unreliable, but fast and cheap level of consistency. It can be used to access and modify non-critical information, or with very rare and not urgent changes.

- **1**0P ... **9**0P - indicates the percentage of nodes that must respond to the request. Allows you to fine-tune queries. The total number of nodes is calculated from the partitioning level and the replication level of the table, but they differ in order to request modification and access to the data. In the table with the replication level 21 and the partitioning level 100, the total number of nodes on which the table entries are stored is a maximum of 2100. When the data modification request with access level 10P is executed, the coordinator will return a response if the request accepted at least 2 nodes (10% of all replicas responsible for the range to which the primary key of the modified record belongs). When requesting access to data with the same parameters, the query is distributed to all parts of the two replicas (10% of all replicas), i.e. for 200 nodes.

It should be noted that a coordinator, after returning a response to the data manipulation request to the client, can continue to distribute the record to all the nodes related to it, regardless of the selected consistency level, in order to receive a reward from these nodes. But to ensure that the record has arrived to all nodes after receiving a successful response by the client, only the consistency level **A**ll could be used.

Cheque

Cheques are the main tool of financial settlements with nodes in Oasis.DDB. Cheques allow to process majority of mutual settlements off-chain, thus providing high speed and low cost of transactions. The user enclose cheques to the record or requests to nodes as a reward for their work. All cheques have a number from a specified range, the identifier of the node to which it was issued and signature of a user.

The cheques should be eventually cashed through a blockchain. However not any cheques can be cashed directly. Cheques are stored on the node they are targeted to. In the process of node and client communication the cheques are accumulated into cumulative cheque and after that original cheques can be discarded. The cumulative cheque can be passed to smart contract for cash.

Details about the structure of cheques are described in the structure of the cheque. There are several types of cheques.

Data access cheque

Data access cheque confirms the off-chain payment fact for the search and data retrieval. They are passed to the nodes in exchange for the result of the query.

Search Cheque

The search cheque pays for finding records on the nodes (for nowis a fixed amount, it will probably be calculated depending on the complexity of the search conditions).

Read cheque

The data received from the nodes are paid with a read cheque (the amount depends on the amount of data transferred).

Data manipulation cheque

Data modification cheque confirms the off-chain payment fact for requests of creation and modification of data. Attaching to the generated requests transmitted to the nodes.

Creation and modification cheque

Cheques for creating and modifying records are used to pay for creating new or updating existing records on the nodes (the amount depends on the amount of data written).

Deletion cheque

Deletion cheques are used to pay for records deletion from nodes.

Storage cheque

Data storage is paid for by storage cheques. That type of cheque is tied to the header of the record, which includes the primary key of the record, the version of the record, and the time of creating the record synchronized with the last block in the blockchain (timestamp). Thus, this type of cheque can not be repaid without a version of the record (or at least its title), for the storage of which it was issued. A storage cheque is issued for the entire period of storage of the record, but it is paid either at the expiration of the storage period of the record or when the record is updated with a new version.

Cumulative cheque

Cumulative cheque, or cheque with a cumulative total, is a final destination cheque. It aggregates cheques of any type, issued by the user, using the numbering ranges. This is the only type of check that is subject to direct cashing through a blockchain smart-contract.

Cheque numbering range

Each cheque should be clearly identified when included in the cheque with a cumulative total, in order to avoid re-inclusion in the final amount. For this purpose, cheque numbering ranges are used. Each range is created for each client on the node and has a unique identifier and version. The range stores an array of sequences of included cheques. When you issue a cheque, it includes the node ID, range identifier and number from this range (see cheque structure).

Issuing cheques

In accordance with the cheque structure, cheques are issued with numbers from the range corresponding to the user's client (application working with Oasis.DDB). The cheque numbering range is issued for a specific copy of the client. If the user is working from several clients then each client will create its own range. The created range can be thought of as a long-term session. Cheques are created and signed automatically by the user. To obtain a range the client generates a new UUID, which becomes a unique range identifier, or uses the existing one. Thus, clients do not need synchronization for work, each cheque is issued within its client's range, and is included by nodes in a cumulative cheque separately. Also, there is no

need to know the state of the node, which allows you to start working immediately, without additional costs of synchronization.

Cheques repayment

Cheques for creating, modifying and storing data can not be cashed directly. To do so they must be included into a current cheque with a cumulative total. This happens on the node's initiative. Cheques are processed on the client in a low-priority background. Client verifies all the cheques, accumulate them in cumulative cheque, sign it and return an updated cumulative cheque to the node.

Data modification cheques repayment

The node sends a cheque with a cumulative total and cheques of the data that it wants to pay off to the client, as well as the latest versions of the cheque numbering ranges to which the data cheques belong, and the missing hashes for the hash root calculation of the branches and leaves of the Merkle tree, the root of which is stored in the cheque with the cumulative total (see cumulative cheque structure). The client checks the signatures of cheques and ranges, and calculates the root of the Merkle tree from the transmitted hashes, then checks that it matches the root transmitted in the cumulative cheque. After the validation of the input data, the user marks the cheques in the ranges as included, increases the total amount of the cumulative cheque by the amount of included checks (signing the new versions of the ranges), and rebuilds the Merkle tree taking into account the new values. After the calculations it replaces the value of the root of the Merkle tree with the new one and signs the resulting check with the cumulative total. After the calculations, the client returns new ranges and a check with a cumulative total of the node.

Data storage cheques repayment

Data storage cheques are paid in the same manner as manipulation and access cheques are, but the cheque is accompanied by the record whose storage is ensured by the cheque. If the cheque is included on the expiration date of the record, a request for the latest version of the record can be executed to verify the validity of the repayment. In response a new storage cheque can be returned, if the storage period needs to be extended. In case the repayment of the storage cheque occurs when a new version of the record appears, the storage cheque and the header of the new version of the record, signed by the user who created the new

record, are attached to the request. Thus, the retention period is not the period of storage for which the cheque is issued, but the time of actual storage, from the time of creation of the current cheque, to the time when the cheque receipt for the new version of the record was created.

You can find more info on cheque repayment in the protocol section

Blaming

Blaming is the mechanism of accusing nodes of incorrect work, and the subsequent establishment of guilt and bringing to justice. All participants in the system can use this mechanism to point out that the system participant is not acting bona fide. To this end, a smart-contract has been implemented in the blockchain, with methods corresponding to each situation that emerges, accepting the identifier of the other system member and other parameters. When the participant accumulates a certain number of blames, the participant is brought to justice by means of contractual penalties.

Healing

Healing is a mechanism for eliminating record's inconsistency. In the distributed system, situations may arise where access to some network members is limited or impossible (so-called partitioning). It is not critical for reading data because information does not change and since the record was distributed to all replicas, so even one node is sufficient to retrieve stored information. Otherwise, when data is modified, inaccessible nodes not get a record. To eliminate this inconsistency, healing is used. The recovery procedure itself occurs when requests for data access are made, with a consistency level higher than that of the system's current. If the coordinator, when searching for records, finds a discrepancy (for example, some replicas return older records, or no records are written to them), he can begin the process of data distribution to non-consistent nodes. To resolve this, the coordinator requests a record from one of the consistent replicas (the entire record, or those parts that have changed from the previous version), and sends them to a non-consistent replica for a fee. This procedure occurs in the background, after a

response to the client and on the initiative of the coordinator, if the required level of consistency could be obtained for the current system state. For example, when accessing the data with request consistency level One and query consistency level All, polling all replicas to find the fastest, an inconsistent node can be detected, but since it does not affect the ability to respond, first of the responses will be returned to the client with the newest version of the record, and then an attempt to update the record on the obsolete nodes will be made. However, in case of inability to obtain requested consistency level from the current system state (there are not enough consistent nodes in the system for the required level of response consistency, for example, quorum), the attempt to forward the newest version of the record will be taken before responding to the client, until the requested consistency is achieved. This approach may slow down responses to requests with the most consistent levels, but it automatically corrects synchronization errors, and ensures the required consistency level eventually, despite a periodic node access problems and system partitioning.

Competitive records race

It is possible there will be two different versions of a record with one version number. For example, two clients changed the same record with the replication level 9 from version 1 to version 2 in different ways, with the lowest level of consistency - One (i.e., the response was received by the clients when an entry hits at least one of the replicas). After returning responses to clients, the coordinators (one for each client they are working with) can continue to distribute the records that have been transferred through them to the remaining nodes. This distribution will occur simultaneously and competitively, and if one of the coordinators would try to write one version of the record to a node that already contains another version of the record, an error will occur. The coordinator that has received errors from most replicas stops distributing their version of the record, and another coordinator who has distributed their version of the record to the majority of nodes, if an error occurs during propagation, passes to this nodes the acknowledgments from the majority of nodes that they have received this entry. The node receiving the acknowledgments from most replicas, checks their validity, and upon successful verification takes a different version of the record. Thus, the coordinator, who managed to distribute the record to the majority of nodes, eventually gets a reward from all the nodes. The same situation can occur during the healing, but in this case

the version of the record stored on most replicas is computed first and race of competitive records does not occur.

Data types

All numeric types in Oasis.DDB are signed, two's complement coded, floating-point numeric types meet the IEEE 754 standard, and time formats are ISO 8601 compliant

Primitive types:

- Boolean - logical type. Can take 2 possible values: true or false.
- Integer - numeric type with size of 32 bits. From -2147483648 up to 2147483647.
- Long - numeric type with size of 64 bits. From -9223372036854775808 to 9223372036854775807.
- Float - single precision floating point numeric type with size of 32 bits.
- Double - double precision floating point numeric type with size of 64 bits.
- Decimal - arbitrary-precision signed numeric type. Size depends on chosen scale and precision.
- String - is a text type. It consists of the specified character encoding and a sequence of octets of the encoded text string.
- Binary - is a raw binary type. A sequence of octets.
- Time - is a time type that includes a date. The size depends on the selected accuracy.
- Duration - is a time type specifying the time interval. The size depends on the selected accuracy.

Collections:

- List - is a sequence of elements, with preserved order.
- Bag - is a set of elements. Depending on the settings, it can be a set of unique elements or a multiset.
- Map - is an associative array of elements with a primitive type as a key. In future it will be extended for usage of composite types as a key.

Compound:

- Structure - is an object with a fixed set of elements. Access to elements can be made like in an associative array by a number, or by name if the elements are named.

Objects structure

This section provides a general description of the structure of objects, without specifying the data types used for each element. This is necessary to simplify the description of operations on objects, using references to their elements.

Descriptions are given in the following format:

$A\ [B,\ ?C,\ D_{0..n}]$, where

A - is a parent element for a structural elements B, C and D, i.e. A contains elements B, C and D.

B - is a mandatory element of the parent element A.

C - is an optional element of the parent element A.

$D_{0..n}$ - is a set of elements of the parent element A, from 0 to n.

Also, links to structural elements are possible:

B^A - is a reference to the structural element B of the parent element A, i.e. B is contained in A.

$C^{A.X.Z}$ - is a reference to the structural element C of the parent element A. The element X is the parent of A, and Z is the parent of X, i.e. Z contains X, X contains A, and A contains C.

D_3^A - is a reference to the third element of the set D of the parent element A.

If an element is a function computed from other elements, the following format is applied:

$A\,[S(B^A),\ B,\ C,\ \#E(B^A, C^A)]$, where

A - is a parent element.

B and C - are the structural elements of the parent element A.

$S(B^A)$ - is a stored structural element of the parent element A, computed from the structural element B of the parent element A, i.e. $S(B^A)$ is contained in A, also is the result of the function from B^A.

$\#E(B^A, C^A)$ - is a structural element of the parent element A that is not stored in the parent element but is calculated from the structural elements B and C of the parent element A.

If the structural composition of an element could not be determined, or depends on factors that can not be described in this format, it has the following format:

$X\,[?]$, where

X - is a parent element whose structural elements could not be determined.

If the structural composition of an element is the same as the other element, or is not important, it can be omitted, in which case the omitted elements are replaced by an ellipsis:

$X\,[Y,\ ...]$, where

X - is a parent element with some unimportant structural elements.

Y - is a structural element of the parent element X which is important in the current context.

If the parent element can have only one structural element of several, depending on the situation, it can be written as follows:

$X\,[Y||Z,\ S(Y^x||Z^x)]$, where

X - is a parent element.

Y and Z - are structural elements, one of which (but not both) must be present in the parent element X.

S - is a structural element of the parent element X, which is calculated from one of the elements Y or Z, depending on their presence.

Structure of the table space

The table space includes the name, access restrictions, and a set of tables within this space.

$Tbs\ [Tsn,\ Rst,\ Tbl_{0..n}]$, where

Tsn - is the name of the table space.

Rst - access restrictions for all elements within this space.

$Tbl_{0..n}$ - is the set of tables in this space.

Tables structure

The table consists of a name, a set of fields, a clustered index, and sets of local indexes, external indexes, dynamic constraints, and smart constraints. Each of the indexes has a set of fields. The clustered and external indexes also contain replication, data, and key parameters, respectively. The external index has a unique name, which allows to use it in the constraint definition and data access requests.

Structure of replication and partitioning parameters

The replication and partitioning parameters consist of the value of the replication level and the value of the level of partitioning. They are placed in a separate structure since they are always stated in pair.

$Rpc\ [Rpl,\ Prl,\ Rpk_{0..r,\ 0..p}]$, where

Rpl - replication level (number of replicas).

Prl - partitioning level (the number of parts per replica).

$Rpk_{0..r,\ 0..p}$ - is the space of replicas of the distribution ranges.

Table structure

$Tbl\,[Tbn,\ Fld_{0..n},\ Cdx,\ Idx_{0..m},\ Edx_{0..l},\ Trs_{0..k}]$, and

$Tbl\,[Cdx[Rpc,\ Fln_{0..a}^{Fld.Tbl}],\ Idx[Idn,\ Fln_{0..b}^{Fld.Tbl}]_{0..m},\ Edx[Edn,\ Rpc,\ Fln_{0..c}^{Fld.Tbl}]_{0..l},\ ...]$, and

$Tbl\,[Trs[Trn,\ Trt,\ Trc]_{0..k},\ ...]$, where

Tbn - is a table name.

$Fld_{0..n}$ - is a set of fields in a table.

Cdx - is a clustered index used to distribute records among nodes.

Rpc^{Cdx} - parameters of table replication and partitioning.

$Fln_{0..a}^{Cdx}$ - the names of the fields involved in primary key index.

$Idx_{0..m}$ - is a set of local indexes.

Idn^{Idx} - is a name of local index.

$Fln_{0..b}^{Idx}$ - names of fields of the table participating in the local index.

$Edx_{0..l}$ - is a set of external non-clustered indexes.

Edn^{Edx} - is a name of external index.

Rpc^{Edx} - external index replication parameters.

$Fln_{0..c}^{Edx}$ - are names of the fields in the table that participate in the external index.

$Trs_{0..k}$ - is a set of dynamic and "smart" constraints for the current table..

Trn - is the name of the constraint unique to the table.

Trt - is restriction type (dynamic or smart) that specifies the format of the constraint condition string.

Trc - conditions of constraint, corresponding to the specified format.

Field structure

The field consists of the field name, type of stored data, parameters specifying the characteristics of the specified data type (for example, size and accuracy) and a set of static constraints.

$Fld\,[Fln,\ T,\ ?Tp,\ ?Frc]$, where

Fln - is a name of field in a table.

T - is a data type.

Tp - parameters of the specified data type.

Frc - conditions of static constraints of the field, corresponding to the field type.

Signature structure

The signature consists of a public key recovery ID and two parts of the signature itself, generated by the ECDSA algorithm.

$Sig\,[Sv,\ Sr,\ Ss]$, where

Sv - is a header of the signature, an ID of key recovery.

Sr and Ss - parts of digital signature.

Record structure

Each record consists of a record header, signature of its hash, a list of modified or created fields with values, and payment cheques for creating and storing the record. In its turn, the hash of record is constructed from hashes of pairs of names and values of all fields (not only changed, but all fields that will be contained in the record after distribution to nodes) and hashes of payment checks for this record. The header contains the name of the tablespace to which the modified table belongs, the name of the table, the values of the primary key fields in the order specified in the clustered index, the version of the record (next version of the record to be changed), the timestamp synchronized with the last block in the blockchain and hash of the record itself. Fields in a record are name-value pairs.

Header structure

$Hdr\ [\#Hash(...),\ Tsn,\ Tbn,\ Pk[?],\ V,\ Et,\ Tim,\ Hen = Hash^{Ent}]$, and

$Hdr\ [\#Hash(Tsn^{Hdr},\ Tbn^{Hdr},\ Pk^{Hdr},\ V^{Hdr},\ Et^{Hdr},\ Tim^{Hdr},\ Hen^{Hdr}),\ ...]$, where

Tsn - is the name of the tablespace which table belongs to.

Tbn - is the name of a table in the tablespace.

Pk - is a set of primary key fields values, given accordingly to the order specified in the primary key index in the table.

V - version of the record.

Et - type of record: new record, modification, deleted record.

Tim - timestamp of the record creation.

Hen - hash of the record to which the header belongs.

$Hash$ - hash of the header of the record is based on the names of the tablespace and table, the values of the primary key, the version of the record, the timestamp and the hash of the record to which the header belongs.

Field structure

$Fld\ [\#Hash(Fln^{Fld},\ Hash(Flv^{Fld})),\ Fln,\ Flv]$, where

Fln - name of the field.

Flv - value of the field.

$Hash$ - hash of name and hashed value.

Record structure

$Ent\ [\#Hash(Hash_{0..n}^{Fld.Ent}),\ Sig(Hash^{Hdr.Ent}),\ Hdr,\ Fld_{0..n},\ Chq_{0..m}]$, where

Sig - signature built on header's hash

Hdr - record's header

$Fld_{0..n}$ - record's fields

$Chq_{0..m}$ -storage and data manipulation cheques

$Hash$ - record hash calculated from all the fields hashes

Request structure

Each request generally includes a signature, a request type, and a consistency level. Manipulation requests include a record, the hash of which is included in the signature generation. Search requests include search query, the hash of which is also included in the request signature. The search conditions include the type of search query, the query string, the search payment cheques and the signature. The hash function of the search query is based on the type of query, query string and hash of the request payment cheques. Only the hash of the search conditions is signed.

Structure of modification request

$Rm\ [?Sig(Cl^{Rm},\ Hash^{Hdr.Ent.Rm}),\ Cl,\ Ent]$, where

Sig - signature, built on the hash of the record, the level of consistency and the type of request.

Cl - is the request consistency level.

Ent - record that should be added or modified.

Structure of search conditions

$Qry\ [\#Hash(Tsn^{Qry},\ Tbn^{Qry},\ Qt^{Qry},\ S^{Qry}),\ Sig(Hash^{Qry}),\ Tsn,\ Tbn,\ Qt,\ S,\ Chq]$, where

Tsn - is the name of the tablespace to which the table belongs.

Tbn - is tablespace name to which a table belongs

Qt - type of search conditions indicating the format used in the search conditions.

S - search conditions, according to the conditions type format.

Chq - is search cheque paid to nodes.

$Hash$ - hash of search request computed from its type, query string, name of table and a tablespace..

Structure of access request

$Rs\ [?Sig(Cl^{Rs},\ Hash^{Qry.Rs}),\ Cl,\ Qry]$, where

Sig - signature, built on the hash of the record, the level of consistency and the type of request.

Cl - is the request consistency level.

Qry - search conditions and payment request

Data storage confirmation structure

A node, after successful storage of a retrieved data, returns a structure consisting of the record's header hash, the node's signature of that hash, and the cheque to pay for the coordinator's work.

$Ace\ [Sig(Hen^{Ace}),\ Hen = Hash^{Hdr.Ent},\ Chq]$, where

Sig - signature of record header's cheque that has received data

Hen - header hash that was received

Chq - cheque issued by a node to coordinator that passed the record

Successful search response structure

In the case of successful data search, the node returns result records consisting of the original record header, the original record signature, the original record fields that were selected for return in the search query, and the add-on fields containing the hashes of the field values, instead of the actual values.

$Srs\ [Enr\ [\#Ent,\ Sig^{Ent.Enr},\ Hdr^{Ent.Enr},\ Fld_{0..n}^{Ent.Enr},\ Fls\ [Fln,\ Flh]_{0..m}]_{0..p}]$, where

Enr - special record returned in search response with some add-on fields.

Ent - original record stored in database.

Sig - signature of the record.

Hdr - header of the original record.

$Fld_{0..n}$ - fields of record that should be returned.

$Fls_{0..m}$ - fields of record that exist in original record but are not requested in query.

Flh - hash function from field value.

Request error structure

A request execution error includes an error hash signature, a record header hash when processing a modification request, or a search condition hash for the access request, an error code, and a text description of the error, if necessary.

$Err\ [\#Hash(H^{Err},\ Ec^{Err},\ ?Msg^{Err}),\ Sig(Hash^{Err}),\ H = (Hash^{Hdr.Ent}||Hash^{Qry}),\ Ec,\ ?Msg]$, where

Sig - signature of error hash by a node which encountered the error.

H - hash of a record header that was not accepted or of the search request that can not be executed.

Ec - error code of request execution.

$?Msg$ - textual description of the error if necessary.

Data access and data manipulation cheques structure

A cheque consists of a number from a specific numbering range, the identifier of this numbering range, the identifier of the node to which it was issued, and the amount of compensation that the node receives when the cheque is redeemed. The signature of the hash formed from all the cheque fields.

$Chq\ [\#Hash,\ Sig(Hash^{Chq}),\ Tim,\ Rid\ ,\ Num\ ,\ Nid_{0..n},\ Amt]$, and

$Chq\ [\#Hash(Tim^{Chq},\ Rid^{Chq},\ Num^{Chq},\ Nid^{Chq}_{0..n},\ Amt^{Chq}),\ ...]$, where

Sig - signature built on the hash of the check

Rid - is the identifier of the range, in which this cheque is issued.

Num - is the number of the cheque within the range.

$Nid_{0..n}$ - are the identifiers of the nodes, to which this cheque was issued

Amt - is the reward amount

$Hash$ - cheque hash, built on the range identifier, cheque number, node ID and cheque amount.

Data storage cheque structure

Storage cheques differ from the access and manipulations cheques in a way the signature of the storage cheque includes the header of the record for which it was issued.

$Chq\ [Sig(Hash^{Hdr.Ent},\ Hash^{Chq}),\ ...]$, where

Sig - a signature built on the hash of the header of the record, for which this cheque was registered and the hash of the cheque itself.

Cheque range structure

The cheques numbering range consists of its identifier, a set of continuous sequences of redeemed cheques, and a signature built on range's hash. The range hash is in turn built on the range identifier and the pairs of cheque numbers (the smallest and largest) of the continuous sequences of redeemed cheques. The full range is stored on the node, and the client generates and stores only its ID and the number of the last issued cheque.

$Rng\ [\#Hash(Id^{Rng},\ Pay[Lcq,\ Hcq]_{0..n}^{Rng}),\ Sig(Hash^{Rng}),\ Id,\ Pay[Lcq,\ Hcq]_{0..n}]$, where

Sig - a signature built on range's hash

Id - is the unique identifier of the range created by the client.

$Pay_{0..n}$ - continuous sequences of canceled cheques

Lcq - is the smallest cheque number in a continuous sequence.

Hcq - is the largest cheque number in a continuous sequence

$Hash$ - range hash, built on the range identifier and the intervals of the canceled checks.

Total cumulative cheque structure

A cheque with a cumulative total consists of a signature, constructed from its hash, node ID, time range, the root of the Merkle tree and the total amount of the cheque. Timestamps of the final cheque are used to limit the inclusion of cheques, the timestamp of which is before the start of the cumulative cheque timestamp. All cheques issued before the start of the cumulative cheque are accepted by the node without payment. When the cumulative cheque is redeemed in the blockchain, the total amount and the timestamp of the latest written check are saved to the blockchain, which becomes the start time should a new cumulative cheque be issued. The hash of the cumulative cheque is based on the node ID, the root of the Merkle tree and the total amount.

Tcq $[\#Hash(Nid^{Tcq},\ Mtr^{Tcq},\ Amt^{Tcq}),\ Sig(Hash^{Tcq}),\ Nid,\ Tms,\ Tml,\ Mtr,\ Amt]$, where

Sig - signature built on cheque's final hash

Nid - node identificator

Tms - minimal timestamp for cheques to be included in this cumulative cheque

Tml - timestamp of the latest issued cheque, included in this cumulative cheque

Mtr - Merkle tree key hash built upon hashes of cheque ranges

Amt - total amount of all included cheques

Protocol

To describe the operations and relationships of elements within the protocol, it is necessary to extend the format of the description with the following notation.

A group is understood as the collection of elements that satisfy the condition inherent to this group. For example, a group of tables, each of which has at least one external index: $\langle Tbl \mid Edx_{0..n}^{Tbl} \neq \varnothing \rangle$.

In case of groups of structural elements, it is useful to consider additional restrictions. Namely, a group of elements can be further limited by the condition of belonging to a particular parent element. If the group to which it belongs is not specified in the group, such group is called absolute, and all the elements specified in the group are included in it. Relative groups indicate, relative to which element it is necessary to select a subset of the elements specified in the group. The links to the group are indicated as follows:

$\langle A \rangle$ - a reference to a group of elements A, relative to the whole system.

$\langle A_{0..n} \rangle \coloneqq A_{0..n}$ - the reference to the group of the set of elements of A without restrictions is equal to this set by definition.

$\langle B^A \rangle$ - reference to the absolute group of structural elements B of the parent element A relative to the whole system. Also $\langle B^A \rangle = \langle B \rangle^{\langle A \rangle}$.

$\langle C^B \rangle^A$ - reference to the group of structural elements C of parent element B relative to the element A.

Also, groups can have restrictions on the inclusion of elements that are specified after the vertical line:

$\langle a \mid a \in \langle A \rangle \rangle \coloneqq \langle A \rangle$ - reference to the absolute group of elements A relative to the whole system. The general case.

$\langle A \mid A \neq B \rangle \coloneqq \langle A \mid \neg B \rangle$ - reference to the absolute group of elements A, except element B.

$\langle M \mid n = N^M \rangle$ - reference to the absolute group of elements M, structured element N of which equals n.

$\langle A[S, X_{0..n}] \mid x \notin \langle X \rangle^A \rangle$ - reference to an absolute group of elements A, group of child elements X of which does not contain element x.

$\langle S^A \mid x \notin \langle X \rangle^A \rangle$ - reference to group S of parent element A whose group of child elements X of parent element A does not contain element x.

Operations and functions as well as computed elements of objects are denoted by round parentheses with arguments necessary for the computation. They can be either named or unnamed.

Functions that do not affect the state of the system are denoted by the equality sign. The absence of side effects allows us to use them as conditions:

$(A,\ B) = Y$ - is a function that allows you to get the element Y of elements A and B and the condition that for elements A and B it is possible to get the element Y, existing in the system.

$(A,\ B) \neq Y$ - is a function that returns an error when trying to get Y of elements A and B. The condition that for elements A and B it is not possible to get the element Y.

$check(A,\ B)$ - named function without definition. The definition can be done separately by the function name. Or the context may imply that definition is known.

Unnamed function without definition serves as simple grouping:

$$(A) \coloneqq A \quad \Rightarrow \quad (A \parallel B) \coloneqq A \parallel B \quad \Rightarrow \quad (A \parallel B)^{X}_{0..n} \coloneqq A^{X}_{0..n} \parallel B^{X}_{0..n}.$$

Operations that change the state of the system are denoted by a function call and the notation of mapping or transition (arrow). To the left of the arrow (denoting the effect), the operation call and the current state of the system if needed are indicated, to the right is the state to which the system has passed as a result of the call:

$(A) \rightarrow B \in \langle B \rangle$ - an operation that generates a new element B (belonging to the absolute group of elements of B) from the element A.

$(X) \rightarrow B \notin \langle B \rangle$ - is an operation that excludes the existing element B (belonging to the absolute group of elements B) from system using the element X. This implies the existence of a function $(X) = B$, that maps an element X of the group $\langle X \rangle$ to an element B of the group $\langle B \rangle$.

To differentiate between elements with the same name we use strokes. For example: $A \neq A' \neq A'' \neq A'''$.

Managing tablespace

Tablespaces are stored in the blockchain, and are managed by calling the functions of smart-contracts.

Creating tablespace

The tablespace is created by executing the function of the smart-contract from the name of the tablespace (if there is not already a tablespace with that name in the system) and a set of access restrictions, provided that there is permission in the access restrictions to create a tablespace with this name for the user who called the creation function. On successful completion, the system creates a tablespace with the specified name and access restrictions.

$(Tsn,\ Rst) \rightarrow Tbs\ [Tsn,\ Rst,\ ...] \in \langle Tbs \rangle \Rightarrow Tsn \in \langle Tsn^{Tbs} \rangle$, if

$Tsn \notin \langle Tsn^{Tbs} \rangle$, and

$createTablespace^{Rst}(Tsn)$, where

Tsn - name of the new tablespace.

Rst - access restrictions for all elements within this tablespace allowing creation of tablespace with this name.

$createTablespace^{Rst}$ - function for checking rights of creation of a tablespace with specified name.

Deleting tablespace

The tablespace is deleted by calling the smart-contract function from the name of the existing tablespace, provided that there is permission in the access restrictions to delete the tablespace for the user who called the delete function.

$(Tsn) \rightarrow Tbs \notin \langle Tbs \rangle \Rightarrow Tsn \notin \langle Tsn^{Tbs} \rangle$, if

$\exists\, Tbs \in \langle Tbs \rangle,\quad Tsn = Tsn^{Tbs}$, and

$deleteTablespace^{Rst.Tbs}(Tsn)$, where

Tsn - name of existing tablespace.

$deleteTablespace^{Rst.Tbs}$ - function for checking rights to delete tablespace with specified name.

Managing tables

Tables, like tablespaces, are stored in the blockchain within those tablespaces, and they are managed by calling smart-contract functions.

Table creation

The table is created in the tablespace by executing the smart-contract function of the name of the tablespace and the table name (if there is no table with the same name in the tablespace), provided that there is permission to create a table with this name for the user who called the creation function in the tablespace access restrictions. On successful completion, a table with the specified name is created within the tablespace.

$(Tsn,\ Tbn) \rightarrow Tbl\ [Tbn,\ ...] \in \langle Tbl \rangle^{Tbs} \Rightarrow Tbn \in \langle Tbn^{Tbl} \rangle^{Tbs}$, if

$\exists\, Tbs \in \langle Tbs \rangle,\quad Tsn\ =\ Tsn^{Tbs}$, and

$Tbn\ \notin \langle Tbn^{Tbl} \rangle^{Tbs}$, and

$createTable^{Rst.Tbs}(Tsn,\ Tbn)$, where

Tbn - name of new table unique within the tablespace.

Tsn - name of existing tablespace which will contain the new table.

$createTable^{Rst.Tbs}$ - function for checking rights to create table with specified name in the specified tablespace.

Table deletion

Table is deleted from the tablespace by calling the smart-contract function of the name of the existing tablespace and the table name in that tablespace, provided that there is permission in the access restrictions to delete the table in the tablespace for the user who called the delete function.

$(Tsn,\ Tbn) \rightarrow Tbl \notin \langle Tbl \rangle^{Tbs} \Rightarrow Tbn \notin \langle Tbn^{Tbl} \rangle^{Tbs}$, if

$\exists\, Tbs \in \langle Tbs \rangle, \quad Tsn = Tsn^{Tbs}$, and

$\exists\, Tbl \in \langle Tbl \rangle^{Tbs}, \quad Tbn = Tbn^{Tbl.Tbs}$, and

$deleteTable^{Rst.Tbs}(Tsn, Tbn)$, where

Tbn - name of the table to be deleted.

Tsn - name of the existing tablespace whose table is to be deleted.

$deleteTable^{Rst.Tbs}$ - function to check rights to delete table with the specified name from the tablespace in access permissions of this tablespace.

Field management

The fields of the table are stored in the blockchain within the table itself, their management, as well as the tables, is performed by calling the functions of smart-contracts.

Field creation

The field is created in the table by executing the smart-contract function of the name of the tablespace, the table name, the name of the new field (if the field does not already exist with this name), the data type for this field, and the parameters of the selected data type, provided that there is permission to create a field with this name for the user who called the create function. On successful completion, a field with the specified name, data type, and data type settings is created in the table.

$(Tsn, Tbn, Fln, T, ?Tp) \rightarrow Fld\,[Fln, T, ?Tp, ...] \in \langle Fld \rangle^{Tbl.Tbs} \Rightarrow Fln \in \langle Fln^{Fld} \rangle^{Tbl.Tbs}$, if

$\exists\, Tbs \in \langle Tbs \rangle, \quad Tsn = Tsn^{Tbs}$, and

$\exists\, Tbl \in \langle Tbl \rangle^{Tbs}, \quad Tbn = Tbn^{Tbl.Tbs}$, and

$Fln \notin \langle Fln^{Fld} \rangle^{Tbl.Tbs}$, and

$createField^{Rst.Tbs}(Tsn, Tbn, Fln)$, where

Fln - name of the new field, unique within the table.

Tbn - name of an existing table in the tablespace.

Tsn - name of the existing tablespace.

T - data type of the new field.

Tp - parameters for the specified data type.

$createField^{Rst.Tbs}$ - function for checking permission to create the field with the specified name within the table in the tablespace, in the access permissions of that tablespace.

Field deletion

The field is deleted from the tablespace table by calling the smart-contract function from the name of the existing tablespace, the table name and the name of the existing field in this table, provided that there is permission in the access restrictions to delete the field in the tablespace table for the user that called the delete function.

$(Tsn,\ Tbn,\ Fln) \rightarrow Fld \notin \langle Fld \rangle^{Tbl.Tbs} \Rightarrow Fln \notin \langle Fln^{Fld} \rangle^{Tbl.Tbs}$, if

$\exists\, Tbs \in \langle Tbs \rangle,\quad Tsn = Tsn^{Tbs}$, and

$\exists\, Tbl \in \langle Tbl \rangle^{Tbs},\quad Tbn = Tbn^{Tbl.Tbs}$, and

$\exists\, Fld \in \langle Fld \rangle^{Tbl.Tbs},\quad Fln = Fln^{Fld.Tbl.Tbs}$, and

$deleteField^{Rst.Tbs}(Tsn,\ Tbn,\ Fln)$, where

Fln - name of the field to be deleted.

Tbn - name of the table existing in the tablespace.

Tsn - name of the existing tablespace.

$deleteField^{Rst.Tbs}$ - function for checking permission to delete the field with the specified name from the table in the tablespace, in the access permissions of that tablespace.

Indexes management

Indexes are created by the functions of smart-contracts in the blockchain and used by nodes when executing queries. The indexes allow you to determine the nodes responsible for the recording and reduce the number of the nodes being polled by searching.

Cluster index creation

$(Tsn,\ Tbn,\ Fln_{0..n},\ Rpl,\ Prl) \rightarrow Cdx^{Tbl.Tbs} = Cdx\ [Rpc[Rpl,\ Prl,\ ...],\ Fln_{0..n}]$, if

$\exists\, Tbs \in \langle Tbs \rangle,\quad Tsn = Tsn^{Tbs}$, and

$\exists\, Tbl \in \langle Tbl \rangle^{Tbs},\quad Tbn = Tbn^{Tbl.Tbs}$, and

$Fln_{0..n} \in \langle Fln^{Fld} \rangle^{Tbl.Tbs},\quad Fln = Fln^{Fld.Tbl.Tbs}$, and

$publishTable^{Rst.Tbs}(Tsn,\ Tbn)$, where

Rpc - parameters for replication and partitioning of table records.

Rpl - table records replication parameters.

Prl - table records partitioning parameters.

$Fln_{0..n}$ - names of the fields in the table that will be used to build the clustered index.

Tbn - name of the existing table in the tablespace.

Tsn - name of the existing tablespace.

$publishTable^{Rst.Tbs}$ - function for checking permission to create the clustered index within the table in tablespace, in the access permissions of that tablespace.

External index creation

$(Tsn,\ Tbn,\ Fln_{0..n},\ Edn,\ Rpl,\ Prl) \rightarrow Edx\ [Edn,\ Rpc[Rpl,\ Prl,\ ...],\ Fln_{0..n}] \in \langle Edx \rangle^{Tbl.Tbs} \Rightarrow$

$\Rightarrow Edn \in \langle Edn^{Edx} \rangle^{Tbl.Tbs}$, if

$\exists\, Tbs \in \langle Tbs \rangle,\quad Tsn = Tsn^{Tbs}$, and

$\exists\, Tbl \in \langle Tbl \rangle^{Tbs}, \quad Tbn = Tbn^{Tbl.Tbs}$, and

$Fln_{0..n} \in \langle Fln^{Fld} \rangle^{Tbl.Tbs}, \quad Fln = Fln^{Fld.Tbl.Tbs}$, and

$Edn \notin \langle Edn^{Edx} \rangle^{Tbl.Tbs}$, and

$createExtIndex^{Rst.Tbs}(Tsn,\ Tbn)$, where

Edn - name of new external index, unique to the table.

Rpc - parameters of replication and partitioning of the index.

$Fln_{0..n}$ - names of the fields in the table that will be used to build the external index.

Tbn -name of the existing table in the tablespace.

Tsn - name of the existing tablespace.

$createExtIndex^{Rst.Tbs}$ - function for checking permission to create the external index with specified name within the table in the tablespace, in the access permissions of that tablespace.

External index deletion

$(Tsn,\ Tbn,\ Edn) \rightarrow Edx \notin \langle Edx \rangle^{Tbl.Tbs} \Rightarrow Edn \notin \langle Edn^{Edx} \rangle^{Tbl.Tbs}$, if

$\exists\, Tbs \in \langle Tbs \rangle, \quad Tsn = Tsn^{Tbs}$, and

$\exists\, Tbl \in \langle Tbl \rangle^{Tbs}, \quad Tbn = Tbn^{Tbl.Tbs}$, and

$\exists\, Edx \in \langle Edx \rangle^{Tbl.Tbs}, \quad Edn = Edn^{Edx.Tbl.Tbs}$, and

$deleteExtIndex^{Rst.Tbs}(Tsn,\ Tbn,\ Edn)$, where

Edn - name of the external index existing in the tablespace.

Tbn - name of the table existing in the tablespace

Tsn - name of the existing tablespace.

$deleteExtIndex^{Rst.Tbs}$ - function for checking permission to delete the external index with the specified name from the table in the tablespace, in the access permissions of that tablespace.

Local index creation

$(Tsn,\ Tbn,\ Fln_{0..n},\ Idn) \rightarrow Idx\ [Idn,\ Fln_{0..n}] \in \langle Idx \rangle^{Tbl.Tbs} \Rightarrow Idn\ \in \langle Idn^{Idx} \rangle^{Tbl.Tbs}$, if

$\exists\, Tbs \in \langle Tbs \rangle,\quad Tsn\ =\ Tsn^{Tbs}$, and

$\exists\, Tbl\ \in \langle Tbl \rangle^{Tbs},\quad Tbn = Tbn^{Tbl.Tbs}$, and

$Fln_{0..n} \in \langle Fln^{Fld} \rangle^{Tbl.Tbs},\quad Fln = Fln^{Fld.Tbl.Tbs}$, and

$Idn\ \notin \langle Idn^{Idx} \rangle^{Tbl.Tbs}$, and

$createIntIndex^{Rst.Tbs}(Tsn,\ Tbn,\ Idn)$, where

Idn - name of the local index, unique to the table

$Fln_{0..n}$ - names of the fields in the table that will be used to build the internal index.

Tbn - name of the table existing in the tablespace

Tsn - name of the existing tablespace

$createIntIndex^{Rst.Tbs}$ - function for checking permission to create the internal index with the specified name within the table in the tablespace, in the access permissions of that tablespace.

Local index deletion

$(Tsn,\ Tbn,\ Idn) \rightarrow Idx \notin \langle Idx \rangle^{Tbl.Tbs} \Rightarrow Idn \notin \langle Idn^{Idx} \rangle^{Tbl.Tbs}$, if

$\exists\, Tbs \in \langle Tbs \rangle,\quad Tsn\ =\ Tsn^{Tbs}$, and

$\exists\, Tbl \in \langle Tbl \rangle^{Tbs},\quad Tbn\ =\ Tbn^{Tbl.Tbs}$, and

$\exists\, Idx \in \langle Idx \rangle^{Tbl.Tbs},\quad Idn = Idn^{Idx.Tbl.Tbs}$, and

$deleteIntIndex^{Rst.Tbs}(Tsn,\ Tbn,\ Idn)$, where

Idn - is a name of the existing internal index in the table

Tbn - is a name of the existing table in the tablespace

Tsn - is a name of the existing tablespace

$deleteIntIndex^{Rst.Tbs}$ -function for checking permission to delete the internal index with the specified name from the table in the tablespace, in the access permissions of that tablespace.

Managing constraints

Constraints are created by the functions of smart-contracts in the blockchain, and are checked by the nodes when executing queries.

Static constraints creation

$(Tsn,\ Tbn,\ Fln,\ Frc) \rightarrow Frc^{Fld.Tbl.Tbs} = Frc$, if

$\exists\, Tbs \in \langle Tbs \rangle,\quad Tsn\ =\ Tsn^{Tbs}$, and

$\exists\, Tbl \in \langle Tbl \rangle^{Tbs},\quad Tbn\ =\ Tbn^{Tbl.Tbs}$, and

$\exists\, Fld \in \langle Fld \rangle^{Tbl.Tbs},\quad Fln\ =\ Fln^{Fld.Tbl.Tbs}$, and

$createFieldRestriction^{Rst.Tbs}(Tsn,\ Tbn,\ Fln)$, where

Fln - name of the field for which restriction is to be created

Frc - constraint condition matching the field format

Tbn - name of the table existing in the tablespace

Tsn - name of the existing tablespace

Creating dynamic and smart constraints

$(Tsn,\ Tbn,\ Trn,\ Trt,\ Trc) \rightarrow Trs\ [Trn,\ Trt,\ Trc] \in \langle Trs \rangle^{Tbl.Tbs} \Rightarrow$

$\Rightarrow Trn \in \langle Trn^{Trs} \rangle^{Tbl.Tbs}$, if

$\exists Tbs \in \langle Tbs \rangle, \quad Tsn = Tsn^{Tbs}$, and

$\exists Tbl \in \langle Tbl \rangle^{Tbs}, \quad Tbn = Tbn^{Tbl.Tbs}$, and

$Trn \notin \langle Trn^{Trs} \rangle^{Tbl.Tbs}$, and

$createTableRestriction^{Rst.Tbs}(Tsn, Tbn, Trn)$, where

Trn - name of a new field table constraint, unique for the table

Trt - constraint type (dynamic or smart) that specifies the format of the constraint condition string.

Trc - constraint conditions that correspond to the specified format.

Tbn - name of a table existing in the tablespace

Tsn - name of the existing tablespace

Record distribution range

When creating a clustered index for a table, a set of record distribution ranges is generated based on the values of the primary keys in accordance with the replication and indexing settings of the index. A distribution range is a subspace in the one-dimensional space of all possible values of the hash function of the selected as the primary key field values. The replication parameters affect the number of ranges in such a way that the value of the partitioning parameter specifies the number of non-intersecting subspaces covering the whole space of the hash values of the function, and the replication parameter specifies the number of replicas (copies) of each range in the system. Thus, the set of distribution ranges is a two-dimensional space of subspaces of possible values of the hash function. This space is determined by the level of replication and the level of partitioning of the table. Alternatively, it can be represented by a three-dimensional space of hash function values equivalent to its projection onto the one-dimensional space of all possible values of hash functions. That is, all values of the hash function in this three-dimensional space belong to the one-dimensional space of all its possible values and are bounded by this one-dimensional space.

Distribution ranges for a single replication level

With respect to one replica, there is a set of distribution ranges, the number of which is equal to the level of the partitioning of the clustered index of the table. For each distribution range, there is a set of hash function values included in it (equal to the number of possible hash values from the values of the cluster table index fields divided by the partitioning level of this index) from the space of all possible hash function values from the cluster table index fields. Thus, any range includes at least one value, and each range of a single replica includes values that are not included in other ranges of this replica.

So:

$P[Epi,\ k_{0..n}]_{0..p}$, and

$P \neq \varnothing, \quad P \subseteq K \Leftrightarrow \langle k \rangle^{P} \subseteq K$, at the same time

$P_a \cap P_b = \varnothing \ \wedge \ |P_a| \approx |P_b|$, where

$p = Prl^{Rpc.Cdx.Tbl}$,

$n = |K| \div p$,

$0 \leq a \leq p, \quad 0 \leq b \leq p, \quad a \neq b$.

K - the space of f hash functions possible values from the fields of the primary key

Epi - distribution range identifier

P - aggregate of distribution ranges is the space of possible values

$k_{0..n}$ - number of hashes including the range

Prl - level of partitioning the clustered index of the table

Replicas of distribution ranges

The number of replicas of all distribution ranges for one space of possible hash function values from the values of the cluster table index fields is equal to the level of replication of this index. Thus, each replica of one space is equivalent to the

remaining replicas of this space if it includes ranges that include the same hash values of the function as the ranges of other replicas of that space, and the set of values of all ranges of a single replica is the set of all possible values of the hash of the values fields of the clustered index of the table.

$R[Eri,\ P[Epi,\ k_{0..n}]_{0..p}]_{0..r} \coloneqq Rpk[Eri,\ Epi,\ k_{o..n}]_{0..r,\ 0..p} \Leftrightarrow k_z^{Rpk_{x,y}} \coloneqq k_z^{P_y.R_x}$, with

$R_a \simeq R_b \Leftrightarrow \langle P \rangle^{R_a} \simeq \langle P \rangle^{R_b} \coloneqq \langle k^P \rangle^{R_a} \equiv \langle k^P \rangle^{R_b} \equiv K$, where

$r = Rpl^{Rpc.Cdx.Tbl}$,

$p = Prl^{Rpc.Cdx.Tbl}$,

$n = |K| \div p$,

$0 \le a \le r, \quad 0 \le b \le r, \quad a \ne b$.

R - range distribution replica.

P - distribution range in a space of potential values.

$k_{0..n}$ - hash-function values, included in the range.

K - space of possible hash-function values.

$Rpk_{0..r,\ 0..p}$ - space of hash function ranges replicas.

Eri - identifier of distribution range replica.

Rpl - table's clustered index replication level.

Prl - table's clustered index partitioning level.

Creating a distribution range

A distribution range is created together with a clustered index.

$(Tsn,\ Tbn,\ Rpl,\ Prl, ...) \rightarrow Cdx\ [Rpc[Rpk[k_{0..n},\ ...]_{0..r,\ 0..p},\ ...],\ ...]$, if

$\exists\, Tbs \in \langle Tbs \rangle, \quad Tsn = Tsn^{Tbs}$, and

$\exists\, Tbl \in \langle Tbl \rangle^{Tbs}, \quad Tbn = Tbn^{Tbl.Tbs}$, where

$p = Prl^{Rpc}$,

$r = Rpl^{Rpc}$,

$n = |K| \div p$.

After creating the space of distribution ranges, they are assigned to the nodes for maintenance. In this case, one node can not serve different ranges of one replica and the same range of different replicas. Also, several nodes can not serve the same range of the same replica.

$\langle Rpk^{Tbl.Tbs} \rangle^{\langle N \rangle} \subseteq \langle Rpk^{Rpc.Tbl.Tbs} \rangle$, while

$\langle Rpk^{Tbl.Tbs} \rangle^{N_x} \not\subset \langle Rpk^{Tbl.Tbs} \rangle^{\langle N \mid \neg N_x \rangle}$, and

$\forall Rpk_{i,j}^{Tbl.Tbs} \in \langle Rpk^{Tbl.Tbs} \rangle^{N_x}, \quad \langle Rpk_{0..r,j} \mid \neg Rpk_{i,j} \rangle^{Tbl.Tbs} \not\subset \langle Rpk^{Tbl.Tbs} \rangle^{N_x} \wedge$

$\wedge \langle Rpk_{i,\,0..p} \mid \neg Rpk_{i,j} \rangle^{Tbl.Tbs} \not\subset \langle Rpk^{Tbl.Tbs} \rangle^{N_x}$, where

$N_{0..n} \coloneqq \langle N \rangle$,

$0 \le x \le n$,

$0 \le i \le r, \quad 0 \le j \le r$.

N - a node registered within the system

If we represent the space of distribution ranges $Rpk_{0..r,\,0..p}$ in the form of a matrix A of dimension $r \times p$, then after assigning the range $a_{ij} \coloneqq Rpk_{i,j}$ to the node, for assignment remains available the matrix of ranges, obtained by excluding the i row and the j column from the matrix A , and excluding elements assigned to other nodes.

For example (for simplicity, we take a hash function with $|K| = 100$):

1. A clustered index of the table with replication level 3 and level of partitioning 5 is created. Hence the number of distribution ranges will be $5 \cdot 3 = 15$, and the number of possible values in one range is $100 \div 5 = 20$.
2. After the index is created, ranges are assigned with nodes serving them. Thus, you need at least 5 nodes to assign for all ranges. The table shows the

distribution of the ranges of 5 nodes A, B, C, D and E, in accordance with the distribution rules for the node. Thus, the distribution of ranges by nodes is as follows:

	r1	r2	r3
p1	A	B	C
p2	D	A	B
p3	E	D	A
p4	C	E	D
p5	B	C	E

node A receives the ranges $\{Rpk_{r1,\,p1}, Rpk_{r2,\,p2}, Rpk_{r3,\,p3}\}$,
node B receives the ranges $\{Rpk_{r1,\,p5}, Rpk_{r2,\,p1}, Rpk_{r3,\,p2}\}$,
node C receives the ranges $\{Rpk_{r1,\,p4}, Rpk_{r2,\,p5}, Rpk_{r3,\,p1}\}$,
node D receives the ranges $\{Rpk_{r1,\,p2}, Rpk_{r2,\,p3}, Rpk_{r3,\,p4}\}$,
node E receives the ranges $\{Rpk_{r1,\,p3}, Rpk_{r2,\,p4}, Rpk_{r3,\,p5}\}$.

3. Thus, each node does not serve all the space of possible values of hash function from the primary key, but only $3/5$ of all possible values, in this case $20 \cdot 3 = 60$.

Distribution of records

After allocating ranges while working with the system, users can create records in the tables. Creating a record requires the mandatory presence of the values of the primary key fields. When the record is transferred to the coordinator for processing from the values of the primary key fields, the hash function (which was used to construct the ranges) is calculated, by which the nodes responsible for processing this record are determined, and the record is transmitted for processing according to the selected level of request consistency.

- Nodes serving a record are identified using record's primary key as follows:

  $Ns_{0..m} = \langle Ns \mid Ns \in \langle N \rangle \wedge h(Pk^{Hdr.Ent}) \in \langle k^{Rpk.Tbl.Tbs} \rangle^{Ns} \rangle$, if

  $\exists\, Tbs \in \langle Tbs \rangle, \quad Tsn^{Hdr.Ent} = Tsn^{Tbs}$, and

  $\exists\, Tbl \in \langle Tbl \rangle^{Tbs}, \quad Tbn^{Hdr.Ent} = Tbn^{Tbl.Tbs}$, where

  N - a node, registered in the system

  $Pk^{Hdr.Ent}$ - field's primary key stored in its header

  h - function's hash used when record was created

  Thus, a record is serviced by the nodes to which the distribution ranges to which the record belongs are assigned, and ranges of which include the hash value of the function from the primary key in the header of this record.

- Determining the nodes without restriction on a primary key, for example for records search, is performed by replicas identifiers:

  $Ns_{0..m} = \langle Ns \mid Ns \in \langle N \rangle \wedge Rpk^{Tbl.Tbs}_{\langle I \rangle,0..p} \subseteq \langle Rpk^{Tbl.Tbs} \rangle^{Ns} \rangle$, where

  $\langle I \rangle \coloneqq i_{0..n} \Rightarrow |\langle I \rangle| = n, \quad 0 \le i \le Rpl^{Tbl.Tbs}, \quad 0 \le n \le Cl.$

  $Rpl^{Tbl.Tbs}$ - table replication level

  I - a set of replicas identificators depending on search consistency level

  Cl - search consistency level

Also:

N - nodes registered in the system.

$Ns_{0..m}$ - nodes maintaining the records.

Rpk - distribution range for the records of the table.

Records management

To save or modify data in the system, the user creates records and sends them to the coordinator. The coordinator in its turn directs the records to the nodes responsible for their maintenance, getting the reward from the nodes, and returns the response to the user.

1. The client generates an unsigned header for a specific table structure:

   $(Tsn,\ Tbn,\ V,\ Et) \rightarrow Hdr\,[Tsn,\ Tbn,\ V,\ Et,\ Tim,\ ...]$, if

   $\exists\, Tbs \in \langle Tbs \rangle, \quad Tsn^{Hdr.Ent} = Tsn^{Tbs}$, and

   $\exists\, Tbl \in \langle Tbl \rangle^{Tbs}, \quad Tbn^{Hdr.Ent} = Tbn^{Tbl.Tbs}$, and

   $createEntry^{Rst.Tbs}(Tsn,\ Tbn)$, where

   Tbn - name of the existing table that should contain new record.

   Tsn - name of existing tablespace containing the table.

   V - record version.

   Et - record type: new, modified, deleted record.

   Tim - record creation timestamp.

   $createEntry^{Rst.Tbs}$ - function to check record creation permissions for the table with specified name in the tablespace, in access permissions for this tablespace.

2. The user provides a set of table field values to the client, which forms the fields of the new record:
   $(Rv_{0..n}) \rightarrow Fld\ [Fln(Rv),\ Flv(Rv)]_{0..n}$, where
   $Rv_{0..n}$ - original values to determine field names and form field values.
   Fld - fields of the record built from source data.
3. The client forms the primary key from the generated record fields:
   $\ldots \rightarrow Pk^{Hdr} = Pk[Flv_{0..k} = \langle Flv^{Fld} \mid Fln^{Fld} \in \langle Fln \rangle^{Cdx.Tbl.Tbs} \rangle]$.
4. The client determines the set of nodes on which the record will be stored and forms cheques of payment for recording and storing data for these nodes:
   $\ldots \rightarrow Chq^{Ent.Tbl.Tbs} = Chq\ [Sig(Hash^{Chq}),\ Rid,\ Num,\ Nid = Nid_{0..x}^{Ns},\ Amt,\ \ldots]_{0..m}$,
   where
   $Ns_{0..x} = \langle Ns \mid Ns \in \langle N \rangle \wedge h(Pk^{Hdr.Ent}) \in \langle k^{Rpk.Tbl.Tbs} \rangle^{Ns} \rangle$.
   N - nodes registered in the system.
   $Ns_{0..x}$ - nodes maintaining records distribution range which contains hash of the record primary key.
   Rid - identifier of cheque range for the current cheque.
   Num - number of the cheque for this range.
   h - hash function used to build records distribution ranges.
   $Rpk^{Tbl.Tbs}$ - distribution range of the table records.
5. The client forms a signed record:
   $\ldots \rightarrow Ent\ [Sig(Hash^{Hdr.Ent}),\ Hdr[Hen = Hash^{Ent}, \ldots],\ Fld_{0..n},\ Chq_{0..m}]$, where
   Ent - new record signed by the user.

Search conditions

The search conditions are formed from a search string in a specific format, a cheque for a search and a signature. The matching of the string to the search format is performed directly on the nodes. In case of an error in the search condition, the node receives payment as if it had performed a search query.

1. The client forms a set of cheques to pay for the search on the nodes (in this case without using a non-clustered index):
   $(Tsn,\ Tbn,\ Cl,\ Qt,\ S) \rightarrow Chq\ [Sig(Hash^{Chq}), Rid, Num,\ Nid_{0..m} = Nid_{0..m}^{Ns}, Amt]$, if
   $\exists\, Tbs \in \langle Tbs \rangle,\quad Tsn^{Hdr.Ent} = Tsn^{Tbs}$, and

$\exists\, Tbl \in \langle Tbl \rangle^{Tbs}, \quad Tbn^{Hdr.Ent} = Tbn^{Tbl.Tbs}$, where

$Ns_{0..m} = \langle Ns \mid Ns \in \langle N \rangle \wedge Rpk^{Tbl.Tbs}_{\langle I \rangle, 0..p} \subseteq \langle Rpk^{Tbl.Tbs} \rangle^{Ns} \rangle$,

$\langle I \rangle \coloneqq i_{0..n} \Rightarrow |\langle I \rangle| = n, \quad 0 \le i \le Rpl^{Tbl.Tbs}, \quad 0 \le n \le Cl$,

$Rpl^{Tbl.Tbs}$ - table replication level.

I - set of replicas identificators depending on search consistency level.

Cl - search consistency level.

Tsn - name of the tablespace to which a table belongs to.

Tbn - name of a table in tablespace.

2. The client forms the signed search conditions:

   $\ldots \rightarrow Qry\ [Sig(Hash^{Qry}),\ Tsn,\ Tbn,\ Qt,\ S,\ Chq]$, where

   Qt - search conditions type with a format chosen in the searchline

   S - search line matching search conditions

Forming a request

Request is generated by the client from the record or search conditions and the consistency level. In this case, the request consistency level may be less than the search conditions consistency level. If the request consistency is less than the consistency of the search conditions, the request is distributed to all the nodes that are specified in the search cheque, and the response is returned from the number of nodes computed by the request consistency level that returned the query results faster than the others. This allows you to increase the speed of access to records by increasing the price of requests, and increase the probability of healing. If the request consistency level is greater than the search condition consistency level, then the level with a lower consistency will be used to respond to the user. In case, if the client himself acts as a coordinator, then the formation of a signature in the requests is optional. Let's consider a more general case when the request signature is required:

$(Cl,\ Qry \parallel Ent) \rightarrow$

$\rightarrow Rm[Sig(Cl^{Rm},\ Hash^{Hdr.Ent.Rm}),\ Cl,\ Ent] \parallel Rs[Sig(Cl^{Rs},\ Hash^{Qry.Rs}),\ Cl,\ Qry]$, where

Rm - data modification request

Rs - data access request

Cl - request consistency level

Node responses

Depending on the type of request, the node generates a response that confirms the processing of the request by this node.

Node response to a data modification request

The response to the modification request includes a signed header's hash, which the node successfully saved. The answer serves as a confirmation to the user of the successful saving of the record, as well as the reason for the healing in case of race of competitive records.

$(Rm[Ent, ...]) \rightarrow Ace\ [Sig(H^{Ace}),\ Hen = Hash^{Hdr.Ent},\ Chq]$, where

Rm - data modification request.

Sig - record's header's hash signature by the node that accepted the data.

Hen - record's header's hash that was accepted.

Chq - cheque issued by a node to the coordinator who received the record.

Ent - the record to be saved.

Node response to a search request

The response to a search request consists of a set of records found on the site that meet the search conditions. Each entry has a signed header hash (as in the case of a response to the data modification request), the record itself with the fields specified in the search conditions and the hashes of values of the missing fields to provide the the modification of the record without additional requests.

$(Rs[Qry,\ ...]) \rightarrow Ent\ [Fld_{0..n},\ ...]_{0..p} \rightarrow$

$\rightarrow Enr\ [\#Ent,\ Sig^{Ent.Enr},\ Hdr = Hdr^{Ent.Enr},\ Fld_{0..m} = Fld^{Ent.Enr}_{0..m},\ Fls_{0..l}]_{0..p} \rightarrow Srs\ [Enr_{0..p}]$

, where

$Fls^{Enr}_{0..l} = \langle Fls(F) \mid F \in Fld^{Ent}_{0..n} \wedge F \notin Fld_{0..m} \rangle^{Enr}$,

$Fls(Fld) \rightarrow Fls\,[Fln^{Fld},\ Flh = Hash^{Flv.Fld}]$.

Rs - search request.

Srs - node response to a search request.

Qry - search parameters.

$Ent_{0..p}$ - found records matching to the search parameters.

$Fld_{0..n}$ - all fields of the found record matching to the search parameters.

$Fld_{0..m}$ - field that have to be returned in accordance with the search parameters.

$Fls_{0..l}$ - supplementary-fields , matching the record fields, that does exists in the real record, but are not requested to be in the response.

When receiving responses from several nodes, the coordinator groups them by the primary key and compares the versions of the records. If several versions of the record are found (write inconsistency), an attempt to heal the system is made. The response to the user is returned only when the level of consistency specified in the request is obtained. In a fully consistent system state, the number and version of records on all nodes of the same replica are equal.

Cheques issue

The general principle of issuing cheques as part of the operations of creating a record and a search queries was considered above. And as mentioned, cheques are issued for payment processing, storage and access to user records. A cheque can be issued to one or several nodes simultaneously. Each issued cheque has a number, within the range of numbers unique to the user, and the identifier of this range. If the node encounters a cheque in the request, with the same number that was already redeemed or expects repayment, it will refuse to process such a request, returning an error. In this case, the range of cheque numbering is considered unreliable, and the client creates a new range to continue operating.

$(Rng,\ Ns_{0..x}) \rightarrow Chq\ [Sig(Hash^{Chq}),\ Tim,\ Rid\ =\ Id^{Rng},\ Num(Rng),\ Nid^{Ns}_{0..x},\ Amt]_{0..m}$,

where

$Num(Rng) \rightarrow Num\ =\ Lcn^{Rng},\ Rng'[Lcn = Lcn^{Rng} + 1, ...].$

$Ns_{0..x}$ - nodes that client should issue cheques for.

Rng - cheque range in which the cheque was issued.

Num - cheque number within the numbering range.

Lcn - number of the latest issued cheque in the range.

Tim - timestamp, synchronized with the last block in the blockchain.

Generating a numbering range is quite trivial. To do so, client uses the generation function of the UUID, which becomes the identifier of the range, and issues checks numbering them from one.

$(Uuid) \rightarrow Rng\ [Id\ =\ Uuid,\ Lcn\ =\ 1,\ ...]$.

If there is no cheque with a cumulative total for the user at the node, the user can create it. For this, he appeals to a smart-contract, receiving from there the amount of the last cashed cumulative cheque for this node and the timestamp of the latest cheque redeemed.

Repayment of cheques

Repayment of cheques redeems them including their amount into the sum of the cumulative cheque for subsequent cashing. To avoid the inclusion of the same cheques into the cumulative cheque multiple times, a quasi-full binary Merkle tree is constructed, based on the range of checks. The basis for checking the inclusion of a check in the final check is the Merkle proof. From the structure it is clear that to check the inclusion of a cheque, it is necessary to store only the ranges of cheques and the root of the Merkle tree, the remaining elements can be calculated from them. If a cheque number collision is found (the node has found a cheque that has already been included in the range, or the node has already received a cheque with the same number), the node returns an error, after which the range of cheque numbering is considered unreliable and the client generates a new one.

The structure of Merkle tree of the cumulative cheque

Each branch contains information about the minimum and maximum cheque numbering range identifier that can be included in it, and the root always includes the entire range of identifier values. The left (0) and right (1) elements of the tree are always sorted by increasing the included identifiers from the minimum to the maximum. Ranges of identifiers of tree elements on one level are never overlap.

$$Mct\,[Mcr(E(Mcb^{Mct})_{\{0,1\}}), \#Mcb(E(Mcb^{Mct} \parallel Rng)_{\{0,1\}})_{0..n}],$$

$$Mcr\,[Mhs, \#E_{\{0,1\}}], \quad Mcb\,[Lid,\ Hid,\ Mhs, \#E_{\{0,1\}}],$$

$$E\,[Lid,\ Hid,\ Mhs], \text{ where}$$

$$E(Mcb) = E[Lid = Lid^{Mcb},\ Hid = Hid^{Mcb},\ Mhs = Mhs^{Mcb}],$$

$$E(Rng) = E[Lid = Id^{Rng},\ Hid = Id^{Rng},\ Mhs = Hash^{Rng}],$$

$$Mcr(E_{\{0,1\}}) = Mcr[Mhs = h(Mhs^{E_0},\ Mhs^{E_1}),\ \#E_{\{0,1\}}],$$

$$Mcb(E_{\{0,1\}}) = Mcb\,[Lid = Lid^{E_0},\ Hid = Hid^{E_1},\ Mhs = h(Mhs^{E_0},\ Mhs^{E_1}),\ \#E_{\{0,1\}}],$$

$$\forall E, \quad Lid^{E} < Hid^{E},$$

$$\forall E_{\{0,1\}}, \quad Hid^{E_0} < Lid^{E_1},$$

$$\forall Mcb, \quad \forall Mcr, \quad (\langle x \mid x \in U \wedge Lid^{E_0} \le x \le Hid^{E_0} \rangle \trianglelefteq \langle y \mid y \in U \wedge Lid^{E_1} \le y \le Hid^{E_1} \rangle)^{Mcb \| Mcr}$$

,

U - multitude of all possible cheque ranges identifiers

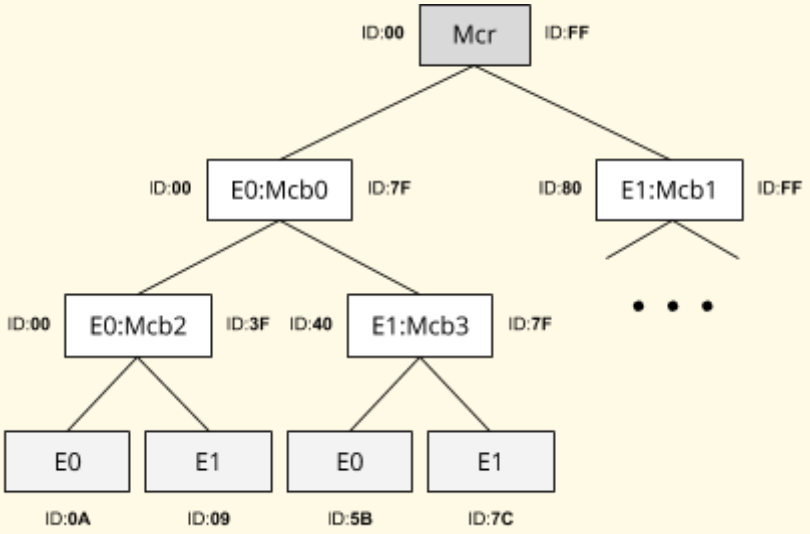
E - Merkle tree element

Mcr - Merkle tree root

Mcb - the tree's branches

Mcl - Merkle tree leafs that are data hashes

Merkle tree depiction



Building a tree

As already mentioned, the Merkle binary tree is quasi-complete, i.e. all its branches are already partitioned in ranges so that each branch from the root includes half the range of the parent. In this case, the hash of the branch, which includes only one leaf hierarchically, is equal to the hash of this leaf. When you include a tree or change an already included leaf, not all the tree is need to be recalculated, but only the hashes of the branches in the hierarchy of which this leaf is located (from the leaf to the root). In this case, a tree leaf is built from a cheque numbering range.

Confirmation of range inclusion

The general principle of verification is that the node sends the user a cheque with a cumulative total, cheques for storage, access and modification of data, their corresponding numbering ranges (if these ranges have already been included in the cumulative cheque) and the branches required for calculation of the tree root (closest to the root). The client checks the compliance of the cheques of storage, access and modification of the data relating to them, marks them in the ranges as redeemed (the hash of the range will be recalculated), including their sums in the total amount of the cheque with the cumulative total and updating the timestamp of the latest paid cheque if necessary. Then recalculates part of the Merkle tree according to the new values of hash ranges, using the attached missing hashes of the branches, replacing the root. And re-signs the changed ranges and the

cumulative cheque. After that, the client returns to the node a new cheque with a cumulative total and the changed ranges whose hashes were included in the tree. In the absence of a cheque with a cumulative total, or when cheques are passed, whose ranges have not yet been included in the tree, the client forms, signs and includes them in the tree with their range ID within the newly formed cheque numbering range with this ID, for the node.

$(Mct,\ \#Mcb_{0..x}^{Mct},\ Rng_{0..y},\ Chq_{0..z}) \rightarrow Mct'[Mcr,\ \#Mcb_{0..n}]$, with

$Mcr^{Mct'} = Mcr(E(Mcb^{Mct} \parallel Mcb^{Mct'} \parallel Rng'(Rng,\ Chq_{...}))_{\{0,1\}})$,

$Mcb_{0..n}^{Mct'} = Mcb(E(Mcb^{Mct} \parallel Mcb^{Mct'} \parallel Rng'(Rng,\ Chq_{...}))_{\{0,1\}})_{0..n}$,

$Mcb_{0..x}^{Mct} = \langle Mcb \mid \langle i \mid i \in U \wedge Lid^{E_0} \le i \le Hid^{E_0} \rangle^{Mcb} \not\subseteq \langle Id \rangle^{Rng_{0..y}} \rangle^{Mct}$,

$Mcb_j^{Mct'} = Mcb_k^{Mct} \Leftrightarrow Lid^{Mcb_j^{Mct'}} = Lid^{Mcb_k^{Mct}} \wedge Hid^{Mcb_j^{Mct'}} = Hid^{Mcb_k^{Mct}}$,

$Rng'(Rng,\ Chq_{0..s}) = Rng'[\#Hash,\ Sig(Hash),\ Pay_{...}, ...]$, where

$Chq_{0..s} = \langle c \mid c \in Chq_{0..z} \wedge Rid^c = Id^{Rng} \rangle$,

$\langle p | p \in \langle Num^{Chq} \rangle \wedge Lcq^{Pay} \le p \le Hcq^{Pay} \rangle^{Rng} \cup \langle Num \rangle^{Chq_{0..s}} \subseteq$

$\subseteq \langle p' | p' \in \langle Num^{Chq} \rangle \wedge Lcq^{Pay} \le p' \le Hcq^{Pay} \rangle^{Rng'}$, where

Mct - initial merkel tree stored is the final cheque.

$Mcb_{0..x}^{Mct}$ - missing branches of the original Merkle tree, necessary to build a new tree.

$Rng_{0..y}$ - the ranges to which cheques have to be included.

$Chq_{0..z}$ - cheques to be included in ranges.

Mct' - a new Merkle tree, built of new ranges and missing branches.

Rng' - new ranges, in which the checks related to them were included.

Receiving rewards

To receive a reward, the smart-contract function is used. The function accepts a cheque with a cumulative total and make transfer to the account of the node, from the account of the user who signed the cumulative cheque, the amount of funds stored in this cheque, minus the amount of all previously transferred funds for this cheque. Also, the maximum timestamp of the cheque stored in the cumulative cheque is saved in blockchain, this is necessary to restore the final cheque in case of its loss.

System Features

As mentioned above, Oasis.DDB is a non-relational, distributed, decentralized database management system that uses **BASE** semantics (**B**asically **A**vailable, **S**oft state, **E**ventual consistency). It is necessary to divide the concepts of distribution and decentralization. Distributed systems can have centralized control center, which manages the operation of the system. Oasis.DDB is a distributed system that does not have a single administration center, and all management settings are dynamically generated and coordinated with the participants of the system.

Also in Oasis.DDB there are no fixed parameters for **CAP** (**C**onsistency, **A**vailability, **P**artition tolerance). Each client chooses the priority of one of the parameters as necessary in the access requests and data changes. In this case, the replication settings of the table affects the availability parameter, and the balance of the specified level of consistency in data access requests and data changes controls consistency and partition tolerance.

Data access restrictions

All records in the Oasis.DDB system are readable. This is dictated by the need of nodes to access records for indexing and for automatic distribution of data in the system. To restrict the reading of critical information, it is recommended to store it in an encrypted form. Unfortunately, the search capabilities will be limited. To solve this problem, it is possible to determine the keywords that will be used for searching, and store them close (in the same record) to encrypted data. It is important to understand that this can partially or completely compromise the encrypted information, and special attention should be paid to the way the

keywords are selected. The keywords stored in the record can be replaced by their hashes, thereby improving the information disguise. However, the search for hashes of keywords will be sensitive to the register and word forms. In addition, over time it is possible to collect a dictionary of matching hashes to the original keywords, thereby compromising them.