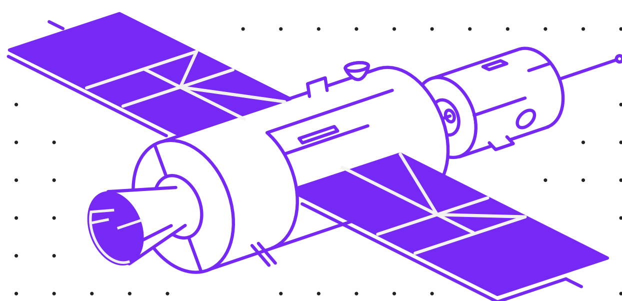
CLUSTERPOINT

TRANSACTION MECHANISM IN CLUSTERPOINT DATABASE



Zigmars Rasscevskis, Clusterpoint
Janis Sermulins, Clusterpoint
Prof. Girts Karnitis, University of Latvia

Copyrights © Clusterpoint Ltd.
info@clusterpoint.com
September 2015

Contents

BASIC CONCEPTS 4
Server software 4
Database architecture 5
DOCUMENTS 6
Document identification 6
Document insert process 7
Shard replica version identification 8
Document insert 10
Document delete 10
Document update 10
SHARD CONSISTENCY 11
Node Reconciliation 11
Back log 11
TRANSACTIONS 14
Transaction log versions 14
Transaction identification 15
Document and node versions used in transaction 16
Transaction workflow 17
SELECT in transaction 20
INSERT in transaction 21
DELETE in transaction 21
UPDATE in transaction 21
Locking document for update or delete during transaction 22
Race condition between transactions 22
Transaction interaction with blind operations 25
Transaction commit 26
Transaction log propagation 26
Transaction rollback 27
MULTI-DOCUMENT SELECT CONSISTENCY 27
ADDITIONAL INFORMATION 27

Baseline Transaction log version number	Transaction log version number assigned to transaction by hub during BEGIN_TRANS operation.
Blind operations	Operations (SELECT, INSERT, UPDATE, DELETE) performed without transactions. Each operation is performed independently and results are only eventually consistent Clusterpoint provides this mode of operation to manage trade-offs between consistency and performance.
Client	User process interacting with Clusterpoint.
Cluster	Set of physical computers or virtual machines containing Clusterpoint software and connected into single network.
Cluster manager	Process that manages information about configuration. Usually cluster has more than one manager.
Database	Sharded collection of documents. There are two types of databases here: client document database and transaction log database.
Document	XML or JSON document.
Document PK	Unique (within database) invariable document ID, given by client or Clusterpoint. Document PK remains the same for all document versions.
Document version	Different versions of a document with the same PK. Initially each document has one version. Document has different version after document update. Each document version is identified by a unique Operation ID and a document version number.
Document version number	Unique (within a database) number, assigned by node to a specific document version.
Document version copies	Equal copies of the same document version, stored in different shard replicas.
Entry Point hub	Hub process that client can connect to. Usually there is more than one entry point hub in the cluster.
Hub	Process responsible for data request orchestration between nodes. Usually there is more than one hub in the cluster. Each hub has a unique ID.
ID	Short name for "identifier".

Node	Process responsible for database data manipulation and querying (INSERT, UPDATE, DELETE, SELECT) in a non-volatile storage on a single computer. Usually there is more than one node in the cluster. Each node has a unique ID.
Node reconciliation	Process during which hub sends to node documents that are missing in shard's stored in that node.
Operation	Common term for data manipulation (INSERT, UPDATE, DELETE, SELECT, BEGIN_TRANS, COMMIT_ TRANS) irrespectively of its type.
Operation ID	Unique (within cluster) number assigned to each operation by entry point hub that gets a request for document or transaction manipulation. Operation ID consists of operation timestamp and entry point hub ID.
PK	Short name for primary key, invariable ID.
Shard	Logical part of database. Document division to shards is determined by hash function. Shard is a virtual unit; shard replicas exist physically.
Shard replica	Replica is a copy of shard that (theoretically) contains all shard documents. Because of possible errors, shard replica in particular time moment can contain part of shard documents. Shard usually has many replicas. All shard replicas are equally important; shard master replica does not exist.
Shard replica version number	Composite (n dimensional) number that contains information about what document versions are stored in shard replica.
Transaction	Logical unit of work containing one or more operations that all must succeed. If any of them fails, all other operations in transaction also fail and database is reversed to the state before the transaction had begun.
Transaction context	Memory structure to store transaction data, e.g. information about document versions read and wrote by this transaction, transaction ID, etc. Transaction context is stored in hub memory until transaction is committed or rolled back. Each transaction always has one context in one hub.
Transaction ID	Operation ID for BEGIN_TRANS operation for given transaction.
Transaction log database	Database containing transaction log records as documents.

Transaction log record	Document created from transaction context during committing or rolling back transaction. Transaction log record is stored in transaction log database and contains information if the transaction was committed or rolled back and the list of shards altered and their versions needed to see all of the changes made by the transaction.
Transaction log record version number	Document version number assigned to the Transaction log record.
Transaction log propagation	Process performed by hubs to ensure that each node serving database gets actual information for each node's served replica about all committed transactions that affect this replica.
Transaction log version number	Composite (n dimensional) number that contains information about all shard versions of transaction log database.

Basic Concepts

Clusterpoint is a document-oriented database server platform for storage and processing of structured and unstructured text data in a distributed fashion on large clusters of commodity hardware. Data records (documents) are logically organized in sharded databases, where each individual partition is referred to as a shard (database shard).

Clusterpoint database runs on cluster of a set of physical computers or virtual machines that contain Clusterpoint software and are connected into a single network. In this paper we will talk only about physical computers, but it could also be virtual machines.

Server software

Clusterpoint server software consists of three main types of processes – manager, hub and node. Usually there are two types of computers in the cluster – one type runs only manager processes and the other type runs node and/or hub processes.

Cluster manager is a process that maintains information about Clusterpoint server configuration, such as: computers, hubs, nodes, database shards and assignment of replicas to nodes. Usually there is more than one machine in the cluster running cluster manager processes, but only one of them is the active cluster manager, other cluster managers are shadow cluster managers. All changes in Clusterpoint architecture (hardware and software) configuration are registered via the active cluster manager process. Other processes (hubs, nodes and shadow cluster managers) contact with the active cluster manager to get information about the

actual configuration. Shadow cluster managers store copy of configuration information. In case if the active cluster manager process fails for any reason (for example, hardware crash), one of the shadow cluster managers becomes the active cluster manager. Hubs and nodes have their own copy of cluster configuration. In case if all cluster managers fail, cluster will be able to continue processing data requests, but changing cluster configuration will not be possible until one of the cluster manager processes starts to work and becomes an active cluster manager.

Hub is a process that performs request dispatching and necessary coordination between nodes to fulfil data manipulation operations and response consolidation for queries in a cluster. Entry point hub gets a document operation request from a client, then sends the request for processing to node and returns the result back to the client. See details in chapters *Document insert* and *Document delete*.

Each node has one hub that it is „responsible" for. Hub's other functionality is to ensure regularly that the node it is „responsible" for has the latest version of documents. One hub can be "responsible" for one or many nodes. This "responsibility" is configured into cluster configuration; usually there is a hub process on each computer running node and one-to-one relation between hubs and nodes.

Node is a process that receives operation request and a document from a hub, then determines necessary manipulation type (INSERT, UPDATE, DELETE, SELECT) and performs the particular manipulation with a document on one computer. It also executes SELECT queries – matching, sorting and aggregating over documents stored in the shard hosted by a particular Node. When inserting or updating, node also assigns document version number as explained in chapter *Document identification*. Cluster has variable amount of nodes - each computer in cluster, operating document data, runs at least one node process. Nodes (computers) can be added to and removed from the cluster at any time. Each node is identified by a unique node ID within a cluster.

Database architecture

The whole database of documents is logically organized in parts – shards. Since sharding spreads the database parts across multiple machines in a cluster, Clusterpoint server uses hash function to determine on which shard and nodes any document is stored.

Shard is a logical part of Clusterpoint database and contains subset of documents. Database is divided in N similarly sized shards and each shard is replicated on several nodes, as defined in cluster configuration. Thus, a shard has many copies - replicas. Shard replicas are strictly ordered – for each shard there are replica 1, replica 2, …, replica N. Each node can store and serve replicas for several shards. See example in Figure 1. Node saves each shard data in separate directory on disk. When number of shards and computers used for saving shards increase, data retrieving throughput of

the cluster increases as they are handled by more Nodes, but overhead of communication between shards increases as well. Therefore, optimal number of database shards should be determined and configured.

Note that particular data record, a document, is not split across several shards and is stored in a shard as a whole unit. Document is allocated to a specific shard by entry point hub, based on a hash of a unique invariable ID – document PK, and document manipulation is performed by nodes, storing replicas of that shard. See details in chapter *Document insert*.

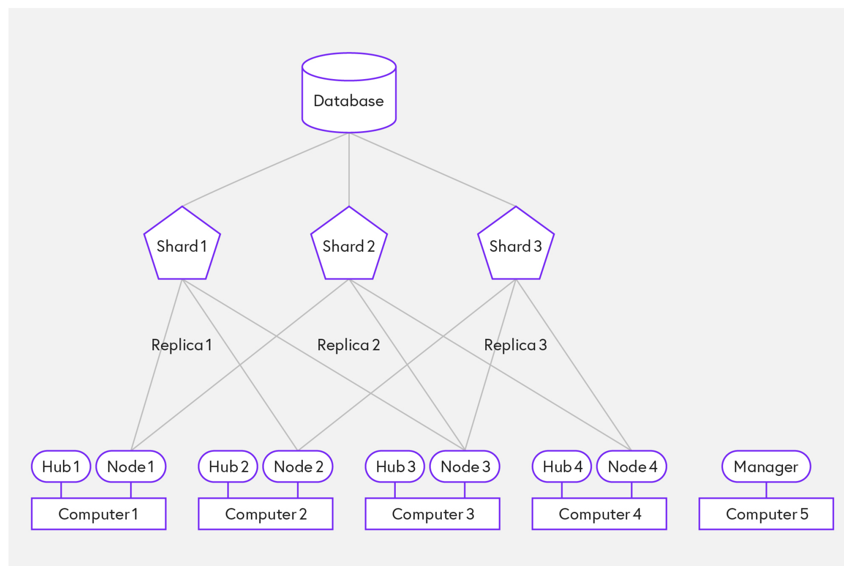


Figure 1

Example in Figure 1 shows Clusterpoint cluster with 5 computers. There are 4 hubs and 4 nodes here, configured one to one, they run on *Computer1* to *Computer4*. Manager process runs on *Computer5*. There is one database consisting of three shards. Each shard has 3 replicas, thus every shard is replicated on 3 different nodes. E.g. *Shard2 Replica1* is on *Node1*, *Shard2 Replica2* is on *Node3, and Shard2 Replica3* is on *Node4*.

Documents

Clusterpoint is a document oriented database, where XML or JSON documents are stored. The document is the main data unit and data is stored and retrieved in a form of documents.

Document identification

Every document in Clusterpoint always has:

1. Document PK: a unique (within database) invariable document ID. Document PK always remains the same for all document versions, independently of operation performed. Document PK is assigned by the client or Clusterpoint.

2. Document version number. Both operations on document - insert and update – always create a new version of a document, thus each document initially has one version, and the version number changes during update manipulations. Document version number is assigned by the node in which this document version is stored at first.

Each document version is identified by a unique Operation ID and unique document version number. Operation ID is assigned by entry point hub.

Document insert process

Let us have a look on the process taking place after a client requests entry point hub to insert a document:

1. Entry point hub calculates hash of document PK to determine shard in which document version must be stored. Because of PK invariability, target shard for a document is always the same. From this point further, we will refer to this particular document as a document version, as each document initially has one version.

2. Entry point hub assigns and appends unique insert operation ID to the document version. Operation ID is composed of a timestamp with millisecond granularity when this operation is issued on entry point hub, sequence number of the operation within this time unit on particular hub and entry point hub's unique ID in the cluster, thus, they three together form a unique sequence.

3. Entry point hub sends the Operation ID, calculated in Step 2, and document version to the node, which contains first replicas, in our case – Replica1 of the shard, in which document version must be stored, as calculated in Step 1. Node and shard location is looked up by hub in the cluster configuration. All shard replicas are strictly numbered and for a specific shard there is the first replica, second replica and so on. First entry point hub tries to connect the first replica; if it fails, then the second replica, etc.

4. The Node receives the Operation ID and document version, and assigns a unique document version number to the document version. The document version number is composed of node's unique ID and node's own unique sequence number for that shard. Each node maintains its own sequence for each shard replica it serves. For example, the first document in Node3, to be stored in Shard5 Replica1, will have version number 3-1_Shard5, the second document will have version number 3-2_Shard5. Further we will shortly refer to document version number as "Node ID – next sequence number for particular shard", thus, 10th document version, stored in a shard's replica on Node3, will have version number 3-10. Operation ID and document version number are integrally appended to the document version. Node records document version in shard on the disk.

5. Node sends back the document version number to the sender hub.

6. Hub sends the document version along with Operation ID and document version number assigned by the first node to all other nodes serving Replica2 – ReplicaN of the particular shard. On other replicas document version is stored with the same document version number that is returned from the first node where the document was stored. Node and shard location are looked up by hub in the cluster configuration. For example, if a document version was first stored in shard replica on Node3 with number 3-1, hub sends this document version to store in shard other replica on Node7, then on Node7 this document version is also stored with version number 3-1.

Shard replica version identification

In order to deliver database consistency, as one of means, the node reconciliation is performed, comparing shard replica version numbers among nodes. Details are described in chapter *Node Reconciliation*.

Shard replica version number is composed of the largest document version numbers, stored in a particular replica. Example in Figure 2 shows how shard replica versions are generated.

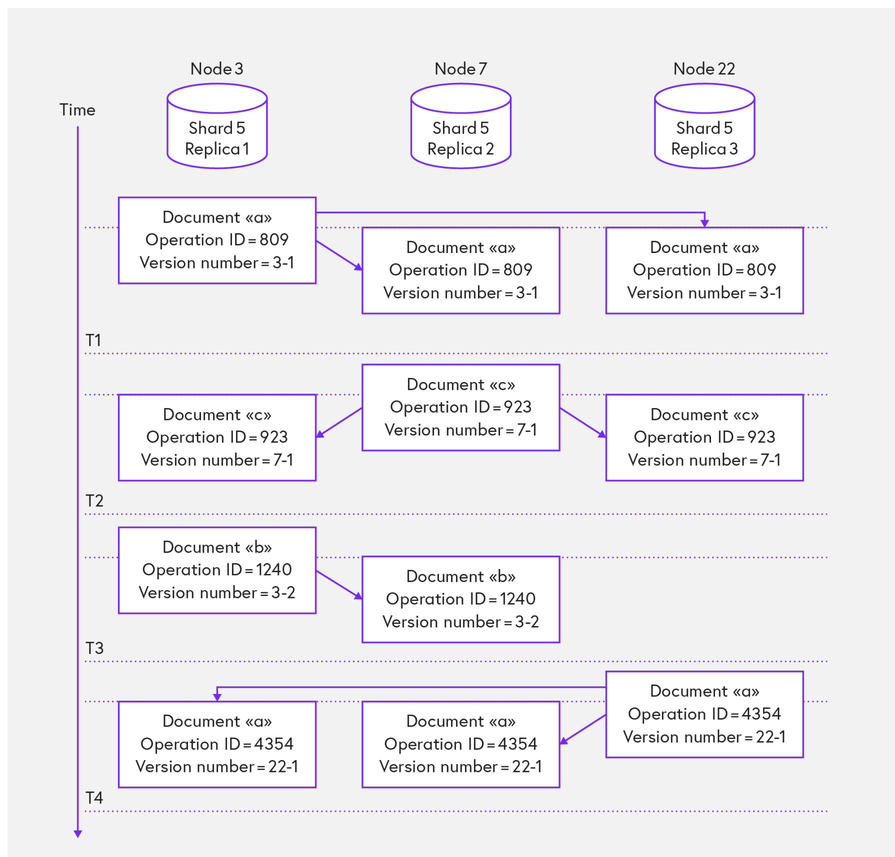


Figure 2

3 nodes (Node3, Node7 and Node22) are shown in this example. There is one shard (Shard5), having 3 replicas that are served by those nodes – one replica per node. Timeline is located at the left side, and time goes down from top to bottom. The column under each node shows document versions, stored on this node in this particular replica of Shard5.

Hub and node operations in the example above explained:

T1. Node3 inserts document "a" with the sender hub's supplied operation ID 809 into Shard5 Replica1 with node's assigned document version number 3-1. Hub replicates this document version to Node7 Shard5 Replica2 and Node22 Shard5 Replica3 with the same data (including document version content, operation ID and document version number).

T2. Node7 inserts document "c" with operation ID 923 into Shard5 Replica2 with document version 7-1 and hub replicates this document version to Node3 Shard5 Replica1 and Node22 Shard5 Replica3 with the same data.

T3. Node3 inserts document "b" with operation ID 1240 into Shard5 Replica1 with document version number 3-2 (next in Node3 own sequence for Shard5), and hub replicates this document version to Node7 Shard5 Replica2 with the same data. Unfortunately, this document version is not replicated to Node22 Shard5 Replica3, because of some network error. The document version will be copied to replicas that missed it during node reconciliation process. See chapter *Node Reconciliation* for more information.

T4. Node22 inserts new version of document "a" (client updates document "a") with operation ID 4354 into Shard5 Replica3 with document version number 22-1, and hub replicates this document version to Node3 Shard5 Replica1 and Node7 Shard5 Replica2 with the same data.

Shard replica version number is composed of the largest version numbers of documents stored in a particular replica. In our example, after data manipulation operation with Op ID = 1240, Node3 Shard5 Replica1 has document from Node3 with largest document version number 3-2 and has a document from Node7 from previous operations (Op ID = 923) with the largest document version number 7-1, therefore, Node3 Shard5 shard Replica1 version is 3-2.7-1. The same version number 3-2.7-1 is for Shard5 Replica 2 on Node7, as this node. But, due to network error in T3, Shard5 Replica3 on Node22 has the document from Node3 with the largest document version number 3-1 and the document from Node7 with the largest document version number 7-1, therefore, Node22 Shard5 Replica3 version number is 3-1.7-1. As seen in this example, it is possible that shard replica versions are different at the same time.

In the example in Picture 1 operations generated the following sequence of shard replicas versions. To sum up, let us have a look at the version number table after a particular operation.

Operation ID	Version numbers		
	Node3 Shard5 Replica1	**Node7 Shard5 Replica2**	**Node22 Shard5 Replica3**
809	3-1	3-1	3-1
923	3-1.7-1	3-1.7-1	3-1.7-1
1240	3-2.7-1	3-2.7-1	3-1.7-1
4354	3-2.7-1.22-1	3-2.7-1.22-1	3-1.7-1.22-1

Document insert

INSERT operations always create a new version of a document. See examples in Figure 2 time T1, T2, T3. For more information see chapter *Document insert process*.

Document delete

During DELETE latest document version is immediately marked as "deleted". Document version that is marked as deleted is no longer readable by subsequent SELECT operations. This algorithm is true for operations out of Transaction; DELETE within transaction is described in chapter *Delete in transaction*.

Deleted versions are physically deleted from persistent storage, when database initiates compaction after configurable percentage or data has been deleted. This threshold manages trade-off between IO necessary for compaction and storage space

Document update

UPDATE operations always consist of INSERT and DELETE operations. The existing document version is never updated. Instead of this, UPDATE always creates a new version of a document and the previous document version is marked as "deleted" and after some time is physically deleted.

Let us describe the process with an example, based on Figure 2 after time T4, when client submits changes to the existing document "a":

1. The client contacts entry point hub with a request to UPDATE a document. Hub generates a unique operation ID, based on timestamp and hub unique ID; let us assume op ID = 10286.
2. Entry point hub determines node and contacts it as it is described step by step in chapter *Document insert process*. The node marks the old document version as deleted, inserts a new document version and sends back the new document version number. Consider the following example (Figure 2):
   At the end of T4 shard replica version number is 3-2.7-1.22-1 after a new

version of existing document "a" sent to shard at op = 4354. Later entry point hub contacts Node3 with a new version for the document "a" and op ID = 10286. Node3 replies that a new version of "a" has been created and the document version number assigned is 3-3.

Entry point hub contacts other nodes with shard replica to update the document "a" version to 3-3. This step is necessary to bring all cluster nodes to consistent state. If some nodes fail to be contacted, they obtain consistency later, when node reconciliation is performed. See details in chapter *Node Reconciliation*.

Shard consistency

It can happen in several scenarios that replicas of the same shard on different nodes are not equal. For example, if one node was down while others were accepting document updates, this node does not have neither information about the latest document versions, nor the document version content. To handle this, a process called node reconciliation is regularly performed; chapter *Node Reconciliation* describes node reconciliation; chapter *Back log* describes how node deals with out of order document versions.

Node Reconciliation

To prevent that some replicas do not have all document versions, hubs, "responsible" for nodes (More about hubs "responsibility" see in chapter *Server software* in section about Hub), constantly poll other nodes containing replicas of the same shards to compare version numbers. If shard replica version number stored on "responsible" for node is lower than its peers, hub requests peers for latest versions of the documents and updates them locally.

Back log

Sometimes it happens that node's shard's replica receives a document which version does not correspond to the correct sequence order on that node's shard's replica. Such documents are stored in node shard's replica's back-log until the correct sequence is restored. In further example we will have one shard with two replicas; one replica on Node3 and the other replica on Node7. In this example shard replicas will be called Node3 and Node7. Documents stored in shard replica on Node3 at different time moments are shown in Figure 3.

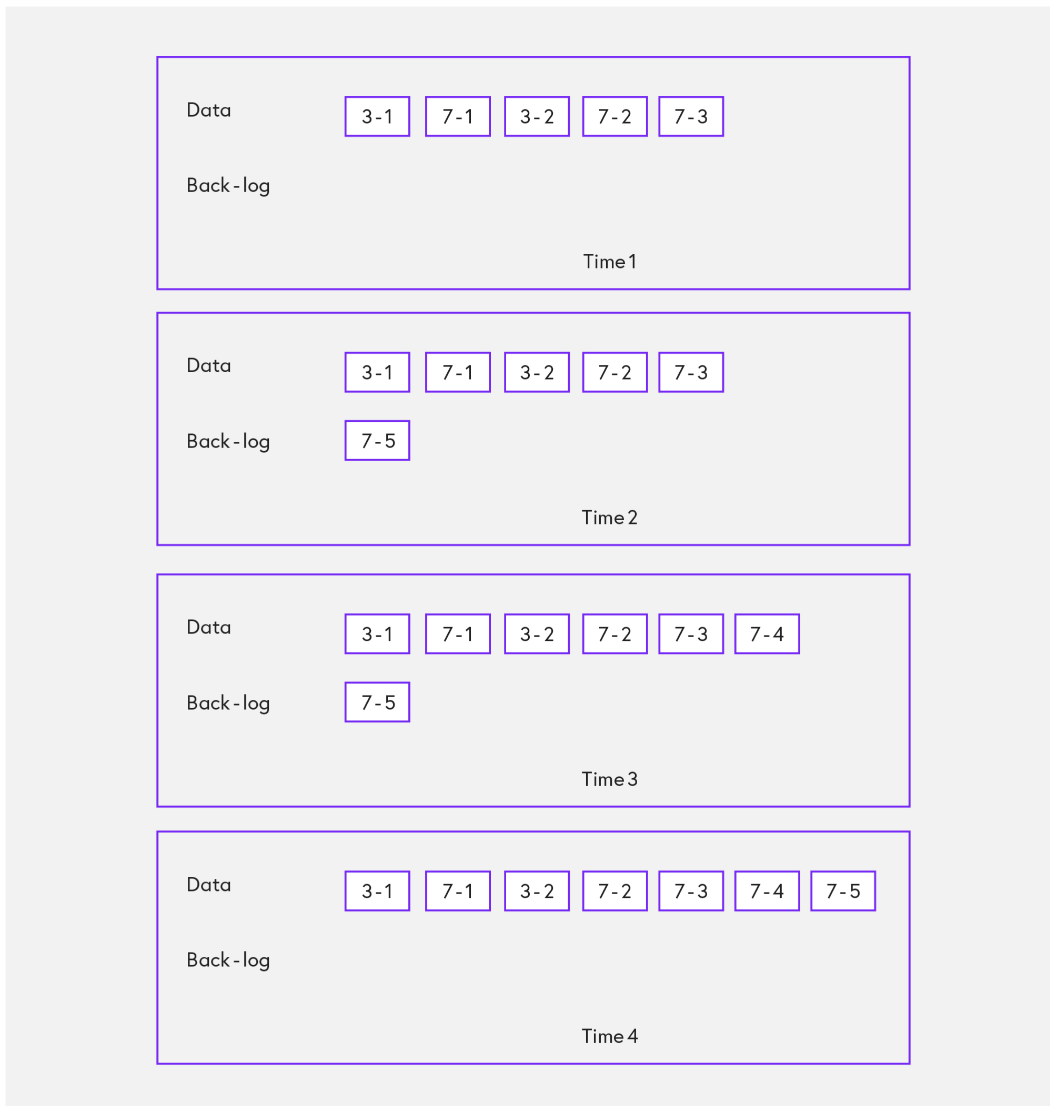


Figure 3

See the example in Figure 3. In Time 1 shard replica on Node3 has shard replica version number 3-2.7-3. It means that replica contains 2 documents from Node3 and 3 documents from Node7. It could happen that in Time 2 Node3 receives document 7-5. This is out of order, because correct sequence should be 7-4, therefore, document 7-5 is placed in back log and the latest document version for this replica version number is still considered to be 3-2.7-3. Then, during the next successful node reconciliation in Time 3, Node3 receives document 7-4 from peers. Now document 7-4 is placed in the shard. After that document 7-5 is moved from back log to shard (Time 4) and replica version number changes to 3-2.7-5.

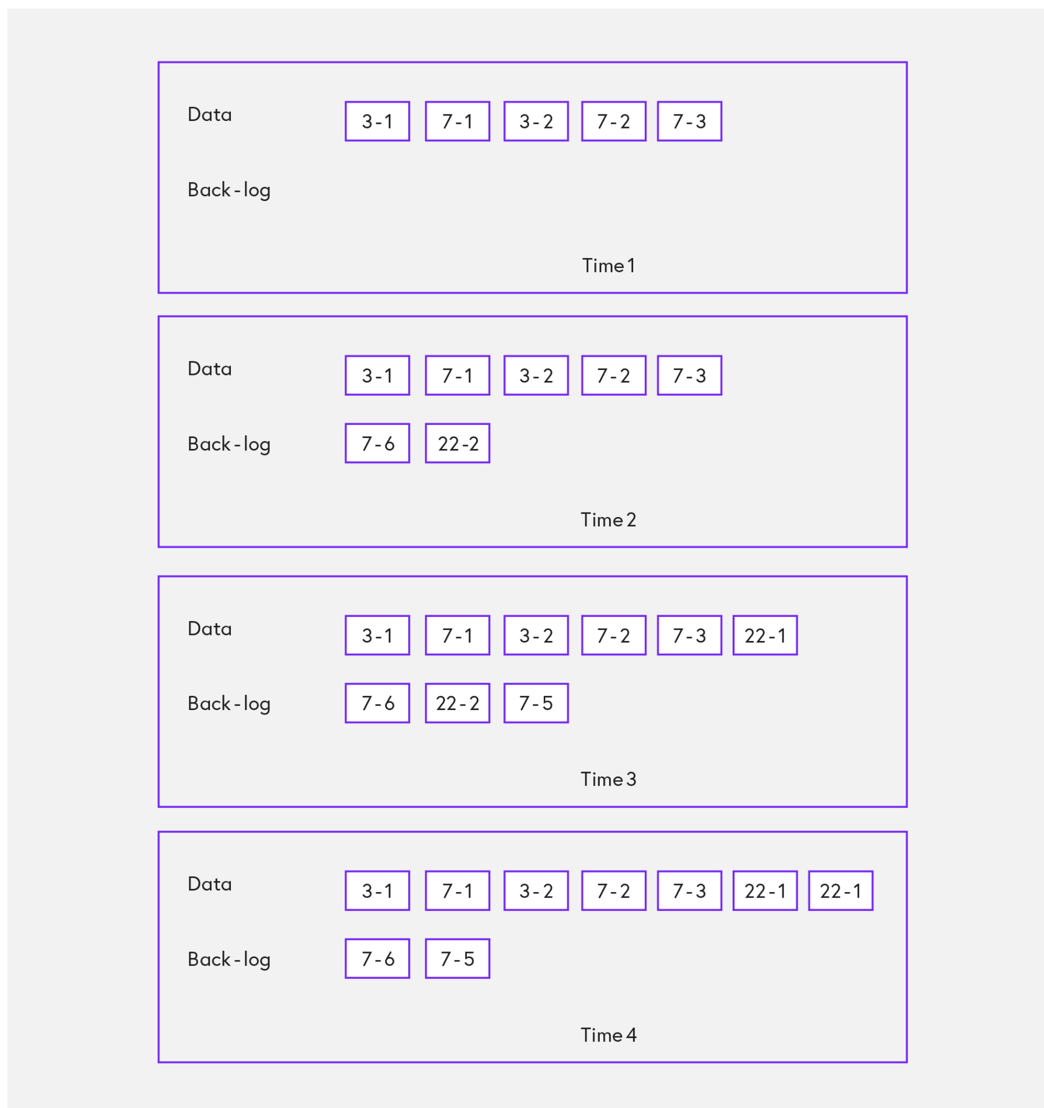


Figure 4

Document version or versions are stored in node back log until correct node sequence is fully restored, and only then each version is moved from back log to the main repository. See the example in the Figure 4. In this case we will have one shard with 3 replicas on Node3, Node7 and Node 22. In Time 1 Node3 contains the same documents that in the previous example in Time 1 (Figure 3). Now we assume that Node3 first receives document 7-6 from Node7 and then document 22-2 from Node 22. Both documents are stored in node back log (Time 2), because both documents are out of order. Then, during node reconciliation, Node3 receives document versions 7-5 and 22-1. Document 7-5 is stored in back log, because it does not form a correct sequence after latest existing document 7-3 from Node 7, but document 22-1 is stored into main repository (Time 3), because of the correct sequence for Node22. After that document 22-2 is moved from back log to the main repository, but documents 7-5 and 7-6 still stay in the back-log (Time 4). When Node3 receives document 7-4, that document is placed in the main repository and documents 7-5 and 7-6 are moved from back-log to the main repository.

Transactions

Support of the transactions consisting of the set of multiple operations is one of the key features of Clusterpoint database server software. It means that in one transaction it is possible to change multiple documents. Transactions are one or more operations that all must succeed. If one operation fails, all the other operations in transaction fail as well, and the database is reversed to the state before the transaction began. The transaction ensures guaranteed consistency of multiple read-modify-writes:

1. Atomicity of updates within transaction: all succeed together or all fail together.
2. Consistency: All changes made in single transaction are all successfully written to database in correct sequence and are all available to every other transaction after successful transaction commit, or all changes are rolled back. Each single transaction preserves consistency.
3. Isolation: each transaction, repeatedly retrieving the same document version, always sees the same content as it was retrieved for the first time. Clusterpoint transactions guarantee the same transaction isolation level as SNAPSHOT isolation level in relational databases.
4. Durability: after hub reported back transaction commit, all changes made in the transaction are available, even if system crashes and then restarts its operation.

Transactions can be rolled back either by client, requesting to roll back, or by server if race condition detected between two transactions, or due to internal failure. More about race condition see in chapter *Race condition between transactions*. Every transaction is initiated by client sending operation BEGIN_TRANS.

Transaction log versions

Each transaction has Transaction context where information about documents inserted, updated or deleted during this transaction is stored. During transaction commit (operation COMMIT_TRANS) a Transaction log record is created from Transaction context and stored as document in Transaction log database. Transaction log database is common Clusterpoint database that is provided to store Transaction log records. When Transaction log record is stored as document version into Transaction log database, Hub receives back document version number for that transaction log record. This document version number is called Transaction log record number.

Each node for each user database shard replica has information, until which transaction (identified by transaction log record number) shard replica has all documents stored in correct order. It means, any shard replica has transaction log

record version number, until which transaction this replica has all documents inserted, updated or deleted by transactions. Transaction log version number are such maximum transaction log record number, that for any user database shard more than half of replicas have at least this Transaction log record number. It means that for each shard more than half replicas are able to answer requests about documents inserted, updated or deleted in transactions until current Transaction log version number.

Process of calculating of Transaction log version number is called discovering of Transaction log version number. From Transaction log version number we can conclude which transactions were committed until the moment this Transaction log version number was discovered.

Transaction identification

Every transaction in Clusterpoint always has:

1. Transaction ID: transaction ID is assigned value of Operation ID for BEGIN_TRANS operation for given transaction. Thereby, each transaction is identified by invariable Transaction ID.

2. Baseline Transaction log version number: Transaction has Transaction log version number, that is the same number that hub has when performs the BEGIN_TRANS operation that starts transaction. More than one transaction can have the same Transaction log version number.

3. Transaction context before committing or rolling back. Transaction context is memory structure in hub RAM. Each transaction has its own transaction context and supplements data there each time an operation is initiated in this transaction. Transaction context contains Transaction ID and Transaction log version number. Transaction context also stores all document Document IDs and their corresponding Operation IDs, read by this transaction.

4. Transaction log record is created from transaction context after transaction commit or rollback. Transaction log record contains Transaction ID and information about all document versions inserted, deleted or modified in a particular transaction.

All document versions additionally have 4 service fields – Transaction ID and Transaction log record version number for transaction that created this document version, and Transaction ID and Transaction log record version number for transaction that deleted this document version.

When Node inserts a new document version, it inserts the transaction ID for transaction that created this document into the document version. When Node deletes document version, it inserts into document version Transaction ID for the transaction that deleted this document.

When committing a transaction, transaction context as Transaction log record (document) is inserted into transaction log database and hub receives back from transaction log database Transaction log record version number for the transaction.

When committing a transaction, which inserts document version, hub sends to node and Node adds to this document version Transaction log record version number of that transaction. Thus, after inserting, the document has 2 filled in fields from 4: Transaction ID and Transaction log record version number.

When committing a transaction, which deletes document version, hub sends to Node and Node adds to deleted document version Transaction log record version number for that transaction. Thus, deleted document has 4 service fields: Transaction ID 1 (as from insert) and Transaction log record version number 1 (as from insert), Transaction ID 2 (as from delete), Transaction log record version number 2 (as from delete).

When committing a transaction, which updates document version, Node commits insert for the new document version and delete for the old document version.

In case node does not receive commit, it adds Transaction log record version number to the documents during Transaction log propagation.

In system settings it can be configured, if Transaction log record version number is inserted into appropriate document version during Transaction committing or during Transaction propagation. Refer to chapter *Transaction log propagation* for more information on this process.

Document and node versions used in transaction

For consistency reasons in case more than one transaction runs in parallel, document versioning is used to ensure that transaction always works with appropriate document version. Appropriate document version is version that is active when transaction is started with command BEGIN_TRANS. Active document version at some moment of time is document version that at that moment is not changed or for which changes are not committed yet.

Let

- Nt is transaction log version number at some moment of time,
- $NIns_{"a"x}$ is Transaction log record version number of transaction that inserted document "a" version x,
- $NDel_{"a"x}$ is Transaction log record version number of transaction that deleted document "a" version x.

Then, for the moment of time, when transaction log version number is Nt, active document "a" version is such document "a" version i that ($NIns_{"a"i} \le Nt < NDel_{"a"i}$) or ($NIns_{"a"i} \le Nt$ and document "a" version i is not deleted).

Client begins transaction by sending to the Entry point hub command BEGIN_TRANS. During BEGIN_TRANS operation hub stores in the transaction context the last discovered Transaction log version number as Basic Transaction log record number.

When any operation with documents is performed, hub always sends to the node transaction log version number for current transaction.

Each node for each replica maintains transaction log version number up to which the node has received and processed all relevant transaction log records for that replica. This is transaction log version number up to which replica has all document versions in the main store (not in the back log). It means that replica can correctly answer only to the operations from such transactions which Baseline Transaction log version numbers is less or equal to the replica's transaction log version number.

When node receives request for some operation from hub executed inside transaction, first node checks if replica's transaction log version number is greater or equal than transaction's Baseline Transaction log version number received from hub. If node's transaction log version number is less than Baseline transaction log version number sent by the hub, then node sends error message back to the hub.

On second step node searches if it has document version active at the moment the transaction began. If it has the necessary document version, node performs requested operation with that document version (more details how SELECT, DELETE, UPDATE operations are carried out see in chapters *Select in transaction, Insert in transaction, Delete in transaction, UPDATE in transaction*). If node has no document version active at the moment the transaction began, node sends error message to hub.

To perform SELECT, DELETE or UPDATE operation with document hub connects node serving shard's first replica containing that document. If node returns error that node does not contain necessary transaction log version number or does not contain document version active for that transaction, hub connect node serving shard second replica etc.

To perform INSERT operation with document hub connects node serving shard's first replica where that document must be saved. If node returns error that node does not contain necessary transaction log version number, hub connects node serving shard's second replica etc.

Transaction workflow

Regular transaction workflow is described in this chapter. Later several exceptions and additions will be described. Typical update transaction workflow example is depicted in Figure 5 and commented afterwards.

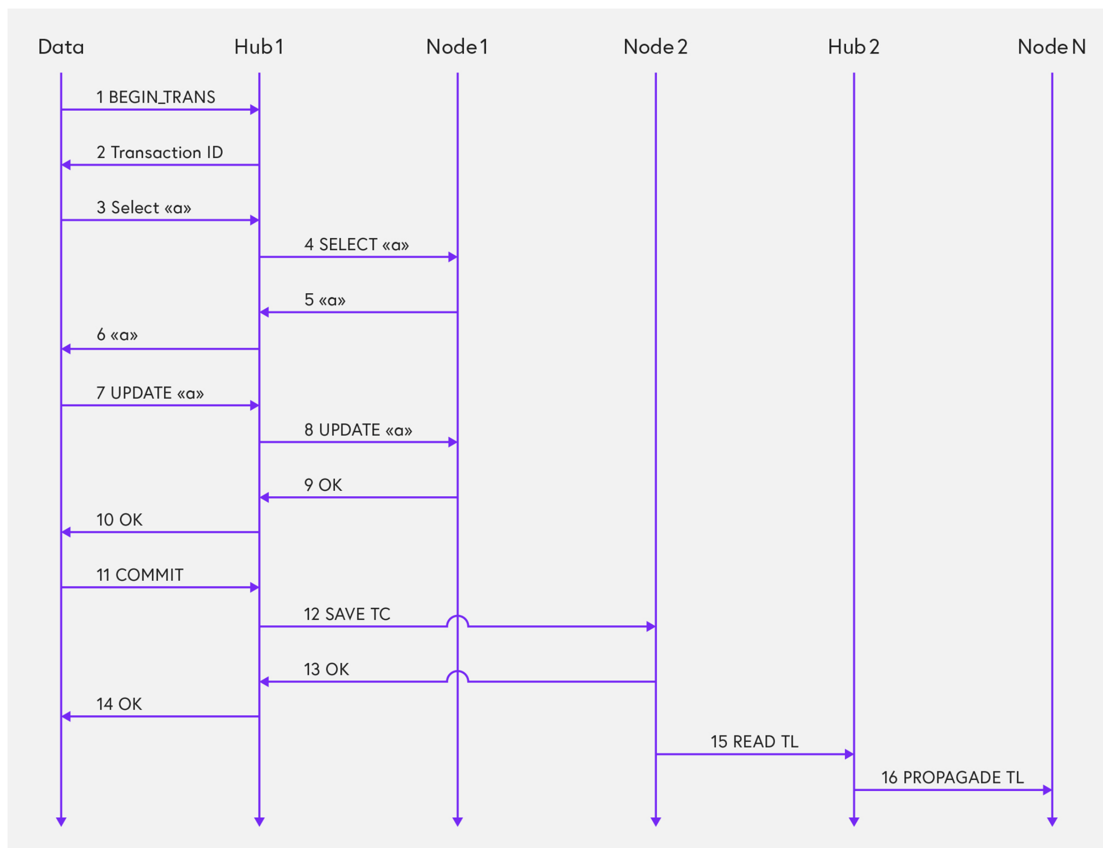


Figure 5

1. Client starts transaction by sending to entry point hub (Hub1) command BEGIN_TRANS.

2. Entry point hub (Hub1) creates empty transaction context and assigns a unique Operation ID to the transaction as invariable Transaction ID. Entry point hub uses the last discovered Transaction log version number Nt and assigns it to the transaction as invariable Baseline Transaction log version number for this transaction. Hub adds Transaction ID and Baseline Transaction log version number to transaction context. Entry point hub sends back Transaction ID to client. Client must include this Transaction ID in every request made in this transaction.

3. Client sends Transaction ID and SELECT document "a" to the Entry point hub Hub1.

4. Entry point hub (Hub1) sends SELECT document "a" and Baseline Transaction log version number Nt (that is stored in this transaction context) to the Node1 that serves shard's first replica containing document "a".

5. Node1 checks if it has appropriate transaction log version number and appropriate document "a" version for Baseline Transaction log version Nt. If Node1 has appropriate transaction log version number and node has appropriate version of document "a", Node1 returns document "a" appropriate

version (even if this document "a" version is deleted or updated). Otherwise Node1 returns error message to Hub1. In case of error message about incorrect transaction log version numbers in Node1, Hub1 asks other node that serves shard second replica for document "a" and so on until some node returns document "a" content. If no node returns document "a", Hub1 returns error message to client.

6. Entry point hub (Hub1) saves in transaction context Operation ID (later it will be referred as "old" operation ID) from document "a" version received from node. Hub1 sends document "a" version to client.

7. Client sends Transaction ID and UPDATE document "a" to Hub1.

8. Hub1 adds to received document "a" version the new Operation ID for UPDATE, the old Operation ID form context (because this document already has been read during the transaction) and Baseline Transaction log version number from transaction context, and sends UPDATE document "a" to appropriate Node (Node1). Thereby, Node1 receives document "a" new version with new Operation ID, Old Operation ID and Baseline Transaction log version number.

9. If Node1 still has document "a" version i with Operation ID for operation inserted document version equal to Old Operation Id and document version is not deleted, first the document is locked, then updated as described before *(*see chapter about *Document insert)* and OK is returned to Hub1.

10. After obtaining OK from the majority of replicas, hub (Hub1) adds the affected shard to the list of shards affected by the transaction and stores the shard version into the transaction context. Hub sends OK to client.

11. Client sends Transaction ID and "COMMIT" to entry point hub (Hub1).

12. Entry point hub (Hub1) saves Transaction context from Hub1 memory as a transaction log record and sends to the node serving appropriate shard of transaction log database (Node2).

13. Node2 saves transaction log record and gives document version number to the Transaction log record, correspondingly the Transaction log version number increases. Node2 sends back the Transaction log record version number to Hub1. Hub1 sends this Transaction log record version number to all nodes serving documents changed in current transaction. These Nodes add Transaction log record version number into document versions changed during this transaction and releases locks from these documents.

14. Hub1 sends OK to client.

15. Some hub (the same or other, in this case Hub2) discovers new Transaction log version number.

16. This hub (Hub2) propagates transaction log to the other nodes *(*see more in chapter Transaction log propagation*)*.

SELECT in transaction

When during transaction a document is SELECTed, it is guaranteed that actual document version will be returned, meaning document version which was current when the particular transaction began. If repeated SELECT is issued in the same transaction, the same document version is returned, even if other transactions have updated or deleted that document. This ensures that all reads within transaction are repeatable and possible conflicts between transactions are resolved upon update of documents. Above mentioned is true not only for single document SELECT, but also for multi-document SELECT database operations.

The only exception to the above is that if a transaction has modified a document 'a' then after the modification the transaction will see the modified document version. This is ensured by storing the affected shard versions in the transaction context in the Hub. When subsequent SELECT operations are issued by the same transaction the shard version required is passed by the Hub to the Node, and Node then given the current transaction ID replaces documents modified by the same transaction with the modified versions. If Node does not have the required shard version it returns an error.

Here is a fragment or typical SELECT workflow during transaction. These are steps 3-6 from transaction workflow described above:

1. Client sends Transaction ID and SELECT document "a" to the Entry point hub Hub1.

2. Entry point hub (Hub1) sends SELECT document "a" and Baseline Transaction log version number Nt (that is stored in this transaction context) to the Node1 that serves shard's first replica containing document "a".

3. Node1 checks if it has appropriate transaction log version number and appropriate document "a" version for transaction log version Nt. If Node1 has appropriate transaction log version number and node has document "a" version with appropriate version, Node1 returns document "a" appropriate version (even if this document "a" version is deleted or updated). Otherwise Node1 returns error message to Hub1. In case of error message about incorrect transaction log version numbers in Node1, Hub1 asks other node that serves shard second replica for document "a" and so on until some node returns document "a" content. If no node returns document "a", Hub1 returns error message to client.

4. Entry point hub (Hub1) saves in transaction context Operation ID (later it will be referred as "old" operation ID) from document "a" version received from node. Hub1 sends document "a" version to client.

Hub maintains in memory transaction context for every transaction in which it saves meta information about all documents used in the transaction: Operation IDs of documents that were returned to client in a particular transaction and during repeated SELECT.

Clusterpoint server guarantees that transaction always accesses the document version which was active when the transaction began or the version modified by the transaction, even if other transaction later updated or deleted it. A fragment about UPDATE and DELETE from transaction race condition described below.

INSERT in transaction

During transaction new document INSERT is performed similarly as described in chapter *Document insert*. During transaction commit hub sends to node committed transaction log record number. Node inserts this transaction log record number into inserted document version metadata as transaction log record number for transaction that inserted this document. More information about document version's technical fields can be found in chapter *Transaction identification*. Until transaction that inserted document version is not committed, newly inserted document version is not available for other transactions.

DELETE in transaction

During DELETE latest document version is immediately marked as "deleted" by specific transaction. Document version that is marked as deleted is no longer readable by subsequent SELECT operations in this transaction. Deleted versions are physically deleted from persistent storage after transaction is committed at the time when database initiates compaction. A compaction occurs periodically after configurable percentage or data has been deleted. This threshold manages trade-off between IO necessary for compaction and storage space. If other transaction requires reading deleted document version, the deleted document version is accessible for reading to this transaction. During transaction commit hub sends to node committed transaction log record number. Node inserts this transaction log record number into deleted document version metadata as transaction log record number for transaction that deleted this document. More information about document version's technical fields is in chapter *Transaction identification*. If other transaction tries to update or delete the document version which is already deleted, this other update or delete operation fails.

UPDATE in transaction

Document versions are not updated. Document UPDATE consists of delete of current document version and insert of new document version. For more details about document version delete and insert see chapters *Insert in transaction, Delete in transaction*.

When a document version is read and then updated during transaction, it is guaranteed that document version read at first read is updated, it is updated is that document version which was current when a particular transaction began.

If other transactions have updated or deleted that document version, the UPDATE operation fails. Details see in further example (Figure 6), Steps 9 – 14. In this example Transaction1 and Transaction2 read document "a", then Transaction1 updates document "a", after that Transaction 2 update on document "a" fails.

If repeated update on the same document in the same transaction is issued after successful update, currently last document version (newly created) is deleted and new document version is inserted.

Locking document for update or delete during transaction

Node locks documents instead of document versions. It means that node stores in memory this node's entire document PK locked at this moment. As all document versions have the same PK, all document versions are locked.

During update or delete a document version is locked with majority locks, e.g. if there are N replicas for shard, the document version must be updated/deleted in more than N/2 replicas. Hub sends update/delete request to all N nodes containing shard replicas with this document version. Node tries to update/delete specific document version. Update/delete can be done if transaction version number of transaction that inserted replica document version is less or equal to current transaction version number and document version is not updated/deleted. If update/delete succeeds in more than N/2 replicas, the operation succeeds. If update/delete succeeds in less or equal to N/2 replicas (for example, if other transaction is locked N/2 or more document replicas for update), the operation fails. If the operation fails, hub sends to nodes that successfully updated document, request to revert update/delete operation.

If locking is not successful, update or delete fails, transaction is automatically rolled back and hub returns error to client. If locking is successful, document is updated or deleted.

This means that if update within transaction succeeds, it is guaranteed that no other transaction will introduce changes between document versions that transaction has previously seen and updated, since only one transaction can lock specific document version on majority of nodes.

Race condition between transactions

Let us assume we have a node which serves shard replica containing document "a" version 3-1 with Operation ID 356 and transaction log record version number for transaction inserted this document version $NIns_{"a"3\text{-}1}$ = 15-4. This document version was inserted by transaction with transaction log record version number Nt = 15-4.

Transaction 15-4 was committed. After that, another transaction on another document was committed; thereby, transaction log record version number for that another transaction is 15-5 and transaction log version number Nt is 15-5. We also assume that transaction log version 15-5 is propagated to this replica. Suppose there are two transactions that run concurrently and try to modify document "a" (Figure 6).

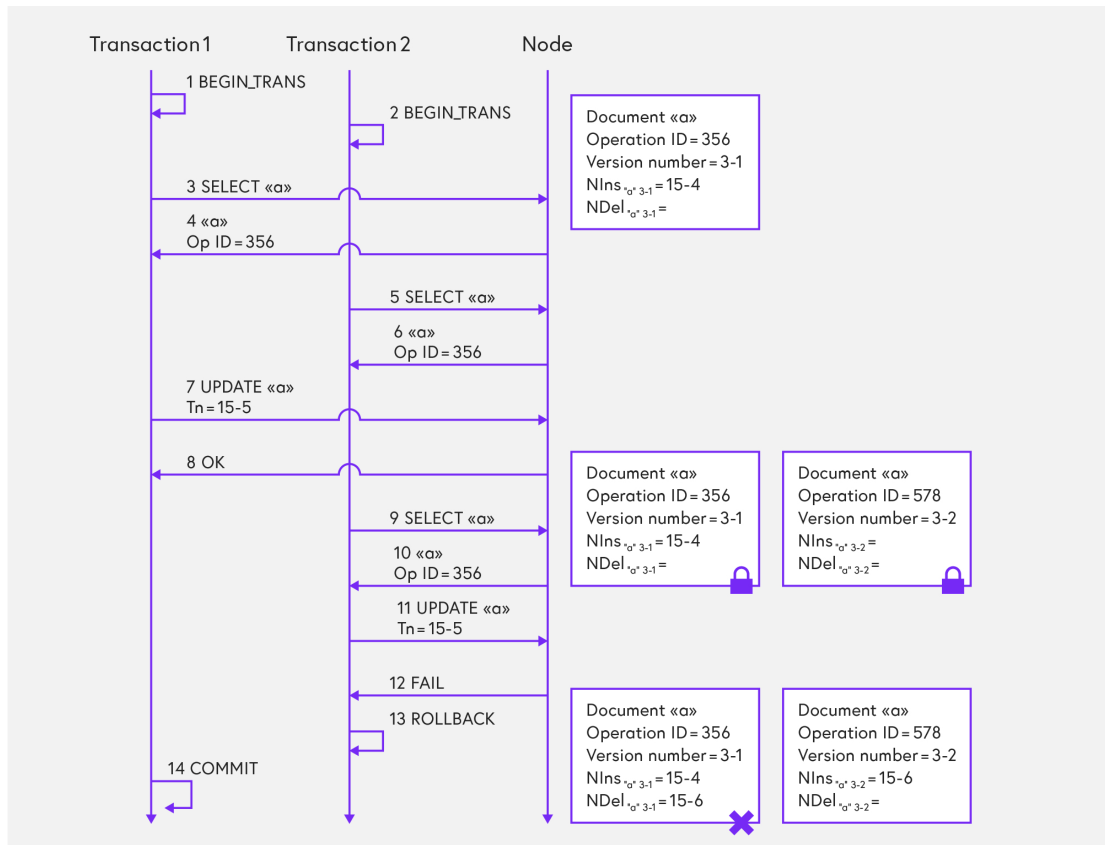


Figure 6

1. Transaction1 is started by BEGIN_TRANS. Transaction log version number Nt is set to 15-5.

2. Transaction2 is started by BEGIN_TRANS. Transaction log version number is Nt is set to 15-5 as there were no transactions committed between Transaction1 BEGIN_TRANS and Transaction2 BEGIN_TRANS.

3. Transaction1 SELECTs document "a". Entry point hub sends request to node for document "a" version with Transaction log version number Nt = 15-5.

4. Node has all document versions for Transaction log version number Nt = 15-5, therefore, it can return document "a" version 3-1 content with Operation ID = 356 and Transaction log record version number for insert transaction $NIns_{"a"3\text{-}1}$ = 15-4.

5. Transaction2 SELECTs document "a". Entry point hub sends request to node for document "a" version with Transaction log version number Nt = 15-5.

6. As in Step 4, Node has all document versions for Transaction log version number 15-5, therefore, it can return document "a" version 3-1 content with Operation ID = 356 and Transaction log record version number for insert transaction $NIns_{"a"3-1}$ = 15-4.

7. Transaction1 performs UPDATE on Document "a". Transaction sends to node a new "a" version with Operation ID = 578 and Transaction log version number for current transaction Nt = 15-5.

8. Node checks that document "a" version active for Nt =15-5 with $NIns_{"a"3-1}$ = 15-4 is not locked for or deleted from another transaction, thereby, node creates a new document version 3-2 for document "a" with Operation ID = 578. This document version has no Transaction log record version number of transaction that inserted document "a" version 3-2 (because transaction is not committed yet). Node locks document "a".

9. Transaction2 SELECTs document "a" with Transaction log version number 15-5 (because transaction SELECTs this document the second time, the same document version as first time will be returned).

10. Node has document "a" version 3-1 with $NIns_{"a"3-1}$ = 15-4, and document "a" newer version 3-2 which is not committed yet. Transaction2 asks for document version with Nt = 15-5, therefore document version 3-1 is returned.

11. Transaction2 performs UPDATE on Document "a" with Transaction log version number 15-5. Transaction2 sends to node a new "a" version with $NIns_{"a"3-1}$ = 15-4.

12. Node sends back error stating that other transaction has updated this document because document "a" version active for transaction with Nt = 15-4 was updated. See details in chapter *UPDATE in transaction*. Transaction2 is failed and is automatically rolled back in step 13.

13. Transaction 2 rollbacks.

14. Transaction 1 commits. Transaction context is written into transaction log record with transaction log record version number 15-6. Transaction log record version number of transaction that inserted document for document version 3-2 $NIns_{"a"3-2}$ is set to 15-6. Transaction log record version number of transaction that deleted document for document version 3-1 $NDel_{"a"3-1}$ is set to 15-6.

Steps 5,6, 9, 10 show that performing read several times within a transaction the same document version is returned even if document was updated by another transaction.

Steps 11, 12 illustrate that it is impossible for a transaction to update document version if this version was successfully read, but after that updated by another transaction.

After some time long enough to finish all transactions that started before document "a" has updated, old version (3-1) of document "a" (with Operation ID = 356) is deleted.

Transaction interaction with blind operations

For each operation with data (SELECT, INSERT, UPDATE, DELETE) client can dynamically choose between two modes how to communicate with Clusterpoint database – to use transaction or not to use transaction. If transaction is not used to perform operation, that operation is called blind operation. Each blind operation is performed independently.

Blind operations are introduced for performance reasons, documents are updated atomically and no transactions are used. As update consists of insert and delete operations, we will describe blind operations by detailing blind updates.

They are targeted to maximize update throughput and compromises consistency, which means that for some period of time different nodes can return different versions of the same document as the actual, because of race condition when two or more operations are attempting to perform at the same time on the same document and therefore, the result of the change in document depends on the built-in thread scheduling algorithm.

Inconsistency among replicas will be resolved during node reconciliation and one of conflicting updates will be considered as actual version of the document. Race condition of blind updates is resolved by operation IDs. The update with the largest operation ID always stays as the actual version and is copied to other replicas. Other conflicting updates are rolled back.

Blind updates ignore document locks placed by running transactions and can change any document in any moment. It means that at the same time while one or more transactions are running, documents can be changed via blind updates.

If operation ID of blind update is less than document ID read by transaction or after document update operation ID in this transaction, transaction does not conflict in any way with this blind update. In other words the two are serializable.

If operation ID of blind update is between document ID read by transaction and document update operation ID in this transaction, then that transaction will be auto rolled back, because the operation ID check will fail.

Also as soon as the blind update is written to some replica and the blind update has a larger operation ID, that replica will now return the blind update instead of the

appropriate version according to the Baseline Transaction log version number of the transaction. Thus interleaving of blind operations with transactions risks breaking the snapshot serializability and introduces non-repeatable reads.

If strong consistency is necessary), all updates should have been performed within transaction.

Transaction commit

When transaction is committed, its transaction context is written as transaction log record (document) in transaction log database and node sends to hub transaction log record version number. Hub receives information from node and sends this Transaction log record version number to the nodes serving documents changed in current transaction. Nodes add Transaction log record version number into document versions changed during this transaction.

Transaction log propagation

Transaction log propagation is performed by hubs to ensure that each node, serving database, gets actual information for each its served replica about all committed transactions even if transaction did not operate with documents from this replica.

When hub discovers new version of transaction log version number, it initiates transaction log propagation. Hub "responsible" for database node asks what the largest transaction log version number for each client document database replica on that node is. Let us consider an example when transaction log version number X for Node1 Replica2 is less than hub's discovered new transaction log version number Y.

1. Hub reads transaction log records from Transaction log database where transaction log record (i.e. document) version number is greater than X (here we can say Replica2 has outdated transaction log version number) and sends them all, for example, 5 records to Node1 for Replica2.

2. Node1 receives those 5 transaction log documents; let us enumerate them from the oldest to the newest as Transaction1 to Transaction5.

3. Node1 processes transaction log records (documents) one-by-one, starting with the oldest one – Transaction1. If Replica2 contains all client document versions which were processed during Transaction1, transaction log version number X for Node1 Replica2 is increased to Transaction1 log version number, let us call it X'. In addition, if Node1 Replica2 contains some document version inserted, updated or deleted by that Transaction1, write lock is released from old and new document versions, because Node1 now knows that Transaction1 is committed. Here we can say that Replica2 now is up to Transaction1.

4. Node1 starts processing Transaction2 log record, next Transaction3 up to Transaction5.

If Node1 Replica2 does not contain all document versions modified by, for example, Transaction3, then Transaction3 log record processing and propagation process is stopped and no other transaction log documents (records) are processed in this propagation time. In this situation Node1 Replica2 log version number equals to Transaction2 transaction log version number, let us call it X".

During the next propagation hub will discover that Node1 Replica2 transaction log version number X" is less than hub's newly discovered new transaction log version number Z and hub will send to Node1 Replica2 Transaction3, Transaction4, Transaction5 and newer transaction records.

Transaction rollback

When transaction is failed either because of race condition or requested to do so by client, entry point hub marks transaction in failure state and contacts every node involved in this transaction to remove document locks associated with this transaction. Failed transactions are automatically rolled back.

Clusterpoint server will anyway remove locks automatically after expiration time, which is greater than configurable maximum lifetime of a transaction, but for performance reasons it is better to remove them explicitly, because while locks are not removed then if other transaction tries to change locked document, change operation will fail.

Multi-document SELECT consistency

The most important part of multi-document operation transactions support is to guarantee that each search operation during the same transaction returns consistent results considering actions performed by this transaction.

When client during transaction sends SELECT request, an entry point hub first assigns transaction log version number to SELECT request before sending it down to nodes for execution. When receiving a SELECT request with transaction log version number nodes check if shard is propagated at least up to this transaction log version number as well as checks if shard is at least at required version. If one of these does not fulfil, node reports failure back to entry point hub and hub requests to node serving another replica. If no node can fulfil SELECT request, error is returned to the client.

Additional information

For more information visit www.clusterpoint.com or e-mail us: info@clusterpoint.com