=nil; Database Management System

Mikhail Komarov

=nil; Foundation

nemo@nil.foundation

March 3, 2021

Abstract

For decades databases and database management systems (DBMS) were the basis for building various applications. DBMS selection in most cases defined the storage data internal structure. With widespread usage of authenticated data structures-based cluster commit log databases has come, lots of new cluster replication and consistency protocols were introduced. Clusered databases replication protocols became the application-level logic, stored data structure itself became even more important, than the way it is getting processed. Existing database management systems became insufficient to handle all the database replication protocols (which turned out to be unique for each of database in the DBMS) storage mechanisms and consistency conditions execution complexity. This paper intends to introduce `=nil;` *Database Management System, able to handle all the modern requirements through moving cluster replication protocol specialization along with cluster transactions consistency conditions definition to database configuration level, specified for each database individually and stored inside the database itself or distributed as a separate module.*

1. Introduction

From the very beginning various database management systems architecture was focused on effectively storing data in the particular format with a strictly defined set of storage consistency conditions, distributing data with a particular, pre-defined replication protocol within particular pre-defined cluster architectures. Introducton of an authenticated cluster protocol commit log concept extended available cluster architecture set with a family of "master-master"-alike replication cluster protocols making possible to store and redistribute data being sure it gets replicated exactly the same way on every cluster node. Such an introduction made advancement to easily achievable clustered databases-based application properties, decreasing the trust required to be given to the data stored because of no way of single cluster node isolated modification of the such a "master-master" replicated storage without achieving cluster consistency possibility was found.

The absence of database management systems able to handle such clusters architecture made databases and applications developers to introduce very own implementations of a data-management piece of software with a hardcoded particular data structure support along with a hardcoded application-specific "master-master" cluster consensus protocol.

Such an approach is an extremely inefficient way of launching an application with specific cluster commit log requirements: financial assets management systems, industrial hardware control and logging, sensible data storages etc. This also creates a problem of handling such datasets with some kind of unified API.

1.1. Cluster Consistency Mechanisms

According to unique consequences, when the data itself along with its' structure is much more important than the software used for handling, and adjusting data replication protocol to the data management software is harder than in fully-controlled "master-slave" clusters (inspite of previous DBMS industry experience when the data-handling software was at least as much important as the data itself), the generalization of data-handling techniques used to build the generic DBMS is required to be done to correctly embed newly introduced cluster architectures into the generic DBMS architecure. Some of these generalizations would lead to moving hardcoded mechanisms to the database configuration level.

Mechanisms selection is reasoned by the necessity of generalization of typical DBMS mechanisms to handle cluster replication protocols of very different designs. This mechanism list includes: cluster commit log handler, cluster replication protocol flow and serialization/marshalling format definitions and handlers, local data storage consistency mechanisms.

1.2. Replication Protocol Features Distribution

According to the assumption every particular database is supposed to have it's unique cluster architecture handling mechanisms, the way to enable and disable particular features right at runtime should be present to avoid the impossible - including all of the cluster architecture mechanisms known. This is supposed to be done with two ways: virtualization and plugins mechanism.

The virtualization of cluster architecture handling mechanisms requires for the virtualization environment to be able to handle generic Turing-complete programming language to ensure every cluster replication protocol can be implemented. This would allow to literally embed the code to the particular database configuration and redistribute it with the database data. Unfortunately the performance of such a solution would be less than with natively-executed code.

This is why it is also required to have plugins mechanism - to ensure performance-demanding databases will get as much as they can from the hardware. But such a solution raises secutiry concerns - every particular plugin implmenenting some replication protocol adapter (the set of cluster replication handling mechanisms) should be manually audited inspite of virtualized implementations.

1.3. Cluster Consistency Conditions

Several database cluster architectures suppose cluster consistency conditions to be represented by imperative language, distributed with the clsuter data itself, (which may be not even Turing-complete one) and executed in virtualized environment (sometimes unique for each protocol) to ensure newly coming data do not violate cluster's master nodes data integrity. Such a mechanism is considered as a part of cluster architecture handling techniques, but being a hevay-loaded part, some replication protocols would require advanced architecture and implementation techniques to make it perform well enough (e.g. sharding the load to other "slave"-alike cluster nodes). This means a necessity for virtualization in case of several protocols, can be replaced with sub-clusterization necessity.

Coming to consistency conditions virtualization. In case some particular replication protocol has its' own unique virtualization environment with its' own bytecode format, it is required to have a way to handle it. This is where pluggable virtualization engines come out handy. Using protocol-specific VM as a DBMS plugin allows to handle even an extremely specific bytecode with no performance troubles.

1.4. Storage Engines

Various clustered databases data structures suppose for the DBMS to handle them differently. Column-oriented data, geospatial data, raw key-value storages - all of them require to be hadled with storage engines implemented in very different ways (some of them should be based on LSM-trees, some of them should be base on B-family trees, some should be done in a completely different way). This fact, by the way, does not mean it is also required to distribute storage engines as a database configuration. Data storage engines are considered as highly resource-demanding components, which are crucial to operate at their maximum performance. That means the only option possible is natively-executed pluggable data storage engines.

1.5. Query Language

Data structures differencies (data relations differencies, whatever they are being called: collections, blobs etc) suppose for such a DBMS to have some query language to handle the set of relations of various complexity. According to the DBMS needs to be able to handle non-relational data structures, this query language has to be an imperative one, so there would be no limitation in query definitions. The most trivial way is to move forward with one more SQL dialect, but this disables unified query language for in-consistency conditions data queries along with all the other replication protocol adapter components data queries, in fact introducing two query languages: a language for replication protocol adapter queries and a language for user data queries. This means, data query language has to be more of library-level DSL implemented in some widespread generic-purpose imperative Turing-complete language (e.g. C++) or compilable to particular's protocol VM bytecode.

Such a query language approach brings questions about query planning and optimizations, which are usually being performed by architecturally separate module with deep understanding of current DBMS load situation. Imperative bytecode-based virtualized queries make nearly impossible to parse and understand particular data queries (DBMS should not try to decompile bytecode-based queries to get an information about data currently requested), that means any query planners and optimizations (getting required data from storage engines) should be embedded right into the query, as a bytecode. This also introduces the possibility for user to completely disable query planners with wiping out related code.

1.6. Paper Contribution

The description above forms a list of requirements for the architecture of DBMS able to handle clustered databases with replication protocol specific for each database. =nil; DBMS being introduced by this paper claims to be the first database satisfying this list of requirements and handling different databases with different "master-master" (but not only) cluster replication protocols within the same dataabse management system.

2. Preliminaries

This section introduces preliminaries, terms and definitions required to describe =nil; DBMS architecture well enough.

2.1. Architecture Elements

At first it is required to describe the generic DBMS architecture components.

2.1.1 Storage Engine

Database management systems are designed to store and manage the data. Such an architecture role usually gets performed by a separate component - storage engine. It usually wraps some internal data structure which handles the actual data. Such an internal data structure is usually a tree-alike data structure, so it is required to remind the formal trees definitions of all the kinds which are going to be discussed within this paper.

2.1.2 Trees Notation

Trees being considered in this paper are mostly ordered rooted trees because of B-Tree-family data structures being most commonly used within the storage engines.

An ordered tree is a tree in which the left-to-right order among siblings is given. An unordered tree is a tree with no order among siblings. In order to formulate these trees, a subclass of partially ordered set theory (or lattice theory) and its algebraic system is being employed rather than graph theory since approximate pattern matching between two trees is considered as an order-preserving mapping between two ordered sets.

Tree is being defined as a subclass of a partially ordered set.

Definition 2.1 (Partially Ordered Set)**.** A partially ordered set (or a poset for short) is a set V with a binary relation $\leq$ (called a partial order), denoted by $(V, \leq)$, that satisfies the following:

1. $\forall x \in V (x \leq x)$ (reflectivity),
2. $\forall x, y \in V (x \leq y \wedge \ y \leq x \Rightarrow x \leq y)$ (antisymmetry),
3. $\forall x, y, z \in V (x \leq y \wedge y \leq z \Rightarrow x \leq z)$ (transitivity)

If the set V in a poset $(V, \leq)$ is finite, it is being said that the poset is finite.

Two elements $x, y \in V$ do not always satisfy either $x \leq y$ or $y \leq x$. Thus,if $x, y \in V$ satisfy either $x \leq y$ or $y \leq x$, two elements x and y are said to be comparable. In contrast, if x and y are not comparable, x and y are said to be incomparable. We write $x \leq y$ if $x \leq y$ and $x \neq y$. Also, we often write $y \leq x$ and $y \geq x$ for $x \leq y$ and $x \leq y$ respectively.

Let $(V, \leq)$ be a poset, and U be a nonempty subset of V. A node $x \in U$ is minimal in $U \Leftrightarrow \forall y \in U : y \leq x \ \exists \ y \leq x$. The node x is called minimum if x is a unique minimal nodes. If any two elements of V are comparable, then we refer to $(V, \leq)$ as a chain or a totally ordered set, and to $\leq$ as a linear order or a total order.

Definition 2.2 (Rooted Trees)**.** A rooted tree T is a non-empty finite poset $(V, \leq)$ that satisfies the following:

1. $\exists! r \in V : x \leq r \ \forall x \in V$
2. $\forall x, y, z \in V, x \leq y \wedge x \leq z \Leftrightarrow y \wedge z$ are comparable.

The elements of V are called nodes (or vertices) of T, and the node r is called the root of T and denoted by $root(T)$.

We refer to the binary relation $\leq$ as the hierarchical order, where, for two nodes $x \leq y$, we say that x is an ancestor of y, and y is a descendent of x. Also, for two nodes $x \leq y$, we say that x is a proper ancestor of y, and y is a proper descendent of x.

For a tree T, and a node $x \in T$, by $(\uparrow x)_T$ (resp. $(\uparrow x)_T$) we denote the set of all ancestors (resp. proper ancestors) of $x \in T$, i.e.

$$(\uparrow x)_T = \{y \in T \mid x \leq y\}, (\uparrow x)_T = \{y \in T \mid x \leq y\}.$$

For a tree T, by $V(T)$, we denote the set of all nodes in T, and b $y \leq T$ the hierarchical order $\leq$ of T for clarity. We also write $x \in T$ instead of $x \in V(T)$ for short.

The parent of a non-root node x, denoted by $par(x)$, is the minimum node y in the set $\{z \in V \mid z \leq x\}$, and conversely, the node x is called a child of $par(x)$. The set of all children of a node x is denoted by $ch(x)$, i.e. $ch(x) = \{y \in V\ root(T) \mid par(y) = x\}$. For any two distinct children of a node, one node is said to be a sibling of the other. A node with no children is called a leaf. The set of all leaves in a tree T is denoted by $leaves(T)$. The depth of a node x is, denoted by $dep(x)$, the number of proper ancestors of x, i.e. $dep(x) = |\{y \mid x \leq y\}|$. The depth of any root node is 0. By $dep(T)$ we denote the maximum depth of T, i.e. $dep(T) = max\{dep(x) \mid x \in T\}$, and call it the depth or height of T. The size of a tree T is the number of nodes in T , denoted by $|T|$. For a node x, the size of $ch(x)$ is denoted by $deg(x)$, and referred to as the degree of x. The maximum number of children for all nodes in a tree T is denoted by $deg(T)$, i.e. $deg(T) = max\{deg(x) \mid x \in T\}$, and referred to as the degree of T.

Note that a rooted tree T pursuant to 2.2 is naturally regarded as a directed graph. In fact, by defining the set of nodes as $V(T)$ and the set of directed edges as $E(T) = \{(x, par(x)) \mid x \in V(T)\ \{root(T)\}\}$, $(V(T), E(T))$, we have a directed graph $G = (E(T), V(T))$.

A tree may be equipped with another order in addition to the hierarchical order. This additional order is called the sibling order, denoted by $\preceq$, and defines the left-to-right relation between nodes.

Definition 2.3 (Rooted Ordered Trees)**.** A rooted ordered tree T is a triplet $(V, \leq, \preceq)$ such that the pair $(V, \leq)$ is a rooted tree, and the pair $(V, \preceq)$ is a non-empty finite poset that satisfies the following:

1. For any $x, y \in V$, two nodes x and y are comparable with respect to the sibling order if and only if x and y are equivalent, or incomparable with respect to the hierarchical order.

2. For any distinct nodes $x, y, x', y' \in V \Leftrightarrow x \leq x', y \leq y' \wedge x' \preceq y' \Rightarrow x \preceq y$.

Back in 1960-s, IBM introduced it's IBM IMS, a hierarchical database for early mainframes, which architecture was not supposed to have a storage component incapsulated in any way. It was a rooted ordered tree-alike data structure, containing all the data, architecturally speaking tightly coupled with pseudo-query language processing.

In 1970 E.D. Codd introduced first concept separating logical data structure organization from it's physical representation in the data storage (e.g. B-Tree) [1]. MySQL was one of the first database management systems benefited from ideas of architecturally incapsulated storage engine. It allowed to extend the MySQL applicability area with making it's performance sufficient for newly discovered suboptimal data emplacement ways.

Definition 2.4 (Storage Engine)**.** Following E.D. Codd results, this paper defines **Storage Engine** as a standardized API component used for abstraction over the actual physical data structure being used for data storage and management.

2.1.3 Storage Engine

2.1.4 Storage Engine Commit Log

Storage engine commit log initial existance reason is to provide the durability property for the stable storage with having a second transaction-by-transaction changelog of the overall data contained. This was used for recovering the data emplaced in stable storage in case of corruption.

Since the **Storage Engine** internal construction may vary from trivial data structure incapsulation with an API similar to `write` (to write some data into the internal data strcuture) and `read` (to read some data from the internal data structure, selecting it with some criteria), to complicated transactional mechanisms, the storage engine commit log construction may vary from "no commit log" to write-ahead log-alike constructions.

The most trivial way to construct a commit log is to represent a set of transactions

This introduces the requirement to generalize the storage engine commit log concept with introducing the standardized API generic enough to handle the trivial `read`/`write` communications and .

2.2. Cluster

2.2.1 Cluster Commit Log

2.3. Consensus Protocol

2.3.1 Master-Slave Consensus Protocols

2.3.2 Master-Master Consensus Protocols

2.4. Networking

2.4.1 Protocol State Simplex

2.4.2 Protocol Structures

2.5. Protocol Adaptor

Definition 2.5 (Protocol Adaptor)**.** Protocol Adaptor is a module consisting of networking modules and consensus protocol definition.

3. Architecture Proposed

Following sections describe the overview of the proposed architecture with terms and concepts defined in "Preliminaries" section **??** able to satisfy all the requirements defined in the "Introduction" section **??**.

4. Framework

In spite of most databases architecture supposes the final implementation to be a solid piece of software, Nil DBMS intends to be based on a stack of frameworks, which are not required to be developed from scratch:

1. Actor model multithreading framework is intended to reduce locks and data flow control, which significally increases performance for a highload applications.

2. Server applications framework. Required to become a communication basis for database infrastructure services.

3. Data storage framework. Incapsulation of a transaction concept, storage engine and write-ahead log access interfaces definition.

4. Communication protocol framework. Building a network communication protocols would require a set of highly customizable serialization techniques and network packet flow control.

Such a solution is required to build API-compatible solutions, specified for a particular platform, with a required-only components enabled.

5. Actor Multithreading Model Framework

The majority of programs today is executed in environments of multiple processing units. A key challenge of program development is to appropriately aggregate resources for the sake of code performance, execution efficiency, and particular application needs. Multi-core CPUs have become an integral part of commodity hardware even in mobiles. Heterogeneous hardware components like graphics processing units (GPUs) and embedded controllers contribute powerful capacities to end systems, while novel computing paradigms arise in emerging distributed ecosystems like cloud computing [5, 10] and mobile crowd sourcing [21]. All these scenarios rely on concurrency [12], many also require distribution. Still the dominant part of current applications is written in some popular imperative language [28].

Imperative programming languages such as C, C++, which are going to be used for an implementation, do not provide language-level concurrency semantics. They were developed before the 'multi-core revolution' started and thus originally aimed at single-core processor machines. Threading libraries were added on top of existing languages that allow to start multiple threads of execution within a process. However, dealing with concurrency is challenging, especially in shared memory environments where parallel access to process-wide memory easily leads to race conditions. The performance and scalability of hand-written synchronization for avoiding race conditions depends on the implementation strategy. Coarse-grained locking is simple, but easily causes queuing and scalability issues, whereas fine-grained locking increases scalability but also complexity and error-proneness due to lock order, for example. Additionally, time-dependent errors make it virtually impossible to verify a concurrent application by systematic testing [13].

A powerful approach to the problems of concurrency and distribution has been formulated in the actor model by Hewitt, Bishop, and Steiger [16]. This formalism describes concurrent entities - 'actors' - that execute independently, do not share state, and communicate by asynchronous message passing. Because actors are self-contained and do not rely on shared resources, race conditions are avoided by design. The message passing communication style facilitates a transparent deployment and applies to (1) concurrency, if actors run on the same machine, (2) heterogeneous environments, if actors on the same machine are bound to different memory regions and processing units, and (3) distribution, whenever actors run on different hosts connected via the network. Actor-based languages like Erlang [3] and frameworks such as Akka [30] or Kilim [25] have been bound to specific niches or use vendor specific APIs (e.g., Casablanca [20]). One major objective of the present work is to make actor programming accessible to a wider community and to broaden its range of applications.

5.1. Shared-Nothing Architecture

Shared-nothing architecture is a distributed-computing architecture in which each node is independent and self-sufficient, and there is no single point of contention across the system. More specifically, none of the nodes share memory or disk storage. People typically contrast SN with systems that keep a large amount of centrally-stored state information, whether in a database, an application server, or any other similar single point of contention.

The advantages of shared-nothing architecture versus a central entity that controls the network (a controller-based architecture) include eliminating any single point of failure, allowing self-healing capabilities and providing an advantage with offering non-disruptive upgrade.

Such an architecure approach is intended to be used inside the particular database management system process because of high accesibility requirements.

6. Server Application Framework

Server application framework intention is to manage long-running applications, so it is required to have a mechanism to extend this application in runtime, some facilities to access paths, ensure single instance instantiation on system, manage and catch signals and so on are also required.

This work is recurrent each time that we need build and deploy an application for particular system, and the way to do this, change a lot on each of these systems.

For instance Windows operates with a concept of 'services' and on Unix-based systems (POSIX) have 'daemons' concept that is used to build long-running executable applications and these two APIs have no similarity. Thus, in this scenarios, is so much difficult to developer to get the application running on Windows or on POSIX as server without a lot of work. The work is harder if it is required to run thea same application in both systems.

Other problem raise when user want provide a way to extend the application using a plug-in mechanism. Like Service/Daemon, the shared modules (DSO) manipulation changes a lot on Windows and POSIX.

Obtain simple thing like paths, arguments, manipulate signal, can be annoying, since it also don't has a common interface to do this on both systems.

Server application framework used (Nil Application - `https://github.com/nil_foundation/nil_application.git`) aims to make significantly easier for the developer get the application running in cross-platform (POSIX/Windows) environment. Nil.Application provides a application environment, or start point for a basic infrastructure to build an system application on Windows or Unix Variants (e.g. Solaris, Linux, MacOS).

The =nil; Application framework uses behaviours modeled using 'aspects' concept proposed by 'Vicente J. Botet Escriba', that allow easy extension and customization of library components. The application modes uses these components internally to achieve the user desirable behaviours.

Nil Application intends to provide some following features:

- Run application as Windows Service;
- Run application as UNIX/POSIX Daemon;
- Plugin extension system;
- Process(executable) Single instance Instantiation support;
- Application SIGNAL/Callbacks customization;

7. Data Storage Framework

7.1. Data Structures Separation Concepts

=nil; DBMS architecture assumes strict data structure types separation by its' itentions concept usage. That means all the framework would consist, besides other parts, of three data management libraries:

1. Data storage management library
2. Storage operations processing library
3. Storage replication library

Such a set of components is required by the separation concept as presented below:

1. Runtime-defined serializable/deserializable data structures used for the data storage management. These ones are required for passing processed data to the storage engine.
2. Compile-time defined data structures used for definition of data management procedures.
3. Runtime-defined serializable/deserializable communication protocol data structures used for networking data transferring.
4. Compile-time defined communication protocol data structures used for managing the network-received/sent data.

Similar techniques are used inside the storage consisdency conditions definitions which is better described in 20.

Basically, such a separation comes from the idea of reflection mechanism, which actually generates these runtime-defined data structures automatically during the compilation stage. But such an approach makes it difficult to manage storage data structure migrations, communication protocol versioning or data management procedures changes without significant changes made all over the project.

7.2. Database Interface Unification

8. Communication Protocol Framework

The implementation of the binary communication protocols can be a tedious, time consuming and error-prone process. Therefore, there is a growing tendency among developers to use third party code generators for data serialization/deserialization. Usually such tools receive description of the protocol data layout in separate source file(s) with a custom grammar, and generate appropriate serialization/deserialization code and necessary abstractions to access the data. There are so many of them: ProtoBuf, Cap'n Proto, MessagePack, Thrift, Kaitai Struct, Protlr. All of these tools are capable of generating C++/Java/Python code. However, the generated code quite often is not performant enough to be used in limited-resource systems, especially bare-metal ones. Either the produced C++ code or the tool itself has at least one of the following limitations:

- Inability to specify binary data layout. Many of the tools use their own serialization format without an ability to provide custom one. It makes them impossible to use to implement already defined and used binary communication protocol.

- Inability to customise underlying types. Most (or all) of the mentioned code generating tools, which do allow customisation of binary data layout, choose to use `std::string` for string fields and/or `std::vector` for lists, as well as serialization/deserialization code is generated to use standard streams (`std::istream` and `std::ostream`). Even if such ability is provided, it is usually "global" one and do not allow substitution of types only for specific messages / fields.
- Limited number of supported data fields or limited number of their serialization options. For example, strings can be serialized by being prefixed with their size (which in turn can have different lengths), or being terminated with `\0`, or having fixed size with `0` padding if the string is too short. There are protocols that use all three variants of strings.
- Poor or weak description grammar without an ability to support conditional serialization/deserialization. For example, having a value (such as single bit in some bitmask field) which determines whether some other optional field exists or not.
- Lack of polymorphic interface to allow implementation of the common code for all the defined messages.
- When polymorphic interface with virtual functions is provided, there is no way to exclude generation of unnecessary virtual functions for a particular embedded application. All the provided virtual functions will probably remain in the final image even if they are not used.
- Lack of efficient built-in way of dispatching the deserialized message object into its appropriate handling function. There is a need to provide a separate dispatch table or map from message ID to some callback function or object.
- Lack of ability to override or complement the generated serialization code with the manually written one where extra logic is required.
- Complex and limited protocol description domain-specific language makes it required for the devleoper to learn another one language to describe the serialization/marshalling, but not the protocol workflow.

9. Pluggability

Plugin system is being widely used in various databases: ElasticSearch, MongoDB, AragnoDB (it is called `ApplicationFeatures` actually). It allows to configure the database behavior without recompiling the core. Extreme parametrization requirements comes with

10. Pluggable Storage Engines

11. Pluggable Virtualization Engine

12. Pluggable Networking

13. Virtualization

Extreme parametrization comes with extreme requirements. Various database node replication protocols, various data structures marshalling formats, various storage consistency conditions, various write-ahead log structures - databases are highly variative at all these params and more. The intention to introduce the open standard for database management systems components requires similarities generalization and factoring out and parameterizing the differencies.

A fine example is a write-ahead log data structure, which is usually a linked list of transactions. The basic requirement for the set internal structure, responsible for the transaction log, is considered

to be a partially ordered. That means lots of data structures are convenient for this position: backward-linked list, double-linked list, hashed trees, concatenation-homomorphic signed lists and trees, literally any set which is convenient for the partially ordered set requirements.

It is believed the reader should be fine with the partial ordered set definition, but below lays the reminder.

Definition 13.1. A partially ordered set is a set P with a binary relation $R \supseteq P \times P$ satisfying all of the following conditions.

- Reflexivity: $(x, x) \in R \forall x \in P$
- Antisymmetry: $(x, y) \in R; (y, x) \in R \Rightarrow x = y$
- Transitivity: $(x, y) \in R and (y, z) \in R \Rightarrow (x, z) \in R$

Pluggability of such differencies approach is widely used across various databases: AragnoDB, MongoDB, ElasticSearch. Inside the trusted context, when all the database nodes are located in a well-known cluster, or are highly localized in some physical place, when the trivial permission system like users with passwords/key access control is enough for the database not to be corrupted and there is a trusted source of verified plugins, which are guaranteed not to harm the operating system it works fine. But outside the trusted context, when even Byzantine fault tolerance conditions are not required to be satisfied, any plugin source code repository cannot be trusted. Inspection of every such a repository can be a time-consuming process. Futhermore, assuming every Nil DBMS database includes replication protocol definition, storage consistency conditions, set of state modifiers and lots of other executable pieces of paramertization for every particular database, audit can last forever. The solution is to isolate this parametrization and execute it in some virtualized environment for the user data and operating system safety.

Next subsections define the virtualized parametrization in terms of [2].

14. Storage Consistency Conditions Definition

Storage consistency conditions differs a lot in every particular DBMS, storage engine or a single-database management software. For instance in MongoDB, write operations are atomic at the document level, and no single write operation can atomically affect more than one document or more than one collection. A denormalized data model with embedded data combines all related data for a represented entity in a single document. This facilitates atomic write operations since a single write operation can insert or update the data for an entity. Normalizing the data would split the data across multiple collections and would require multiple write operations that are not atomic collectively.

However, schemas that facilitate atomic writes may limit ways that applications can use the data or may limit ways to modify applications.

Futhermore, MongoDB uses schema validation for every insert or update with a validation rule specialized for every collection. That guarantees the mailformed query would break the particular database node storage consistency, so it gets rejected. [3]

For example, the following example specifies validation rules using MongoDB JSON schema:

```
db.createCollection("students", {
   validator: {
      $jsonSchema: {
         bsonType: "object",
         required: [ "name", "year", "major", "gpa" ],
         properties: {
            name: {
```

```
                bsonType: "string",
                description: "must be a string and is required"
            },
            gender: {
                bsonType: "string",
                description: "must be a string and is not required"
            },
            year: {
                bsonType: "int",
                minimum: 2017,
                maximum: 3017,
                exclusiveMaximum: false,
                description: "must be an integer in [ 2017, 3017 ] and is required"
            },
            major: {
                enum: [ "Math", "English", "Computer Science", "History", null ],
                description: "can only be one of the enum values and is required"
            },
            gpa: {
                bsonType: [ "double" ],
                minimum: 0,
                description: "must be a double and is required"
            }
        }
    }
  }
})
```

Nil DBMS storage consistency conditions definition should be wiser. It should include the ability for the database to define the complex schema and current data storage state validation with turing-complete language because of the maintenance of wide variety of supported databases is required.

Considering the storage consistency conditions definitions could be categorized to several types:

1. On-transaction stateless conditions make sure the transaction data is consistent. A fine example - MongoDB schema validators.

2. On-transaction stateful conditions make the current storage state checks for the particular transaction before it could be considered applied. MongoDB Compass tool implements the declarative basic domain-specific language, which basically allows such a validation, but no turing-complete operations are available.

3. In-storage continious stateful and stateless checks makes all the database node change their state simultaneously.

This makes no questions in case of no extraoridnary performance demands are made - database architecture supposes the built-in interpreter in every node executes these definitions, whether they are defined in turing-complete language or not.

In case of time-consuming computing has to be made to decide if the transaction should be accepted or not, n-distanced executables are proposed to be used. The particular protocol resolving the n-distanced computing results depends on the database.

Stored executables are considered to be able to communicate with the outside via API, which can be an in-memory binary communication protocol (in case of 0-distanced executables) or a network/socket-based communication protocol, depending on the distance of the stored executable

(in case of n-distanced executables). Such a communication-type split is required to reduce latency within data-management tasks.

Following prototype consistency condition definition example in C++ would demonstrate the proposed.

14.1. Consistency Conditions Definitions Communication Protocol

First of all, stored executables are proposed to be a virtualized full-featured processes, just like it is in Unix systems. That means no custom initialization function is allowed in generic n-distanced case. That is why the generic API definition would be required for the preprocessing (AST-tree parsing-based presumably) could be done in a way convenient for the particular recommended execution distance.

```
/**
    @class api
    @brief Generic API class.

    Introspection toolchain would use this class to identify the consistency
    conditions API classes to generate the convenient for the selected execution
    distance communication way.

    @note This class intends to be included implicitly
*/
template<typename ApiType, typename ImplType>
class api : public ... {
    public:

    typedef ApiType api_type;
    typeder ImplType impl_type;

    ...

};
```

[1].

Next generic implicitly included part of the storage consistency definition includes the generic consistency definition way. Distance definition is made with a library-level mark for the preprocessing toolkit.

```
/**
    @class consistency
    @brief Represents consistency condition definition

    @note Platform-dependent primitives definitions could be replaced by
    more complex endiannes-independent types
*/
    template<template<uint32_t D> typename T = distance<D>>
    class consistency_condition<T<D>> {
    public:
        typedef T distance_type;

        virtual bool handle_value(const Type1 &value) override {
```

[1] More about proposed communication protocol definition way is described in [4]

```
        }

        virtual bool handle_value(const Type2 &value) override {

        }

        ...

        virtual bool handle_value(const TypeN &value) override {

        }

        ...

    };
```

Now particular-distanced consistency conditions are required to be defined. More about database state management way is described in **??**.

Following example roughly sketches the 0-distansed consistency condition gets transaction applied only and only if some other database has some named data structure with value stored equal to the value getting proposed in the transaction.

```
class condition_0 : public consistency_condition<distance<0>> {
    public:

    virtual bool handle_value(const Type1 &value) override {
        // Example transaction consistency condition dependent on the current
        // state of another database. Supposed to return true if selection went
        // well

        database &db = application.get_databases()["bitcoin"];

        return db.select<bitcoin::key>([&](const bitcoin::key &v) {
            v.balance == value.balance;
        }));
    }

    virtual bool handle_value(const Type2 &value) override {
        ...
    }

    ...

    virtual bool handle_value(const TypeN &value) override {
        ...
    }
};

class condition_1 : public consistency_condition<distance<1>> {
    public:

    virtual bool handle_value(const Type1 &value) override {

    }

    virtual bool handle_value(const Type2 &value) override {

    }
```

```

    ...

    virtual bool handle_value(const TypeN &value) override {

    }
};

class condition_42 : public consistency_condition<distance<42>> {
    public:

    virtual bool handle_value(const Type1 &value) override {

    }

    virtual bool handle_value(const Type2 &value) override {

    }

    ...

    virtual bool handle_value(const TypeN &value) override {

    }
};
```

```
/**
    @class consistency_api
    @brief Represents consistency conditions definition communication API.

    This class represents the stored executable communication API and depending
    on the executable distance set for the particular consistency condition
    (resolved as consistency_condition::distance_value) it would be preprocessed
    as binary in-memory or network communication protocol.

    @note templated api class is required for the reflection engine to be able
    to introspect class members definition.
*/
class consistency_api : public api<consistency_api, consistency_0> {
    public:

    consistency_api(const impl_type *input_impl) : impl(input_impl) {

    }

    template<typename T>
    void handle_value(T value) {
        impl.handle_value(value);
    }

    template<typename T>
    T get_value(const &T) {
        return impl.method();
    }

    protected:

    impl_type *impl;
```

```
};
```

```
class consistency_api : public api<consistency_api, consistency_1> {
    public:

    consistency_api(const impl_type *input_impl) : impl(input_impl) {

    }

    template<typename T>
    void handle_value(T value) {
        impl.handle_value(value);
    }

    template<typename T>
    T get_value(const &T) {
        return impl.method();
    }

    protected:

    impl_type *impl;
};
```

14.2. 0-distanced Executables

0-distanced executables usage deserves the separate subsection because of the data storage integration and interaction way is slightly different from outside of the process execution devices.

15. Communication Protocol Definition

Inspite of storage consistency conditions can be defined as n-distanced executables $\forall\ n \in \mathbb{Z}_+$, database node replication protocol transport definition is bounded to the 0-distanced ones, because of this being a basement for the network-replication process.

16. Data Consistency

17. ACID Compilancy

18. MVCC

19. Authenticated Data Structures Write-Ahead Log

Recent introduction of merkle-tree hashed write-ahead log databases introduces requirements to make the write-ahead log parametrization available.

20. Data Management

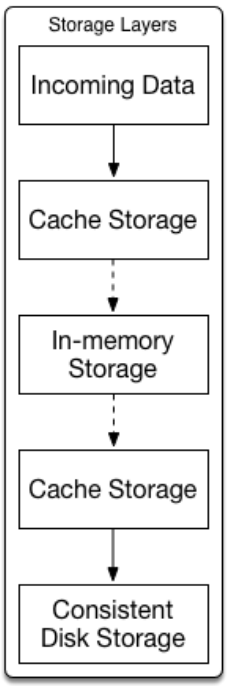

Figure 1: *4-layered storage*

Database storage engines and corresponding data management ways are used to get built around the data concept the database management system getting them used, which is a fine way to create a single-purposed database management system. Fine example is a ClickHouse database [5], which is designed to be a column-oriented database suitable for analytics processing. Coming to the set of storages, which intention is to be used for multiple different types of data, several types of solutions comes up:

- Use the compromise database with compromise-performanced data storage engine (p.e. MongoDB [3]) and to be sure the performance of processing any type of data is a trade-off.
- Use the multi-model database, which (p.e. FoundationDB [6] or AragnoDB [7] which "cast" different model types into a key-value storage (AragnoDB casts the data to RocksDB storage). But such an approach means significant performance drop. FoundationDB SQL layer performance is, for example, twice as lower than MySQL [8].

So that is why Nil DBMS intends to be a multi-level storage database with customizable storage engines. Multi-level storage means ordered storage engine sequence, organized to provide the highest response rate possible. Such layers could be a caching layer (p.e. LRU-caching), consistent in-memory storage, on-disk reduced dataset caching layer, consistent on-disk storage. An example of 4-layered data storage with caching in-memory storage, consistent in-memory storage, caching on-disk storage and consistent on-disk storage at the end is presented in ??.

Such a levelling is only applicable to the particular database - it is basically a part of database configuration. Such an approach leads to the composite database concept.

21. Composite Database

Coming to database management systems, almost every of such a system has its' own database concept. Most of these concepts mean this is an abstraction over database scheme, data itself, write-ahead log in predefined format, data integrity conditions definitions, sometimes it also includes users authorization data. Composite database concept extends the common database term definition. It proposes to:

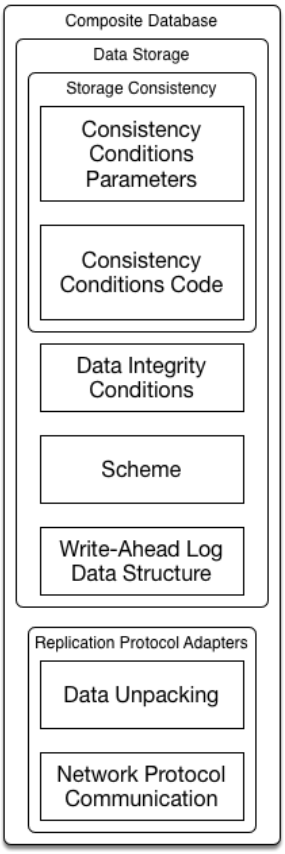

Figure 2: *Composite Database*

- Store and process database user permissions using the same storage engine and with the same techniques as the business-data itself. This leads to the extension of available authorization ways set and allows developers to define custom user authentication techniques (p.e. cryptographic user authentication schemes).
- Store and process database consistency conditions definition using the same storage engine and with the same techiniques as the business-data itself.
- Store and process database replication instructions using the same storage engine and with the same techniques as the business-data itself. These replication instructions intended to be implemented as adapters of several types:
  - Particular data-handling software transport protocol adapters: p.e. MongoDB daemon adapter or bitcoind daemon adapter.
  - Particular data scheme replication instructions: p.e. consistency conditions required for the currently replicating transaction to be applied.
- Store and process as database metadata storage scheme/pseudo-scheme.
- Store and process as executable in virtualized enviroment database metadata write-ahead log data structure definition. Most of databases do not separate the write-ahead log definition from the particular storage engine used, but such a factoring out allows to handle authenticated write-ahead log databases (p.e. bitcoin database).

22. Data Structures Separation Concepts

Nil DBMS uses the concept of a strict data structure types separation:

1. Runtime-defined serializable/deserializable data structures used for the data storage management.
2. Compile-time defined data structures used for definition of data management procedures.
3. Runtime-defined serializable/deserializable communication protocol data structures used for networking data transferring.

4. Compile-time defined communication protocol data structures used for managing the network-received/sent data.

Basically, such a separation comes from the idea of reflection mechanism, which actually generates these runtime-defined data structures automatically during the compilation stage. But such an approach makes it difficult to manage storage data structure migrations, communication protocol versioning or data management procedures changes without significant changes made all over the project.

Following rough and simple example would make it clear:

```
class A {
public:
    int64_t a;
    char *b;
}

template<typename T, ...>
class static_reflection_structure {
public:
    ...
}

template<>
class static_reflection_structure<A, ...> {
    ...

    const static uint16_t member_count = 2;

    ...
}

template<typename T, typename String, typename SequenceContainer>
class dynamic_reflection_structure : public some_base<...> {
public:
    ...

    virtual String get_typename() override {
        return "T";
    }

    uint16_t get_member_count() {

    }

    SequenceContainer<String> get_members_list() {

    }

    ...
};

template<typename String, typename SequenceContainer>
class runtime_structure<A> : public some_base<...> {
public:
    ...

    /**
        In case of runtime-defined data identifier would be
        required to change - it can be changed to any string still being a
```

```
            representative class for A
            */
            virtual String get_typename() override {
                return "A";
            }

            uint16_t get_member_count() {
                return 2;
            }

            ...
        };

```

It is assumed for every data structure to have a default-generated such a reflected meta-information structure set until the specific flag was passed to the compilation suite. (Note, it is a compilation suite because of no modified compiler intends to be used, only preprocessing and code pre-generating). In case of special serializable data structure requirements this can be set explicitly without using code generation.

Such a meta-information intends to be used in virtualized executables and in the particular framework definitions as well.

23. No Query Language

Inspite of traditional approach to data management inside the DBMSs' with custom domain-specific language interpreters (some of them, p.e. SQL, went too far and got standartized), Nil DBMS proposes the turing-complete language data accessors/mutators definition.[2] Accessing or mutating data intends to be implemented at library-level in combination with code introspection. Such an approach enables the database user to perform cross-database access or mutation (select or insert).

Following example would demonstrate the selection:

```

#include <nil/database/select.hpp>
#include <nil/container/sequence.hpp>

...
class data_object ... {...}; ///< dynamically reflected object which
runtime-used metadata can be set automatically (to match the actual
programming-language semantics metadata) or manually to make it convenient for
optimizations and concept-separation.
...

... function() {
    ... &website_database = application.db["website"];
    website_database.select<data_object>.get<by_id>([&](... &comparison_object) {
        if (comparison_object.id > 50) {
            if (comparison_object.field == some_other_value) {
                ...

                // Here follows any other turing-complete selection conditions

                ...
```

[2] More about NoDSL protocol approach can be found in [4]

```
                return true;
            }
        }
    });
}

```

Example provided assumes the lambda passed to `get<by_id>()` would return `true` if the object considered in the index is convenient to be returned and `false` if it is not.

Such a data management way, in case of the selection being performed as 0-distanced executable, intends the data contained in the index `by_id` (which is also an automatically of manually introspected object, actually) to be copied into virtualized environment (kind of MVCC-session initialization) or accesed via user-space shared memory-like mechanism. In case of the selection gets performed as n-distanced executable, an extensive introspection of a selector function intends to be performed, and the actual computation gets split between (cluster?) nodes connected.

Following example demonstrates the insertion or mutation of the indexed data storage with cryptographically-authenticated transactions:

```

#include <nil/database/insert.hpp>
#include <nil/database/modify.hpp>

void operation_example() {

}

...
/**
    Dynamically reflected object which runtime-used metadata can be set
    automatically (to match the actual programming-language semantics metadata)
    or manually to make it convenient for optimizations and concept-separation.
*/
class data_object ... {...};

...

... function(... params) {
    // A database identifier of any other type can be used, it is not insisted
    // to be a string
    ... &bitcoin_database = application.db["bitcoin"];

    bitcoin::transaction trx;
    trx.operations = {[&](

    bitcoin::signed_transaction strx = trx.sign(params.key_imported);
    bitcoin_database.push_transaction(strx); ///< Such an example would be fine
    in case of prepared and pre-signed transaction availability.

    ... &website_database = application.db["website"];

    website::transaction trx = {.
}


```

24. Query Sharding

25. Performance

References

1. Codd E. F. A Relational Model of Data for Large Shared Data Banks // Commun. ACM. New York, NY, USA, 1970. Vol. 13, no. 6. p. 377–387. URL: https://doi.org/10.1145/362384.362685.
2. Komarov M. Storage-Centric Computing Architecture. 2018. URL: https://github.com/nemo1369/nil_foundation/dbms/execution_architecture/main.pdf.
3. Inc. M. Data Models. https://docs.mongodb.com/manual/data-modeling/. 2018. [Online; accessed 19-June-2018].
4. Komarov M. NoDSL Communication Protocol Approach. 2018. URL: https://github.com/nemo1369/nil_foundation/libs/protocol/main.pdf.
5. LLC Y. ClickHouse. https://clickhouse.yandex. 2016. [Online; accessed March 3, 2021].
6. Inc. A. FoundationDB. https://foundationdb.org. 2018. [Online; accessed March 3, 2021].
7. Inc. A. AragnoDB. https://aragnodb.com. 2018. [Online; accessed March 3, 2021].
8. Hugg J. FoundationDB's Lesson: A Fast Key-Value Store is Not Enough. https://www.voltdb.com/blog/2015/04/01/foundationdbs-lesson-fast-key-value-store-not-enough/. 2015. [Online; accessed March 3, 2021].
9. Kelner J. A., Maymounkov P. Electric routing and concurrent flow cutting // CoRR. 2009. Vol. abs/0909.2859. URL: http://arxiv.org/abs/0909.2859.
10. Native Actors – A Scalable Software Platform for Distributed, Heterogeneous Environments / D. Charousset, T. C. Schmidt, R. Hiesgen et al. // Proc. of the 4rd ACM SIGPLAN Conference on Systems, Programming, and Applications (SPLASH '13), Workshop AGERE! New York, NY, USA: ACM, 2013. Oct. P. 87–96.
11. Charousset D., Hiesgen R., Schmidt T. C. Revisiting Actor Programming in C++ // Computer Languages, Systems & Structures. 2016. Vol. 45. P. 105–131. URL: http://dx.doi.org/10.1016/j.cl.2016.01.002.
12. Komarov M. Storage-Centric Computing Architecture. 2018. URL: https://github.com/nemo1369/nil_foundation/dbms/execution_architecture/main.pdf.
13. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging / C. Mohan, D. Haderle, B. Lindsay et al. // ACM Trans. Database Syst. New York, NY, USA, 1992. Vol. 17, no. 1. p. 94–162. URL: https://doi.org/10.1145/128765.128770.