International Journal of Geo-Information

MDPI

Article

A Generic and Extensible Core and Prototype of Consistent, Distributed, and Resilient LIS

Zdravko Galić * and Mario Vuzem

Faculty of Electrical Engineering and Computing, University of Zagreb, Unska 3, 10000 Zagreb, Croatia; mario.vuzem@fer.hr

* Correspondence: zdravko.galic@fer.hr; Tel.: +385-1-6129-531

Received: 10 June 2020; Accepted: 9 July 2020; Published: 13 July 2020

Abstract: The majority of the existing land information systems (LIS) are centralized, transaction processing systems based on object-relational database management systems for data storage, management, and retrieval. These traditional database management systems are dominantly based on a share-everything or share disk architecture and face challenges in meeting the performance and scalability requirements of distributed, data-intensive systems, including LIS. They support vertical, rather than horizontal scalability, which is of particular importance in distributed systems. In some cases, due to legal, administrative, or infrastructure constraints, LIS need to be distributed rather than centralized systems. Distributed computing systems and share-nothing architecture have become very popular, including new data processing platforms and frameworks with horizontal scalability and fault tolerance capabilities. In this paper, we present cdrLIS—a generic and extensible core of LIS based on relevant international standards and the NewSQL database management system (DBMS) that enables the implementation of consistent, distributed, highly-available, and resilient LIS. A generic core is implemented in the Go programming language and can be easily extended and adopted towards the implementation of a specific country profile. cdrLIS can be deployed either on a computer cluster or on cloud computing platforms and thus support the design and building of a new generation of distributed and resilient data-intensive applications and information systems in the land administration domain.

Keywords: distributed database; distributed SQL; Go; LADM; LIS; NewSQL

1. Introduction

The majority of the existing land information systems (LIS) are dominantly centralized, on-line transaction processing (OLTP) systems, built on mission-critical relational or object-relational database management systems (DBMS) such as Oracle, Microsoft SQL Server, IBM DB2, MySQL, and PostgreSQL. These traditional DBMS have been the key technology for data storage, retrieval, and management. Although all of them have a distributed version, they were designed in a different era when hardware characteristics were much different from today: (i) processors are thousands of times faster; (ii) memories are thousands of times larger; and (iii) disk volumes have increased enormously. These DBMS have been criticized for their "one size fits all" paradigm, i.e., the fact that the traditional DBMS architecture has been used to support data-intensive applications in different domains and with widely varying capabilities and requirements [1].

The modern computing environment is becoming largely distributed; enterprises and institutions have geographically distributed and interconnected data centers, forming distributed systems. Similarly, due to legal, administrative, or infrastructure constraints in the land administration domain, there is a need to design and implement distributed rather than centralized LIS. However, traditional share-everything DBMS are not designed to take advantage of distributed computing and

face serious problems in meeting the scalability, consistency, resiliency, and performance requirements of distributed data-intensive applications and systems. They only support vertical (scale-up) rather than horizontal (scale-out) scaling, which is the prerequisite for building a new generation of distributed LIS.

Land administration (LA) agencies often purchase and install additional CPUs and memory at database servers to get more power from these systems. However, that standard approach to vertical scaling is costly, requires expensive hardware and DBMS upgrades, as well as adds development complexity and increases overhead and maintenance costs. The crucial problem with vertical scaling is that the cost growth is not linear: a machine with twice as much CPU, RAM, and disk capacity as another usually costs much more than double as much, but due to bottlenecks, it cannot handle double the workload. Therefore, this architecture makes it complicated to build distributed LIS that deliver the expected resilience, consistency, and maintainability. What is actually needed is a distributed DBMS that scales-out linearly and limitlessly by adding new commodity servers to a share-nothing (share-nothing cluster—a number of nodes that do not share particular resources, most notably disk and memory) DBMS cluster.

During the last two decades, the advances in web technology, mobile devices, and IoT have resulted in the explosion of structured, semi-structured, and unstructured data. Consequently, building data-intensive applications and systems imposed a variety of requirements on DBMS, including: (i) horizontal scalability; (ii) high availability/fault tolerance, i.e., resilience; (iii) transaction reliability to support strongly consistent data; and (iv) database schema maintainability. The fact that achieving these mutually exclusive requirements through the traditional DBMS is very difficult or even impossible [2] triggered the development of NoSQL DBMS.

One of the key features of NoSQL DBMS is neglecting ACIDtransactions and the relational model in favor of eventual consistency. Although eventual consistency enables high availability, omitting strong (ACID) consistency support and eliminating SQL resulted in moving back to a low-level DBMS programming interface, thus significantly increasing the application complexity to handle potential inconsistent data [3]. If a network partition occurs, NoSQL DBMS will return the query result, even if the result is not the most current at that given time instant. Therefore, it is not surprising that NoSQL DBMS have not been relevant for many enterprise applications and systems, including LIS, simply because these applications and systems cannot give up strict transaction and consistency requirements.

A transaction is a unit of consistent and reliable read and write operations. LIS is a typical OLTP system, and the transaction services of an underlying data management system must provide ACID properties [4,5]:

- Atomicity ensures the all-or-nothing execution of transactions. In other words, transactions are atomic, and each transaction is treated as a single "unit" that either completely succeeds or completely fails.
- Consistency indicates that a transaction execution can only bring the database from one valid state to another.
- Isolation refers to the fact that each transaction must appear to be executed as if no other transaction is executing at the same time, i.e., the effects of concurrent transactions are shielded from each other until they are committed.
- Durability guarantees that the effects of a committed transaction on the database are permanent and must never be lost.

Distributed systems, cluster computing, and the share-nothing architecture became very popular during the last decade, including many Big Data processing platforms/frameworks with scalability and fault tolerance capabilities. They use large amounts of commodity hardware to store and analyze big volumes of data in a highly distributed, scalable, and cost-effective way. These new systems (Apache Hadoop [6], Apache Spark [7], and Apache Flink [8], among others) are optimized for

massive parallel data intensive computations, including their extensions and adaptations for spatial and spatio-temporal data: SpatialHadoop [9,10], GeoSpark [11], and MobyDick [12]. Unfortunately, these systems do not have transaction services and are therefore not suitable for building OLTP systems, such as LIS.

Not surprisingly, NewSQL—a new class of modern relational DBMS—has emerged. The NewSQL DBMS provide high throughput and performances as do NoSQL DBMS, guarantee strict ACID transactions, and preserve the relational model including SQL (Table 1). These features and capabilities enable applications to execute a large number of concurrent transactions using SQL, and developers do not have to write logic to deal with the eventual consistency as they would in a NoSQL DBMS [3,13].

Table 1. Traditional SQL vs. NoSQL vs. NewSQL database management systems (DBMS).

	Pros	Cons
Traditional SQL	• SQL • ACID	• Do not scale horizontally • Vertical scaling requires expensive HW, replication, and sharding the database for peak volumes
NoSQL	• Horizontal scalability • Availability	• Sacrificed ACID • Low-level programming interface • Data management and consistency are embedded in application logic
NewSQL	• SQL • ACID • Horizontal scalability	

In some cases, due to legal, administrative, or infrastructure constraints, the LIS needs to be designed and implemented as a distributed rather than centralized system. However, there is no (published) evidence that state-of-the-art distributed DBMS, in particular NewSQL, are successfully applied in building distributed LIS. The work presented in [14] also confirmed that although adoption of distributed NoSQL/NewSQL DBMS in many sectors is occurring, scaled uptake in building LIS is negligible.

The Land Administration Data Model (LADM) [15,16] is a well-known international standard for the domain of land administration. It is a conceptual, generic domain model for designing and building LIS and has been already extended and adapted to a number of particular profiles. The model covers the basic data related components of land administration: (i) party related data; (ii) data on rights, restrictions, and responsibilities (RRR) and the basic administrative units where RRR apply; and (iii) data on spatial units and on surveying and topology/geometry. LADM provides an extensible basis for the development and refinement of LIS, based on a model driven architecture (MDA). Several LADM-based country profiles have been developed, but only a limited number of them have been implemented and are operational [17]. Recently, there have been activities to build prototypes in the 3D domain, but these prototypes were not generic and focused primarily on either visualizing 3D spatial units [18–21] or, to a lesser extent, mapping the Unified Modeling Language (UML) conceptual model to database schema or data exchange format [22].

In general, most of the LIS are developed from scratch (we do not consider the adaptation of an already existing LIS to a specific country context relevant to our discussion), and to the best of our knowledge, there is no a generic LADM-compliant software library that can be extended and reused in developing LIS. In this paper, we present our on-going development of a generic and extensible prototype of consistent, distributed, and resilient LIS compliant with the LADM standard. The novelty of our approach is that it is designed from the ground up in a general and comprehensive way to support building a new class of LIS based on LADM.

The main purpose and contribution of this paper are twofold:

(i) To present generic and extensible LADM-compliant software packages implemented in the Go programming language. The packages can be reused and extended in developing LIS, either centralized or distributed.

(ii) To demonstrate the implementation of a single-site distributed LIS using a state-of-the-art NewSQL DBMS. The implementation of geo-replicated distributed LIS using a spatially enabled NewSQL DBMS is straightforward.

The rest of the paper is structured as follows. In Section 2, we discuss the main features of distributed and NewSQL DBMS that are crucial for designing and building a new generation of consistent, highly available, and resilient land information systems and applications. A general overview of our generic and extensible LADM-compliant core library, including some specific implementation in the Go programming language, is shown in Section 3. Section 4 presents the prototype architecture and ensuring resilience, reliability, and consistency. Section 5 concludes the paper and discusses future directions.

2. State-of-the-Art: Distributed and NewSQL DBMS

Before proceeding further, this section provides a brief overview of the state-of-the-art distributed and NewSQL DBMS, as well as their main features and importance for building a new generation of consistent, distributed, and resilient LIS.

A distributed system is a collection of autonomous computing elements (nodes) that appear to its users as a single coherent system. A node can be either a hardware device or a software process, and modern distributed systems consist of all kinds of nodes, ranging from very big high-performance computers to small plug computers [23]. Recently, distributed computing systems and the share-nothing architecture became very popular, including new data processing platforms/frameworks with scalability and fault tolerance capabilities. These new systems/frameworks use large amounts of commodity hardware to store, manage, and analyze data in a highly distributed, scalable, and cost-effective way.

The primary goal of a distributed system is conceptually simple: it should ideally be just a fault-tolerant and more scalable version of a centralized system. A distributed system should preserve the simplicity and consistency of a centralized system, leverage distribution and replication to boost high availability and fault tolerance and resilience by masking failures, provide scalability, and reduce latency [24].

There are several reasons for building distributed LIS, i.e., to distribute an LIS database across multiple nodes:

(i) Distribution and autonomy of land administration units: Decentralized land administration units/offices are administratively and geographically distributed. Each unit/office may have the authority to create its own local data, which it wants to control.
(ii) Data communications costs and reliability: Although the data communication costs have decreased recently, the cost of transmitting large amounts of spatial data across communication networks or handling a large volume of transactions from remote offices/units can be expensive. In such cases, it is more economical to locate data and applications close to where they are needed. In addition, dependency on data communications always involves a risk, so placing data geographically close to users can be a reliable way to support the fast access to data [25].
(iii) Scalability: If the data volume or transaction workload grows bigger than a single node can handle, the data and workload can be distributed across multiple nodes.
(iv) High availability/fault tolerance/resilience: LIS should be highly available, fault-tolerant, and resilient systems. Whenever a failure occurs, i.e., whenever a part of the system fails, the system should be available and continue to operate correctly.

A distributed database is a collection of multiple, logically interrelated databases located at the nodes of distributed systems. A distributed DBMS is the software system that manages the distributed database and makes distribution invisible to the users; the users view a unified database, while underlying data are physically distributed across the nodes. An important feature of distributed

DBMS is that it is logically integrated, but physically distributed, i.e., it is a single logical database that is physically spread across nodes in multiple locations connected by a data communications network. A distributed database system refers jointly to the distributed database and the distributed DBMS [5]. Consequently, a distributed LIS is an LIS built on a distributed database system.

There are two possible types of distributed LIS/DBMS:

(i) Single-site, typically characterized by a computer cluster in one data center.
(ii) Geo-distributed; the sites (i.e., data centers) are connected by wide area networks (WAN).

It should be emphasized here that a centralized LIS can be built up using a single-site distributed DBMS. Moreover, it is possible to build a distributed LIS that has multiple single-site clusters interconnected by a WAN.

There are two standard ways to distribute data across multiple nodes [26]:

(i) Replication: a fundamental concept of distributed systems for achieving trusted decentralization, low-latency access to data, scalability, high availability, fault tolerance, and resilience. Replication involves creating and distributing copies of the same data across the nodes and ensuring their consistency.
(ii) Partitioning: splitting a big dataset into partitions, distributing them across the nodes, and ensuring their consistency. The main motivation for partitioning is scalability: different partitions can be stored on different nodes in a share-nothing cluster, and the query load can be distributed across many nodes.

The NewSQL DBMS combine replication and partition so that copies of each partition are stored on multiple nodes. The state-of-the-art NewSQL DBMS share the following common features: (i) relational data model and distributed SQL; (ii) strong consistency via ACID transaction; (iii) horizontal scalability using data partition in share-nothing clusters of commodity machines; and (iv) high availability and resilience through data replication. They include (alphabetically): ClustrixDB [27], CockroachDB [28], FaunaDB [29], HyPer [30], MemSQL [31,32], NuoDB [33], SAP HANA [34], Spanner [35], TiDB [36], VoltDB [37], and YugabyteDB [38]. They are based on distributed architectures that operate on share-nothing clusters and have components to support consistency, high availability, fault tolerance, and resilience, as well as distributed SQL.

This is very important because it allows the DBMS to send the query to the data rather than the traditional approach of bringing the data to the query, which results in significantly less network traffic. Finally, most of the techniques that NewSQL DBMS apply have existed in traditional, either academic or commercial, DBMS. However, they were only implemented one-at-a-time in a single DBMS and never all together. What is really innovative about NewSQL DBMS is that they incorporate and implement these techniques into single platforms [13].

However, only a few of them support the object-relational model and geospatial data types (SAP HANA and YugabyteDB, the former being a COTS and the latter an open-source DBMS; MemSQL supports a very limited set of geospatial data types and operations based on a spherical model similar to Google Earth), thus making them potentially applicable in designing and building a new generation of LIS. We have already witnessed gradual geospatial support in traditional DBMS. Therefore, we can expect that in the future, more and more NewSQL DBMS will use the object relational model and thus directly support the building of a new generation of distributed LIS (either geo-distributed or single-site) with a distinguished set of features, of which we highlight strong consistency, horizontal scalability, and resilience/reliability/high availability.

It should be noted here that it would be possible to build a consistent, distributed, and highly available LIS using a distributed version of traditional DBMS (Oracle, DB2, SQL server) and an additional set of products. However, the means by which this can be achieved are much more complex and expensive. That approach requires a number of additional technological layers, and still, the system can allow potentially costly anomalies in the database and may have horizontal scalability

issues. As an illustration, a distributed and highly available Oracle-based system typically requires a number of additional licensed products: real application cluster (RAC)/active data guard, global data services, and sharding, which significantly increase both system complexity and cost. On the contrary, NewSQL DBMS support and provide all necessary concepts and techniques out-of-the-box (and in our case, as an open source database system), do not require additional expensive technologies and complex maintenance, and significantly increase the return on investment.

3. Generic and Extensible LADM Packages

In accordance with the model driven architecture (MDA) principles, we use the Go programming language [39] for building a generic and extensible platform-specific model. Unfortunately, MDA modeling and development tools (like Sparx Enterprise Architect, Rational Software Architect, etc.) do not support PIM to PIS transformation for the newer programming languages: Go, Scala, Kotlin, and Rust, among others. Go is an open source programming language for building simple, reliable, and efficient software [40] that could take advantage of distributed systems, but has a radically different approach to object-oriented programming.

What makes the object-oriented paradigm in Go different from class-based object-oriented programming languages (like Java, C++, Objective-C) is that it does not explicitly support either classes or inheritance (is-a relationship). When object-oriented programming became popular, inheritance was considered as one of its biggest advantages. However, after a few decades, it turned out that inheritance had some serious drawbacks when it came to maintaining large systems. Consequently, instead of using both aggregation and inheritance like most other object-oriented languages, Go supports the creation of custom types and aggregation (has-a relationships) via embedding. Here is an example of embedding VersionedObjectas an anonymous field inside LASpatialUnit:

```
type VersionedObject struct {
        BeginLifespanVersion time.Time
        EndLifespanVersion *time.Time
        Quality     *metadata.DQ_Element
        Source      *metadata.CI_ResponsibleParty
}

type LASpatialUnit struct {
        common.VersionedObject
        ExtAddressID     *external.ExtAddress
        . . .
        ReferencePoint   *geometry.GMPoint
        SuID     common.Oid
        SurfaceRelation          *LASurfaceRelationType
        RelationSu       []LARequiredRelationshipSpatialUnit     // relationSu
        . . .
        Baunit           []SuBAunit                      // suBaunit
        SuHierarchy      *SuHierarchy            // suHierarchy
        SpatialUnitGroups        []SuSuGroup                     // suSuGroup
        MinusBfs         []BfsSpatialUnitMinus           // minus
        PlusBfs          []BfsSpatialUnitPlus            // plus
}
```

Operations defined on the LA_SpatialUnit UML class are implemented as methods whose receiver is an aggregate type LASpatialUnit:

```
func (su LASpatialUnit) AreaClosed() bool {
 closed, _ := geos.Must(su.Boundary()).IsClosed()
 return closed
}

func (su LASpatialUnit) ComputeArea() LAAreaValue {
 var av LAAreaValue
```

```
 multiSurface := su.CreateArea()
 area, _ := multiSurface.Area()
 av.AreaSize, av.Type = Area(area), CalculatedArea
 return av
}

func (su LASpatialUnit) CreateArea() *geometry.GMMultiSurface {
 msBoundary := geos.Must(su.Boundary())
 tempMultiSurface := geos.Must(geos.EmptyPolygon())
 var ms []*geos.Geometry
 nGeometry, _ := msBoundary.NGeometry()
 for i := 0; i < nGeometry; i++ {
  . . .
  ms = append(ms, surface)
  tempMultiSurface = geos.Must(tempMultiSurface.Union(surface))
 }
 multiSurface := geos.Must(tempMultiSurface.Clone())
 for _, surface := range ms {
  if related, _ := surface.RelatePat(multiSurface, "2FF1FF212"); related {
   multiSurface = geos.Must(multiSurface.Difference(surface))
  }
 }
 return &geometry.GMMultiSurface{GMObject:
          geometry.GMObject{Geometry: *multiSurface}
          }
}
```

Consequently, LALegalSpaceBuildingUnit

```
type LALegalSpaceBuildingUnit struct {
 common.VersionedObject
 SpatialUnit                *LASpatialUnit
 ExtPhysicalBuildingUnitID  *external.ExtPhysicalBuildingUnit
 Type           *LABuildingUnitType
}
```

i.e., the named aggregated field SpatialUnit, can use the methods defined on LA_SpatialUnit.

A specific aspect of object-oriented programming in Go is that interfaces, values, and methods are kept separate: interfaces are used to specify method signatures; structure types (structs) are used to specify aggregated and embedded values; and methods are used to specify operations on custom types. There is no explicit connection between a custom type's methods and any particular interface—but if a type satisfies an interface, i.e., possesses all the methods the interface requires, that type is considered an instance of that interface [40,41].

This approach is more flexible than traditional inheritance, as each object is loosely coupled, and changes to one type do not really cause dramatic changes down the line. In essence, compositions combined with interfaces meet all the demands of an object-oriented system, without the complexity and limitations of inheritance.

We used the CRUDer interface type (and followed the Go convention for interface names, which is that they should end with "er") to express abstractions about the behaviors of all types in the context of CRUD operations at the persistence layer:

```
type CRUDer interface {
 Create(value interface{}) (interface{}, error)
 Read(where ...interface{}) (interface{}, error)
 ReadAll(where ...interface{}) (interface{}, error)
 Update(value interface{}) (interface{}, error)
 Delete(value interface{}) error
}
```

All LADM/cdrLIS concrete CRUDs in the context of object-relational mapping are instances of the CRUDer interface, i.e., they satisfy the interface by possessing all the methods the interface requires. Here is an example for the Create operation of LASpatialUnit:

```
type LASpatialUnitCRUD struct {
 DB *gorm.DB
}

func (crud LASpatialUnitCRUD) Create(spatialUnitIn interface{}) (interface{},error) {
 tx := crud.DB.Begin()
 spatialUnit := spatialUnitIn.(ladm.LASpatialUnit)
 spatialUnit.ID = spatialUnit.SuID.String()
 currentTime := time.Now()
 spatialUnit.BeginLifespanVersion = currentTime
 spatialUnit.EndLifespanVersion = nil
 writer := tx.Set("gorm:save_associations", false).Create(&spatialUnit)
 if writer.Error != nil {
  tx.Rollback()
  return nil, writer.Error
 }
 commit := tx.Commit()
 if commit.Error != nil{
  return nil, commit.Error
 }
 return &spatialUnit, nil
}
```

Because the CRUDer interface should be generic, arguments of the methods are anonymous interfaces. The concrete implementation of CRUDer operates on the concrete object, so the input data must first be cast to a concrete object type. The prepared object is persisted in the database by calling the DB...Create() method, and all associated objects are ignored by setting the GORMoption save_associations to false before calling DB...Create(). The creation and persistence of associated objects is accomplished by their corresponding CRUDers.

Our generic core relies on gogeos packages [42] to manage and use geospatial data types and operations on them. The gogeos packages provide bindings to the GEOS C++ library [43] that implements the geometry model and API according to the OGC Simple Features Specification for SQL and [44].

The LADM conceptual model, described as a set of UML class diagrams, Go language, and NewSQL DBMS are grounded in different paradigms—the former two in object-oriented, the latter in relational. Each technology requires that those who use it have a particular view of a universe of discourse. Incompatibilities between these views manifest as problems of an object-relational impedance mismatch. An object-relational application combines artifacts from both object-oriented and relational paradigms, and software development requires the resolution of impedance mismatch problems [45]. In order to isolate the object-oriented paradigm from the relational paradigm (i.e., from the need to understand the SQL language and the LADM schema of a NewSQL database), we used the object-relational mapping package GORM [46].

Here, we only scratched the surface of our generic and extensible Go software packages; the implementation details and source code are available at cdrLIS's GitHub site [47].

4. Architecture, Resilience, and Consistency

4.1. Entity-Relationship Logical Data Model

Figure 1 shows the cdrLIS entity-relationship (ER) logical data model consisting of core LADM entities and relationships between them. As stated in Section 3, the current version of our prototype is focused on 2D, so the 3D-relevant entities are currently omitted.

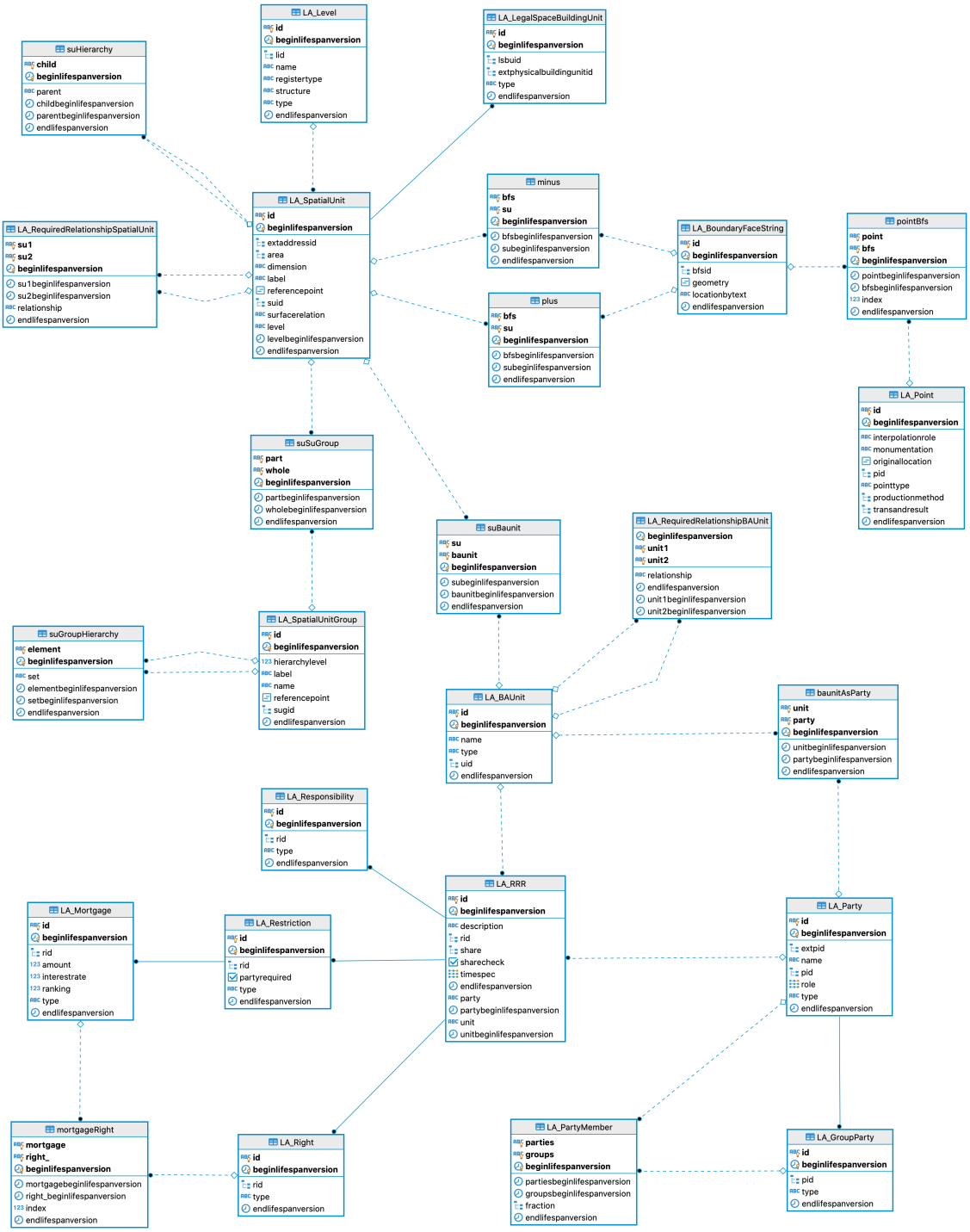

Figure 1. Entity-relationship (ER) model of cdrLIS.

Furthermore, for the sake of simplicity, entities LA_AdministrativeSource, LA_SpatialSource, LA_LegalSpaceUtilityNetwork, as well as attribute data types are not shown.

4.2. Software Architecture

The cdrLIS software architecture consists of four layers, which are encapsulated in four corresponding Go packages: (i) main application layer; (ii) http handlers layer; (iii) CRUD layer; and (iv) model layer. We describe them very briefly here.

(i) Main application layer:
cdrLIS is implemented as a REST web service, and this top layer contains only the Go main method, which runs the whole application and establishes communication between mutually independent components of the lower layers. Only this layer is aware of specific implementation details as: database connection, http handlers and their CRUDers, endpoint URLs, web service port, etc.

```
func main() {
 ...
 db, error := gorm.Open(dbDialect, dbArgs)
 defer db.Close()
 . . .
 sunitCRUD := crud.LASpatialUnitCRUD{DB: db}
 sunitHandler := handler.SpatialUnitHandler{SpatialUnitCRUD: sunitCRUD,
                 LevelCRUD: levelCRUD}
 router := httprouter.New()
 router.GET("/spatialunit",sunitHandler.GetSpatialUnits)
 router.POST("/spatialunit",sunitHandler.CreateSpatialUnit)
 router.GET("/spatialunit/:namespace/:localId",sunitHandler.GetSpatialUnit)
 router.PUT("/spatialunit/:namespace/:localId",sunitHandler.UpdateSpatialUnit)
 router.DELETE("/spatialunit/:namespace/:localId",sunitHandler.DeleteSpatialUnit)
 . . .
 handler := cors.Default().Handler(router)
 http.ListenAndServe(":3000", handler)
}
```

(ii) Http handlers layer:
This layer defines the http handlers for the REST API web service. Each handler contains logic on how to handle GET, POST, PUT, and DELETE requests, which CRUDers should be called, and how to return results. Here is an illustration of the GET handler for SpatialUnit:

```
type SpatialUnitHandler struct {
 SpatialUnitCRUD CRUDer
 LevelCRUD   CRUDer
}

func (handler *SpatialUnitHandler) GetSpatialUnit (w http.ResponseWriter,
                 r *http.Request,
                 p httprouter.Params)
{
 uid := common.Oid{Namespace:p.ByName("namespace"),LocalID:p.ByName("localId")}
 suUnit, error := handler.SpatialUnitCRUD.Read(uid)
 . . .
 respondJSON(w, 200, suUnit)
}
```

(iii) CRUD layer:
The components of this layer (structure types and methods) implement the persistence of the structure type values (objects) in the database. The basic abstraction in this layer is the CRUDer interface that defines prototype methods for CRUD functionalities (Read, ReadAll, Create, Update, Delete). Each data type that uses the CRUDer interface must have its own CRUDer implementation. A concrete implementation of CRUDer defines the way each object is persisted in a database. CRUDer implementations depend only on the GORM package, not on the specific database. The CRUDer interface and an example of the Create method for SpatialUnit are shown in Section 3.

(iv) Model layer:
This layer is actually a mapping of the LADM model to a platform-specific model, i.e., the mapping of UML classes into the corresponding Go structure types (structs). Since we use GORM, the structure type definitions also include GORM tags for object-relationship mapping, as well as JSON tags for serialization.

The UML sequence diagram in Figure 2 shows the roles and communications between these layers in the case of creating an LASpatialUnit object.

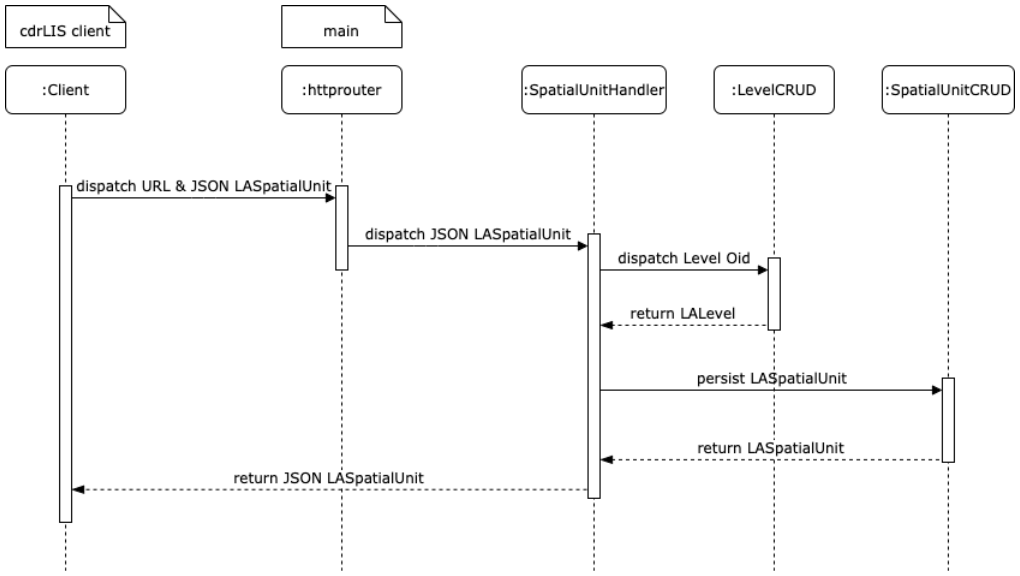

Figure 2. UML sequence diagram: creation of LASpatialUnit.

Regardless of country-specific business processes that need to be supported by the LIS, each country-specific LIS requires CRUD operations on all LADM entities/objects; this is precisely the core of our prototype. Adopting our generic core and prototype in the case of simply adding entities/objects would require two steps: (i) extending the database model; and (ii) extending all three application layers, namely CRUD, http handler, and main. Adding new classes and attributes to the CRUD layer can be simply accomplished using the embedding concept available in the Go language. From a software engineering perspective, extending the http handler and main layers does not require any radical programming intervention and can be considered trivial.

However, we should also point out the fact that any country-specific business processes and business rules in the domain of land administration and cadaster may require more complex operations than the set of operations in our CRUD layer. In this case, building a country-specific LIS using our generic core and prototype would require the implementation of more complex CRUD operations, including completely new interfaces, and the overall implementation effort could be significantly heavier. Unfortunately, our generic core and prototype do not include support for generic processes of initial data acquisition, data maintenance, and data publication. In that regard, the implementation of these processes, the specification of which was announced and is expected in the second edition of LADM [48], could significantly increase the scope and complexity of our generic core and prototype, but on the other hand, reduce implementation efforts in building a country-specific LIS.

4.3. System Architecture

Figure 3 shows the simplified system architecture of our single-site distributed cdrLIS prototype. In line with the model driven architecture principles, we used YugabyteDB—a NewSQL distributed DBMS—and its object-relational model as a database platform-specific model. It is an open-source derivative of Google Spanner [35], designed for (i) horizontal scalability, (ii) strong consistency, and (iii) disaster resilience and built on top of a custom distributed key/value storage engine forked from RocksDB [49]. The cluster of five nodes functions as a highly available and resilient distributed database with fault tolerance factor $f_t = 2$ (i.e., the system can tolerate two node failures; see Section 4.4), in which each node performs CRUD operations. In our case, nodes are commodity physical machines running the Linux operating system, but they also could be virtual machines or containers.

Clients can connect to any of the nodes to perform CRUD operations on the cdrLIS database cluster. They interact with a distributed SQL query layer that replicates, stores, and retrieves data using already mentioned, strongly consistent, and distributed key/value storage engine.

The distributed query execution layer distributes queries across nodes of the cluster. The queries are accepted by a leader node, which then requests other nodes (followers) to execute their part of the query, and sends the aggregated results back to the client.

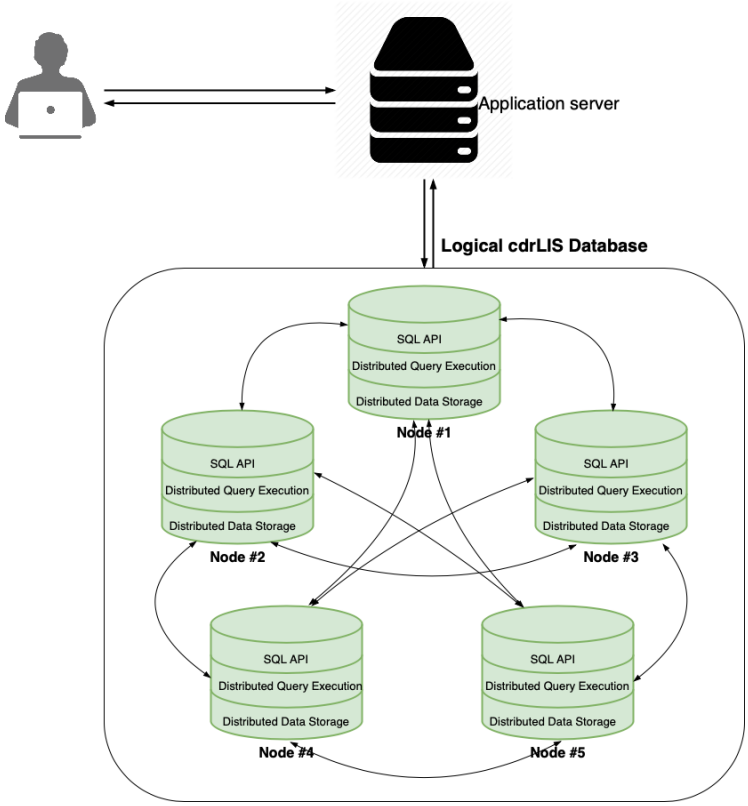

Figure 3. Single-site distributed architecture.

The cdrLIS can be deployed on a public cloud, on-premise data center, i.e., on a cluster of commodity machines, or a Docker container [50]. It can also run natively, as a stateful application, in Kubernetes [51] and similar container-orchestration environments for automating application deployment, scaling, and management. By virtue of the inherited features of YugabyteDB, cdrLIS is a consistent and, according to the CAPtheorem [52], partition tolerant (CP) system. It has all the features of a NewSQL distributed system, which we will explain in a bit more detail in the next two subsections.

4.4. Fault Tolerance, Reliability, and Resilience

Reliability refers to both the resiliency of a system to various types of failures and the capability to recover from them. A distributed LIS should be tolerant of system failures and continue to provide services even when failures occur. An important goal in designing and implementing distributed LIS is to build the system that is able to recover automatically from partial failures without affecting the overall performance. In particular, whenever a failure occurs, the LIS should continue to operate while repairs are being made. In other words, a distributed LIS is expected to be reliable, fault tolerant, and resilient. As we pointed out in Section 2, replication is fundamental for achieving fault tolerance, high availability, and resilience.

An extension of an embedded, key-value database is responsible for partitioning, replication, transactions, and persistence. The cdrLIS database is physically stored as a separate set of documents, and each document is mapped to key-values in RocksDB—the underlying per-node storage engine.

The cdrLIS tables are managed by RocksDB as multiple horizontal partitions (Figure 4). The tables are automatically partitioned into multiple partitions with the primary key for each row uniquely specifying the partition and making partitioning invisible to users. Partitioning is performed either by the hash of the primary key or by the primary key range.

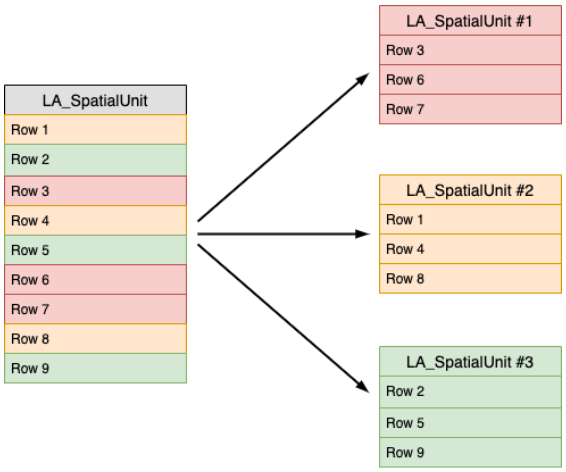

Figure 4. An example of horizontal partitioning.

To achieve fault tolerance f_t (i.e., the capability of a system to provide its services even in the presence of faults—in our context, it is the maximum number of node failures the system can survive while continuing to preserve the correctness of data), the cdrLIS database cluster has to be configured with a replication factor $r_f = 2f_t + 1$. That implies that each partition is replicated on r_f nodes (Figure 5), i.e., the system can tolerate $(r_f - 1)/2$ node failures.

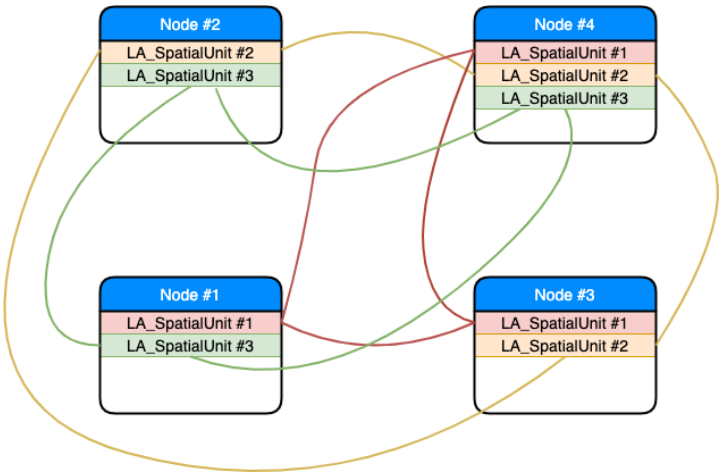

Figure 5. An example of combining replication and partitioning in a cluster with four nodes.

Each partition consists of a set of partition-peers, each of which stores one copy of the data belonging to the partition. There are r_f partition-peers for a partition hosted on r_f different nodes in a cluster.

The system is based on a synchronous replication, i.e., all writes are propagated to a quorum of copies of the partitions before being considered committed. Data are up to date with synchronous writes across nodes; if any of the cluster nodes fails, the data are consistent and not lost.

High availability, i.e., fault tolerance is achieved by having an active replica that is ready to take over as a new leader in a matter of seconds after the failure of the current leader and serve requests.

4.5. Consistency

Both database consistency and transaction consistency are guaranteed in cdrLIS. Database consistency is achieved by semantic integrity constraints specified as a set of structural integrity rules using SQL assertions and a set of behavioral constraints embedded in cdrLIS Go packages.

Transactions are an abstract layer that allows an application to pretend that concurrency problems and faults (both hardware and software) do not exist. Transaction consistency refers to the operation of concurrent transactions, i.e., ensuring database consistency when concurrent access and failures occur. By virtue of our NewSQL DBMS, the strictest transaction isolation level, serializable snapshot isolation [53], is guaranteed. Serializable snapshot isolation emulates serial transaction execution for all committed transactions; i.e., as if transactions had been executed serially, one after another, rather than concurrently. Additionally, one transaction does not block another transaction: SQL reads do not block SQL updates, and vice versa.

In order to guarantee fault tolerance of distributed systems in the case of node failures, the strong consistency of transaction inherently requires that updates should be synchronously committed at multiple nodes. The basic idea behind strong transactional consistency (also known as linearizability or atomic consistency) is simple: to make a distributed system appear as if there is only a single database, and all CRUD operations on it are atomic [26]. However, replicating partitions across the nodes poses consistency problems that each distributed system should solve efficiently.

Consensus is a key component to providing fault-tolerant services such as synchronously replicated data stores and non-blocking atomic transactions. Consequently, NewSQL systems use consensus protocols, i.e., Paxos [54,55] or the Raft consensus algorithm to enforce strong consistency in the presence of failures. Consensus algorithms enable a database cluster to work as a coherent group that can survive the failures of a minority of the nodes (f_t). This is achieved via a majority voting mechanism: any change to the data requires a majority of nodes to agree to the change.

In our case, the Raft algorithm [56] achieves consensus by first electing a leader, then giving the leader complete responsibility for managing the replicated log. The leader accepts log entries from clients, replicates them on followers, and tells them when it is safe to apply log entries to their state machines. Followers are passive: they only respond to requests from leaders. The client queries are handled by the leader: if a client sends query to a follower, the follower redirects it to the leader. If a leader fails or becomes disconnected from the other nodes, a new leader is elected.

5. Conclusions

Distributed computing and the share-nothing architecture became very popular, including new data processing systems, platforms, frameworks, and NoSQL and NewSQL DBMS with horizontal scalability and fault tolerance capabilities. These new data processing and management systems use large amounts of commodity hardware to store, manage, and analyze data in a highly distributed, scalable, and cost-effective way. The key feature of NoSQL DBMS is neglecting ACID transaction capabilities and the (object-)relational model in favor of eventual consistency and other data models. However, many mission-critical enterprises applications, including LIS, cannot give up the strict transaction and consistency requirements and, consequently, are not able to use NoSQL DBMS.

Data-intensive OLTP applications and systems, including LIS, should be primarily designed and implemented as resilient, scalable, and maintainable systems that guarantee consistency. They should be built on the "invest in data/system resilience, not disaster recovery" paradigm, thus eliminating complex and expensive backup infrastructure. NewSQL DBMS are designed and developed exactly for these purposes, and they should be seriously considered in designing, building, and deploying the next generation of LIS. Although designed to support the OLTP systems with ACID-transactional workloads, scalability, and resilience, distributed NewSQL DBMS have not yet been adopted in the LA domain.

In this paper, we have presented our generic and extensible LADM-compliant core framework and a prototype of consistent, distributed, and resilient LIS based on a NewSQL database management

system. Our Go generic core framework can be extended in two directions: (i) to support specific types of spatial unit, i.e., spatial profiles (polygon based, topological based, or similar); and (ii) to support the implementation of a specific country profile. The framework is also generic in the sense that it can be used in building of both distributed and centralized LIS, as well as other applications in the land administration domain including data acquisition tools.

As future work, we aim to extend our generic framework with 2D polygon based and topological based spatial profiles. Additional future work will also include performance improvement, i.e., spatial partitioning of data among nodes at the database level, and an extension towards supporting 3D spatial units.

Author Contributions: Conceptualization, Zdravko Galić; methodology, Zdravko Galić; software, Mario Vuzem and Zdravko Galić; validation, Zdravko Galić and Mario Vuzem; formal analysis, Zdravko Galić; writing, original draft preparation, Zdravko Galić; writing, review and editing, Zdravko Galić; visualization, Mario Vuzem; supervision, Zdravko Galić; project administration, Zdravko Galić. All authors read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Acknowledgments: The authors would like to thank the anonymous reviewers for their valuable comments and suggestions for improving the quality of the paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Stonebraker, M.; Çetintemel, U. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*; Brodie, M.L., Ed.; ACM/Morgan & Claypool: New York, NY, USA, 2019; pp. 441–462. [CrossRef]
2. Davoudian, A.; Chen, L.; Liu, M. A Survey on NoSQL Stores. *ACM Comput. Surv.* **2018**, *51*. [CrossRef]
3. Stonebraker, M. New Opportunities for New SQL. *Commun. ACM* **2012**, *55*, 10–11. [CrossRef]
4. Garcia-Molina, H.; Ullman, J.D.; Widom, J. *Database Systems—The Complete Book*, 2nd ed.; Pearson Education: London, UK, 2009.
5. Özsu, M.T.; Valduriez, P. *Principles of Distributed Database Systems*, 4th ed.; Springer: Berlin, Germany, 2020. [CrossRef]
6. Apache Software Foundation. Apache Hadoop. 2020. Available online: https://hadoop.apache.org (accessed on 4 March 2020).
7. Apache Software Foundation. Apache Spark. 2020. Available online: https://spark.apache.org (accessed on 4 March 2020).
8. Apache Software Foundation. Apache Flink. 2020. Available online: https://flink.apache.org (accessed on 4 March 2020).
9. Eldawy, A.; Mokbel, M.F. SpatialHadoop: A MapReduce framework for spatial data. In Proceedings of the 31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, Korea, 13–17 April 2015; Gehrke, J., Lehner, W., Shim, K., Cha, S.K., Lohman, G.M., Eds.; IEEE Computer Society: Piscataway, NJ, USA, 2015; pp. 1352–1363. [CrossRef]
10. Belussi, A.; Migliorini, S.; Eldawy, A. Skewness-Based Partitioning in SpatialHadoop. *ISPRS Int. J. Geo-Inform.* **2020**, *9*, 201. [CrossRef]
11. Yu, J.; Zhang, Z.; Sarwat, M. Spatial data management in Apache Spark: the GeoSpark perspective and beyond. *GeoInformatica* **2019**, *23*, 37–78. [CrossRef]
12. Galic, Z.; Meskovic, E.; Osmanovic, D. Distributed processing of big mobility data as spatio-temporal data streams. *GeoInformatica* **2017**, *21*, 263–291. [CrossRef]
13. Pavlo, A.; Aslett, M. What's Really New with NewSQL? *SIGMOD Rec.* **2016**, *45*, 45–55. [CrossRef]
14. Bennett, R.M.; Pickering, M.; Sargent, J. Transformations, Transitions, or Tall Tales? A Global Review of the Uptake and Impact of NoSQL, Blockchain, and Big Data Analytics on the Land Administration Sector. *Land Use Policy* **2019**, *83*, 435–448. [CrossRef]
15. *ISO 19152:2012 Geographic Information—Land Administration Domain Model (LADM)*; International Organization for Standardization: Geneva, Switzerland, 2012.

16. Lemmen, C.; van Oosterom, P.; Bennett, R. The Land Administration Domain Model. *Land Use Policy* **2015**, *49*, 535–545. [CrossRef]
17. Kalogianni, E.; Kalantari, M.; Dimopolou, E.; van Oosterom, P. LADM Country Profiles Development: Aspects to be Reflected and Considered. In Proceedings of the 8th Land Administration Domain Model Workshop, Kuala Lumpur, Malaysia, 1–3 October 2019; van Oosterom, P., Lemmen, C., Rahman, A.A., Eds.; International Federation of Surveyors (FIG): Copenhagen, Denmark, 2019; pp. 287–302.
18. Ying, S.; Guo, R.; Li, L.; van Oosterom, P.; Ledoux, H.; Stoter, J. Design and Development of a 3D Cadastral System Prototype based on the LADM and 3D Topology. In Proceedings of the 2nd International Workshop on 3D Cadastres, Delft, The Netherlands, 16–18 November 2011; pp. 167–188.
19. Vandysheva, N.; Sapelnikov, S.; van Oosterom, P.; de Vries, M.; Spiering, B.; Wouters, R.; Hoogeveen, A.; Penkov, V. The 3D Cadastre Prototype and Pilot in the Russian Federation. In Proceedings of the FIG Working Week 2012, Rome, Italy, 6–10 October 2012; pp. 1–16.
20. Zulkifli, N.A.; Rahman, A.A.; Jamil, H.; Teng, C.H.; Tan, L.C.; Looi, K.S.; Chan, K.L.; Van Oosterom, P. Development of a Prototype for the Assessment of the Malaysian LADM Country Profile. In Proceedings of the FIG Congress 2014, Kuala Lumpur, Malaysia, 16–21 June 2014; pp. 1–18.
21. Visnjevac, N.; Mihajlovic, R.; Soskic, M.; Cvijetinovic, Z.; Bajat, B. Prototype of the 3D Cadastral System Based on a NoSQL Database and a JavaScript Visualization Application. *ISPRS Int. J. Geo-Inform.* **2019**, *8*, 227. [CrossRef]
22. Kalogianni, E.; Dimopoulou, E.; van Oosterom, P. A 3D LADM prototype implementation in INTERLIS. In *Advances in 3D Geoinformation*; Abdul-Rahman, A., Ed.; Springer: Berlin, Germany, 2017; pp. 137–157.
23. Van Steen, M.; Tanenbaum, A. *Distributed Systems*, 3rd ed.; CreateSpace Independent Publishing Platform: Scotts Valley, CA, USA, 2018.
24. Viotti, P.; Vukolic, M. Consistency in Non-Transactional Distributed Storage Systems. *ACM Comput. Surv.* **2016**, *49*, 19:1–19:34. [CrossRef]
25. van Steen, M.; Tanenbaum, A.S. A brief introduction to distributed systems. *Computing* **2016**, *98*, 967–1009. [CrossRef]
26. Kleppmann, M. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*; O'Reilly: Newton, MA, USA, 2017.
27. MariaDB. ClustrixDB. 2020. Available online: http://clustrix.com (accessed on 24 March 2020).
28. CockroachDB Labs. CockroachDB. 2020. Available online: https://www.cockroachlabs.com/product/ (accessed on 24 March 2020).
29. Fauna, Inc. FaunaDB. 2020. Available online: https://fauna.com (accessed on 24 March 2020).
30. Kemper, A.; Neumann, T. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, Hannover, Germany, 11–16 April 2011; Abiteboul, S., Böhm, K., Koch, C., Tan, K., Eds.; IEEE Computer Society: Piscataway, NJ, USA, 2011; pp. 195–206. [CrossRef]
31. Shamgunov, N. The MemSQL in-Memory Database System. In Proceedings of the 2nd International Workshop on in Memory Data Management and Analytics, IMDM 2014, Hangzhou, China, 1 September 2014.
32. MemSQL Inc. MemSQL. 2020. Available online: https://www.memsql.com (accessed on 24 March 2020).
33. NuoDB, Inc. NuoDB. Available online: https://www.nuodb.com (accessed on 24 March 2020).
34. Lee, J.; Muehle, M.; May, N.; Faerber, F.; Sikka, V.; Plattner, H.; Krüger, J.; Grund, M. High-Performance Transaction Processing in SAP HANA. *IEEE Data Eng. Bull.* **2013**, *36*, 28–33.
35. Corbett, J.C.; Dean, J.; Epstein, M.; Fikes, A.; Frost, C.; Furman, J.J.; Ghemawat, S.; Gubarev, A.; Heiser, C.; Hochschild, P.; et al. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* **2013**, *31*, 8:1–8:22. [CrossRef]
36. PingCAP Inc. TiDB. 2020. Available online: https://pingcap.com/en/ (accessed on 24 March 2020).
37. Stonebraker, M.; Weisberg, A. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* **2013**, *36*, 21–27.
38. Yugabyte, Inc. YugabyteDB. 2020. Available online: https://www.yugabyte.com (accessed on 24 March 2020).
39. Donovan, A.A.A.; Kernighan, B.W. *The Go Programming Language*; Addison-Wesley: Boston, MA, USA, 2016.
40. Google Inc. Go. 2020. Available online: https://golang.org (accessed on 7 April 2020).
41. Summerfield, M. *Programming in Go*; Addison-Wesley: Boston, MA, USA, 2016.
42. Smith, P. gogeos. 2020. Available online: https://github.com/paulsmith/gogeos (accessed on 10 March 2020).

43. OSGeo. GEOS—Geometry Engine, Open Source. Available online: https://trac.osgeo.org/geos/ (accessed on 10 March 2020).
44. *Geographic Information—Simple Feature Access—Part 2: SQL Option*; International Organization for Standardization: Geneva, Switzerland, 2004.
45. Ireland, C.; Bowers, D.; Newton, M.; Waugh, K. A Classification of Object-Relational Impedance Mismatch. In Proceedings of the First International Conference on Advances in Databases, Knowledge, and Data Applications, DBKDS 2009, Gosier, Guadeloupe, France, 1–6 March 2009; Chen, Q., Cuzzocrea, A., Hara, T., Hunt, E., Popescu, M., Eds.; IEEE Computer Society: Piscataway, NJ, USA, 2009; pp. 36–43. [CrossRef]
46. Pathreon. Package Gorm. 2020. Available online: https://pkg.go.dev/github.com/jinzhu/gorm (accessed on 10 March 2020).
47. Galić, Z.; Vuzem, M. cdrLIS. 2020. Available online: https://github.com/cdrlis/cdrLIS (accessed on 8 May 2020).
48. Lemmen, C.; Van Oosterom, P.; Kalantari, M.; Unger, E.; De Zeeuw, C. *OGC White Paper on Land Administration*; Open Geospatial Consortium: Wayland, MA, USA, 2019; Available online: https://docs.opengeospatial.org/wp/18-008r1/18-008r1.html (accessed on 3 July 2020).
49. Facebook. RocksDB. 2020. Available online: https://rocksdb.org (accessed on 3 June 2020).
50. Docker Inc. Docker. 2020. Available online: https://www.docker.com (accessed on 3 June 2020).
51. The Linux Foundation. Kubernetes. 2020. Available online: https://kubernetes.io (accessed on 3 June 2020).
52. Brewer, E. CAP Twelve Years Later: How the "Rules" Have Changed. *Computer* **2012**, *49*, 23–29. [CrossRef]
53. Cahill, M.J.; Röhm, U.; Fekete, A.D. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.* **2009**, *34*, 20:1–20:42. [CrossRef]
54. Lamport, L. The Part-Time Parliament. *ACM Trans. Comput. Syst.* **1998**, *16*, 133–169. [CrossRef]
55. Van Renesse, R.; Altinbuken, D. Paxos Made Moderately Complex. *ACM Comput. Surv.* **2015**, *47*, 42:1–42:36. [CrossRef]
56. Ongaro, D.; Ousterhout, J.K. In Search of an Understandable Consensus Algorithm. In Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC'14, Philadelphia, PA, USA, 19–20 June 2014; Gibson, G., Zeldovich, N., Eds.; USENIX Association: Berkeley, CA, USA, 2014; pp. 305–319.