# Ganos: A Multidimensional, Dynamic, and Scene-Oriented Cloud-Native Spatial Database Engine

Jiong Xie, Zhen Chen, Jianwei Liu, Fang Wang, Feifei Li, Zhida Chen, Yinpei Liu, Songlu Cai, Zhenhua Fan, Fei Xiao, Yue Chen
Alibaba Group
{xiejiong.xj,erchen.cz,jerven.ljw,solar.wf,lifeifei,zhida.chen,yinpei.lyp,zijia,huanzhi.fzh,jibo.xf,beiqi.cy}@alibaba-inc.com

## ABSTRACT

Recently, the trend of developing digital twins for smart cities has driven a need for managing large-scale multidimensional, dynamic, and scene-oriented spatial data. Due to larger data scale and more complex data structure, queries over such data are more complicated and expensive than those on traditional spatial data, which poses challenges to the system efficiency and deployment costs. The existing spatial databases have limited support in both data types and operations. Therefore, a new-generation spatial database with excellent performance and effective deployment costs is needed.

This paper presents Ganos, a cloud-native spatial database engine of PolarDB for PostgreSQL that is developed by Alibaba Cloud, to efficiently manage multidimensional, dynamic, and scene-oriented spatial data. Ganos models 3D space and spatio-temporal dynamics as first-class citizens. Also, it natively supports spatial/spatio-temporal data types such as 3DMesh, Trajectory, Raster, PointCloud, etc. Besides, it implements a novel extended-storage mechanism that utilizes cloud-native object storage to reduce storage costs and enable uniform operations on the data in different storages. To facilitate processing "big" queries, Ganos extends PolarDB and provides spatial-oriented multi-level parallelism under the architecture of decoupling compute from storage in cloud-native databases, which achieves elasticity and excellent query performance. We demonstrate Ganos in real-life case studies. The performance of Ganos is evaluated using real datasets, and promising results are obtained. Finally, based on the extensive deployment and application of Ganos, the lessons learned from our customers and the expectations of modern cloud applications for new spatial database features are discussed.

**PVLDB Reference Format:**
Jiong Xie, Zhen Chen, Jianwei Liu, Fang Wang, Feifei Li, Zhida Chen, Yinpei Liu, Songlu Cai, Zhenhua Fan, Fei Xiao, Yue Chen. Ganos: A Multidimensional, Dynamic, and Scene-Oriented Cloud-Native Spatial Database Engine. PVLDB, 15(12): 3483 - 3495, 2022.
doi:10.14778/3554821.3554838

## 1 INTRODUCTION

In recent years, we have entered a new era of digital twins of physical worlds. The city digital twins technology develops digitization copies of cities and manages them for the bidirectional interaction between digital and real worlds [20, 30–32]. It has broad applications in urban planning, smart traffic management, automated environment monitoring, etc. [30, 31]. With the development of sensors and the city digital twins technology, multidimensional, dynamic, and scene-oriented spatial (MDS, for short) data is generated at an unprecedented speed. Compared with traditional spatial data that is often static and two-dimensional (2D), MDS data can model real-life objects much more precisely in three aspects: (1) It models the three-dimensional (3D) space of our world; (2) It stores the information of an object at different time points, which reflects the dynamic characteristics of our world over time; (3) It stores scene-oriented information of an object that is rarely considered by traditional spatial data. Here, the scene-oriented information can be visual or behavior information, or any other information that can describe the object, e.g., textures of a building, locking and unlocking events in bike-sharing. It should be noted that the concept of MDS data is a superset of traditional spatial data. Several cases of MDS data include the building information modeling (BIM) that represents real-world buildings as a 3D entity with textures and materials, and the trajectory of an unmanned aerial vehicle (UAV) that is composed of 3D points and remote sensing images collected with timestamp information.

The development of a spatial database management system (DBMS) to manage large-scale MDS data is the foundation of city digitization [15, 29, 31, 32]. However, it brings new challenges to the system design. First, the MDS data has much more complex structure and much larger data size than traditional spatial data does. For instance, the size of a BIM object can be a few orders of magnitude larger than that of a Point of Interest (POI) object. Moreover, different MDS data differs greatly in both data structure and size. It is challenging for a DBMS to manage large-scale MDS data of different types and to provide scalable operations on the data. Second, the DBMS should support different types of queries, including spatial, spatio-temporal, scene-oriented, and *cross-model* queries. A cross-model query requires computations on different MDS data. For example, a query for the no-fly zones of UAVs requires computations on BIM data, digital elevation model (DEM) data, and Trajectory data. Besides, due to the complexity of the data structure and spatial operations, queries on the MDS data can be time-consuming, e.g., taking hours to finish. Thus poses challenging requirements on the DBMS in both its functionality and efficiency.

Much work has been done on managing spatial data. This paper focuses on relational DBMS (RDBMS) because it is most commonly used for OLTP and complex queries over spatial data. The existing spatial RDBMS [3, 6, 10, 11, 19, 21] have good support for traditional

Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554838

spatial data, but they have limited support for MDS data in both data types and operations. The prosperity of cloud computing has facilitated the development of cloud-native spatial RDBMS. Several compatible editions of on-premise spatial RDBMS are implemented, e.g., [9, 11], but they have limited support for MDS data as well. To overcome these challenges, this paper presents a new spatial RDBMS engine called Ganos (the name comes from the goddess of earth Gaea and the god of time Chronos).

Ganos is built on PolarDB for PostgreSQL (hereafter simplified as PolarDB). PolarDB [26] is a cloud-native relational database service developed by Alibaba Cloud. Ganos enables PolarDB to store, index, and query MDS data. It is provided as a value-added service on PolarDB for cloud users who have spatial applications. Ganos is designed as a cloud-native spatial RDBMS because of the following reasons. First, with an increasing number of applications moving to the cloud, Database-as-a-service (also called DBaaS) has shown its potential to process large-scale data and queries, making it a reasonable choice for managing MDS data. Second, compared to on-premise databases, the features of DBaaS, e.g., cloud-native object storage and decoupling computation from storage, allow designing a system with good performance and elasticity. This is important for users who care about performance and deployment costs.

Ganos considers MDS data as first-class citizens and implements a type hierarchy of MDS data. Ganos implements systems-supplied data types including 3DMesh, Trajectory, Raster, PointCloud, etc. Based on this, Ganos provides a systematic framework of managing MDS data, including defining data structures, implementing data storage and indexing, and implementing operations on MDS data. These features distinguish Ganos from other spatial RDBMS. Each data type encapsulates multiple fields and is implemented as a compact binary structure called Spatial Large Object (SLOB). This is beneficial to reducing the storage space and the complexity of table schema. Ganos uses multidimensional R-trees to manage SLOBs.

Besides, Ganos extends the query functionality of PolarDB by supporting spatial queries, spatio-temporal queries, scene-oriented queries, and cross-model queries. To realize these extensions, Ganos adds over one thousand operations to PolarDB, many of which are optimized considering the characteristics of MDS data. This allows Ganos to achieve optimizations in processing queries. Moreover, Ganos implements operations that leverage cloud features to achieve good performance and elasticity. Specifically, Ganos implements an extended-storage mechanism that allows storing large-scale spatial data in a database and in a cheaper cloud-native object storage, thus reducing the deployment costs significantly. Ganos also provides spatial-oriented multi-level parallelism that makes good use of one-write-multiple-reads computing resources to accelerate the processing of *big* queries on MDS data.

**Product Impact.** Since 2018, Ganos has been deployed as a cloud service, and it has shown success in over 30 fields of applications.

The contributions of this paper are summarized as follows:

- A spatial RDBMS engine called Ganos is designed and implemented to manage real-world MDS data. Ganos considers MDS data as first-class citizens and provides a systematic framework of managing MDS data, including defining data structures, implementing data storage and indexing, and implementing data operations.
- A brand-new practice of moving from traditional on-premise spatial RDBMS to cloud-native spatial RDBMS is shown. Ganos implements a set of operations that leverage cloud features to achieve good performance and elasticity. Specifically, it implements an extended-storage mechanism that reduces deployment costs significantly. Besides, Ganos provides spatial-oriented multi-level parallelism to accelerate the processing of big queries on MDS data.
- Lessons learned from the customers are shared. By deploying Ganos as a cloud service to power many brand-new applications such as smart city planning, autonomous driving, and DB for GeoAI. Last but not least, city digital twins will bring a new direction to the development of spatial DBMS.

## 2 RELATED WORK

Extensive studies have been conducted to develop systems for spatial data in both industry and academia. This paper focuses on reviewing full-fledged RDBMS that have comprehensive features (e.g., SQL support, data integrity, and query plan optimizations) because they are most related to our work.

**Spatial RDBMS**. To handle ubiquitous spatial data and to overcome the limitations of traditional RDBMS, commercial database vendors have extended their products to provide native support for spatial data, e.g., IBM DB2 Spatial Extender [10, 19], SQL Server Spatial [11], Oracle Spatial [3]. Some spatial RDBMS are developed based on or by adding spatial support to the open-source RDBMS, e.g., PostGIS [6] built on PostgreSQL [4], MySQL [2], and SpatiaLite [21] built on SQLite [8]. Among them, PostGIS [6] is arguably the de facto standard of open-source spatial RDBMS, which adds support for geometry, geography, raster, and other types to PostgreSQL.

These spatial RDBMS have achieved great success in processing traditional static and 2D spatial data, and they have served many applications well. However, they supported limited types of MDS data and operations. For example, Oracle Spatial is probably the most powerful spatial RDBMS that supports a rich set of spatial data types and operations, which has inspired our work a lot. As Ganos does, Oracle Spatial implements 3D geometric data and PointCloud as system-supplied data types. However, it does not fully implement the scene-oriented attributes like textures, and it does not implement Trajectory as a system-supplied data type [7]. Therefore, it has limited support for scene-oriented, and spatio-temporal queries, which are essential for building city digital twins systems. This problem will be solved by Ganos.

**Extensions on spatial RDBMS.** This line of studies makes extensions based on spatial RDBMS. This paper categorizes the studies into application-level and DBMS-level extensions according to whether they implement changes in DBMS. (1) Application-level extensions [18, 27, 28, 34, 35]: Logothetis et al. [28] developed an open source Building Information Modeling (BIM) tool for the BIM process, which leverages PostGIS to extend FreeCAD [1] into BIM. Zhang et al. [35] extended PostgreSQL to manage raster species distribution data. Alexandre et al. [18] developed a spatio-temporal database based on TimeDB [17] and Oracle Spatial to support temporal and spatial data, respectively. AST-PostGIS [27] implements advanced spatial data types by enforcing spatial integrity constraints on PostGIS geometric types. 3DCityDatabase [34] is an open source 3D spatial database based on PostGIS or Oracle Spatial to manage 3D city models. (2) DBMS-level extensions [13, 25, 36]: MobilityDB [13, 36] and PG-TRAJECTORY [25] extend spatial RDBMS at the DBMS level. They natively support Trajectory by implementing it as a system-supplied data type and providing a set of data operations, both of which are built on PostGIS.

Compared to the above systems, Ganos natively supports more MDS data types and can be applied to more applications. Ganos makes changes in DBMS and implements plenty of operations on MDS data. Meanwhile, it optimizes the operations by considering the characteristics of MDS carefully. MobilityDB and PG-TRAJECTORY belong to DBMS-level extensions as Ganos does, but they only add support for Trajectory.

**Cloud-native spatial RDBMS.** An increasing number of enterprises are migrating their data to cloud-native databases. Major cloud vendors are providing DBaaS such as AWS Aurora [33] and Azure SQL DB [12]. They implement compatible editions of spatial RDBMS to manage spatial data. For example, Aurora PostgreSQL [9] supports PostGIS, and Azure SQL DB has the spatial features of SQL Server. They support spatial data as much as the base spatial RDBMS can afford. In comparison to them, Ganos is a spatial database engine of PolarDB that enables PolarDB with the capability to manage spatial data by making changes in its kernel. Ganos provides better support for MDS data in three aspects. Firstly, it implements a set of system-supplied MDS data types, which cover many real applications. Secondly, it supports spatial, spatio-temporal, scene-oriented, and cross-model queries on MDS data. Lastly, it considers the characteristics of MDS data and leverages cloud features to optimize a large set of operations, including data storage, indexing, and querying.

## 3 SYSTEM OVERVIEW

Ganos is a spatial database engine that is built on PolarDB [26]. PolarDB is a cloud-native relational database system developed by Alibaba Cloud. It provides high elasticity and concurrency, and it is built on and fully compatible with PostgreSQL. Figure 1 shows the architecture of Ganos and the interaction between Ganos and PolarDB, where the left picture shows the architecture of PolarDB and the right one shows the architecture of Ganos. This paper first gives a brief introduction to PolarDB and then introduces Ganos.

**PolarDB.** PolarDB adopts the shared-storage architecture that decouples compute from storage. The computation nodes and storage nodes are connected through a low-latency remote direct memory access (RDMA) network. PolarDB supports on-demand scaling of computation nodes and auto-scaling of storage nodes. At the computation layer, PolarDB has one primary node called RW node and many read-only (RO) nodes. The primary node conducts both read and write operations, while the RO node only conducts read operations. PolarDB implements a failover mechanism to achieve high availability. Once the primary node crashes, PolarDB will select a RO node as the new primary node. At the storage layer, PolarDB is based on PolarFS [16], a distributed file system with ultra-low latency, high throughput, and high availability.

**Ganos overview.** Ganos is the spatial database engine of PolarDB, which enables PolarDB to store, index, and query MDS data. Since Ganos is built on PolarDB, Ganos inherits the functions and features of PolarDB, including relational data support, high availability, and elasticity. At the DBMS level, Ganos implements a rich set of system-supplied spatial data types and adds plenty of operations to PolarDB. As shown in Figure 1, Ganos extends PolarDB in four aspects, including data models, access methods, extended storage, and query processing. In the following, these aspects are introduced, respectively.

**Data models.** Ganos models real-world objects as MDS data and implements a type hierarchy of MDS data, where Geometry is implemented following the hierarchy that is defined by OGC [5] and SQL/MM Spatial [24]. Ganos implements a set of system-supplied MDS data types including 3DMesh, Trajectory, PointCloud, Raster, etc., each of which contains the information of real-world objects. As the data of different types have different data fields, Ganos introduces SLOB to simplify the storage of database tables. A SLOB encapsulates multiple fields of one MDS object into one data structure, e.g., geometry, textures, and materials of 3DMesh, and the data structure is implemented as a compact binary structure. SLOBs are stored in the same column of a table. This design avoids using a schema for each type of data and is conducive to operations on different types of data. Besides, it is easy to extend to support more system-supplied data types in the future. For more detailed information, please refer to Section 4.

**Access methods.** To accelerate the processing of queries, Ganos implements indexes for MDS data types. In addition to a B-tree for equality search, based on generalized search tree (GiST) [23], Ganos provides an index framework named GiST+ to enrich the access methods of GiST. GiST is an index framework that can be used to build disk-based search trees including R-tree [22] and B-tree [14]. GiST+ extends GiST in the way that GiST+ can select among indexes for answering a query, when there are different GiST+-based indexes built on the same set of columns.

**Extended storage.** Based on the storage model of PolarDB, Ganos implements a novel mechanism called extended storage to store MDS data, which can reduce the deployment costs of Ganos significantly. Extended storage allows storing MDS data not only in database tables, but also on Alibaba Cloud Object Storage Service (OSS). To meet different needs, Ganos provides two modes of extended OSS storage, i.e., hot/cold data separation and heterogeneous file access. To hide the underlying storage details, Ganos implements a spatial object locator that enables data operations to access objects stored in database tables or on OSS in a unified manner. More details are provided in Section 5.

**Query processing.** This component consists of three parts, and each of them will be introduced in the following.

*Query types.* Ganos supports multiple types of queries, including spatial, spatio-temporal, scene-oriented, and cross-model queries. This paper mainly introduces the scene-oriented and cross-model queries because they are rarely considered by other spatial RDBMS. A scene-oriented query conducts operations based on the scene-oriented information, e.g., conducting an intersection on a 3DMesh object will change its textures and materials. A cross-model query conducts operations on MDS data of different types. Ganos implements over 1,000 operations to support these queries. Details are provided in Section 6.1.

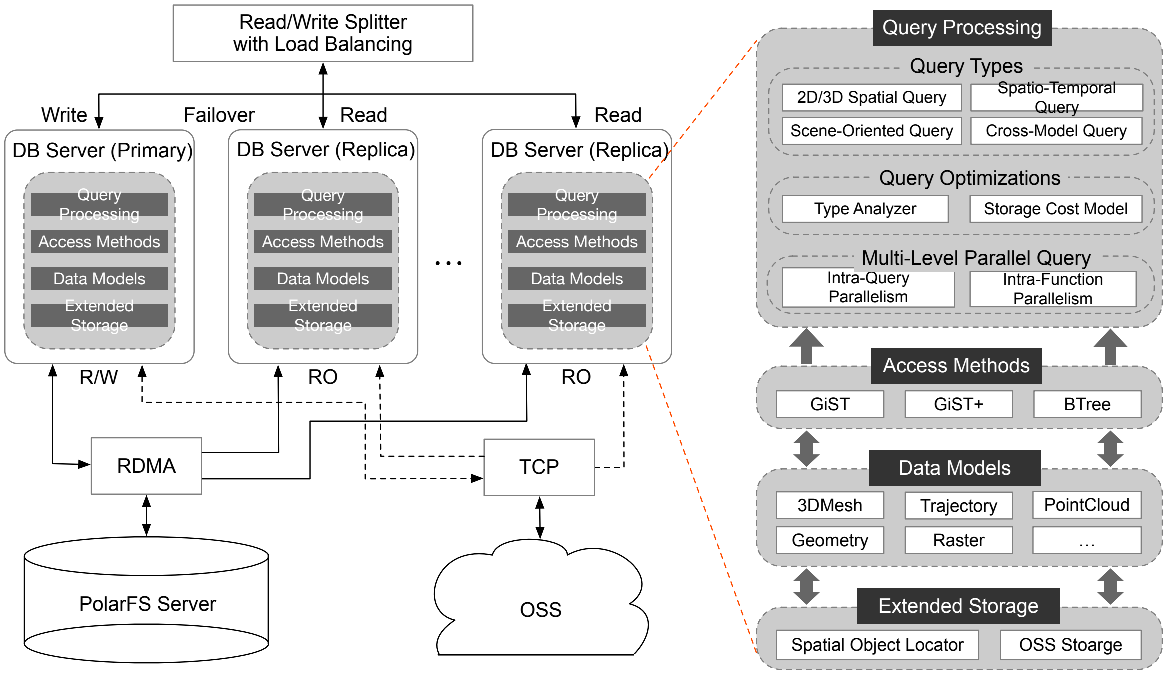


**Figure 1: Architecture of Ganos and the relation between Ganos and PolarDB.**

*Multi-level parallelism.* Queries on MDS data are usually more complicated and take a longer time to finish than those on traditional spatial data. This paper refers to a query that takes hours to finish as a *big* query. To accelerate the processing of big queries, Ganos extends the parallelism mechanism of PolarDB and implements spatial-oriented multi-level parallelism, including intra-query parallelism (IQP) and intra-function parallelism (IFP). IQP parallelizes the processing of a big query by assigning it to many RO nodes, each of which processes a disjoint subset of SLOBs independently. To mitigate the potential load imbalance problem that is caused by the existence of spatial objects with drastic size differences, Ganos introduces IFP, and leverages it to work with IQP. For more detailed information, please refer to Section 6.2.

*Query optimizations.* Ganos implements query optimizations to improve the query processing efficiency. Since Ganos introduces the extended storage mechanism, the query plan of PolarDB is not usable because its cost model dost not consider I/O costs of OSS. Therefore, Ganos implements the Spatial Type Analyzer of PolarDB to maintain storage distribution histogram. Also, it improves the cost model to consider the I/O costs of OSS. Based on this, query plans tend to use index scans for queries that consider objects on OSS. For more detailed information, please refer to Section 6.3.

## 4 DATA MODELS

In this section, the modeling and supporting of MDS data are described. In Section 4.1, a representative set of system-supplied MDS data types of Ganos is introduced. In Section 4.2, the implementation of these data types in Ganos is explained. In Section 4.3, the indexes used in Ganos are explained.

### 4.1 Data Types

Ganos models real-world objects as multidimensional, dynamic, and scene-oriented data. Ganos supports a rich set of data types, and objects of different data types have different attributes that are categorized into *SpatialAttr*, *TemporalAttr*, *SceneAttr*, and *GeneralAttr*:

- *SpatialAttr* contains the spatial information of an object that can be a set of 2D/3D points or a spatial range in 2D/3D space.
- *TemporalAttr* contains the temporal information of an object, which can be a set of timestamps or a temporal interval.
- *SceneAttr* contains the scene-oriented information of an object, which can be a set of images, RGB colors, lighting models, or a composite data structure.
- *GeneralAttr* contains the general attributes of an object that do not belong to the other three types of attributes.

Ganos represents the MDS data as the data with attributes of *SpatialAttr*, and possibly *TemporalAttr*, *SceneAttr*, and *GeneralAttr*, i.e., an MDS object can have *SpatialAttr* only, or have *SpatialAttr* and other attributes that are *TemporalAttr*, *SceneAttr*, or *GeneralAttr*. Figure 2 shows the hierarchy of the data types that are supported by Ganos, where Geometry is implemented under the hierarchy defined by OGC [5] and SQL/MM Spatial [24]. In the following, 3DMesh, Trajectory, Raster, and PointCloud are introduced in detail.

**3DMesh.** 3DMesh represents a 3D object with scene-oriented information, which is denoted as $3DMesh = (Shape, Visuals, General)$. *Shape* represents a 3D geometry. $Visuals = (textures, materials, mappings, UVcoords)$, where *textures* together with *materials* describe the visual information, such as color and pattern. *mappings* represents the association of *materials* and *textures* with *geometry*, and *UVCoords* represents the pixel coordinates for visualization.

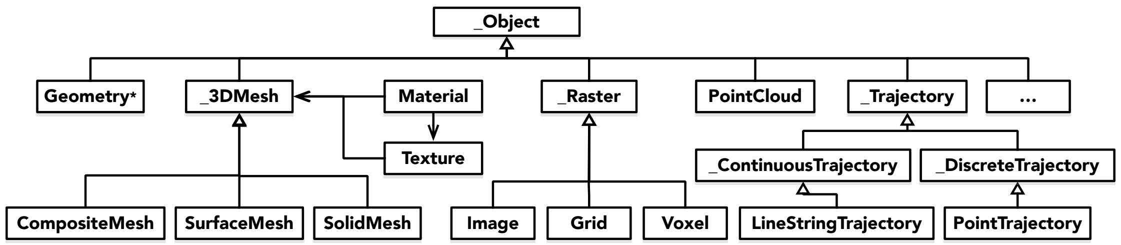


**Figure 2: Hierarchy of Ganos data types.**

*General* represents the general attributes, e.g., the weight of each vertex of *Shape*. In *3DMesh*, *Shape* belongs to *SpatialAttr*, *Visuals* belongs to *SceneAttr*, and *General* belongs to *GeneralAttr*. Based on the structure of *Shape*, 3DMesh can be categorized into SurfaceMesh, SolidMesh, and CompositeMesh, where SurfaceMesh represents a hollow 3D object, SolidMesh represents a solid 3D object, and CompositeMesh represents a 3D object that has a composite structure of SolidMesh and SurfaceMesh.

**Trajectory.** Trajectory is used to represent a moving object whose location changes over time. It is denoted as $Trajectory = (TPoints, Events)$. $TPoints = \{(p_1, t_1, A_1), \cdots, (p_n, t_n, A_n)\}$, where $p_i$ is a 2D/3D point, $t_i$ is a timestamp, and $A_i$ is a set of general attributes. $Events = \{(e_1, t^e_1), \cdots, (e_m, t^e_m)\}$, where $e_j$ is an event, and $t^e_j$ is a timestamp. In *Trajectory*, $p$ belongs to *SpatialAttr*, $t$ and $t^e$ belong to *TemporalAttr*, $A$ belongs to *GeneralAttr*, and event $e$ belongs to *SceneAttr*. Based on the sampling precision, Trajectory can be categorized into ContinuousTrajectory and DiscreteTrajectory.

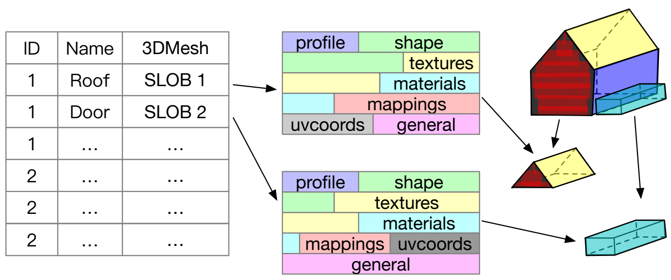
**Raster.** Raster is a gridded data where each cell is associated with a geolocation. It is denoted as $Raster = (Footprint, Time, Matrix)$, where *Footprint* represents a spatial range, *Time* represents a temporal range, and *Matrix* represents a multidimensional array of cells. *Matrix* stores scene-oriented information such as temperature and spectrum. In *Raster*, *Footprint* belongs to *SpatialAttr*, *Time* belongs to *TemporalAttr*, and *Matrix* belongs to *SceneAttr*. Based on the information that is stored in *Matrix* and its usage, Raster can be categorized into Image, Grid, and Voxel.

**PointCloud.** PointCloud is a collection of 3D points. It is denoted as $PointCloud = \{(p_1, A_1), \cdots, (p_n, A_n)\}$, where $p_i$ is a 3D point, and $A_i$ is a set of general attributes. In *PointCloud*, $p$ belongs to *SpatialAttr*, and $A$ belongs to *GeneralAttr*.

## 4.2 Data Type Implementation

Each data type is divided into two parts, *profile* and *details*. *profile* contains spatial information, metadata, and possibly temporal and scene-oriented information, which is a summary of the data and is used for filtering. *details* contains the detailed information of the data, which comprises a distinct set of attributes of the data type. Both parts are implemented as a binary sequence, and thus each object is stored as a long binary sequence, i.e., a concatenation of two binary sequences, in a table. We refer to it as a SLOB. The benefits of introducing SLOBs are two-fold. On the one hand, it follows the manner that PolarDB extends data types, which makes SLOBs compatible with other data types in PolarDB. On the other hand, the binary structure is flexible and compact that improves the performance of both computation and storage.

Figure 3 shows the structure of 3DMesh, as an example of a SLOB. At the front part, it is *profile*. The following part is *details*, which contains *shape*, *textures*, *materials*, *mapping*, *uvcoords*, and *general*. A building is decomposed into multiple components that are of 3DMesh type, e.g., a building has a roof and many doors. Each component object is implemented as a SLOB, and it is stored as a cell in the table. They share the same building id, indicating that they belong to the same building.



| ID | Name | 3DMesh |
|---|---|---|
| 1 | Roof | SLOB 1 |
| 1 | Door | SLOB 2 |
| 1 | ... | ... |
| 2 | ... | ... |
| 2 | ... | ... |
| 2 | ... | ... |

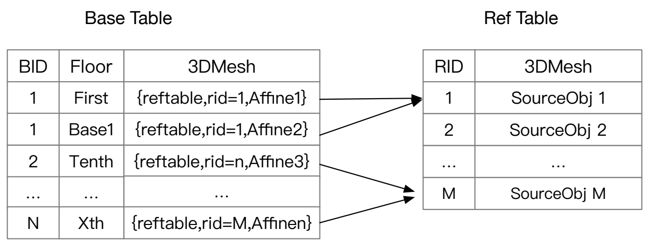
**Figure 3: Example of SLOBs in a 3DMesh data.**

In real life, there exist many objects that have minor differences. For example, in BIM, a building has many doors that have no difference in appearance but only different locations. It is a waste in storage to store each of them. Therefore, Ganos develops a technique called instance referencing. The idea is to use a source object to represent many objects having minor differences, and to use a reference table to store the source objects. The other tables store the ids of the source objects and a set of transformation information that directs how to transform the source object into the real object to be used. Figure 4 shows an example of instance referencing, where the first floor and base one of building one can be obtained by transforming SourceObject1, and the tenth floor of building two and $X$th floor of building $N$ can be obtained by transforming SourceObject$M$. It is possible that different buildings have objects referencing the same source object, e.g., they are produced by the same manufacturer. It should be noted that the source object and the object referencing it have the same data structure, and thus Ganos conducts operations on them indifferently.

## 4.3 Indexes

In PolarDB, the suggested way of adding indexes is to develop indexes that are based on GiST [23]. Following PolarDB, Ganos

*Geometry follows the hierarchy defined by OGC and SQL/MM Spatial


Base Table

| BID | Floor | 3DMesh |
|---|---|---|
| 1 | First | {reftable,rid=1,Affine1} |
| 1 | Base1 | {reftable,rid=1,Affine2} |
| 2 | Tenth | {reftable,rid=n,Affine3} |
| ... | ... | ... |
| N | Xth | {reftable,rid=M,Affinen} |

Ref Table

| RID | 3DMesh |
|---|---|
| 1 | SourceObj 1 |
| 2 | SourceObj 2 |
| ... | ... |
| M | SourceObj M |

**Figure 4: Example of instance referencing.**

develop indexes based on GiST+ which is an index framework that enriches the access methods of GiST. The extension includes adding a set of function interfaces to estimate index costs. Ganos implements a set of GiST+-based indexes (mostly R-tree) that organize the data based on different dimensions, e.g., $(x, y)$, $(x, y, z)$, or $(x, y, z, t)$. Thus allows Ganos to have multiple GiST+-based indexes on the same set of data, which is useful for answering the queries that consider different dimensions. For example, if users have a need for the queries on $(x, y, z)$ and $(x, y, z, t)$, respectively, Ganos allows users to build two GiST+-based indexes that are on $(x, y, z)$ and $(x, y, z, t)$, respectively, and selects between the two to answer a query on-the-fly. Even though there exists no GiST+-based index of which the indexed dimensions are the same to the queried dimensions, another one can still be used to answer that query if there exist common indexed dimensions between them. For example, a GiST+-based index built on $(x, y, z)$ can answer a query having a constraint on $(x, y, t)$ as they share common dimensions $(x, y)$. This is an improvement over a GiST-based index which does not work here, and it allows Ganos to optimize the processing of less often queries of which the queried dimensions are not indexed exactly. Moreover, Ganos implements a strategy to select the best GiST+-based index to answer a query when multiple candidates exist, which is based on the similarity of indexed and queried dimensions.

## 5 EXTENDED STORAGE

This section introduces the storage mechanism of Ganos.

### 5.1 Storage Mode

Figure 5 shows an overview of extended storage. As explained, a SLOB has *profile* and *details*, where *profile* is a summary of the object and *details* is the detailed information of the object. If *details* is small, it is stored inline in the table (the top one in the figure). Otherwise, it is stored in a TOAST (The Oversized-Attribute Storage Technique) table (the second one in the figure), which is used by PolarDB to store large objects. Both cases belong to storing in the database. Operations on the data stored in the database are efficient. However, the deployment costs are expensive when the data is of large scale. Since most queries can be answered using *profile*, while only a few queries require *details*, Ganos provides extended storage that separates the storages of *profile* and *details*, which stores *profile* in a database table and stores *details* on OSS. There are two modes of extended storage aiming at different scenarios, which are hot/cold data separation (the third and fourth ones in the figure) and heterogeneous file access (the bottom one in the figure).

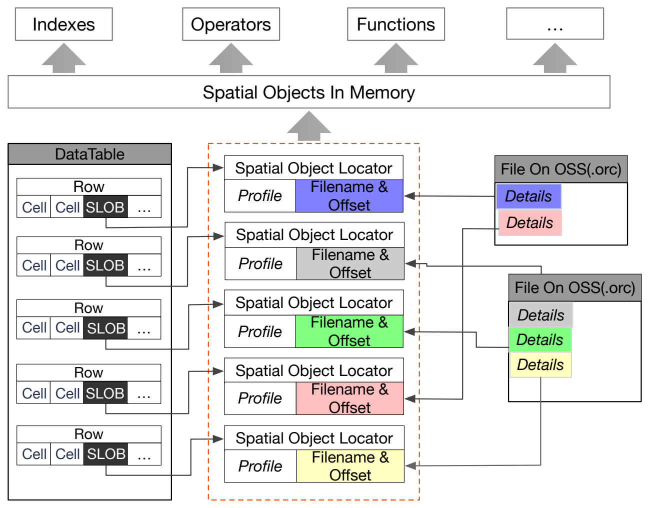


**Figure 5: Overview of extended storage.**

**Hot/cold data separation.** There exist large-scale *cold* data stored in database tables that are rarely used. Meanwhile, it is common that most queries consider a smaller fraction of data, i.e., *hot* data. For the users who care about deployment costs and are willing to tolerate the performance loss of a small ratio of queries, it is better to store the cold data on OSS because OSS is much cheaper than the database storage.

Periodically, Ganos exports *details* of objects that are rarely used to OSS files, which are created and managed by Ganos. It means that an OSS file stores the *details* of many objects that were stored in the database previously. During this process, Ganos changes *details* of each exported object into its address on OSS, which is composed of a file location and an offset in the file.

**Heterogeneous file access.** There are many sources of MDS data. Files of different formats are used to store the data from them. These heterogeneous files are of large size but for each of them only a small part is necessary for most queries, which need to be stored in the database. The files can be stored on OSS. To achieve this, Ganos extracts *profile* from each object in the files, pairing it with the file location, and writes the pair into a SLOB that is stored in a database table then. In this mode, an OSS file stores *details* of one SLOB only.

There exists difference between the two modes. Hot/cold data separation exports *details* of objects that were in the database previously to OSS files. It is useful for reducing the storage cost of database, especially for data of large scale such as Trajectory data. Heterogeneous file access extracts *profile* from objects stored on OSS and writes them into the database. An example is to extract *profile* from files that store Raster data.

### 5.2 Spatial Object Locator

At the central part of Figure 5 is spatial object locator (Locator, for short), which is the core of extended storage. It provides transparent access to data in different storages, and is responsible for maintaining the storage information. Specifically, when an operation only requires *profile* of a SLOB, e.g., creating spatial indexes, and filtering using bounding boxes, Locator can return *profile* directly without reading *details* from OSS, and when an operation requires *details* of a SLOB, e.g., computing topological relationship, and spatial clipping, Locator checks it is *details* or an address that is stored in the SLOB. If meeting an address, Locator automatically reads *details* from OSS and returns it to the operation. Otherwise, Locator returns *details* that is stored in the SLOB directly. By doing this, operations can consider all the objects as being stored in the database without knowing the storage details, and the same index can be used to organize all the objects based on their *profile*. Besides, the storage cost of the database is reduced greatly because *profile* and the address information are much smaller than *details* in size.

In summary, Locator connects the higher-level access methods and the storage layer. It hides the underlying storage details, and allows access methods to operate on the objects in a unified way.

## 5.3 Cache Mechanisms

Ganos provides cache mechanisms that include write caching and read caching to work with the extended storage.

**Write caching.** For the hot/cold data separation, Ganos utilizes write caching to reduce the network costs of exporting cold data from the database to OSS. It is to maintain a write cache to store all the cold data that is to be exported. When the cache is full, Ganos setups a network connection with OSS, and transfers all the cached data to OSS. The write cache reduces the network costs by avoiding frequent network connections and data transferring.

**Read caching.** A SQL query usually requires conducting operations on one object multiple times. This does not matter for objects that are stored in the database because PolarDB provides a read cache to avoid reading the same object multiple times from a table. However, for an object whose *details* is stored on OSS, PolarDB does not write the object's *details* into the cache after reading it from OSS, but writes its address into the cache. Therefore, it will result in multiple times of OSS readings. To solve this problem, in one query session, Ganos builds and maintains a read cache that adopts a least recently used (LRU) policy. After reading *details* of an object from OSS, Ganos writes it into the read cache. The following operations on that object will be redirected to the read cache, thus avoiding reading the same object from OSS multiple times.

## 6 QUERY PROCESSING

## 6.1 Query Types

Ganos supports multiple types of queries, including spatial queries, spatio-temporal queries, scene-oriented queries, and cross-model queries:

- Spatial queries only consider spatial dimensions. They include 3D relationships, 3D analysis, and 3D processing operations in 3D scenarios.
- Spatio-temporal queries consider both spatial and temporal dimensions. They include spatio-temporal relationships, spatio-temporal analysis, and spatio-temporal processing operations.
- Scene-oriented queries is a new query type supported by Ganos. They contain the scene information and require operations to construct, edit and process the scenes.

**Table 1: Functions for MDS data types**

| Category | Geom. | Raster | Traj. | 3DMesh | PointCloud | Count |
|---|---|---|---|---|---|---|
| 3D spatial relationship | ✓ | | ✓ | ✓ | ✓ | 69 |
| | Example: ST_3DIntersects(3DMesh, 3DMesh) - Check if two 3DMeshes spatially intersect in 3D | | | | | |
| 3D spatial analysis | ✓ | | ✓ | ✓ | ✓ | 52 |
| | Example: ST_3DBuffer(Geometry) - Compute a geometry that contains all points whose distance to the geometry is less than or equal to a given distance in 3D | | | | | |
| 3D spatial processing | ✓ | | ✓ | ✓ | ✓ | 90 |
| | Example: ST_3DIntersection(Pointcloud, Geometry) - Compute a new pointcloud representing the point-set intersection of input pointcloud and geometry in 3D | | | | | |
| Spatio-temporal relationship | | ✓ | ✓ | | | 31 |
| | Example: ST_3DIntersects(Raster, Raster) - Check if two rasters intersects in both intersects in both spatial(footprint) and temporal dimensions | | | | | |
| Spatio-temporal analysis | | ✓ | ✓ | | | 57 |
| | Example: ST_LCSSSimilarity(Trajectory, Trajectory) - Compute the similarity of two trajectories using LCSS algorithm with spatial and temporal criteria | | | | | |
| Spatio-temporal processing | | ✓ | ✓ | | | 53 |
| | Example: ST_Intersection(Trajectory, Trajectory) - Compute the same temporal points of two trajectories | | | | | |
| Scene edit | | ✓ | ✓ | ✓ | | 45 |
| | Example: ST_AddMaterial(3DMesh, Material) - Add a material to a 3DMesh | | | | | |
| Scene processing | | ✓ | ✓ | ✓ | | 56 |
| | Example: ST_Simplify(3DMesh) - Compute a simplified version of the given 3DMesh with geometry and other scene-oriented information | | | | | |
| Cross model processing | ✓ | ✓ | ✓ | ✓ | ✓ | 75 |
| | Example: ST_Intersects(Trajectory, 3DMesh) - Check if Trajectory and 3DMesh spatially intersects in 3D | | | | | |

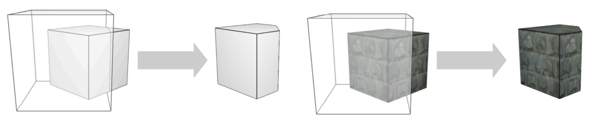


**Figure 6: Geometry intersection vs. scene intersection.**

- Cross-model queries are hybrid queries that involve multiple data types, e.g., overlay analysis of 3DMesh and Trajectory data.

To support above query types, Ganos provides a rich set of data types and functions, which are listed in Table 1. To improve the query performance, the most appropriate index will be used by functions according to the query type.

Note that the functions in Ganos are polymorphic, which implies the same function will behave differently according to the input arguments. For example, when two trajectory objects are fed to the intersection function, the spatial and temporal relationships will be analyzed. Figure 6 shows that if two 3DMesh objects are taken as input in the same function, the spatial information and scene-related information (e.g., textures and materials) will be considered.

## 6.2 Parallel Execution

Some queries on MDS data are extremely time-consuming, e.g., 3D building surface area calculations and trajectory similarity calculations. There are various reasons that make them slow. (1) Some queries may involve millions of rows even after most irrelevant data is filtered by indexes such as R-tress. (2) One SLOB can grow up to several MBs and operations on SLOBs are complicated.

To accelerate these queries, this paper proposes spatial-oriented multi-level parallelism in PolarDB, including IQP and IFP. The overview of the parallelism is shown in Figure 7.

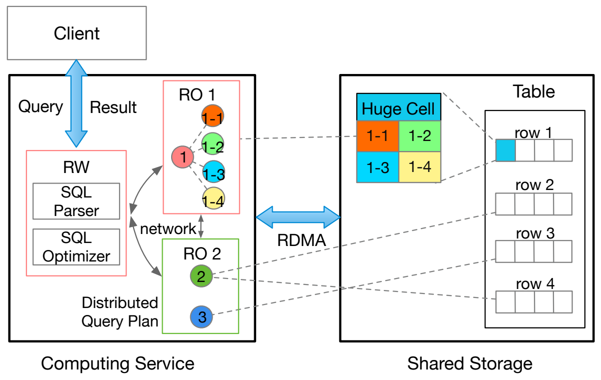


Figure 7: Overview of Ganos parallelism.

**Intra-query parallelism (IQP)**. The IQP splits the millions of rows into multiple data slices, which can be processed simultaneously on several RO nodes. IQP starts with the generation of a distributed plan. The plan decides the degree of IQP and the nodes participating in the query. The default size of each data slice is 4MB (512 pages), and the default data slice assignment scheme is hash-based. Ganos also provides a dynamic data slice assignment scheme, which assigns a data slice to an idle process on-the-fly by keeping the data slices in a queue, and a leader process being responsible for the assignment of the data slices to the processes having finished their job. A user can set a different data slice size and use the dynamic data slice assignment scheme based on his/her needs. For example, if a user wants better load balance, he/she can set the size of each data slice as one page, and adopt the dynamic data slice assignment scheme.

When executing the plan, each node may start multiple processes. Processes can communicate with others through network protocols or shared memory, depending on whether they are on the same node. Because PolarDB decouples storage and computation, all processes in all nodes can access the entire data from the shared storage. Thus, many costly operations of parallel executions can be avoided, e.g., data shuffling. Besides, the parallel degree can be adjusted flexibly for different queries.

**Intra-function parallelism (IFP)**. IFP accelerates the processing of huge cells, i.e., SLOBs of great size, in a tuple that is compute-intensive and cannot be split into small data slices through IQP. When a worker process meets a huge cell, IFP divides the huge cell into several small cells by spatio-temporal information and processes these small cells with subprocesses in parallel. For the Trajectory data, the huge cell can be segmented by the temporal dimension; for the 3DMesh, it can be segmented by spatial grid.

Figure 8 gives an example of IFP. The black squares in the left-most picture are the meshes (in the *Geometry* attribute) stored in one 3DMesh SLOB. We want to combine all the meshes into a single shape efficiently. IFP first classifies them into 16 grids (green grids in Figure 8), and starts subprocesses to compute the results in each of them. When all subprocesses are finished, their results are aggregated and illustrated in the second picture. IFP again classifies them into four larger grids and repeats this procedure. The final result is given when all the grids are aggregated.

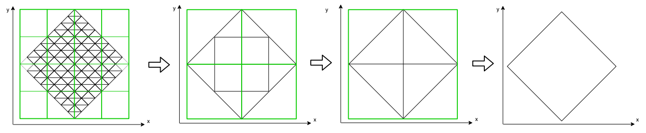


Figure 8: Example of intra-function parallelism.

Figure 7 gives an example of the whole procedure. One RW and two RO nodes are employed to speed up the union processing for the 3D building data containing four rows, in which the first row has a huge cell. The processing consists of the following steps: (1) The RW node uses the query optimizer to generate a distributed query plan, and distributes the plan to the RO1 and RO2 nodes. (2) The RO1 node reads the first row. Since the record contains a huge cell, the cell is divided into four MBRs. Therefore, the RO1 node will create four subprocesses to process the huge cell. RO1 adopts the IFP mechanism. (3) Worker process 2 in RO2 node will process row 2 and row 4 data. This is because worker process 2 is idle after processing row 2 and will continue processing the next row; worker process 3 in RO2 node will process row 3. RO1 and RO2 collectively adopt the IQP mechanism. (4) After all worker processes finish the tasks, RW aggregates all the results and returns them to the client.

## 6.3 Query Optimizations

Since the access performance of OSS is two orders of magnitude slower than that of local disk, query performance is affected when spatial data are stored on OSS. To reduce the impact of reading data from OSS, Ganos implements the spatial type analyzer and optimizes cost model for OSS storage.

**Spatial Type Analyzer.** The query optimizer of PolarDB uses a type analyzer to collect statistics of data tables and store them in a metadata table. Then, the statistics are used to estimate the selectivity of the different query predicates and select the optimal execution plan. The type analyzer provides analysis functions that are triggered by running ANALYZE command. Ganos implements its analyzer for each extended type to collect the distribution histograms of tables. For example, the geometry analyzer collects spatial distribution histogram, and the trajectory analyzer collects spatial and temporal distribution histograms. As Ganos supports the OSS storage, Ganos type analyzers need to collect additional storage distribution histogram. They also collect the tuples and the number and percentage of tuples using OSS storage.

**Storage Cost Model.** The cost evaluation model is used to select the optimal execution plan by query optimizer of PolarDB. It considers the pages scanning costs, tuples processing costs, functions and operators execution costs, etc. It does not consider the costs of accessing data that are stored on OSS, thus making it not accurate for estimating the costs of a query plan that requires *details* stored on OSS. To solve this problem, Ganos improves the cost model to consider the costs of reading OSS. Specifically, Ganos adds a new part into the cost model: $C_{\text{new}} = C_{\text{old}} + \Delta C$, where $C_{\text{old}}$ represents the costs output by the old cost model, and $\Delta C$ represents the extra costs of reading OSS data in a query.

The value of $\Delta C$ is computed differently based on the scanning method to be used. Sequence scan (seqscan, for short) and indexscan are the two most common scanning methods. Seqscan scans a table sequentially. The extra costs of seqscan are computed by multiplying the number of tuples in a table whose *details* are stored on OSS and the average cost of reading a *details* from OSS. Indexscan scans a table using an index, which reduces the number of tuples to be accessed by utilizing the filtering effect of the index. The extra cost of indexscan are computed by multiplying the number of tuples whose *details* are stored on OSS and that cannot be filtered by the index, and the average cost of reading a *details* from OSS.

The improved cost model is used to select the optimal query plan for a query that asks to access *details* of many SLOBs on OSS. In particular, for a query that has a high selectivity, the old cost model will prefer using seqscan because it is smaller in cost. However, it is probably better to use indexscan because that the costs of reading OSS data overwhelm the extra scanning costs. The improved cost model takes into consideration the costs of reading OSS data, and thus assist optimizer in selecting the optimal query plan.

## 7 EVALUATION

### 7.1 Experimental Setup

**Testbed.** Four PolarDB instances with Ganos are deployed on the nodes equipped with 32 CPUs and 256GB memories. One instance works as the RW node, and the other three instances work as RO nodes. These database instances are connected to a distributed storage by a high-speed RDMA network.

**Datasets.** Ganos is evaluated with one public dataset and three private datasets that are from real applications. The details of these datasets are listed below:

- OSM (201GB) is a public dataset that we crawled from online.[1] It contains 96,648,669 trajectories. The average number of points in one trajectory is 49.8 and the time span is 20 years.
- DT4 (100MB) is a dataset composed of four different types of data: UAV trajectory, digital elevation model (DEM), no-fly zone, and BIM data for the restricted-fly zone. It is used in a digital twins scene to demonstrate the capability of Ganos to handle cross-model queries considering different types of data.
- BIM1000 (428GB) is a BIM dataset. It contains the information of 1,000 buildings composed of 11,106,095 components. Each component has 313 points and 327 surfaces on average and 896,638 points and 1,062,881 surfaces at most.
- RASTER1718 (519GB) is a satellite image dataset of two-meter resolution. It contains 1,718 images, and the size of each image is 300MB on average and 2.4GB at maximum.

### 7.2 Use Case Study

In this case, the DT4 dataset is used to demonstrate the processing of a cross-model query in a city digital twin scene with different data types (as shown in Figure 9).

[1]https://www.openstreetmap.org/traces

**Figure 9: A cross-model query in a city digital twins scene.**

It is necessary to conduct an operation guideline check on UAV's flight trajectories to confirm that all the operations are legal. The requirements for a legal flight include:

(1) The maximum height from the ground of the flight must be lower than a certain threshold. Since the elevation recorded by GPS is based on the reference ellipsoid, a further relative height calculation is needed according to the DEM data.
(2) The maximum flight speed must be lower than a certain threshold. The flight speed has been recorded on the track.
(3) UAV must not touch the restricted-fly zone. The restricted-fly zone is a 3D space that no flight is allowed inside, e.g., the space whose distance from a building is less than 100 meters. The restricted-fly zones are usually generated by BIM data.
(4) UAV must not cross the no-fly zone. The no-fly zone is a 3D space where no flight can cross at any height.

The data used in this case study include UAV's flight of 3D Trajectory type, DEM of Raster type, no-fly zones of Geometry type, and BIM of 3DMesh type. They are stored in different tables of the database. Since no-fly zone check and restricted-fly zone check are similar, this paper only takes restricted-fly zone check for example. SQL 1 shows the SQL statement to conduct the check.

The relative flight height is calculated by the difference of the ground height based on the DEM and the height recorded by the GPS. The maximum flight speed is determined by calling a function to obtain the speed values recorded in the UAV trajectory. Whether the flight crosses the restricted-fly zone is checked by calling a function named ST_3DIntersects.

This case study illustrates the capabilities of Ganos to support MDS data and the interoperability for different types of MDS data. By using simple SQL to perform the complex cross-model queries in the applications of digital twins, users can benefit from reducing the development costs and complexity of the systems.

### 7.3 Evaluation of Extended Storage

This section evaluates the extended storage of Ganos using the OSM dataset. As stated in Section 5, Ganos designs a hot/cold separation mode and a heterogeneous file access mode for different scenarios. In this section, we illustrate the performance of the hot/cold separation mode along with the R-tree index based on GiST+, while the heterogeneous file access mode will be illustrated later in *Q*2 of

**SQL 1:** SQL statements

```
/* Max relative height. 1.st_addz to get ground elevation;
   2.get relative height; 3.get maximum relative height.   */
WITH height AS (
 SELECT st_z((st_dumppoints(traj)).geom) - st_z((st_addz(rast, traj)).geom)
  AS h
 FROM t_trajectory, t_dem WHERE t_dem.id = 1 AND t_trajectory.id = 1),
  max_height AS (
 SELECT max(h) AS max_relative_height FROM height ),
/* Max speed. 1.st_attrsfloat8 to get the speed; 2.get the
   maximum speed.                                          */
max_speed AS(
 SELECT max(speed) AS max_fly_speed FROM (
 SELECT unnest(st_attrsfloat8(traj, 'speed')) AS speed
 FROM t_trajectory WHERE id = 1 ) t ),
/* Buffer zones touch check. 1.st_3dbuffer to buffer
   buildings; 2.st_3dintersects to check if trajectory
   touches the buffer zones.                               */
cross_buiding AS(
 SELECT NOT EXISTS (
  SELECT 1 FROM t_trajectory, t_building
  WHERE st_3dintersects( st_3dbuffer(t_building.m, 100), t_trajectory.traj)
  AND t_trajectory.id = 1) AS cross_restricted_zone )
/* Final check.                                            */
SELECT max_relative_height < 500 AND max_fly_speed < 60 AND
  cross_restricted_zone
FROM max_height, max_speed, cross_building.
```

Section 7.4. Based on customers' situations, this section assumes 20% of the dataset is hot data, and 80% of queries search for hot data. In addition, this section assumes the hot data is the more recent part (judged by trajectories' end times) of the dataset.

In the following experiments, the percentage of data stored on OSS is varied. Figure 10 shows the change of table size after 0%,20%,40%,60%,and 80% of the dataset is exported to OSS. After the original data are imported into the database, it is compressed and occupies 109GB. The table size in the database decreases linearly, and the file size on OSS increases linearly as more data is exported to OSS. The cold data in the database has a size of about 88GB in the database, and when it is exported to OSS, it occupies only 19GB in the database, achieving a 21.6% compression rate. The compression rate in the database is in reverse proportion to the size of the SLOBs' *details* data; for longer trajectories with rich attribute and event information, e.g., AIS ship tracks, the compression rate can be less than 1%.

Figure 11 shows the index construction time against the ratio of on-OSS data. The index construction time is roughly 40 minutes when the ratio of on-OSS data varies from 0% to 80%. Ganos can obtain the MBRs from a Locator to build an index without accessing the OSS, and thus the ratio of on-OSS data has minor influence on the index construction time. Without a Locator, the index construction time takes over twelve hours even on only 20% on-OSS data, which is not included in the figure. The experiment demonstrates the effectiveness of the Locator.

The performance of biased range queries is tested with $500m \times 500m \times 7$days query ranges. Biased queries assume that the centers of the query ranges have similar spatio-temporal distribution as the dataset does. Since Ganos provides transparent accesses to OSS, each query can access both in-database and on-OSS data. Figure 12 illustrates the QPS of the queries. It can be seen that storing cold data on OSS reduces QPS. When all cold data is stored on OSS, the QPS is roughly halved. Figure 13 shows the change of latency distribution (measured by frequency per minute) with different ratios of on-OSS data. The results are aggregated by six response-time ranges: 0–50ms, 50–100ms, 100–500ms, 500–1000ms, 1,000–2,000ms, and more than 2,000ms. It can be seen that most queries are finished in 50ms, because most irrelevant data is pruned by the indexes. Moreover, when the ratio of on-OSS data increases, there appear more queries with longer latencies.

Additionally, the performance of range queries with different query ranges is compared. Range queries are performed with $100m\times 100m$, $500m \times 500m$, $2000m \times 2000m$ spatial ranges and 1-day, 7-day, 30-day temporal ranges. The QPSs of the queries are illustrated for tables with 0% (marked as In-DB) and 80% (marked as On-OSS) data exported to OSS in Figure 16. For larger query ranges, the QPS of both setups decreases. The performance of the extended storage is affected more because the index has weaker pruning power on larger query ranges. For the 1-day temporal ranges, the QPS with the extended storage is 60% of that without OSS; for the 30-day temporal ranges, the QPS with the extended storage is 35% of that without OSS.

The experiments illustrate the abilities of Ganos on exporting data to OSS, building indexes on tables with extended storage, and querying through indexes with transparent accesses to OSS. The results show that OSS can reduce storage cost with an acceptable sacrifice of QPS. Although reading data from OSS is slow, with the help of the indexes, the query performance on spatio-temporal queries can become acceptable. The results also show that the extended storage suits better for queries with small query ranges and without a strong requirement on latency. However, it affects the latency, and harms query performance when the query range is large.

## 7.4 Evaluation of Parallel Query Processing

In this set of experiments, we evaluate the performance of the spatial-oriented multi-level parallelism of Ganos. Two big queries are used for the evaluation, which are described below.

*Q*1**:** This query asks to project each building in the BIM1000 dataset into the *xy*-plane and compute its area.

*Q*2**:** This query asks to collect the statistics of each Raster data in the dataset RASTER1718, which is stored on OSS.

**Systems to compare against.** The PostgreSQL compatible parallelism (hereafter simplified as PostgreSQL) is used, which is one of the parallel query strategies supported by PolarDB and is inherited from PostgreSQL. It can only parallelize the processing of a query in one node and cannot work with IFP.

*7.4.1 Results for Q1.* Figure 14 shows the results of running *Q*1, where IFP-*x* (resp. IQP-*x*) represents that the degree of the parallelism of IFP (resp. IQP) is *x*. It takes more than three hours to run *Q*1 without parallelism, so we do not show that result in this figure. It can be seen that the parallelism of Ganos performs much better than PostgreSQL. For example, the latency of IQP-24 and IFP-4 is 66% lower than that of PostgreSQL whose parallelism degree is 96. There are several phenomena that worth discussing, which are described in the following.

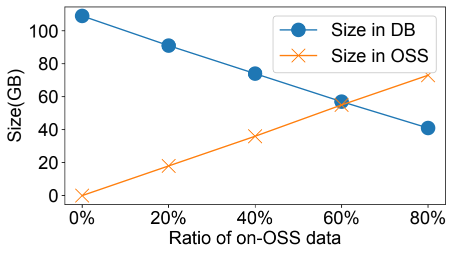


**Figure 10: Size vs. ratio of on-OSS data.**

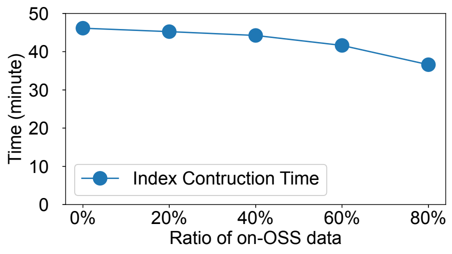


**Figure 11: Index construction time.**

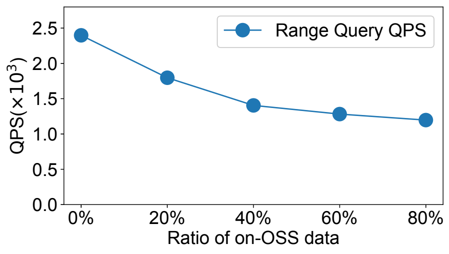


**Figure 12: QPS vs. ratio of on-OSS data.**

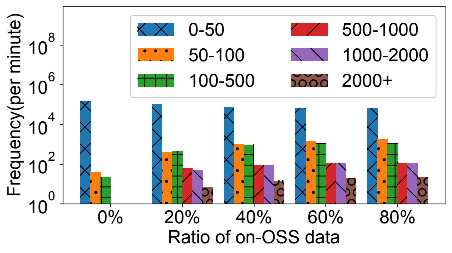


**Figure 13: Latency distribution.**

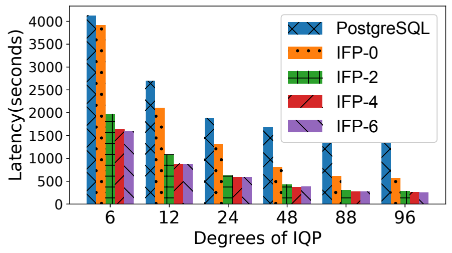


**Figure 14: Q1 latency vs. #parallelism.**

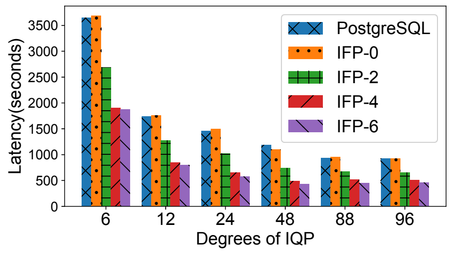
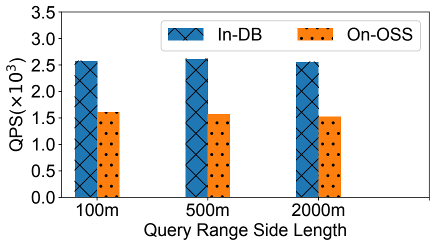
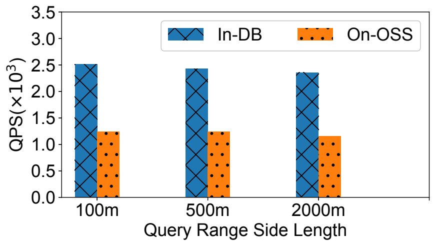


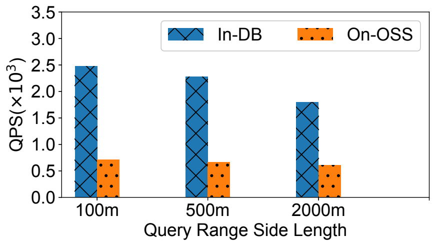
**Figure 15: Q2 latency vs. #parallelism.**



(a) QPS for 1 day



(b) QPS for 7 days



(c) QPS for 30 days

**Figure 16: QPS for different queries.**

First, the IQP without IFP (called IFP-0) is more efficient than PostgreSQL. For instance, PostgreSQL takes 1.4x (resp. 2x) more time than IFP-0 when IQP is 24 (resp. 48). The main reasons are: (1) The inter-process communication mechanisms of Ganos and PostgreSQL are different. In PostgreSQL, the worker processes need to ask the leader process for data. It causes extra waiting time if there are many worker processes. In comparison, each process of Ganos can obtain the data for processing independently, which reduces the inter-process communication costs. (2) The IO mechanisms of Ganos and PostgreSQL are different. In PostgreSQL, there is only one process that can obtain data at one time. While in Ganos, processes can obtain data simultaneously.

Second, the IFP facilitates the processing of huge cells. Figure 14 shows that: (1) The latency of IFP-2 is almost half of that of IFP-0 at the same IQP degree, e.g., 577 seconds for IQP-96 and IFP-0 vs. 288 seconds for IQP-96 and IFP-2. (2) With the same degree of parallelism, IQP+IFP outperforms IQP only. For instance, the latency of IQP-24 and IFP-2 is 628 seconds, while the latency of IQP-48 and IFP-0 is 810 seconds. This is because IFP can parallelize the processing of huge cells, thus improving the query performance.

Third, the improvement in performance becomes smaller with the increasing of the degree of the parallelism. That is because more time is spent on scheduling and communication if there are more workers, and the aggregating progress by the leader process cannot be parallelized neither.

Even so, Ganos is more scalable than PostgreSQL, e.g., IQP-96 and IFP-0 has a smaller latency than IQP-88 and IFP-0, i.e., 577 seconds vs. 617 seconds, while the latency of PostgreSQL is 1,728 seconds when the degree of parallelism is 96, which is larger than that when the degree of parallelism is 88, i.e., 1,673 seconds.

*7.4.2 Results for Q2.* Figure 15 shows the results of running *Q*2 (It takes about six hours to run the query without any parallelism, and the result is not shown in this figure). Similar patterns can be found that the parallelism of Ganos performs better than PostgreSQL, e.g., 655 seconds for IQP-24 and IFP-4 vs. 921 seconds for PostgreSQL when the degree of parallelism is 96. In the following, we discuss different patterns that are not shown in Figure 14.

First, PostgreSQL performs slightly better than using IQP w/o IFP when the degree of parallelism is 6, 12, or 24. This is due to load imbalance caused by the data skew. IQP parallelizes the processing of data at the granularity of blocks, while PostgreSQL parallelizes at the granularity of tuples. Since PostgreSQL uses a finer granularity than IQP, data skew is more likely to occur in IQP when the degree of parallelism is smaller.

Second, the difference among the evaluated parallelisms for the same IQP degree is more obvious than that in Figure 14. For instance, in Figure 15, the latency for IQP-48 and IFP-4 is 66% of that for IQP-48 and IFP-2, while in Figure 14, it is 87%. This is because that more huge cells are invoked for the dataset RASTER1718 than for the BIM1000 dataset, which cannot be parallelized in IQP.

Overall, the experimental results prove two points: First, the parallelism of Ganos performs better than PostgreSQL in most cases. Second, it is better to use IQP and IFP collectively than using IQP only when there exist huge cells.

## 8 LESSONS LEARNED

After it was released in 2018, Ganos has offered service in Alibaba Cloud for over four years. Various spatial/spatio-temporal applications from small Internet applications to large enterprise applications have been implemented with Ganos. Most of these applications focus on traditional 2D static spatial data management. However, in recent years, a lot of new applications have emerged with new data types like digital twins, location sensing, etc. They represent the future trend and lead the traditional spatial databases to the new era. This section introduces several novel applications, the challenges encountered, and our solutions.

**Multidimensional scenes and 3D analytics.** The integration of BIM and 3D GIS is the foundation of multidimensional scenes construction and 3D aided analytics. BIM data contains a huge number of delicate components and poses great challenges in the complexity analysis and computational efficiency. One of our digital government customers, who is responsible for the urban planning and construction of a State-Level New Area, aims to make full use of the BIM data for the approval of the construction project. One crucial step is the fast computation of the overall ground projection of the target building. However, for a large auditorium with a peculiar roof structure, its BIM data contains millions of polygons, which cannot be quickly computed in the external middleware of the systems due to the unbearable overhead of network transmission.

Ganos utilizes the built-in 3DMesh data type to manage the BIM data, and adopts spatial-oriented multi-level parallelism to achieve in-database computation acceleration. As a result, Ganos improves the efficiency of projection computation by nearly 100 times and solves the problem for customers. The inspiration from these scenarios is that in the urban digital twin business, customers have changed from only focusing on 3D visualization to 3D analysis and computing. The integration of spatial databases and cloud-native technology will be more and more important in this field.

**Querying dynamic data.** The moving objects such as vehicles, ships, and aircrafts are crucial elements for building city digital twins systems. Ganos has been applied to a series of MOD-based (Moving Object Database) scenarios in the cloud, including LBS bike-sharing management, agricultural credit systems, transportation systems, and airlines systems etc. Following are several practical problems encountered in the applications.

*Scene management.* Some bike-sharing service providers try to model the locking and unlocking events and record the complete itinerary of a rent event. Ganos solves this problem by adding scene-oriented event processing in the trajectory models.

*Extended storage.* Some LBS service providers store the trajectories data into the databases with customized JSON formats and periodically extract the cold data into OSS to reduce the storage cost. This strategy makes it difficult to access the cold data. Ganos elegantly solves this problem by providing built-in Trajectory types.

*Cross-modal queries.* Some vehicles service providers would like to support cross-modal queries. For example, based on the overlay analysis of Trajectory data and Raster data, they want to compute the climbing height of the vehicles. Ganos offers the SQL interfaces to support this type of query.

**Database for GeoAI** Satellite photographing is an effective method to obtain the data in future digital twins applications. The integration of GeoAI (geographical AI) and remote sensing images can significantly enhance large-scale geographical spatial data analytics and processing ability. In recent years, GeoAI providers have an increasing demand for temporal and spatial resolutions of remote sensing images. The traditional static-tiles based Raster data management cannot meet the following requirements: (1) The databases can directly manipulate OSS-based raw image files to avoid redundant storage. (2) Remote sensing images should be structurally stored in spatial databases to enable spatial and spatio-temporal queries. (3) Support parallel mechanism to improve the query efficiency. To meet these requirements, Ganos supports the Raster data type natively, allowing the integration with heterogeneous files on OSS. In addition, Ganos uses spatial indexes and spatial-oriented multi-level parallelism to accelerate queries for the Raster data. The above solutions have been extensively applied to the national territory spatial plan and governance, dynamic monitoring of ecological environment, water resource management (watershed protection, river and lake supervision), financial credit, etc.

## 9 CONCLUSION

With the rapid development of smart cities, digital twins, and cloud computing, the existing spatial relational databases cannot meet the requirement of modern applications for MDS data processing. To address this issue, in Alibaba, a cloud-native spatial database engine called Ganos is designed and implemented on PolarDB. Ganos provides a systematic framework of data models, access methods, and operations for MDS data. Especially, Ganos optimizes the processing of queries on MDS data through cloud-native capabilities, which provides a new practice of moving from traditional on-premise spatial database to cloud-native spatial database. The future work will leverage GPU resources on the cloud to accelerate Ganos and utilize the PolarDB serverless framework to achieve better dynamic resource provisioning.

## REFERENCES

[1] KiCad Services Corp. [n.d.]. *FreeCAD: Your own 3D parametric modeler.* KiCad Services Corp. Retrieved January 25, 2022 from https://www.freecadweb.org/

[2] Oracle [n.d.]. *MySQL Documentation.* Oracle. Retrieved January 25, 2022 from https://dev.mysql.com/doc/

[3] Oracle [n.d.]. *Oracle's Spatial Database.* Oracle. Retrieved January 25, 2022 from https://www.oracle.com/database/spatial/

[4] The PostgreSQL Global Development Group [n.d.]. *PostgreSQL: The World's Most Advanced Open Source Relational Database.* The PostgreSQL Global Development Group. Retrieved January 25, 2022 from https://www.postgresql.org/

[5] Open Geospatial Consortium [n.d.]. *Simple Feature Access, Part 1: Common Architecture.* Open Geospatial Consortium. Retrieved January 25, 2022 from http://www.opengeospatial.org/standards/sfa/

[6] PostGIS Project Steering Committee (PSC) [n.d.]. *Spatial and Geographic objects for PostgreSQL.* PostGIS Project Steering Committee (PSC). Retrieved January 25, 2022 from https://postgis.net/

[7] Oracle [n.d.]. *Spatial Developer's Guide.* Oracle. Retrieved February 23, 2022 from https://docs.oracle.com/en/database/oracle/oracle-database/21/spatl/

[8] SQLite Consortium [n.d.]. *SQLite.* SQLite Consortium. Retrieved February 7, 2022 from https://www.sqlite.org/index.html

[9] Amazon [n.d.]. *Working with Amazon Aurora PostgreSQL.* Amazon. Retrieved January 25, 2022 from https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Aurora.AuroraPostgreSQL.html

[10] David M. Adler. 2001. DB2 Spatial Extender-Spatial data within the RDBMS. In *Proceedings of 27th International Conference on Very Large Data Bases (PVLDB '01).* 687–690.

[11] Mladen Andzic, Van To, Mike Ray, Craig Guyer, Saisang Cai, and Douglas Laudenschlager. [n.d.]. *Spatial Data (SQL Server).* Microsoft. Retrieved January 25, 2022 from https://docs.microsoft.com/en-us/sql/relational-databases/spatial/spatial-data-sql-server?view=sql-server-ver15

[12] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, et al. 2019. Socrates: The new SQL server in the cloud. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '17).* 1743–1756.

[13] Mohamed Bakli, Mahmoud Sakr, and Esteban Zimanyi. 2019. Distributed moving object data management in MobilityDB. In *Proceedings of the 8th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data (BigSpatial '19).* 1–10.

[14] Rudolf Bayer and Edward McCreight. 2002. Organization and maintenance of large ordered indexes. In *Software Pioneers: Contributions to Software Engineering.* Springer, 245–262.

[15] Martin Breunig, Patrick Erik Bradley, Markus Jahn, Paul Kuper, Nima Mazroob, Norbert Rösch, Mulhim Al-Doori, Emmanuel Stefanakis, and Mojgan Jadidi. 2020. Geospatial data management research: Progress and future directions. *ISPRS International Journal of Geo-Information* 9, 2 (2020), 95.

[16] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proceedings of the VLDB Endowment* 11, 12, 1849–1862.

[17] Alexandre Carvalho, Cristina Ribeiro, and António Augusto Sousa. 2006. Spatial timedb-valid time support in spatial dbms. In *Proceedings of 2nd International Advanced Database Conference (IADC '06).*

[18] Alexandre Carvalho, Cristina Ribeiro, and A. Augusto Sousa. 2006. A spatio-temporal database system based on timedb and oracle spatial. In *Research and Practical Issues of Enterprise Information Systems.* Springer, 11–20.

[19] Judith R. Davis. 1998. IBM's DB2 spatial extender: Managing geo-spatial information within the DBMS. *IBM corporation* (May 1998).

[20] Li Deren, Yu Wenbo, and Shao Zhenfeng. 2021. Smart city based on digital twins. *Computational Urban Science* 1, 1 (2021), 1–11.

[21] Alessandro Furieri. [n.d.]. *SpatiaLite.* Retrieved February 7, 2022 from https://www.gaia-gis.it/fossil/libspatialite/index

[22] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data (SIGMOD '84).* 47–57.

[23] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. 1995. *Generalized search trees for database systems.*

[24] ISO/IEC. 2016. Information Technology—Database Languages—SQL Multimedia and Application Packages—Part3: Spatial.

[25] Ahmet Kucuk, Shah Muhammad Hamdi, Berkay Aydin, Michael A Schuh, and Rafal A Angryk. 2016. Pg-trajectory: A postgresql/postgis based data model for spatiotemporal trajectories. In *IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom '16).* IEEE, 81–88.

[26] Feifei Li. 2019. Cloud-native database systems at Alibaba: Opportunities and challenges. In *Proceedings of the VLDB Endowment (PVLDB '17, Vol. 12).* 2263–2272.

[27] Luís Eduardo Oliveira Lizardo and Clodoveu Augusto Davis Jr. 2017. A PostGIS extension to support advanced spatial data types and integrity constraints. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '17).* 1–10.

[28] S. Logothetis, E. Valari, E. Karachaliou, and E. Stylianidis. 2017. Spatial DMBS architecture for a free and open source BIM. *International Archives of the Photogrammetry, Remote Sensing & Spatial Information Sciences* 42 (2017).

[29] N Mazroob Semnani, PV Kuper, M Breunig, and M Al-Doori. 2018. Towards an intelligent platform for big 3d geospatial data management. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences* 4 (2018), 133–140.

[30] Georgios Mylonas, Athanasios Kalogeras, Georgios Kalogeras, Christos Anagnostopoulos, Christos Alexakos, and Luis Muñoz. 2021. Digital Twins From Smart Manufacturing to Smart Cities: A Survey. *IEEE Access* 9 (2021), 143222–143249.

[31] Ehab Shahat, Chang T Hyun, and Chunho Yeom. 2021. City digital twin potentials: A review and research agenda. *Sustainability* 13, 6 (2021), 3386.

[32] Sara Shirowzhan, Willie Tan, and Samad ME Sepasgozar. 2020. Digital twin and CyberGIS for improving connectivity and measuring the impact of infrastructure construction planning in smart cities. *ISPRS International Journal of Geo-Information* 9, 4 (2020), 240.

[33] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17).* 1041–1052.

[34] Z. Yao, C. Nagel, F. Kunde, G. Hudra, P. Willkomm, A. Donaubauer, and TH Kolbe. 2018. 3DCityDB - a 3D geodatabase solution for the management, analysis, and visualization of semantic 3D city models based on CityGML. *Open Geospatial Data, Software and Standards* 3, 5 (2018), 1–26.

[35] Jianting Zhang, Michael Gertz, and Le Gruenwald. 2009. Efficiently managing large-scale raster species distribution data in PostgreSQL. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '09).* 316–325.

[36] Esteban Zimányi, Mahmoud Sakr, and Arthur Lesuisse. 2020. MobilityDB: A mobility database based on PostgreSQL and PostGIS. *ACM Transactions on Database Systems (TODS)* 45, 4 (2020), 1–42.