OrientDB: A NoSQL, Open Source MMDMS

Daniel Ritter[1,2], Luigi Dell'Aquila[1], Andrii Lomakin[1] and Emanuele Tagliaferri[1]

[1]OrientDB / SAP SE, Walldorf (Baden), Germany
[2]Hasso Plattner Institute, University of Potsdam, Potsdam, Germany

Abstract

OrientDB is a full-function, NoSQL MMDMS (multi-model database management system), addressing the big data *variety* problem with one single, multi-model store and a SQL-based, multi-model query language. It combines graph and semi-structured data management with object-oriented, text and spatial capabilities, and features a variety of deployment and distribution / replication options, transactional / ACID storage and indexing, making it a commercially successful MMDMS. With that, OrientDB is well-suited for novel adaptations of applications like smart logistics, asset management, social data storage and analysis, and other use cases that require multiple perspectives on the data.

While OrientDB's initial open source release was in 2010, many improvements like the multi-model API / query language and full ACID support came with a recent re-design. This paper gives the first comprehensive description of the revamped system, focusing on its data model, query language, distribution models and software architecture. We assess the current state of OrientDB's performance compared to other multi-model, but also "best-in-class" NoSQL single-model document systems.

Keywords

Graph data management, Multi-model database, NoSQL, Object-oriented, Semi-structured data

1. Introduction

Modern analytical, business and smart applications (e. g., in the areas of logistics, asset management, and social network analysis) require efficient storage and access to large amounts of multi-model data [1, 2, 3]. Data management platforms addressing the multi-model or in general the big data *variety* problem could be more generally referred to as multi-model data platforms [1] (e. g., as in the Q3/2021 Forrester wave[1] that especially stresses on the importance of such a polyglot persistence model for modern applications).

In the literature, these multi-model data platforms are commonly differentiated into polystores (e. g., BigDAWG [4]) and multi-model database management system (MMDMS) [1, 5], according to the number of separate data stores: polystores have multiple, federated data stores and MMDMSs have one single, integrated data store. In practice, there are several (commercial) multi-model data platforms[2] that are (a) more on the polystore side, e. g., from Microsoft, Oracle, and SAP, or (b) on the multi-model database side, e. g., from ArangoDB, Couchbase, DataStax, Redis Labs, MarkLogic, or (c) aspiring "best-in-class" NoSQL systems like MongoDB (document) and Neo4j (graph) (all in the Forrester Wave).

While all of them support most of the data models such as relational / table, graph, document or geospatial, we limit the systems relevant to this work to those selected for our evaluation shown in Tab. 1. Although MongoDB (representing (c)) is a commercial, document store, it started to extend support for other data models like table, graph and key-value. Similarly, Redis (cf. (b)) is originally a key-value database, which added support for document and graph. Postgres (cf. (a)) and OrientDB (community edition) do not appear in the Forrester Wave report, since they do not have a commercial offering per se. However, for Postgres several extensions exist that, e. g., add JSON document, key-value (hstore) and time series (timescaledb) support to its relational core. The selected systems all support time, geospatial and text.

In this paper we give the first comprehensive description of key aspects of one of the earliest NoSQL, open-source MMDMSs that was designed from ground up with a multi-model data design, namely OrientDB[3]. As such, OrientDB uniquely combines object-oriented principles like inheritance with semi-structured document (schema-less and strict / mixed schema) and their relationships in a (TinkerPop Blueprints compatible) graph with native and Gremlin graph traversal and pattern matching support. OrientDB's design decicions were influenced by the key MMDMS differentiators, namely:

(1) single, unified data model covering multiple data formats for complex data modeling (→ **data model**)
(2) SQL-like, multi-model query / operations, e. g., traversal, matching, lookup (→ **querying data**)
(3) single multi-model, ACID-transacted store / persistence (→ **persisting data**)

daniel.ritter@{sap.com|hpi.uni-potsdam.de} (D. Ritter)
https://github.com/dritter-sap (D. Ritter)
0000-0001-6146-3365 (D. Ritter)

CEUR Workshop Proceedings (CEUR-WS.org)

[1]Forrester wave – Multimodel data platforms, visited 11/2021: https://bit.ly/3iopfdI

[2]Acknowledging that the list is not complete and does not mention early MMDMSs like Berkeley DB (1994) or Virtuoso (1998).

[3]OrientDB, visited 11/2021: https://git.io/JRY6H

Table 1
Multi-model database management systems

	Table	Graph	Document	Key-Value	Object-oriented	Text	Time	Spatial
MongoDB (2009)	👍	👍	👍	👍	👎	👍	👍	👍
Postgres/JSON (2013)	👍	👍	👍	👍	👍	👍	👍	👍
Redis (2009)	👎	👍	👍	👍	👎	👎	👍	👍
OrientDB (2010)	👎	👍	👍	👍	👍	👍	👎	👍

👍: supported, 👍: partially supported, 👎: not supported

(4) seamless scaling (→ **distribution model**)

OrientDB aims to cover all of them, distinguishing it from existing single-model databases that are growing into multi-model data platforms (in Tab. 1: MongoDB misses (2) and partially (3), Postgres and Redis require extensions / modules for (1), Postgres could ease (4), and Redis lacks (2)). Its multi-model versatility (cf. differentiators (1)–(4)) lets OrientDB compete with single-model document stores [6] and graph databases [7] and helps it to maintain high ranks as a NoSQL database system[4]. The multi-model concepts and design decisions that OrientDB followed are presented in this work and might further inspire designs of other MMDMSs.

This paper is organised as follows: Sect. 2 covers the *data model* (cf. differentiator (1)) in Sect. 2.1 as well as *querying data* (cf. (2)) in Sect. 2.2. Section 2.3 gives insights into *persisting data* (cf. (3)). In Sect. 3 the general system architecture is introduced and the *distribution model* (cf. (4)) is explained. Section 4 shares initial performance numbers on how OrientDB compares to selected systems in Tab. 1 and Sect. 5 concludes the paper.

2. Multi-model data

In this section we describe OrientDB's data model (cf. differentiator (1)), its unified SQL-based query capabilities over multi-model data (cf. (2)), and unified data storage and manipulation (cf. (3)).

2.1. Data definition

As many NoSQL databases, OrientDB exclusively focuses on the main non-relational data categories (i. e., graph, document, key-value [8]), as shown in Tab. 1. However, for a single, unified, and expressive data model (cf. differentiator (1)), its multi-model data definition uniquely combines semi-structured / document with graph data and object-oriented principles.

Base entities For a better understanding, Tab. 2 sets the main entities of OrientDB's data model into context to its relational, graph and document counterparts. Instead of a relational *table*, a graph *vertex* / *edge* or a document *collection*, OrientDB defines *class* — known from object oriented programming — as its top level entity (i. e., object-oriented features like inheritance and polymorphism are usable throughout the entire data model).

Table 2
Data models, entities (partially adapted from [8])

Relational	Graph	Document	**OrientDB**
Table	Vertex / Edge	Collection	**Class**
Row	Vertex	Document	**Vertex / document**
Column	Property	Key-value pair / attribute	**Property / attribute**
Ref. / join	Edge	- (join)	**Edge / link**

Considering a *row* table, OrientDB follows graph and document approaches and offers *vertex* and *document* entities, while in correspondence to *column*, *property* (i. e., property graph) and *attribute* (i. e., key-value pairs) entities are available.

The equivalent to a table *join* in graph databases is the *edge* entity, which is the same in OrientDB. In addition, documents might directly refer to other documents via *link* entities, usually not found in document stores.

Logical, physical model entities In Fig. 1, the basic entities are set into context to each other and connected to the most important (physical) storage entity. That storage entity is called *cluster* and denotes the place, where the data of a class is stored, possibly in different physical locations. In fact, usually multiple clusters are automatically created or assigned for each class (e. g., one cluster per CPU core or one per data center). A cluster has an identifier *id*, information on whether it is the *default* cluster of a class and a *selection* strategy that is considered, when adding new data (e. g., round-robin, balanced). Further storage details on clusters are discussed in Sect. 2.3.

Classes are object-oriented, schema constructs that might have a *super class* (e. g., for inheritance) and properties with constraints / rules. OrientDB supports flexible / schema-less data, but also strict and mixed-schema. With properties and constraints, the latter can be realized. OrientDB ships several default classes like `OUser` and `ORole` for security, vertex `V` and edge `E`, among others, from which properties can be derived.

The main entity — in which all data is stored — is a *record*, which gets assigned a record identifier `@rid` (RID), a version `@version` and a class `@class`. The RID is of the format #<CID>:<RP>, where <CID> is the cluster

[4] db-engines.com: "graph+dbms" and "document+store", 8/2021

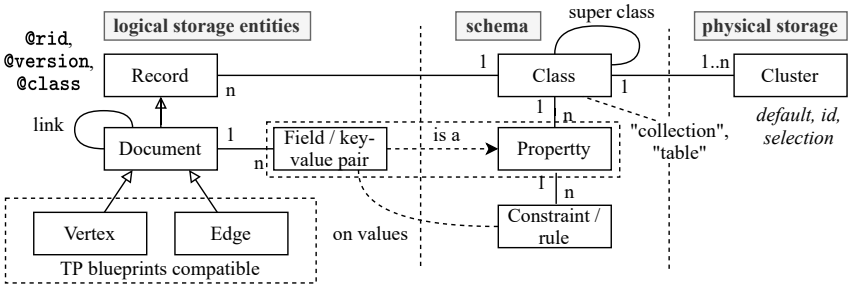


Figure 1: OrientDB storage data model

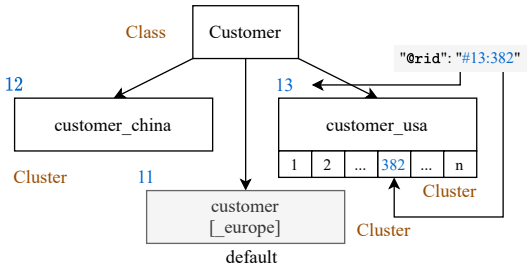


Figure 2: Clusters and record identifiers `@rid`

identifier and <RP> is the record's position in that cluster, thus it is a direct pointer to its physical storage location (cf. Ex. 1), similar to CTID in PostgreSQL.

Example 1 (Clusters, RIDs). *Figure 2 shows a `Customer` class with three clusters (i. e., identified by* 11–13*) and default cluster* 11*. A record with RID* #13:382 *can be found in cluster* 13 *at position* 382. ■

OrientDB has different record types with *document* as the most relevant. A document represents semi-structured data by a set of JSON *field* elements or *key-value* pairs, denoting properties of the record's class (e. g., for strict / mixed schema). Unlike other document stores, documents in OrientDB can have links to other documents, where a *link* is a set of RIDs. While in a schema-less setup no properties are required, the creation of an index for a field requires a property.

From a graph perspective, *vertex* and *edge* entities are documents, thus allowing them to be simple property graphs with key-value pairs or carrying more complex data like JSON documents (cf. Ex. 2).

Example 2 (Class, document, graph). *OrientDB's multi-model, SQL DDL, DML allow for the creation of a class that is of type vertex `V` as shown in Listing 1. The class extends built-in `V` to specify that the documents in `person` can be vertices in a graph (line 2). As in SQL, there are several way to `insert` data. For simplicity, two JSON documents are inserted into the `person` class, the second one with a RID, indicating a document link with label `likes` (lines 4–6). Since the documents are vertices in a graph, they can also be related by creating an edge class `friend` and specifying an edge using the RIDs of the previously inserted documents (i. e., Linda is Frank's friend and she likes Frank) (lines 8–9).*

Listing 1.: OrientDB DDL, DML for graph, document

```
1 --- collection as vertex
2 CREATE CLASS person EXTENDS V;
3 --- insert documents
4 INSERT INTO person CONTENT {"name": "Frank"};
5 INSERT INTO person CONTENT {"name": "Linda",
6   "likes": "#13:382"}
7 --- collection as edge
8 CREATE CLASS friend EXTENDS E;
9 CREATE EDGE friend FROM #13:382 to #13:37;
```

Listing 2.: Records / documents in OrientDB

```
1 {"@rid": "13:382", "@class": "person",
2  "name": "Frank", ... }
3
4  {"@rid": "13:37", "@class": "person",
5  "name": "Linda", "likes": "#13:382", ... }
```

The corresponding JSON records in OrientDB are shown in Listing 2. Notably, the RIDs and class assignments are part of the documents and can be used as shown above. ■

Sizing / limits While the number of databases within an OrientDB server is not limited, each database can have $2^{15} - 1$ clusters. A cluster can store $2^{63} - 1$ records, which allows up to $2^{78} - 1$ records per database. The maximum size of a record / document is 2GB (cf. 16MB in MongoDB and 512MB in Redis). There is no limit on the number of properties for a schema-less, and 2×10^9 properties per database for schema-full usage.

Summary OrientDB specifies a single, unified data model with an object-oriented class entity at its core. The object-oriented features allow for expressive modeling, including key-value or JSON document records and complex graph data at vertices and edges.

2.2. Querying data

On top of the unified data model, OrientDB specifies *OSQL* with SQL-like query capabilities and operations (cf. differentiator (2)), which we will briefly introduce by example for document, graph, text and geospatial (including object-oriented capabilities).

Document queries OSQL queries on documents are mostly similar to standard SQL. Notable deviations denote the dot-notation for specifying fields in nested JSON objects (e. g., `address.city`) and working with arrays (e. g., `UNWIND` returns the entries of a JSON array as single lines).

Besides the usual SQL keywords like `DISTINCT`, object-oriented extensions are added (e. g., to check the type of a class with `INSTANCEOF`).

Graph queries Apart from the support of TinkerPop / Gremlin queries on its graphs, OSQL allows for graph traversals and pattern matching (cf. Ex. 3).

Example 3 (Graph traversal, pattern matching). *An example of graph traversal on the documents of Listing 2 is depicted in Listing 3 and pattern matching is illustrated in Listing 4.*

Listing 3.: Graph traversal: Breadth-first search

```
1 TRAVERSE friend FROM #13:37
2   WHILE $depth <= 3 BREADTH_FIRST;
```

Listing 4.: Graph pattern matching

```
1 MATCH
2   {class: Person, as: people, where: (name =
      ↪ 'Linda')}
3 RETURN people
```

Queries in Listings 3 and 4 return records #13:382, #13:37, respectively. ■

Graph traversals are defined using `TRAVERSE` instead of `SELECT`, but traversals can be embedded into queries as sub-queries. In traversal queries, the `FROM`-clause can contain classes, clusters, one or more record identifiers and sub-queries. Similar to a `WHERE`-clause, the `WHILE` condition limits the traversal, while the result set can be limited by `LIMIT` (not shown). During a traversal, projections help to restrict to fields that should be followed. While in Listing 3 vertices of type *friend* are specified, OSQL supports `*`, `ALL()`, and `ANY()`. When specifying a class, polymorphic traversals can be specified. For instance, when *customer* is a *person*, then specifying `customer.name` will also traverse *person* vertices. OSQL supports different search strategies like `BREADTH_FIRST`, which can be limited by selections on context variables like `$depth` (nesting depth) or `$path` (path traversed). Similarly, linked documents are traversed and traversals can be directed by using keywords like `IN()` or `OUT()`.

Graph pattern matching requires the `MATCH` keyword with JSON input for a valid target `CLASS`, an alias for a node pattern (e. g., `people`), and a `WHERE`-clause that matches a node in the pattern.

Text and geospatial queries The text and geospatial features are based on the data model in Sect. 2.1 and both require index creation (cf. Ex. 4).

Example 4 (Graph traversal, pattern matching). *We recall that for indexed fields, a property has to be added to the class, as shown in Listing 5.*

Listing 5.: Text and spatial indexes

```
1 --- property, fulltext index
2 CREATE PROPERTY person.name STRING;
3 CREATE INDEX name ON Person(name)
4   FULLTEXT ENGINE LUCENE;
5 --- Geospatial index
6 CREATE PROPERTY person.location EMBEDDED
  ↪  OPoint;
7 INSERT INTO person SET location =
  ↪  St_GeomFromText(
8   "POINT (51.498308 -0.176882)")
9   WHERE name = "Frank";
10 CREATE INDEX person.location ON
  ↪  person(location)
11   SPATIAL ENGINE LUCENE;
```

For spatial coordinates (e. g., numeric), decimal degree values can be parsed from String using `St_GeomFromText` (e. g., by updating Frank's record).

Listing 6.: Text and spatial queries

```
1 --- text search using SEARCH_CLASS function
2 SELECT FROM person
3   WHERE SEARCH_CLASS("+name:Fran*") = true
4 --- spatial search using NEAR operator
5 SELECT *,$distance FROM person
6   WHERE [location,$spatial]
7   NEAR [51.495449,-0.17625,{"maxDistance":1}]
```

The syntax to support spatial arguments (i. e., $spatial) is taken from Lucene. ■

In contrast to all other supported index types (e. g., hash, B-tree / range), text and spatial indexes are created in Lucene[5], which was therefore embedded into OrientDB (cf. `ENGINE LUCENE`, `ENGINE SPATIAL ENGINE LUCENE`). When creating indexes in Lucene, additional metadata can be passed to the underlying engines to configure according to their full capabilities (e. g., analyzers, parsers).

Text queries require function `SEARCH_CLASS` (line 3) to specify complex fulltext searches (e. g., regular expressions) in a query's `WHERE`-clause. Further search functions like `SEARCH_FIELDS` (search index for more than one field) enable the full spectrum of Lucene's search capabilities.

[5] Apache Lucene, visited 11/2021: https://lucene.apache.org/

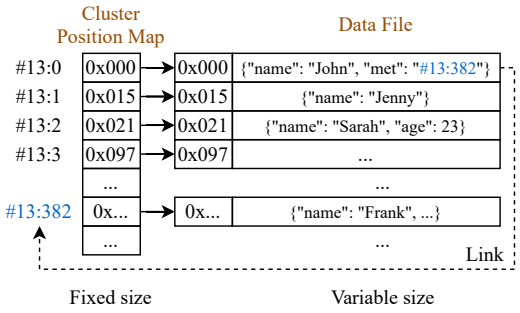


Figure 3: Record storage

Similarly, geospatial queries use the syntax of the underlying engine to work with multiple geometries (e. g., `Point`, `Line`), using functions like `ST_AsText`, `ST_Within`, `ST_Disjoint` on indexed fields. Built-in context fields like `$distance` (line 5) return distances sorted from nearest to furthest and the `NEAR` (line 7) operator finds all points near a given geo location. The special keyword `maxDistance` (line 7) limits the search space.

Summary OrientDB defines a SQL-like, multi-model query language that covers all aspects of the unified data model (i. e., including object-oriented, document, graph, key-value, text, and geospatial). External engines can be added, however, which loose their state, if not persistent.

2.3. Persisting data

As one of the first MMDMSs, OrientDB provides a single, multi-model persistence with ACID transactions (cf. differentiator (3)), whose main aspects we briefly introduce.

Indexes OrientDB defines several built-in indexes like Hash- and SB-Tree [9] for fast lookup and sequential record access. The indexes can be set as `UNIQUE` (i. e., not allowing for duplicate keys), which is also guaranteed in distributed setups. *Composite keys* allow for searches in multiple indexes at the same time. In addition, custom indexes can be specified, configured and loaded into the system. For instance, fulltext and geospatial indexes are provided that way through the Lucene search engine.

There can be up to two billion indexes per database, without limitations regarding the number of indexes per class (cf. 64 indexes per collection in MongoDB).

Transactions, Storage Compared to many other NoSQL databases, OrientDB supports ACID transactions with isolation levels *READ COMMITTED* (default, distributed) and *REPEATABLE READS* (single instance). During the commit of a transaction, records are physically stored using its main storage entity (i. e., cluster). Each cluster is split in pages, which contain system information (e. g., checksum for integrity / recovery) and record data (i. e., paginated storage). On disk, the data is stored in variable size *data files* as shown in Fig. 3. The RIDs are mapped through a fixed size, append-only *cluster position map* through cluster pointers (i. e., page identifier, record position in page). When deleting a record a "tombstone" is referenced that will be cleaned by compaction.

For data and index recovery a *write-ahead log* (WAL) is written. In case of indexes, the storage can be configured to recover without the need to rebuild indexes. To reduce disk access, the paginated storage has a two-level disk cache, consisting of a *W-TinyLFU* [10] read cache and *WOW* [11] write cache. The read cache specifies a main cache with segmented LRU eviction policy [12] and least-frequently used (LFU) cache admission policy based on the approximate TinyLFU data structure [10] that is combined with a window cache (for new items) based on a least-recently used eviction without an admission policy (i. e., admitting every new item). Together this allows for fine-grained locking and better cache hit-ratios compared to other replacement policies. The write cache maintains a queue of short lived pages. The read cache asks the write cache to load data from disk. If the data is not in the write cache's queue, it will be loaded from disk. Since OrientDB is developed in Java, an off-heap memory pool is used for file cache allocations to avoid JVM garbage collection performance penalties.

Summary OrientDB uses ACID transactions for change operations on its single multi-model storage. The data is queried through extensible index capabilities, while data is accessed through a RID mapping to data pages on disk (i. e., mostly independent of the database size).

3. System architecture, Distribution / Replication

In this section we introduce the general system architecture and deployment / distribution options for horizontal scaling (cf. differentiator (4)). In the architecture we locate the introduced single, multi-model data and storage components from Sect. 2 (cf. differentiators (1)–(3)).

3.1. Component architecture

The main deployment unit of OrientDB is the ODB Server shown in Fig. 4. The server accepts local OSQL and remote JDBC and REST calls. In addition, specialized integration like graph / Gremlin is available through modules provided by the open source community. The official database administration tool is OrientDB studio, which uses the REST API. While business applications usually use JDBC and REST, other clients use OrientDB's binary protocol to access the server. Besides data import and migration tools like ETL, Neo4j (community projects) and teleporter (relational data migration tool). OrientDB's active open source community contributed several language bindings / connectors for Python, .Net, NodeJS that complement the standard Java, JDBC, and REST drivers. Subsequently, we briefly introduce internal layers that were recently significantly reworked: multi-model API,

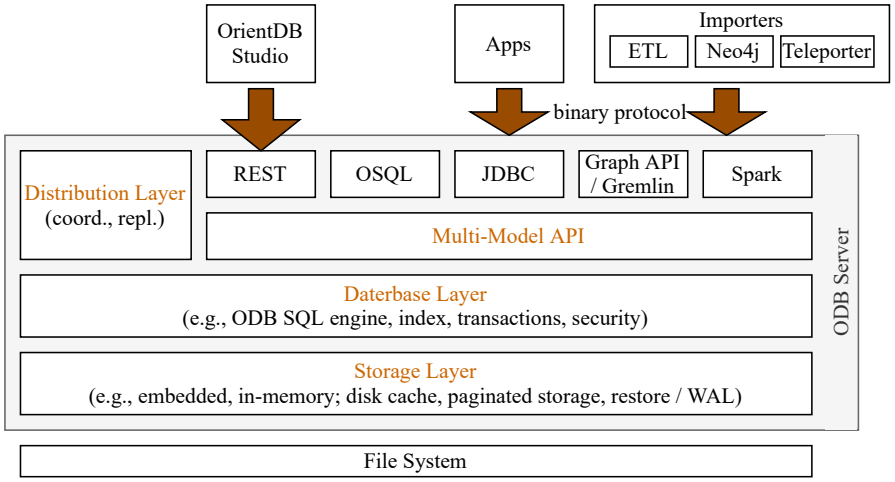


Figure 4: OrientDB architecture

database, and storage layers. The distribution layer will be discussed in Sect. 3.2.

Multi-model API OrientDB originally started out with separate document and graph APIs. From version 3.0, the access to the database layer is managed by a unified, multi-model API that we introduced in Sect. 2.2. The new API makes use of the data model from Sect. 2.1 to give a combined perspective on multi-model data. The old document and graph APIs as well as access via TinkerPop 2.x is deprecated and support for TinkerPop 3.x was added (cf. Graph API / Gremlin).

Database layer The OSQL optimizer and execution are part of the database engine. The incoming OSQL statements are transformed into logical, physical plans that leverage the indexes from Sect. 2.3 (cf. Appendix A).

Regarding security, OrientDB supports authentication through Kerberos and LDAP integration, database encryption and a sophistiacated security concept reaching from clients over server down to record-level.

Storage layer The storage components, introduced in Sect. 2.3 are seamlessly integrated into the ODB server to support full ACID transactions with disk-based storage. Despite the exception of Lucene (text, spatial), which are kept in their native storage, built-in and custom indexes operate on the same ODB storage.

3.2. Distribution / Replication

The ODB server can be deployed in different (distributed) setups that allow applications to scale OrientDB from small to larger installations, as shown in Fig. 5: (1) embedded, (2) standalone, (3) replicated, and (4) mixed.

OrientDB can be *embedded* into one application (1), either completely in-memory (e. g., for testing / CI) or with local storage (e. g., for simple micro-services). Multiple applications can start with a single, *standalone* OrientDB instance (2) that can be evolved from the embedded one

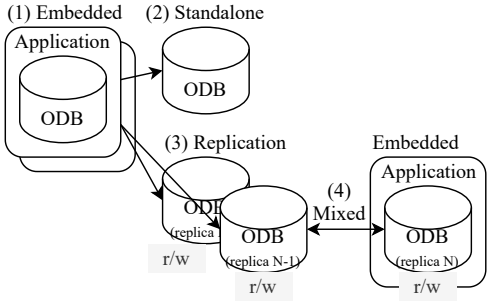


Figure 5: Distribution and replication

(cf. variant (1)) and scaled-out to several OrientDB instances through *replication*. Embedded instances can provide replicas, and thus run in a *mixed* deployment (4).

A seamless evolution or scaling is possible through a *Raft*-based [13] auto-discovery mechanism that identifies OrientDB instances, stores the runtime cluster configuration and synchronizes certain operations between nodes. After nodes have been added to the distributed system, OrientDB uses a variant of *Fast Paxos* [14, 15] to support distributed transactions (i. e., fast rounds, but bigger quorums than in classic Paxos). In that way, OrientDB supports multi-leader replication, which allows several nodes to perform change operations (insert, update, delete). To avoid quorum nodes lagging behind and requiring expensive quroum rounds to catch up (especially after failure), each node records change operations per transaction as persistent version counters. In case a node missed some operations, it can easily detect that by comparing the local version with the one in the next received message and catch up by direct synchronization with another replica (similar to [16]). With that, OrientDB supports distributed, unique indexes and fault-tolerance without slowing down a quorum.

4. Experiments

In this section we conduct a preliminary performance assessment of OrientDB compared to the databases in Tab. 1, i. e., well-established NoSQL databases (MongoDB, Redis), and one extended RDBMS (Postgres/JSON), using the well-known YCSB benchmark [17].

4.1. Setup, Limitations

The YCSB benchmark is an open-source, NoSQL database benchmark with a broad coverage of database systems (including all selected databases from Tab. 1). The benchmark features several workloads of JSON documents that cover important operations like read, insert, update and read-modify-write as well as configuration parameters for scale-factors (e. g., number of records and fields per record, field length).

Setup For our measurements, we run all workloads (A–F) and use the default parameters (e. g., single user and record, records with 10 fields of length 100). We set the number of documents per workload to five million records, operation count to 500k and the batch size to 10k. If a workload requires an index, a hash index is set.

Out of the multitude of polystore and multi-model NoSQL systems, we decided to choose MongoDB as one of the leading document stores, Postgres/JSON as one of the mostly used RDBMS with multiple NoSQL extensions, and Redis, as a representative key-value store (cf. Tab. 1). For all databases we benchmarked the latest server versions available, for which we updated the OrientDB driver to version 3.1.3 and added support for the `scan` operations in workload E[6].

All measurements are conducted on two Intel X5650 CPUs with 2.67GHzs (12 cores), 24GB RAM, Windows 10 operating system, and JDK 1.8.

Limitations YCSB originally targeted key-value stores, and thus only works with "flat" key-value pair JSON documents (i. e., no nesting, arrays). However, even wide-column and document stores provide YCSB drivers, due to the lack of open source benchmark alternatives. Recently, MongoDB tried to adapt their workloads to TPC-C [18], but had to admit that neither YCSB nor TPC-C sufficiently address their workloads.

Since OrientDB is an MMDMS, one could expect a suitable multi-model benchmark. In fact, there are polystore benchmarks (e. g., PolyBench [19]) and at least one multi-model benchmark (UniBench [20]), but none of the benchmarks is available as open-source (e. g., for database driver development).

Due to the lack of more suitable benchmarks and the rather simple nature of YCSB's data sets and workloads, the results are considered preliminary, but give interesting insights into relative rather than their absolute performance, which we consider valuable.

[6] YCSB OrientDB 3.1.*x* port, visited 11/2021: https://github.com/brianfrankcooper/YCSB/pull/1468

4.2. Preliminary results

The preliminary results of our benchmark runs are shown in Fig. 6 as time per request in μ seconds (lower bar is better) for MongoDB (short *mdb*), OrientDB (*odb*), Postgres/JSON (*pqj*), and Redis (*rds*), which we briefly discuss for each workload and concurrent user scaling.

Workload A: Update-heavy, read The first workload is a combination of 50% read and upgrade operations with a Zipfian record selection. Fig. 6a shows that this key-value store workload is best for Redis, while OrientDB is slightly better for read operations compared to MongoDB and Postgres. The update performance is similar for the three non key-value stores.

Workload B: Mostly read, update Similarly, Redis dominates the read-heavy workload with 95% read operations with Zipfian distribution in Fig. 6b. The focus on read operations leads to better results for OrientDB and Postgres, which have slightly slower update compared to read operations.

Workload C: Read-only The Zipfian read-only workload underpins the previous observations on read operations for OrientDB and Postgres, which gain ground on Redis, shown in Fig. 6c. Hereby, OrientDB might profit from its WTinyLFU read cache implementation that keeps a mix of LRU and heavily read older records in memory and its physical RID mapping.

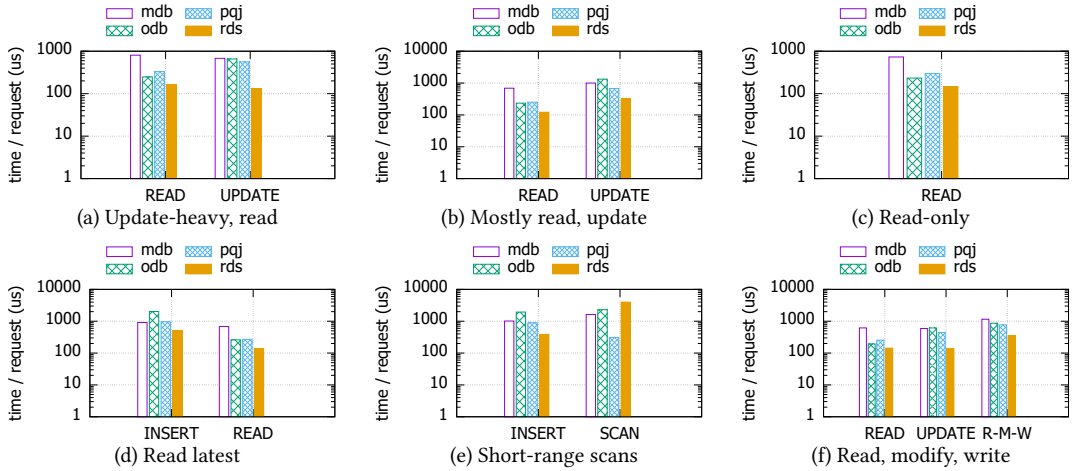
Workload D: Read latest Consequently, the 95% read and 5% insert workload for reading the latest inserts shows a similar read performance for all databases. Unlike for the similar update case of workload B, OrientDB is slightly behind MongoDB and Postgres, while Redis shows similar results to its update performance.

Workload E: Short-ranges In this scan operation dominated workload, the inserts (again only 5%) are similar to that of workload D. The scans, however, show best performance for Postgres and worst for Redis, while MongoDB and OrientDB are closely left in-between.

Workload F: Read-modify-write For the combined r-m-w workload, all databases perform similar to the previous workloads, with Redis on top, then OrientDB with slightly faster read performance than MongoDB and Postgres, but similar update and r-m-w time.

Concurrent user scaling Since the results for workloads A–F were for a single user, and thus can be considered as baseline performance, we now briefly study workloads B and F for 64 concurrent users in Fig. 7.

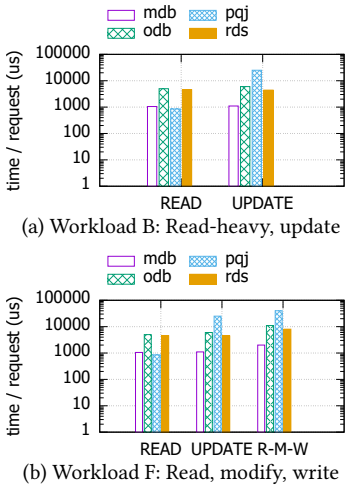
MongoDB and Postgres perform slightly better for read operations than OrientDB and Redis in both workloads, while MongoDB shows the fastest updates, followed by



(a) Update-heavy, read (b) Mostly read, update (c) Read-only

(d) Read latest (e) Short-range scans (f) Read, modify, write

Figure 6: YCSB benchmark workloads (single instance and user)



(a) Workload B: Read-heavy, update

(b) Workload F: Read, modify, write

Figure 7: YCSB workloads B, F (single instance, 64 users)

Redis and OrientDB (cf. Figs. 7a and 7b). Notably, for read and update operations, MongoDB's performance remains similar to the single user benchmarks. Redis and OrientDB are factors of 5–10 slower than their single user performance and Postgres even beyond that for update and r-m-v workloads.

4.3. Discussion

While it is not surprising that all single user YCSB workloads were best for the key-value store Redis, we made two notable observations: (1) good single instance scaling of MongoDB from one to 64 users, (2) OrientDB with competitive results compared to "best-in-class" document and key-value stores.

Firstly, the slightly better update and insert performance of MongoDB and Redis could be explained by their slightly more relaxed ACID guarantees (e. g., with single-document focus, compare-and-set operation focus). For single user workloads, MongoDB seems to have additional overhead (i. e., MongoDB's hash index is actually a B-tree, cf. [21]) that amortizes for multiple users in a single database instance (cf. observation (1)). Secondly, despite being implemented in Java, OrientDB's unified data model and query capabilities on one single persistence, combined with proven database technology (e. g., W-TinyLFU buffer cache) and off-heap memory usage make it competitive to the other databases in our experiments (cf. observation (2)).

5. Conclusions

This paper gives the first comprehensive description of a recently revamped OrientDB, a commercially successful NoSQL, open source MMDMS. While more and more database systems strive to become multi-model data platforms, OrientDB early on addressed NoSQL multi-model key differentiators, such as (1) a single, unified data model, (2), SQL-like, multi-model query and operations, (3) a single, multi-model ACID-transacted store, and (4) a seamless scaling. Although OrientDB is one of the earliest NoSQL MMDMSs, we showed that it is competitive compared to "best-in-class" document and key-value stores.

While multi-model data platforms are on a rise, we found that future work should consider standardized, open benchmark initiatives for MMDMSs as well as for single NoSQL areas like document stores (similar to YCSB for key-value). For OrientDB in particular, there are several areas of future improvements like adding all custom indexes to the ODB storage, and further improving transacted change operation performance.

Acknowledgements We are especially indebted to Dr. Spranz and Dr. Pepke, who made this work possible, and we thank the fellow OrientDB contributors, mainly Enrico Risa and Colin Leister, for the joint journey through the OrientDB 3.0 to 3.2 releases.

References

[1] J. Lu, I. Holubová, B. Cautis, Multi-model databases and tightly integrated polystores: Current practices, comparisons, and open challenges, in: A. Cuzzocrea, J. Allan, N. W. Paton, D. Srivastava, R. Agrawal, A. Z. Broder, M. J. Zaki, K. S. Candan, A. Labrinidis, A. Schuster, H. Wang (Eds.), ACM International Conference on Information and Knowledge Management (CIKM), ACM, 2018, pp. 2301–2302. doi:10.1145/3269206.3274269.

[2] S. Abiteboul, M. Arenas, P. Barceló, M. Bienvenu, D. Calvanese, C. David, R. Hull, E. Hüllermeier, B. Kimelfeld, L. Libkin, W. Martens, T. Milo, F. Murlak, F. Neven, M. Ortiz, T. Schwentick, J. Stoyanovich, J. Su, D. Suciu, V. Vianu, K. Yi, Research directions for principles of data management (dagstuhl perspectives workshop 16151), Dagstuhl Manifestos 7 (2018) 1–29. doi:10.4230/DagMan.7.1.1.

[3] D. Abadi, A. Ailamaki, D. Andersen, P. Bailis, M. Balazinska, P. Bernstein, P. Boncz, S. Chaudhuri, A. Cheung, A. Doan, et al., The seattle report on database research, ACM SIGMOD Record 48 (2020) 44–53.

[4] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, S. B. Zdonik, The bigdawg polystore system, ACM SIGMOD Record 44 (2015) 11–16. doi:10.1145/2814710.2814713.

[5] G. Mihai, Multi-model database systems: The state of affairs, Economics and Applied Informatics (2020) 211–215.

[6] M. Besta, E. Peter, R. Gerstenberger, M. Fischer, M. Podstawski, C. Barthels, G. Alonso, T. Hoefler, Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries, CoRR abs/1910.09017 (2019). arXiv:1910.09017.

[7] D. Fernandes, J. Bernardino, Graph databases comparison: Allegrograph, arangodb, infinitegraph, neo4j, and orientdb, in: J. Bernardino, C. Quix (Eds.), International Conference on Data Science, Technology and Applications (DATA), SciTePress, 2018, pp. 373–380. doi:10.5220/0006910203730380.

[8] J. Dann, D. Ritter, H. Fröning, Non-relational databases on FPGAs: Survey, design decisions, challenges, CoRR abs/2007.07595 (2020). arXiv:2007.07595.

[9] P. E. O'Neil, The sb-tree an index-sequential structure for high-performance sequential access, Acta Informatica 29 (1992) 241–265.

[10] G. Einziger, R. Friedman, B. Manes, TinyLFU: A highly efficient cache admission policy, ACM Trans. Storage 13 (2017) 35:1–35:31. URL: https://doi.org/10.1145/3149371. doi:10.1145/3149371.

[11] B. S. Gill, D. S. Modha, WOW: wise ordering for writes - combining spatial and temporal locality in non-volatile caches, in: G. Gibson (Ed.), File and Storage Technologies (FAST), USENIX, 2005.

[12] R. Karedla, J. S. Love, B. G. Wherry, Caching strategies to improve disk system performance, Computer 27 (1994) 38–46. URL: https://doi.org/10.1109/2.268884. doi:10.1109/2.268884.

[13] D. Ongaro, J. K. Ousterhout, USENIX annual technical conference (ATC), USENIX Association, 2014, pp. 305–319.

[14] L. Lamport, Fast paxos, Distributed Comput. 19 (2006) 79–103. doi:10.1007/s00446-006-0005-x.

[15] W. Zhao, Fast paxos made easy: Theory and implementation, Int. J. Distributed Syst. Technol. 6 (2015) 15–33. doi:10.4018/ijdst.2015010102.

[16] S. Zhou, S. Mu, Fault-tolerant replication with pull-based consensus in MongoDB, in: J. Mickens, R. Teixeira (Eds.), USENIX Symposium on Networked Systems Design and Implementation, (NSDI), USENIX Association, 2021, pp. 687–703.

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears, Benchmarking cloud serving systems with YCSB, in: ACM Symposium on Cloud Computing (SoCC), 2010, pp. 143–154. doi:10.1145/1807128.1807152.

[18] A. Kamsky, Adapting TPC-C benchmark to measure performance of multi-document transactions in mongodb, Proc. VLDB Endow. 12 (2019) 2254–2262. doi:10.14778/3352063.3352140.

[19] J. Karimov, T. Rabl, V. Markl, Polybench: The first benchmark for polystores, in: TPC Technology Conference on Performance Evaluation and Benchmarking (TPCTC), Springer, 2018, pp. 24–41.

[20] C. Zhang, J. Lu, P. Xu, Y. Chen, Unibench: A benchmark for multi-model database management systems, in: R. Nambiar, M. Poess (Eds.), TPC Technology Conference on Performance Evaluation and Benchmarking (TPCTC), volume 11135 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 7–23. doi:10.1007/978-3-030-11404-6_2.

[21] G. Graefe, Modern b-tree techniques, Found. Trends Databases 3 (2011) 203–402. URL: https://doi.org/10.1561/1900000028. doi:10.1561/1900000028.

A. OSQL query model

The new OSQL query model from OrientDB version 3.0.x is sketched in Fig. 8. OSQL is defined based on a JavaCC[7]

[7] JavaCC parser generator, visited 11/2021: https://javacc.github.io/javacc/documentation/grammar.html

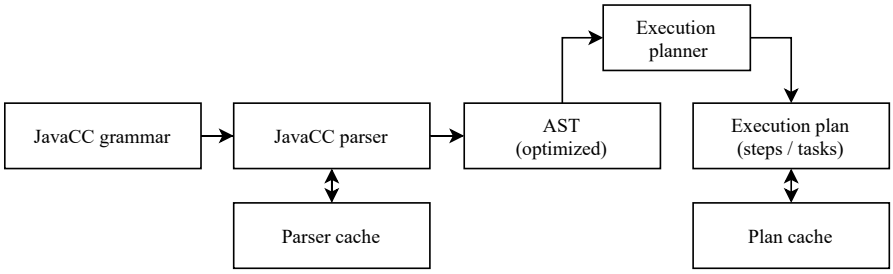


Figure 8: OSQL query model

grammar, which is used to generate an OSQL parser that creates a query AST. To speed up the generated JavaCC parser, recurring queries are cached (i. e., logical plan cache). Several AST-level optimizations are applied (e. g., task push-down, index look-ups on link chains) that (partially) rewrite the AST (e. g., JSON path / chain dot notation becomes a set of sub-queries).

An execution planner creates an execution plan from the (optimized) AST, using pre-defined execution steps. The resulting physical plan itself is "executable" in the sense that its steps (e. g., fetch data, project, filter) and is cached (physical plan cache).