<https://cratedb.com/docs>

# CrateDB

CrateDB supports both the HTTP protocol and the PostgreSQL wire protocol, which ensures that many clients that work with PostgreSQL, will also work with CrateDB. Through corresponding drivers, CrateDB is compatible with JDBC, ODBC, and other database API specifications.

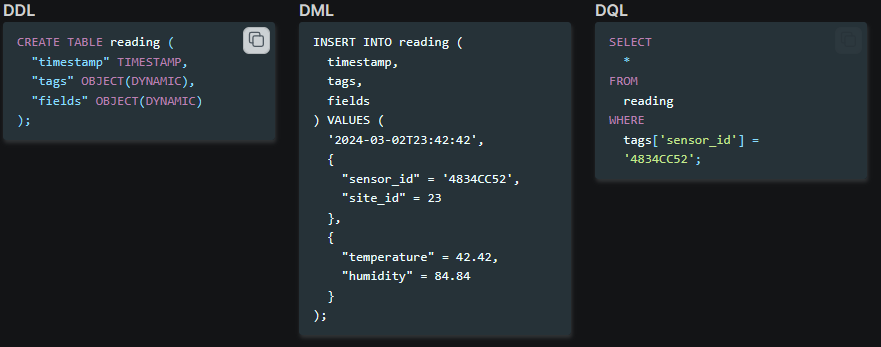
CrateDB relies on Lucene for storage and inherits components from Elasticsearch / OpenSearch for cluster consensus. Fundamental concepts of CrateDB are familiar to Elasticsearch users, because both are actually using the same implementation.

**Document Store**

Learn how to efficiently store JSON documents or other structured data, also nested, using CrateDB’s OBJECT and ARRAY container data types, and how to query this data with ease.

CrateDB combines the advantages of typical SQL databases and strict schemas with the dynamic properties of NoSQL databases. While traditional object-relational databases allow you to store and process JSON data only opaquely, CrateDB handles objects as first-level citizens.

Even when using dynamic objects, i.e. when working without a strict object schema, all attributes are indexed by default, and can be queried efficiently.



For columns of type OBJECT, CrateDB supports different policies about the behaviour with undefined attributes, namely STRICT, DYNAMIC, and IGNORED, see [Object column policy](https://cratedb.com/docs/crate/reference/en/latest/general/ddl/data-types.html#type-object-column-policy).

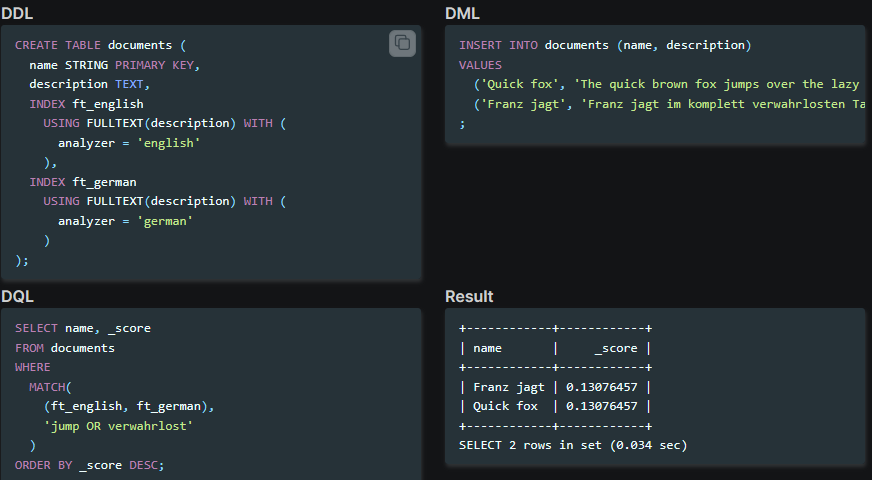
* STRICT: Reject any sub-column that is not defined upfront.
* DYNAMIC: INSERT operations may dynamically add new sub-columns to the object definition. This is the default setting.
* IGNORED: Also means DYNAMIC, but dynamically added sub-columns do not cause a schema update, and the new values will not be indexed. Because IGNORED columns are not recorded in the schema, you can insert mixed types into them. For example, one row may insert an integer and the next row may insert an object. Objects with a STRICT or DYNAMIC column policy do not allow this.

# Full-Text Search

CrateDB is an exceptional choice for handling complex queries and large-scale data sets. One of its standout features are its full-text search capabilities, built on top of the powerful Lucene library. This makes it a great fit for organizing, searching, and analyzing extensive datasets.

[Full-text search](https://en.wikipedia.org/wiki/Full_text_search) leverages the [BM25](https://en.wikipedia.org/wiki/Okapi_BM25) search ranking algorithm, effectively implementing the storage and retrieval parts of a [search engine](https://en.wikipedia.org/wiki/Search_engine).

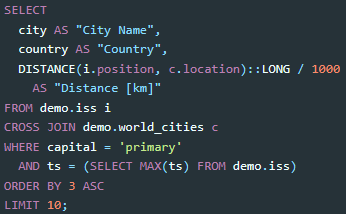
To create a “full-text” index, the field value is first analyzed and, based on the used analyzer, split into smaller units, such as individual words, a processing step called tokenization. A full-text index is then created for each text unit separately.



# Geospatial Search

**CrateDB supports location data for efficiently storing and querying geographic and spatial/geospatial data.**

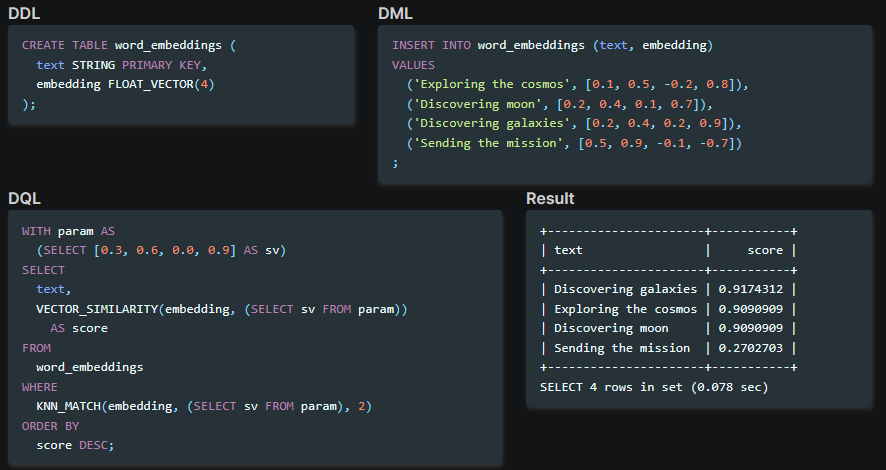
CrateDB can be used as a database to conduct geospatial search operations building upon the Prefix Tree and BKD-tree index structures of Apache Lucene.



# Vector Search

CrateDB can be used as a [vector database](https://en.wikipedia.org/wiki/Vector_database) (VDBMS) for storing and retrieving vector embeddings based on the FLOAT\_VECTOR data type and its accompanying KNN\_MATCH and VECTOR\_SIMILARITY functions, effectively conducting HNSW semantic similarity searches on them, also known as vector search.

CrateDB uses Lucene as a storage layer, so it inherits the implementation and concepts of Lucene Vector Search, in the same spirit as Elasticsearch.



# BLOB Store

**CrateDB provides a blob/object storage subsystem accessible via HTTP, similar to AWS S3.**

CrateDB includes support to store [binary large objects](https://en.wikipedia.org/wiki/Object_storage), using its [Blobs](https://cratedb.com/docs/crate/reference/en/latest/general/blobs.html#blob-support) feature / subsystem. By utilizing CrateDB’s cluster features, the files can be replicated and sharded just like regular data.



# Clustering

**CrateDB provides scalability through partitioning, sharding, and replication.**

CrateDB uses a shared-nothing architecture to form high-availability, resilient database clusters with minimal effort of configuration, effectively implementing a distributed SQL database.

Sharding and partitioning are techniques used to distribute data evenly across multiple nodes in a cluster, ensuring data scalability, availability, and performance.

Replication can be applied to increase redundancy, which reduces the chance of data loss, and to improve read performance.

## Sharding:

In CrateDB, tables are split into a configured number of shards. Then, the shards are distributed across multiple nodes of the database cluster. Each shard in CrateDB is stored in a dedicated Lucene index.

You can think of shards as a self-contained part of a table, that includes both a subset of records and the corresponding indexing structures.

Figuring out how many shards to use for your tables requires you to think about the type of data you are processing, the types of queries you are running, and the type of hardware you are using.

If you have fewer shards than CPUs in the cluster, this is called *under-allocation*, and it means you’re not getting the best performance out of CrateDB.

Whenever possible, CrateDB will parallelize query workloads and distribute them across the whole cluster. The more CPUs this query workload can be distributed across, the faster the query will run.

However, if most nodes have more shards per table than they have CPUs, you could actually see performance degradation. Each shard comes with a cost in terms of open files, RAM, and CPU cycles. Smaller shards also means small shard indexes, which can adversely affect computed search term relevance.

## Partitioning:

CrateDB also supports splitting up data across another dimension with partitioning. Tables can be partitioned by defining partition columns. You can think of a partition as a set of shards.

Partitioned tables optimize access efficiency when querying data, because only a subset of data needs to be addressed and acquired.

Each partition can be backed up and restored individually, for efficient operations.

Tables allow to change the number of shards even after creation time for future partitions. This feature enables you to start out with few shards per partition, and scale up the number of shards for later partitions once traffic and ingest rates increase over the lifetime of your application or system.

## Replication:

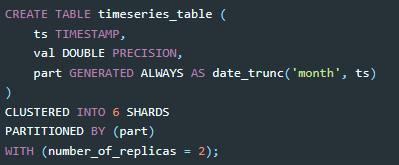
You can configure CrateDB to replicate tables. When you configure replication, CrateDB will ensure that every table shard has one or more copies available at all times.

Replication can also improve read performance because any increase in the number of shards distributed across a cluster also increases the opportunities for CrateDB to parallelize query execution across multiple nodes.

## Example:

With a monthly throughput of 300 GB, partitioning your table by month, and using six shards, each shard will manage 50 GB of data, which is within the recommended size range (5 - 50 GB).

Through replication, the table will store three copies of your data, in order to reduce the chance of permanent data loss.



## Architecture

In a CrateDB cluster all nodes have a direct link to all other nodes; this is known as [full mesh](https://en.wikipedia.org/wiki/Network_topology#Mesh) topology. Due to simplicity reasons every node maintains a one-way connections to every other node in the network. The network topology of a 5 node cluster looks like this:

### bootstrapping

Starting a CrateDB cluster for the first time requires the initial list of master-eligible nodes to be defined. This is known as cluster bootstrapping.

### Master election

In a CrateDB cluster there can only be one master node at any single time. The cluster only becomes available to serve requests once a master has been elected, and a new election takes place if the current master node becomes unavailable.

The master node is responsible for making changes to the global cluster state. The cluster [elects the master node](https://cratedb.com/docs/crate/reference/en/latest/concepts/clustering.html#concept-master-election) from the configured list of master-eligible nodes the first time a cluster is bootstrapped.

CrateDB requires a [quorum](https://en.wikipedia.org/wiki/Quorum_(distributed_computing)) of nodes before a master can be elected. A quorum ensures that the cluster does not elect multiple masters in the event of a network partition (also known as a [split-brain](https://en.wikipedia.org/wiki/Split-brain_(computing)) scenario).

### [Cluster meta data](https://cratedb.com/docs/crate/reference/en/latest/concepts/storage-consistency.html#id6)

Cluster meta data is held in the so called “Cluster State”, which contains the following information:

* Tables schemas.
* Primary and replica shard locations. Basically just a mapping from shard number to the storage node.
* Status of each shard, which tells if a shard is currently ready for use or has any other state like “initializing”, “recovering” or cannot be assigned at all.
* Information about discovered nodes and their status.
* Configuration information.

Every node has its own copy of the cluster state. However there is only one node allowed to change the cluster state at runtime. This node is called the “master” node and gets auto-elected. The “master” node has no special configuration at all, all nodes are master-eligible by default, and any master-eligible node can be elected as the master. There is also an automatic re-election if the current master node goes down for some reason.

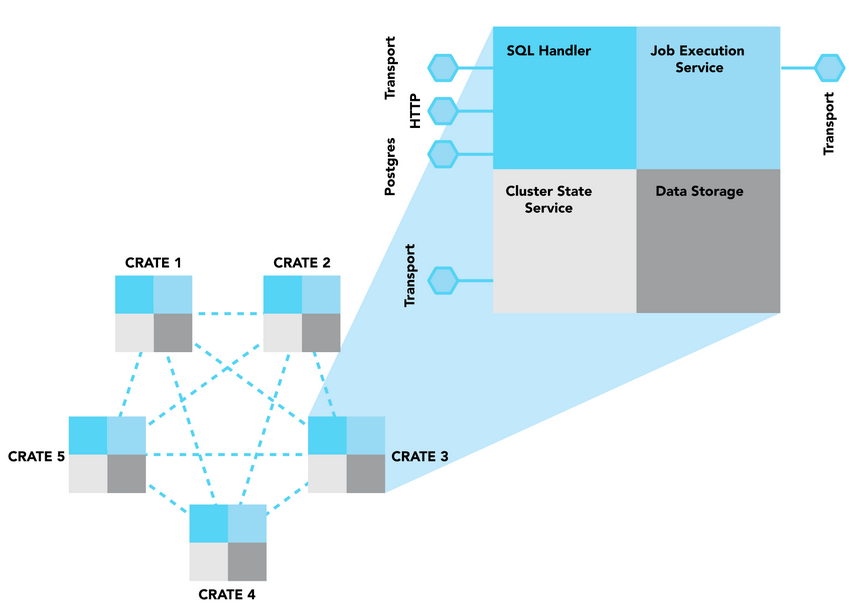
### multi-zone setup

In some cases, it may be necessary to run a cluster across multiple data centers or availability zones



This means that CrateDB will try to allocate [shards](https://cratedb.com/docs/crate/reference/en/latest/general/ddl/sharding.html) and their [replicas](https://cratedb.com/docs/crate/reference/en/latest/general/ddl/replication.html) according to the zone tags, so that a shard and its replica are not on a node with the same zone value.

## [Components of a CrateDB Node](https://cratedb.com/docs/crate/reference/en/latest/concepts/clustering.html#id6)



### [SQL Handler](https://cratedb.com/docs/crate/reference/en/latest/concepts/clustering.html#id7)

The SQL Handler part of a node is responsible for three aspects:

1. handling incoming client requests,
2. parsing and analyzing the SQL statement from the request and
3. creating an execution plan based on the analyzed statement ([abstract syntax tree](https://en.wikipedia.org/wiki/Abstract_syntax_tree))

The SQL Handler is the only of the four components that interfaces with the “outside world”. CrateDB supports three protocols to handle client requests:

1. HTTP
2. a Binary Transport Protocol
3. the PostgreSQL Wire Protocol

### [Job Execution Service](https://cratedb.com/docs/crate/reference/en/latest/concepts/clustering.html#id8)

The Job Execution Service is responsible for the execution of a plan (“job”). The phases of the job and the resulting operations are already defined in the execution plan. A job usually consists of multiple operations that are distributed via the Transport Protocol to the involved nodes, be it the local node and/or one or multiple remote nodes. Jobs maintain IDs of their individual operations. This allows CrateDB to “track” (or for example “kill”) distributed queries.

### [Cluster State Service](https://cratedb.com/docs/crate/reference/en/latest/concepts/clustering.html#id9)

The three main functions of the Cluster State Service are:

1. cluster state management,
2. election of the master node and
3. node discovery, thus being the main component for cluster building (as described in section [Multi-node setup: Clusters](https://cratedb.com/docs/crate/reference/en/latest/concepts/clustering.html#concept-clusters)).

### [Data storage](https://cratedb.com/docs/crate/reference/en/latest/concepts/clustering.html#id10)

The data storage component handles operations to store and retrieve data from disk based on the execution plan.

In CrateDB, the data stored in the tables is sharded, meaning that tables are divided and (usually) stored across multiple nodes. Each shard is a separate Lucene index that is stored physically on the filesystem. Reads and writes are operating on a shard level.

# Insert

Every insert is first applied to the primary shard

After the primary shard has been updated, the insert is then individually communicated in parallel to every configured replica shard

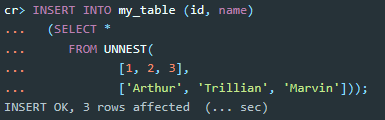
CrateDB will not return a response until all replica shards have been updated

## [Single inserts](https://cratedb.com/docs/guide/performance/inserts/methods.html#id3)

[Single inserts](https://cratedb.com/docs/crate/reference/en/latest/general/dml.html#dml-inserting-data) are the most basic sort of insert statement, and look like this:



## [UNNEST](https://cratedb.com/docs/guide/performance/inserts/methods.html#id4)

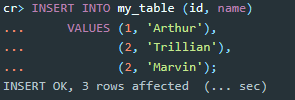


You should see a dramatic improvement in performance over single inserts.

Specifically, the advantages are:

* Significantly less internal network traffic
* The query only needs to be parsed, planned, and executed once
* If [translog.durability](https://cratedb.com/docs/crate/reference/en/latest/sql/reference/create_table.html#translog-durability) is set to REQUEST (the default), an insert using UNNEST flushes the disk once for every shard written to

## [Multiple value expressions](https://cratedb.com/docs/guide/performance/inserts/methods.html#id5)



This method of doing bulk inserts is usually slower than the UNNEST method, because parsing is more expensive. The query looks nicer for humans though.

## [refresh\_interval](https://cratedb.com/docs/crate/reference/en/latest/sql/statements/create-table.html#id17)

In CrateDB new written records are not immediately visible. A user has to either invoke the [REFRESH](https://cratedb.com/docs/crate/reference/en/latest/sql/statements/refresh.html#sql-refresh) statement or wait for an automatic background refresh.

# Snapshots

**CrateDB provides a backup mechanism based on snapshots.**

CrateDB, like Elasticsearch, uses snapshots to perform cluster-wide backups of your data.

A snapshot is a backup of a running CrateDB cluster. You can use snapshots for different purposes.

* Regularly back up a cluster with no downtime
* Recover data after deletion or a hardware failure
* Transfer data between clusters
* Reduce your storage costs by out-phasing partitions into cold and frozen data tier repositories and archives

# Storage Layer

The CrateDB storage layer is based on Lucene. By default, all fields are indexed, nested or not, but the indexing can be turned off selectively.

This page enumerates some concepts of Lucene, and the article [Indexing and Storage in CrateDB](https://cratedb.com/blog/indexing-and-storage-in-cratedb) goes into more details by exploring its internal workings.

## Lucene

Lucene offers scalable and high-performance indexing which enables efficient search and aggregations over documents and rapid updates to the existing documents. Solr and Elasticsearch are building upon the same technologies.

* **Documents**

A single record in Lucene is called “document”, which is a unit of information for search and indexing that contains a set of fields, where each field has a name and value. A Lucene index can store an arbitrary number of documents, with an arbitrary number of different fields.

* **Append-only segments**

A Lucene index is composed of one or more sub-indexes. A sub-index is called a segment, it is immutable, and built from a set of documents. When new documents are added to the existing index, they are added to the next segment, while previous segments are never modified. If the number of segments becomes too large, the system may decide to merge some segments and discard the freed ones. This way, adding a new document does not require rebuilding the whole index structure completely.

* **Column store**

For text values, other than storing the row data as-is (and indexing each value by default), each value term is stored into a [column-based store](https://cratedb.com/docs/crate/reference/en/latest/general/ddl/storage.html) by default, which offers performance improvements for global aggregations and groupings, and enables efficient ordering, because the data for one column is packed at one place.

In CrateDB, the column store is enabled by default and can be disabled only for text fields, not for other primitive types. Furthermore, CrateDB does not support storing values for container and geospatial types in the column store.

### Reducing storage

The following techniques can help to reduce the average size of one shard.

Please note reducing disk usage often comes at the cost of performance.

If there are columns that will not be used in aggregations (joins) and groupings (group by, order by), it will have no impact on performance and might make sense to reduce its storage footprint.

Things that can be done:

* disable indexing
* disable the columnar store
* change the compression algorithm
* review the data schema

## Data structures

By default, CrateDB indexes all data in every field, and each indexed field has a dedicated, optimized data structure. For example, text fields are stored in inverted indices, and numeric and geo fields are stored in BKD trees.

This section enumerates the three main Lucene data structures that are used within CrateDB: Inverted indexes for text values, BKD trees for numeric values, and DocValues.

* **Inverted index**

The Lucene indexing strategy for text fields relies on a data structure called inverted index, which is defined as a “data structure storing a mapping from content, such as words and numbers, to its location in the database file, document or set of documents”.

Depending on the configuration of a column, the index can be plain (default) or full-text. An index of type “plain” indexes content of one or more fields without analyzing and tokenizing their values into terms. To create a “full-text” index, the field value is first analyzed and based on the used analyzer, split into smaller units, such as individual words. A full-text index is then created for each text unit separately.

The inverted index enables a very efficient search over textual data.

* **BKD tree**

To optimize numeric range queries, Lucene uses an implementation of the Block KD (BKD) tree data structure. The BKD tree index structure is suitable for indexing large multi-dimensional point data sets. It is an I/O-efficient dynamic data structure based on the KD tree. Contrary to its predecessors, the BKD tree maintains its high space utilization and excellent query and update performance regardless of the number of updates performed on it.

Numeric range queries based on BKD trees can efficiently search numerical fields, including fields defined as TIMESTAMP types, supporting performant date range queries.

* **DocValues**

Because Lucene’s inverted index data structure implementation is not optimal for finding field values by given document identifier, and for performing column-oriented retrieval of data, the DocValues data structure is used for those purposes instead.

DocValues is a column-based data storage built at document index time. They store all field values that are not analyzed as strings in a compact column, making it more effective for sorting and aggregations.

# Cross-Cluster Replication

Overview

Cross-cluster replication, also called logical replication, is a method of data replication across multiple clusters.

About

CrateDB uses a “publish and subscribe” model where subscribers pull data from the publications of the publisher they subscribed to.

Details

Logical replication is useful for different use cases.

* Consolidating data from multiple clusters into a single one for aggregated reports.
* Ensure high availability if one cluster becomes unavailable.
* Replicating between different compatible versions of CrateDB. Replicating tables created on a cluster with higher major/minor version to a cluster with lower major/minor version is not supported.

# Memory Configuration

CrateDB is a Java application running on top of a Java Virtual Machine (JVM).

For optimal performance, you must configure the amount of memory that is available to the JVM for **heap** allocations. The **heap** is a memory region used for allocations of objects. For example, if you invoke a SELECT statement, parts of the result set are temporarily allocated on the **heap** memory.

The JVM has automatic memory management and frees up memory using [Garbage Collection](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)) (GC). CrateDB, as of version 4.1, defaults to use a garbage collection implementation called [G1GC](https://docs.oracle.com/javase/10/gctuning/garbage-first-garbage-collector.htm). This implementation performs well for heap sizes ranging from several GB to ten or more. It tries to provide the best balance between latency and throughput. Still, the garbage collection times **increase** with a bigger heap, leading to **increased** latency. Therefore, your heap size shouldn’t be too large.

On [x64 architectures](https://en.wikipedia.org/wiki/X86-64), the [HotSpot Java Virtual Machine](https://www.oracle.com/java/technologies/javase/javase-core-technologies-apis.html) (JVM) uses a performance optimization technique called [Compressed Oops](https://wiki.openjdk.java.net/display/HotSpot/CompressedOops). This technique allows the JVM to use 4 bytes instead of 8 bytes for object references. This saves a lot of memory.

Unfortunately, the JVM can only address up to about 32 GB of memory with *Compressed Oops*, and the optimization is disabled if a heap size of more than 32 GB is configured. For this reason, you should aim to stay below 30.5 GB.