SingleStore DB
A Powerful Distributed Database

Overview

SingleStoreDB (formerly known as MemSQL) is a distributed SQL database designed for performance, scalability, and real-time analytics.

It combines in-memory **rowstore** and disk-based **columnstore** for fast OLTP and OLAP processing.

Compatible with MySQL and cloud-native, it scales easily across nodes. Ideal for dashboards, IoT, streaming data, and **low-latency analytics**.

The **Community Edition** offers robust features with some limitations

Why Single Store?

We need a fast caching layer, which is queryable.

Apache Ignite

✅ **Pros:**

- **In-memory distributed database**, blazing fast
- Can act as **caching layer**, **compute grid**, **SQL engine**
- Has **persistence mode**, so data isn't lost
- Can perform **co-located joins** with indexing

❌ **Cons:**

- Setup is comparatively Complex
- **.NET integration is weak**, more Java-friendly
- Lacks native **dashboards** tooling

RediSQL

SQLite inside Redis

✅ **Pros**

- Fast for **small, flat datasets**
- Adds SQL-style queries to Redis
- Low-latency reads for **lookups, small metadata**

❌ **Cons**

- Not suitable for **large, relational, or complex analytical data**
- **Single-threaded bottleneck** (inherits Redis's threading model)
- Lacks advanced indexing and vectorization like SingleStore
- Poor support for **heavy concurrent writes** or **bulk scans**
- **Persistence** is in pro version only.
- Lacks native **dashboards** tooling

Different Databases With Use Cases

Use Case	Recommended Databases	Rationale
Real-Time Analytics	SingleStoreDB, PostgreSQL, MongoDB	SingleStoreDB offers hybrid OLTP/OLAP capabilities; PostgreSQL provides strong consistency; MongoDB offers flexibility and scalability.
Enterprise Applications	SQL Server, PostgreSQL	SQL Server integrates well with Microsoft tools; PostgreSQL offers open-source flexibility with enterprise-grade features.
Caching Layer	Redis, SingleStoreDB	Redis provides ultra-fast caching; SingleStoreDB can cache frequently accessed data in-memory.
Flexible Schema Applications	MongoDB, PostgreSQL	MongoDB allows for schema-less design; PostgreSQL offers JSON support with relational integrity.
High-Volume Transactions	SQL Server, SingleStoreDB	SQL Server provides robust transactional support; SingleStoreDB handles high-throughput workloads efficiently.
Cloud-Native Applications	SingleStoreDB, MongoDB	Both databases offer cloud-native features and scalability, with SingleStoreDB providing SQL capabilities and MongoDB offering schema flexibility.

Single Store Components

Cluster

A **cluster** in **SingleStoreDB** is a **distributed system** composed of multiple nodes (machines or containers) working together to provide **scalable**, **high-performance**, and **fault-tolerant** SQL-based data processing.

Cluster Components

Component	Purpose
Master Aggregator	Controls metadata, DDL statements, and cluster coordination.
Child Aggregators	Optional aggregators that help scale read/query performance.
Leaf Nodes	Store actual data and perform most of the heavy computation and storage.

Key Characteristics of a SingleStore Cluster

Aspect	Description
Distributed	Data is **sharded** and distributed across multiple **leaf nodes**.
Parallel Processing	Queries are split and run in parallel across leaf nodes for faster performance.
Separation of Roles	Aggregators handle SQL parsing & routing, while leaf nodes handle data/storage.
Horizontal Scalability	You can add more nodes to scale both compute and storage.
High Availability	Clusters can be set up with redundancy and failover for reliability.

How It Works?

Client connects to an aggregator (Master or Child).

1. Aggregator **parses and plans the query**.
2. Aggregator **distributes the work** to relevant **leaf nodes**.
3. Leaf nodes **process the data in parallel**.
4. Aggregator **merges the results** and returns the final response.

Nodes

1. Aggregator (AG)

- Acts as the **SQL interface** to the cluster.
- Parses, optimizes, and coordinates query execution.
- Aggregators **do not store data** themselves.
- Typically, you deploy **multiple aggregators** for **high availability and load balancing**.

Nodes

2. Master Aggregator

- One aggregator is designated as the **Master Aggregator**.
- Responsibilities:
  - Manages **cluster metadata**
  - Processes **DDL commands** (CREATE, ALTER, DROP)
  - Propagates metadata changes to leaf nodes and child aggregators
- Acts as the **control plane** of the cluster.

Nodes

3. Child Aggregator

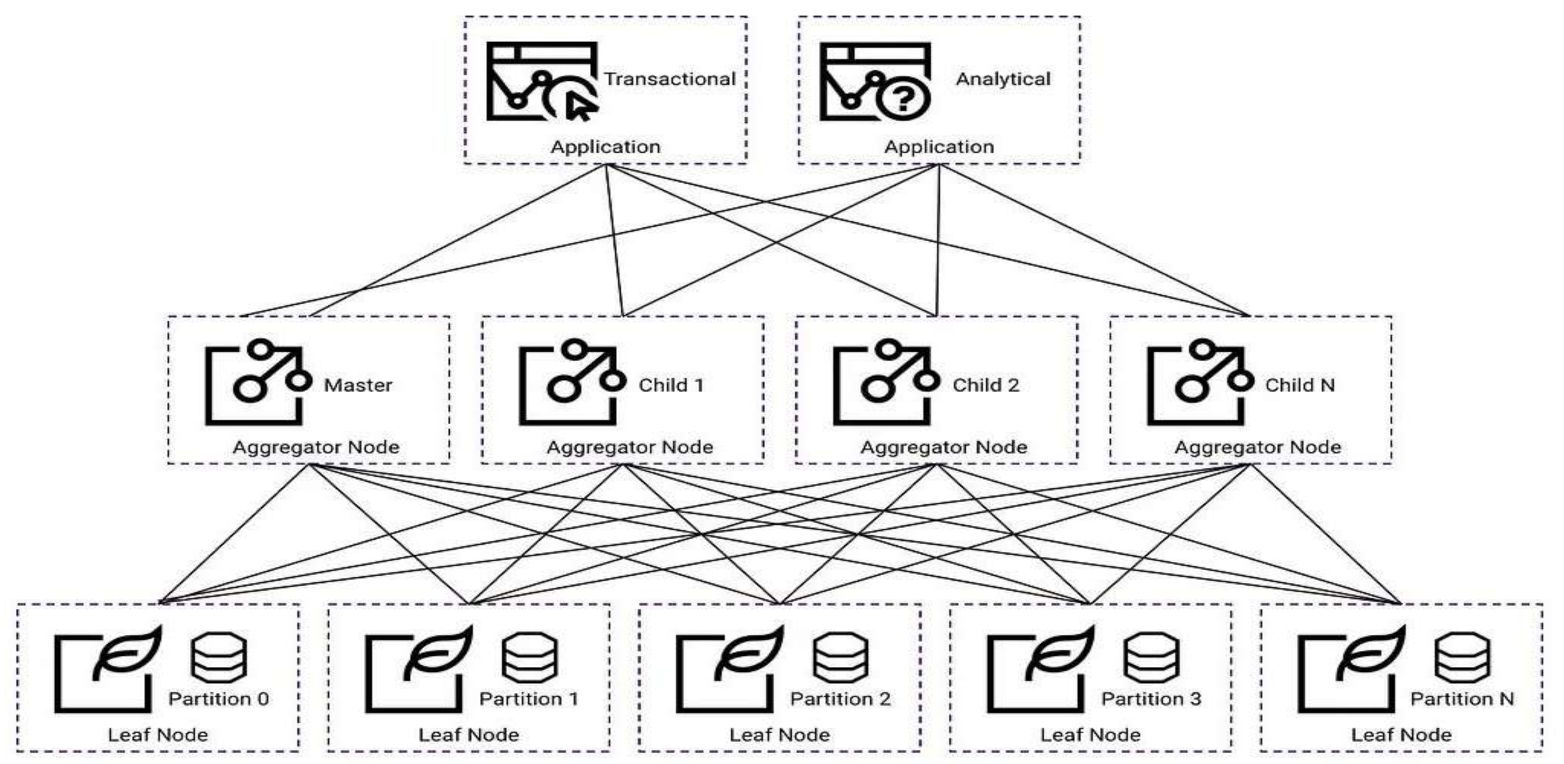
- Optional but useful for **read scaling** and load distribution.
- Forwards client queries to the leaf nodes just like the master aggregator, **without managing metadata**.
- Helps distribute query load when many concurrent users are querying.

Nodes

4. Leaf Node

- The **workhorse** of SingleStoreDB.
- Stores data in **rowstore and/or columnstore formats**.
- Executes **local parts of distributed queries**, such as filtering, joins, aggregations.
- Leaf nodes are **sharded** — data is partitioned across them based on sharding key.

Architecture



Tables

There are two types of tables in the SingleStore DB.

Columnstore and **Rowstore.**

Tables

Feature	Rowstore	Columnstore
Storage Format	Row-oriented (traditional RDBMS style)	Column-oriented (OLAP style)
Best For	OLTP (Transactional) workloads	OLAP (Analytical) workloads
Typical Use Cases	Real-time inserts, updates, deletes, lookups	Large-scale aggregations, reporting, analytics
Data Access Pattern	Fast access to entire rows	Efficient access to specific columns
Compression	Minimal compression	High compression using dictionary & run-length
Indexing	Supports secondary indexes	Uses segment-level metadata for query pruning

Tables

Feature	Rowstore	Columnstore
Insert/Update Speed	Fast for small, frequent writes	Slower for frequent updates; best for bulk loads
Query Performance	Fast for point queries, small joins	Fast for column scans, group by, aggregates
Memory Usage	Higher memory use	Optimized due to compression
Storage Location	In-Memory (rowstore tables reside in memory)	On-Disk (columnstore tables persist to disk)

When to use what

Rowstore

Use **rowstore** when your workload:

- Is transaction-heavy (inserts/updates)
- Requires fast key lookups
- Benefits from low-latency

Columnstore

Use **columnstore** when your workload:

- A large number of rows are scanned sequentially (i.e. millions of rows or >5% of the table)
- Aggregation happens over only a few columns (e.g. <10 columns)
- Benefits from storage savings and scan efficiency

Important Keys

1. **Shard Key**
Defines how data is distributed across partitions for parallel processing.

2. **Primary Key**
Uniquely identifies each row; enforces uniqueness and creates a clustered index.

3. **Unique Key**
Ensures all values in specified columns are unique; allows multiple per table.

4. **Sort Key (Columnstore only)**
Defines the storage order of columnstore data to improve range query performance.

Is a RAM-based rowstore faster than a disk-based columnstore?

No, columnstore tables are faster on some workloads – if the workload is batch inserts and sequential reads (e.g. an analytical workload with lots of scans) a columnstore can be significantly faster.

[Source1: Click Here]
[Source2: Click Here]

Do columnstore tables use memory?

Absolutely – SingleStore Helios uses the operating system disk buffer cache to cache segment files in memory. Good performance in columnstore can only be achieved when there is enough memory to cache the working set. In addition, columnstore tables use a **rowstore buffer table** as a special segment to **batch writes** to the disk.

[Source1: Click Here]

Connection With ASP.NET

Install SingleStoreConnector

Use NuGet Package Manager command: Install-Package SingleStoreConnector.

Create Connection String

Add your SingleStoreDB connection under <connectionStrings> in Web.config.

Example Config

```
server=your_host; port=3306; user=your_user; password=your_password; database=your_db;
```

Connection With ASP.NET

```
public class ProductRepository
{
        private readonly string _connectionString;

        public ProductRepository()
        {
        _connectionString = ConfigurationManager.ConnectionStrings["SingleStoreDb"].ConnectionString;
        }

        public List<Product> GetAllProducts()
        {
        var products = new List<Product>();

        using (var connection = new SingleStoreConnection(_connectionString))
        {
                connection.Open();
                using (var command = new SingleStoreCommand("SELECT Id, Name, Price FROM Products", connection))
                using (var reader = command.ExecuteReader())
        {
           while (reader.Read())
                {
                products.Add(new Product
                {
                        Id = reader.GetInt32("Id"),
                        Name = reader.GetString("Name"),
                        Price = reader.GetDecimal("Price")
                });
                }
        }
        }
        }

        return products;
        }
}
```

Fast Member

FastMember is a .NET library that provides high-performance access to object members (properties and fields) using dynamic code generation. It allows for fast reading/writing of members and efficient object-to-datatable mapping, making it ideal for performance-critical applications.

Why FastMember Is Faster Than Reflection?

FastMember uses **IL (Intermediate Language) code generation and caching**, avoiding the overhead of slow runtime type inspection that traditional reflection involves. Reflection has bad effect on high concurrent call.

Why FastMember Is Faster Than ORMs?

ORMs often include **extensive features** (tracking, validation, metadata, etc.), introducing overhead. FastMember focuses solely on **raw member access**, which keeps it lightweight and faster for direct property mapping.

Comparison Table: Reflection vs FastMember

Feature	Reflection	FastMember
Speed	Slower (runtime inspection)	Faster (cached IL-based access) Almost as fast as direct property access
Memory Allocation	Higher (new objects for access)	Lower (reuses accessors)
Complexity	Verbose and error-prone	Simple API
Object to DataTable	Manual or via other libs	Built-in support
Use Case Suitability	One-off access	High-performance loops and mappings
Caching Support	Manual	Automatic

FastMember Code Example

```
using FastMember;
 var person = new Person();
var accessor = TypeAccessor.Create(typeof(Person));
 // Set values
accessor[person, "Name"] = "Alice";
accessor[person, "Age"] = 30;
accessor[person, "IsActive"] = true;
// Get values
var name = (string)accessor[person, "Name"];
var age = (int)accessor[person, "Age"];
var isActive = (bool)accessor[person, "IsActive"];
```

Thank You