# Introduction

## What Is Embedded DB?

Embedded databases are database management systems (DBMSs) which are tightly **integrated / embedded with an application** and completely controlled by the application. This means an embedded DB is NOT shared among multiple applications, and especially, it **DON'T need a server** to run. When a database server is used as an embedded database, it is called a **database engine**.

Structurally, embedded databases may be relational, or non-relational/NoSQL. But in this document, we only talk about relational SQL databases.

For example, SQLite is an embedded database because it's **server-less** (it's self-contained in local machine). On the other hand, external databases (such as MySQL, MS SQL Server, etc.) requires a client and server architecture to interact over a network.

## Embedded DB vs External DB

|  |  |  |
| --- | --- | --- |
|  | **Embedded DB** | **External DB** |
| **Architecture** | Runs on **local** machine  + Directly stores data in files on local machine  + File format is cross-platform, easily copied and moved.  + Little configurations are required | Runs on **server**  + Stores data in files on a sever  + Only portable after being exported to a file and upload to another server  + More configurations are required |
| **Examples** | SQLite, MS SQL Server Compact, LevelDB, RocksDB, Berkley D, Firebird SQL | MySQL, MS SQL Server, Oracle 12c, MongoDB, PostgreSQL, MariaDB |
| **DB size** | Limited | Much more storage |
| **Speed** | Usually extremely quick | Usually much slower |
| **Data type** | Lesssupport | Moresupport |
| **Complex operations** | Less support | More support |
| **Multiple access** | Lacks of user management functionality  + Not suitable for multiple user access | Has a well-constructed user management system  + Can handle multiple users and grant various levels of permission |
| **Security** | Lacks of inbuilt authentication mechanism  + Database files can be accessed by anyone | Has many inbuilt security features. This includes authentication with a username, password, SSH, etc. |
| **Syntax** | Can be SQL or NoSQL | Can be SQL or NoSQL |
| **Use cases** | - Read and write directly from the disk  - Develop:  + Standalone, self-contained, no-network apps  + Embedded systems  + Mobile phones, cameras, IoT devices | - Read and write directly via a server  - Develop:  + Web-based apps  + Distributed systems  - Multiple user access, strong security and authentication features |

## Pros and Cons

**Pros:**

* **Extremely quick**: Network calls lead to significant latency. Because an embedded DB doesn't need any network call, it significantly reduces latency and DB access time.
* **Very lightweight**: Most embedded DBs occupy only some hundreds of KBs in the disk. In term of memory, they load the data when needed, rather than reading the entire file and hold it in memory, which significantly reduce memory use during run time.
* **Easy to set up**: Most embedded DBs are very easy to install. Just download the libraries, do some local configurations, and you're ready to go. No need any server-client configuration, or network setups.

**Cons:**

* **Limited DB size**: Most embedded DBs limit their database capability to some GBs. For example, MS SQL Server Compact limited DB size is only 4 GB.
* **Limited access**: Embedded DBs do not allow multiple accesses. Also, they don't have any specific user management functionality.
* **Security**: An embedded DB is inherently zero- or low-administration because the 'care-and-feeding' of the DBMS is carried out by the application itself. This means they do not have an inbuilt authentication mechanism, and thus, database files can be accessed by anyone.

## Types

Like other DBMSs, embedded databases are divided into two categories:

* SQL embedded databases
* NoSQL embedded databases

Differences between SQL and NoSQL embedded DBMSs:

|  |  |  |
| --- | --- | --- |
|  | **SQL** | **NoSQL** |
| **Full names** | Primarily called RDBMS or Relational databases | Primarily called as Non-relational or distributed database |
| **Design** | Uses SQL syntax and queries to analyze and get the data for further insights. They are used for OLAP systems. | Consists of various kinds of database technologies. These databases were developed in response to the demands presented for the development of the modern application. |
| **Query Language** | Structured query language (SQL) | No declarative query language |
| **Variations** | Only table-based databases | Can be document-based, key-value pairs, graph databases |
| **Schema** | Have a predefined schema | Use dynamic schema for unstructured data |
| **Scalability** | Vertically scalable **(1)** | Horizontally scalable **(2)** |
| **Examples** | MySQL, Oracle, Postgres, MS SQL Server  SQLite, MS SQL Server Compact | MongoDB, Redis, Neo4j, Cassandra, Hbase  LevelDB, RocksDB, LMDB |
| **Hierarchical data storage** | Not suitable for hierarchical data storage. | More suitable for the hierarchical data store as it supports key-value pair method. |
| **Consistency** | Should be configured for strong consistency. | Depends on DBMS as some offers strong consistency like MongoDB, whereas others offer only offers eventual consistency, like Cassandra. |
| **Best features** | Cross-platform support, secure and consistent | Easy to use, high performance, and flexible schemas **(3**) |
| **Use cases** | - Should be used when data validity is super important.  - Should be used when need to support dynamic queries. | - Should be used when it's more important to have fast data than correct data.  - Should be used when need to scale based on changing requirements. |
| **Hardware** | - Specialized DB hardware (Oracle Exadata, etc.)  - Highly available network (InfiniBand, Fabric Path, etc.)  - Highly available storage (SAN, RAID, etc.) | - Commodity hardware  - Commodity network (Ethernet, etc.)  - Commodity drives storage (standard HDDs, JBOD) |
| **Model** | ACID (Atomicity, Consistency, Isolation, and Durability) | BASE (Basically Available, Soft state, Eventually Consistent) |

**Complement points:**

**(1)**: *Vertical scaling* – In SQL databases, you typically scale by adding more power (CPU, RAM) to an existing machine.

**(2)**: *Horizontal scaling* – In NoSQL databases, you typically scale by adding more machines (PCs) into your pool of resources.

**(3)**: *Flexible data models* – Flexible schemas in NoSQL databases allow to easily make changes to your database as requirements change. You can iterate quickly and continuously integrate new application features to provide value to your users faster.

## Common Embedded DB Engines

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **SQLite** | **MS SQL Server Compact** | **LevelDB** | **RocksDB** | **Berkley DB** | **Firebird SQL** |
| **Architecture** | - SQL  - Table-based model  - Single-thread | - SQL  - Table-based model | - NoSQL  - Key-value model  - Multi-thread | - NoSQL  - Key-value model  - Multi-thread  - Built on top of LevelDB | - NoSQL  - Key-value model  - Multi-thread | - SQL  - Table-based model  - Multi-thread |
| **Developer** | D. Richard Hipp | Microsoft | Google | Facebook | Oracle | Firebird Project |
| **License** | Public domain  (Free) | Redistribution license  (Free) | BSD 3-Clause License  (Free) | BSD 3-Clause License  (Free) | Dual Licensed  (Free and Commercial) | LGPL |
| **Open-source code** | [Here](http://www.sqlite.org/src/) | No | [Here](http://www.github.com/google/leveldb) | [Here](https://github.com/facebook/rocksdb) | [Here](https://github.com/berkeleydb) | [Here](github.com/FirebirdSQL/firebird) |
| **Written in** | C | N/A | C++ | C++ | C | C++ |
| **Supported languages** | C, C++, C#, Java, JavaScript, Go, Perl, PHP, Python, Swift, Ruby, etc. | C, C++, C# | C++, Python, PHP, Go, Node.js | C++, Java | C++, C#, Java, Perl, PHP, Python, Ruby, etc. | C++, C#, VB, Java, PHP, Perl, Python, etc. |
| **OS** | Windows, macOS, Linux, Android, iOS, Symbian, etc. | Windows | Windows, macOS, Linux, Android | Windows, macOS, Linux, FreeBSD, OpenBSD, Solaris, AIX | Windows, macOS, Linux, Android, iOS, FreeBSD, OpenBS, etc. | Windows, macOS, Linux |
| **Size**  **(Footprint)** | ~700 KB | N/A | ~350 KB | N/A | ~1200 KB | ~4-5 MB |
| **Max DB size support** | 140 TB | 4 GB | N/A | N/A | 256 TB | 32 TB |

# Overall Architecture

External DB (MySQL, PostgreSQL, etc.) requires a separate server to operate. The applications use TCP/IP protocol to send and receive requests. This is called client/server architecture.



On the other hand, embedded DB is integrated with the application. So, the app can read/write directly from the database files stored on disk.



## Multiple Accesses + Multiple Users

Multi-user means that many tasks (possibly on many computers) can open connections to the database at the same time. The users in this context are threads in the web server which are accessing the database.

Different databases have different solutions for handling multiple connections working with the database at once. Generally, reading is not a problem, as multiple reading operations can overlap without disturbing each other. But only one connection can write data in a specific unit at a a time.

The difference between concurrency for databases is basically how large units they lock when someone is writing.

For example:

* MySQL has an advanced system where records, blocks or tables can be locked depending on the need, usually a single record, sometimes a table.
* SQLite has a simpler system where it only locks the entire database.

The impact of this difference is seen when you have multiple threads in the webserver, where some threads want to read data and others want to write data. MySQL can read from one table and write into another at the same time without problem. SQLite has to suspend all incoming read requests whenever someone wants to write something, wait for all current reads to finish, do the write, and then open up for reading operations again.

Look at [ScimoreDB](http://www.scimore.com/Embedded.html). It's an embedded database that supports multi-process (or user) read and write access. It also can work as a client-server database.

ScimoreDB is discontinued and therefore not included in the DB-Engines ranking.

If concurrent writes are an issue, you may want to look at [Berkeley DB](http://bit.ly/gYaDfo). The SQL API is completely SQLite compatible. In fact, it incorporates the SQLite executor top of Berkeley DB's storage engine, which **does** support multiple concurrent write operations.

SQLite multi accesses:

<https://www.sqlite.org/faq.html#q5>

<https://stackoverflow.com/a/5102105>

Embedded server database into app:

<https://www.mysql.com/oem/>

<https://www.percona.com/resources/videos/embedding-sql-engine-your-application>

<https://www.h2database.com/html/main.html>

# SQLite

## SQLite vs MySQL

|  |  |  |
| --- | --- | --- |
|  | **SQLite** | **MySQL** |
| **Architecture** | Runs on **local** machine  + Directly stores data in a single file  + No configurations are required  + Easy to set up | Runs on **server** |
| **Data type support** | **Limited** support  (Blob, Integer, Null, Text, Real) | **Wide-range** support  (Tinyint, Smallint, Mediumint, Int, Bigint, Double, Float, Real, Decimal, Double precision, Numeric, Timestamp, Date, Datetime, Char, Varchar, Year, Tinytext, Tinyblob, Blob, Text, MediumBlob, MediumText, Enum, Set, Longblob, Longtext) |
| **Storage** | SQLite library is about 250 KB in size | MySQL server is about 600 MB |
| **Multiple access** | Does **not have any specific user management** functionality  + Not suitable for multiple user access | Has a **well-constructed user management** system  + Can handle multiple users and grant various levels of permission |
| **Scalability** | Suitable for **smaller databases**  + As the database grows, the memory requirement gets larger.  + Performance optimization is harder. | Easily scalable and can handle a **bigger database** with less effort. |
| **Security** | Does **not have an inbuilt authentication mechanism**  + Database files can be accessed by anyone | **Has many inbuilt security features**. This includes authentication with a username, password, and SSH. |
| **Syntax** | Uses **standard SQL** syntax with minor alterations | Slightly different syntax as compared to conventional SQL |
| **When to Use** | - Developing small standalone apps  - Read and write directly from the disk  - Basic development and testing | - Web-based applications  - Large database and more scalability  - Multiple user access  - Require strong security and authentication features |

## Migration from MySQL to SQLite

<https://stackoverflow.com/a/9933603>