# 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 |
| **Storage** | Can be on-disk or in-memory | Can be on-disk or in-memory |
| **Examples** | SQLite, MS SQL Server Compact, LevelDB, RocksDB, Berkeley DB, 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** | Very limited 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 support very limited multiple accesses (more details at [this section](#_Single_Access_vs.)). 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

Reference: <https://db-engines.com/en/ranking/>

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **SQLite** | **LevelDB** | **RocksDB** | **Berkeley DB** | **Firebird SQL** |
| **SQL or NoSQL** | SQL | NoSQL | NoSQL | NoSQL | SQL |
| **Model** | Table-based | Key-value | Key-value | Key-value | Table-based |
| **In-memory support (1)** | Yes | No | Yes | Yes | No |
| **Concurrency** | - Multi-process  - Multi-threads | - Single-process  - Multi-threads (read-write) | - Multi-process (read-only)  - Multi-threads | - Multi-process  - Multi-threads |  |
| **Developer** | D. Richard Hipp | Google | Facebook | Oracle | Firebird Project |
| **License** | Public domain  (Free) | BSD 3-Clause License  (Free) | BSD 3-Clause License  (Free) | Dual Licensed  (Free-Commercial) | LGPL |
| **Open-source code** | [Here](http://www.sqlite.org/src/) | [Here](http://www.github.com/google/leveldb) | [Here](https://github.com/facebook/rocksdb) | [Here](https://github.com/berkeleydb) | [Here](file:///C:\Users\ADMIN\AppData\Roaming\Microsoft\Word\github.com\FirebirdSQL\firebird) |
| **Written in** | C | C++ | C++ | C, C++ or Java (based on edition) | C, C++ |
| **Transaction concepts** | ACID | No | Yes | ACID | ACID |
| **Supported languages** | C, C++, C#, Java, JavaScript, Go, Perl, PHP, Python, Swift, Ruby, etc. | C++, Python, Java, Go, Node.js | C++, Java | C, 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, 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 | ~350 KB | N/A | ~1200 KB | ~4-5 MB |
| **Max DB size support** | 140 TB | N/A | N/A | 256 TB | 32 TB |

**Complement points**:

(1) and (2) – *In-memory and on-disk database*s: These terms are mentioned in [this section](#_In-Memory_vs_On-Disk).

**Other embedded DBs:**

* **Neo4j**: It is an open-source embedded graph database developed by Neo Technologies.
* **Apache** **Derby**: It is an open-source relational database which is fully based on (written/implemented in) Java. Derby can be used as an external DB (with server) or an embedded DB (with an engine embedded in to Java applications and running in the same JVM as the application).
* **IBM Informix**: It is a commercial, relational database which can be used as an external DB or an embedded DB.
* **SolidDB**: It is an in-memory embeddable relational database owned formerly by IBM but divested and now owned by Unicom Systems of Unicom Global.
* **OrientDB**: It is an open-source NoSQL embedded database which supports multiple models (key-value, document-based, graph). It is written in Java, and support various languages.
* **eXtremeDB**: It is an in-memory embedded relational database for IoT connected devices and time series analyses.
* **UnQLite**: It is an in-process software library which implements a self-contained, serverless, zero-configuration, transactional NoSQL database engine
* **Raima**: It is a commercial, relational database which can be used as an external DB or an embedded DB.

# 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.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| |  | | --- | |  | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

# Choose An Embedded Database

## Single Access vs. Multiple Accesses

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. For example:

* 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.

About multiple-access capability, embedded databases can be categorized into following groups:

|  |  |  |  |
| --- | --- | --- | --- |
| **Process** | **Thread** | **Description** | **Examples** |
| Single | Single | - The DB only allows operations in a single thread of a single process.  - Because all operations run serially, they're readable and writable without any issue. |  |
| Single | Multiple | - The DB doesn't allow two instances to be open at the same time. All options are for a **single process.**  - Within a process, the same object may be safely shared (**read/write**) by multiple concurrent threads without any external synchronization (the DB implicitly does the required synchronization). | [LevelDB](https://chromium.googlesource.com/external/leveldb/+/HEAD/doc/index.md#concurrency)  (Sol [here](https://github.com/juliangruber/multilevel)) |
| Multiple | Multiple | - The DB allows two instances to be opened in **read-only** mode from multiple processes at the same time.  - Within a process, **read-only** operations are allowed in multiple threads. |  |
| - The DB allows two instances to be opened in **read-only** mode from multiple processes at the same time.  - Within a process, **read/write** operations are allowed in multiple threads. | [RocksDB](https://github.com/aayushKumarJarvis/rocks-wiki/blob/master/RocksDB-FAQ.md) |
| - The DB allows two instances to be opened in **read-write** mode from multiple concurrent processes. But at any instant in time, **only one writer** can modify data (the DB implicitly implements locking/ blocking to ensure this behavior).  - Within a process, **read/write** operations are allowed in multiple threads. | [SQLite](https://www.sqlite.org/faq.html#:~:text=SQLite%20allows%20multiple%20processes%20to,only%20takes%20a%20few%20milliseconds.) **(1)**,  [Berkeley DB](https://docs.oracle.com/cd/E17275_01/html/programmer_reference/cam.html) **(1)**  [LMDB](https://github.com/jnwatson/py-lmdb/issues/114) |

**Complement points**:

1. *SQLite, Berkeley DB*: In these DBs, when another process tries writing, the **entire DB is locked** until the previous process finished writing. Nothing, even reading, can access the DB at all. But the locking duration normally only takes a few milliseconds.

On the other hand, external databases (client/server) usually have an advanced system where records, blocks or tables can be locked depending on the need, usually a single record, sometimes a table.

**Limitation of embedded DBs:**

External databases (client/server DB, such as MySQL, PostgreSQL, or Oracle) usually support a higher level of concurrency and allow multiple processes to be writing to the same database at the same time. This is possible in a client/server database because there is always a single well-controlled server process available to coordinate access. If your application has a need for a lot of concurrency, then you should consider using a client/server database. But experience suggests that most applications need much less concurrency than their designers imagine.

## In-Memory vs. On-Disk Database

|  |  |  |
| --- | --- | --- |
|  | **In-memory (1)** | **In-disk** |
| **Storage location** | - The whole DB stored in main **memory**  - No need to perform disk I/O to query or update data | - The whole DB stored on **disk**  - Disk I/O needed to move data into main memory when needed |
| **Speed** | Much faster **(2)** | Much slower |
| **Data loss** | Data can be **volatile** and **persistent** depending on the DB.  Data loss **(3)** can happen. | Data is always **persisted** to disk |
| **Data structure** | Specialized data structures and **index** structures assume data is always in main memory. | Traditional data structures like **B-Trees** **(4)** designed to store tables and indices efficiently on disk. |
| **DB size** | Database size **limited** by the amount of main memory.  Note: There are in-memory databases which can work with data sets larger than available RAM | **Virtually** **unlimited** database size. |
| **Use cases** | Optimized for specialized workloads; i.e., communications industry-specific HLR/HSS workloads. | Support very broad set of workloads, i.e., OLTP, data warehousing, mixed workloads, etc. |

**Complement points**:

1. *In-memory*: Also called in-memory database (IMDB) or main memory database (MMDB)
2. *Speed*: Memory databases are faster than disk databases because accessing data in memory eliminates latency when switching between disk blocks and seek time when querying the data. In addition, the internal optimization algorithms are simpler and execute fewer CPU instructions.
3. *Data loss*: It is a possibility when working with in-memory DBs. There are some approaches used to reduce data loss, but it's still a possibility:

* **Persistence**: Might support two modes for opening a database in-memory:
* **Volatile** – The DB is empty when first opened and all contents are discarded when it's closed.
* **Persistent** – The DB is populated from content on disk when opened, and changed data (inserts, updates, deletes) is written to disk when closed. This way, data loss is limited to what has happened since the last persistence operation.

Persistent is implemented using 2 mechanisms in combination:

* Periodically take a snapshot of the in-memory data in the data node and write it to disk. This is referred to as a *Local Checkpoint* (LCP)
* Each change is written to a log buffer. These buffers are periodically flushed to a disk-based log file. This is coordinated across all data nodes in the Cluster and is referred to as a *Global Checkpoint* (GCP)

More details [here](http://www.clusterdb.com/mysql-cluster/how-can-a-database-be-in-memory-and-durable-at-the-same-time).

* **Replication**: Many DB systems support replicating changes to another DB instance (or another DB system). This way allows data that may have been lost from the in-memory copy to be recovered from the replicated copy.

1. *B-Trees*: A disk-based database commonly uses a B-Tree based index to limit the number of disk access required to locate a row.

**Can an on-disk databases be used in-memory?**

Yes! **In-memory data can be loaded from disk when opened and saved to disk when closed**, or when "save points" are requested. Save points are *atomic* (the 'A' in ACID), meaning that all changes to the data since the last save point are written to disk together, or none of them are. This makes them transactional, but not on the same transaction boundaries as an on-disk database.

**Are all embedded databases are in-memory?**

Most embedded DB engines used a hard-disk for data storage as the amount of memory available would not allow for sufficient data volume. As the size of memory in embedded devices increase, many vendors began adding in-memory capabilities. Today many database engines, embedded or external, have supports for in- memory.

## License

Databases with Dual Licensed (such as Berkeley DB) don't charge if you use it in redistribute projects (the complete source code must be available and freely redistributable under reasonable condition).

On the other hand, databases with Public domain license or with BSD license (such as SQLite, LevelDB, RocksDB) are totally free.

# 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>

# RocksDB

Why LevelDB:

[leveldb/LICENSE at master · google/leveldb · GitHub](https://github.com/google/leveldb/blob/master/LICENSE)

## Features

RocksDB provides all of the features of LevelDB, including:

* Keys and values are arbitrary byte arrays.
* Data is sorted by key.
* The basic operations are Put, Get, Write, Delete. They're thread safe.
* Callers can provide a custom comparison function to override the sort order.
* Multiple changes can be made in one atomic batch.
* Users can create a transient snapshot to get a consistent view of data.
* Forward and backward iteration is supported over the data.
* Data is automatically compressed using the Snappy compression library.
* External activity (file system operations, etc.) is relayed through a virtual interface, so users can customize the OS interactions.

Plus:

* Column Families
* Transactions and WriteBatchWithIndex
* Backup and checkpoints
* Merge operators
* Compaction Filters
* RocksDB Java
* Manual compactions run in parallel with automatic compactions
* Persistent Cache
* Bulk loading
* Forward Iterators/ Tailing iterator
* Single deleteTri
* Delete files in range
* Pin iterator key/value

Why RocksDB?

leveldb may get corrupted after power loss

## Installation

Stable release version: <https://github.com/facebook/rocksdb/releases>

## Use

First check development FAQs: <https://github.com/facebook/rocksdb/wiki/RocksDB-FAQ>

# Berkeley DB

## Why Berkeley DB?

* **Type**: NoSQL (key-value model)
* **License**: Open source (with either free or commercial license)
* **Supported languages**: C, C++, Java, PHP, and many more.
* **Developer**: Oracle (with long-term development/maintenance plan, full documents/references, great community, etc.)
* **Multiple access**: Allows multiple applications to access and modify the DB. Its principle is "multiple reader – one writer" at a time. More details here: <https://stackoverflow.com/a/66028601/14835442>
* **In-memory support**: In addition to in-disk storage, Berkeley DB supports in-memory supports which improve execution time while offers always unlimited DB size.

## Download

**Warning:**

Don't clone source code from <https://github.com/berkeleydb/libdb>, because it's very out of date.

Download here: <https://www.oracle.com/database/technologies/related/berkeleydb-downloads.html>

Note: Log in with this account (username: [triho1110@gmail.com](mailto:triho1110@gmail.com), password: Honhantri11101995)

## Build

Full guide: <https://docs.oracle.com/database/bdb181/html/installation/index.html>

### Linux

**Prerequisite:**

First of all, make sure to install following packages:

$ sudo apt -y install gcc

$ sudo apt -y install make

$ sudo apt install libdb-dev # Libraries and header files of C for Berkeley DB on Linux

$ sudo apt install libdb++-dev # Libraries and header files of C++ for Berkeley DB on Linux

**1. Build the libraries**

Change to the <db-root-dir>/build\_unix directory and enter the following commands:

../dist/configure

This symbolic link is created in step 1.

Its actual path is:

<db-root-dir>/build\_unix/.libs/libdb-<ver>.so

$ whereis libdb

/usr/lib/x86\_64-linux-gnu/libdb.so 🡪 0Kb

/usr/lib/x86\_64-linux-gnu/libdb.a 🡪 0Kb

make

**2. Install the libraries**

Run:

make install

**3. Run a sample**

Sample codes, in several different language, are included with Berkeley DB.

Let's pick up a C sample in <db-root-dir>/examples/c/getting\_started directory, then build and run it with following commands:

$ gcc -o read gettingstarted\_common.c example\_database\_read.c -ldb

$ ./read # If it returns outputs, we're successful

**Tips:**

* To rebuild Berkeley DB, run:

$ make clean

$ make

* To uninstall Berkeley DB, run:

$ make uninstall

* To reconfigure and rebuild Berkeley DB (in other words, start from scratch), run:

$ make realclean

$ ../dist/configure

$ make

* To build multiple UNIX versions of Berkeley DB in the same source tree, create a new directory at the same level as the build\_unix directory, and then configure and build in that directory as described previously.

### Windows

## Use

Full API references: <https://docs.oracle.com/database/bdb181/html/>

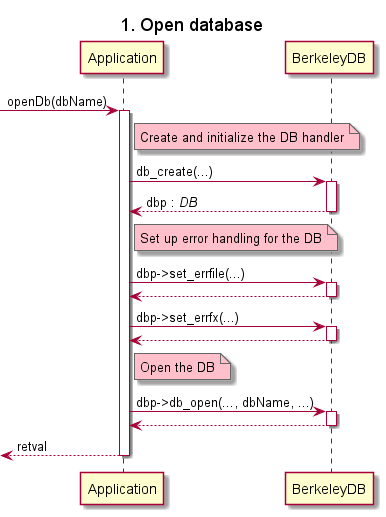
### Flows of Basic Operations

Following sequence diagrams demonstrate the basic flow of database manipulation using Berkeley DB.

**Note**: APIs used in the diagrams is for C language.

#### Open Database

The flow of **opening a database** is as below:



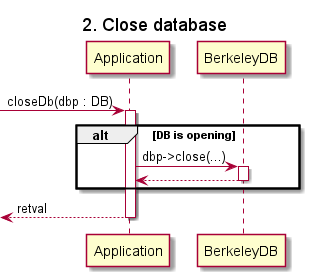
**Details:**

You interact with most DB APIs using special structures that contain pointers to functions (also named 'callbacks'). The variable that you use to access these callbacks is referred to as a **handler**. To use a database, you need to obtain a handle to that database.

Note that by default, DB structure does not create databases if they do not already exist. To override this behavior, specify the DB\_CREATE flag on the DB->open() method.

#### Close Database

The flow of **closing database** is as below:



**Details:**

Once you are done using the database, you must close it. You use the DB->close() method to do this.

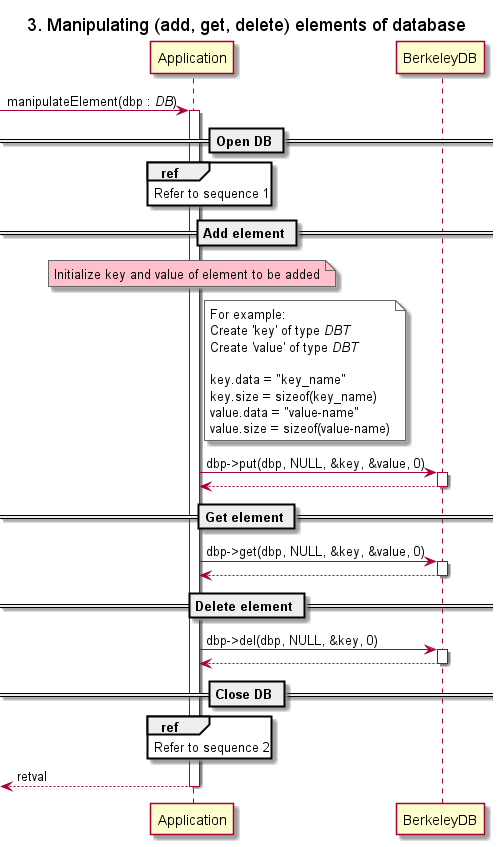
Closing a database causes it to become unusable until it is opened again.

Notes:

* Should close any open cursors before closing your database. Active cursors during a database close can cause unexpected results, especially if any of those cursors are writing to the database.
* Always make sure that all database accesses have completed before closing the database.

#### Manipulating (Add, Get, Delete) Elements

The flow of manipulating (add, get, delete) elements of database is as below:

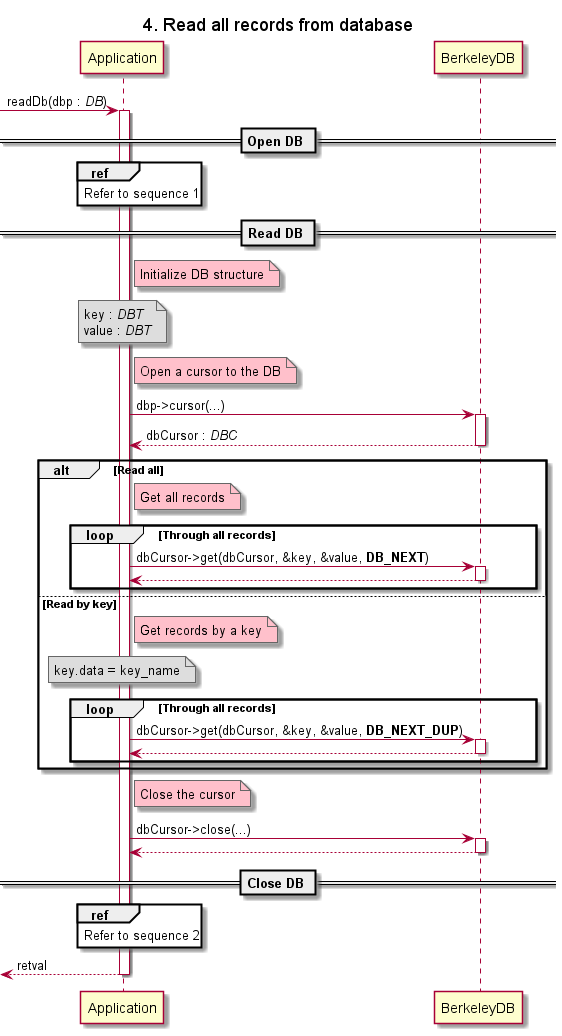


**Details:**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Description** | **API** |
| **Get** | Retrieving a record from DB. | get() |
| **Add**  **(or Put)** | Storing a record to DB.  **Sorting:**  Once a record is stored, it's sorted according to whatever sort order is in use by the DB. Sorting is mostly performed based on the key, but sometimes the data is considered too.  **Duplicate records:**  If you put a record having key already existing in the DB, then the existing record is replaced with the new data. But, if the DB supports duplicate records (that is, records with identical keys but different data), then that new record is stored as a duplicate record and any existing records are not overwritten.  If a DB supports duplicate records, then you can use a DB handler to retrieve only the first record in a set of duplicate records. | put() |
| **Delete** |  |  |

#### Read All Records

The flow of **reading all records** from database is as below:



**Details:**

In addition to using a DB handler, you can read/write data using a **cursor**. It's iterator that you can use to walk over records in a DB (from the first record to the last, and from the last to the first).

You can also use cursors to seek to a record.

In the event that a DB supports duplicate records, cursors are the only way you can access all the records in a set of duplicates.

### Access Methods

An access method can be **selected only when the database is created**. API usage is generally identical across all access methods. That is, while some exceptions exist, mechanically you interact with the library in the same way regardless of which access method you have selected.

For choosing an access method, consider:

* What you want to use as a key
* How good the performance

Berkeley DB supports following access methods:

|  |  |
| --- | --- |
| **Access Method** | **Description** |
| B-Tree | Data is stored in a sorted, balanced tree structure.  Both the key and the data for B-Tree records can be arbitrarily complex **(1)**.  Also, although not the default behavior, it is possible for two records to use keys that compare as equals. When this occurs, the records are considered to be duplicates of one another. |
| Hash | Data is stored in an extended linear hash table.  Like B-Tree, the key and the data used for Hash records can be of arbitrarily complex data.  Also, like B-Tree, duplicate records are optionally supported. |
| Queue | Data is stored in a queue as fixed-length records. Each record uses a logical record number as its key.  Queue is designed for fast inserts at the tail of the queue, and it has a special operation that deletes and returns a record from the head of the queue.  Queue is unusual in that it provides record level locking. This can provide beneficial performance improvements in applications requiring concurrent access to the queue. |
| Recno | Data is stored in either fixed or variable-length records. Like Queue, Recno records use logical record numbers as keys. |

1. – *Arbitrarily complex*: Data can contain single values (integer, string, etc.) or complex types (structure).

Choose an access method: https://docs.oracle.com/database/bdb181/html/gsg/CXX/accessmethods.html

Other:

Secondary Database

DB provides a special kind of a database called a secondary database. Secondary databases serve as an index into normal databases (primary database). Secondary databases are interesting because DB records can hold complex data types, but seeking to a given record is performed only based on that record's key. If you wanted to be able to seek to a record based on some piece of information that is not the key, then you enable this through the use of secondary databases.

getopt()

<https://www.gnu.org/software/libc/manual/html_node/Getopt.html>

[8 Ways to Measure Execution Time in C/C++ | Level Up Coding (gitconnected.com)](https://levelup.gitconnected.com/8-ways-to-measure-execution-time-in-c-c-48634458d0f9)

[Measure execution time with high precision in C/C++ - GeeksforGeeks](https://www.geeksforgeeks.org/measure-execution-time-with-high-precision-in-c-c/)

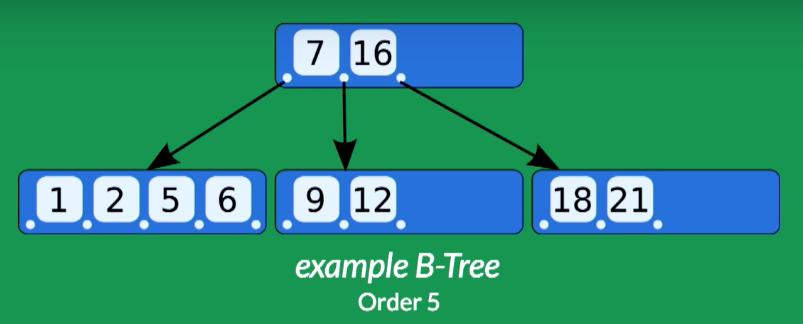
[benchmarking - Execution time of C program - Stack Overflow](https://stackoverflow.com/questions/5248915/execution-time-of-c-program)

# B-Tree

Must watch this video first: <https://www.youtube.com/watch?v=C_q5ccN84C8&ab_channel=FullstackAcademy>

## What Is B-Tree?

**Example:**



**Root node**

**Child node**

**Key**

The above B-Tree has:

* 1 root containing 2 keys: 7 and 16
* 3 child nodes. The left most node containing keys whose values are < 7. The middle node containing keys whose values are > 7 and < 16. The right most node containing keys whose values are > 16.
* Order of 5 (because its node – the left most one – has at most 5 children)

**General properties:**

A B-Tree of order **m** has:

* All leaves appear in the same level.
* Every node has at most **m** children.
* A non-leaf node with **k** children contains **k-1** keys.
* The root has at least two children if it is not a left node.
* Every non-left node (except the root) has at least a **ceiling of m/2** children.

**Searching, inserting and deleting:**

Watch the video: 7:28

**Time complexity:**

|  |  |
| --- | --- |
| Algorithm | Time Complexity |
| Search | O(log n) |
| Insert | O(log n) |
| Delete | O(log n) |

**When is B-Tree helpful?**

To understand the use of B-Trees, we must think of the huge amount of data which cannot fit in main memory. So, we have to put the data in disk. But accessing and reading data from disk takes much more time than from main memory.

In this case, we can take advantage of B-Trees to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, etc.) require O(h) disk accesses where h is the height of the tree. But the height of B-Trees is kept low by putting maximum possible keys in a B-Tree node, so total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree, etc.

To conclude, **whenever you deal with some kind of external memory and the time to access the data of a node greatly exceeds the time spent processing that data (such as big databases)**, consider using B-Trees.