Vinh Quang Le, Tri Nhan Ho

vinhlq@neal-technology.com, trihn@neal-technology.com

Abstract

TBD

A Research ON EMBedded Database SYSTEM

Apply Embedded Database on GPU Device

Table of Contents

[Abstract 4](#_Toc63173031)

[Overview 5](#_Toc63173032)

[Embedded DB 6](#_Toc63173033)

[I. Embedded Database Overview 6](#_Toc63173034)

[1. What is Embedded Database 6](#_Toc63173035)

[2. Embedded Database vs. External Database 6](#_Toc63173036)

[3. Embedded Database Types 8](#_Toc63173037)

[4. Embedded Database Engines 9](#_Toc63173038)

[II. Embedded Database Feasibility 11](#_Toc63173039)

[1. Pros/Cons of Embedded Database 11](#_Toc63173040)

[2. Use cases 11](#_Toc63173041)

[III. Choose an Embedded Database 12](#_Toc63173042)

[1. Single Access vs. Multiple Accesses 12](#_Toc63173043)

[2. In-Memory vs. On-Disk Database 13](#_Toc63173044)

[3. License 14](#_Toc63173045)

[IV. Berkeley DB 15](#_Toc63173046)

[1. Why Berkeley DB? 15](#_Toc63173047)

[2. Downloading 15](#_Toc63173048)

[3. Building 15](#_Toc63173049)

[4. Examples 16](#_Toc63173050)

[GPU (Graphic Processing Unit) 20](#_Toc63173051)

[I. What is GPU, networking using GPU 20](#_Toc63173052)

[II. Architecture 20](#_Toc63173053)

[1. CPU vs GPU Architecture 20](#_Toc63173054)

[2. GPU CUDA 21](#_Toc63173055)

[3. Basic Units of CUDA 22](#_Toc63173056)

[4. CUDA Memory types 23](#_Toc63173057)

[III. GPU System Application 25](#_Toc63173058)

[1. GPU as an accelerated device (CPU-GPU combination) 25](#_Toc63173059)

[2. GPU as a control server (only GPU) 26](#_Toc63173060)

[Embedded Database on GPU 27](#_Toc63173061)

[I. Current Embedded Database Engine 27](#_Toc63173062)

[II. Comparison 27](#_Toc63173063)

[Demonstration 28](#_Toc63173064)

[I. Demonstration overview 28](#_Toc63173065)

[II. Basic Design 28](#_Toc63173066)

[III. Measure performance 28](#_Toc63173067)

[IV. Evaluation result 28](#_Toc63173068)

[Conclusion 29](#_Toc63173069)

[Reference 30](#_Toc63173070)

[Figure 1 External Database Structure 7](#_Toc62465140)

[Figure 2 Embedded Database Structure 8](#_Toc62465141)

[Figure 3 CPU & GPU Architecture 16](#_Toc62465142)

[Figure 4 CPU & GPU combination 17](#_Toc62465143)

[Figure 5 Basic unit of CUDA architecture 18](#_Toc62465144)

[Figure 6 CUDA memory type structure 19](#_Toc62465145)

[Figure 7 GPU as an accelerated device structure 21](#_Toc62465146)

[Figure 8 GPU as a control server 22](#_Toc62465147)

# Abstract

TBD

# Overview

TBD

# Embedded DB

## Embedded Database Overview

### What is Embedded Database

Embedded database systems are database management systems (DBMSs) which are tightly **integrated 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.

Structurally, embedded databases may be relational, or non-relational/NoSQL.

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

### Embedded Database vs. External Database

The following tables describes some main differences between embedded databases and external databases:

|  |  |  |
| --- | --- | --- |
|  | **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 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, Berkley D, Firebird SQL | MySQL, MS SQL Server, Oracle 12c, MongoDB, PostgreSQL, MariaDB |
| **DB size** | Limited | Much more storage |
| **Speed** | Much quicker | 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. |
| **Query Language** | 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 |

The following block diagrams describe how these databases are different in terms of 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.



Figure 1 External Database Structure

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



Figure 2 Embedded Database Structure

### Embedded Database Types

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

* SQL embedded databases
* NoSQL embedded databases

Differences between SQL and NoSQL embedded DBMSs:

|  |  |  |
| --- | --- | --- |
|  | **Embedded SQL** | **Embedded 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** | SQLite, MS SQL Server Compact | 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.

### Embedded Database Engines

There are some common embedded DB engines with great supports, including:

* **SQLite**: The most used database engine in the world. It's widely used in mobile phones, televisions, cameras, home electronic devices, and most computers.
* **MS SQL Server Compact**: A compact relational database produced by Microsoft for applications that run on mobile devices and desktops.
* **LevelDB**: A high-performance embedded DB produced by Google for key-value data in byte arrays. It's not a SQL database, and does not have a relational data model and it does not support SQL queries.
* **RocksDB**: A high-performance embedded DB produced by Facebook for key-value data. It's builds on top of LevelDB. It comes off as an exceptional embedded DB but without support for replication.
* **Berkley DB**: A high-performance embedded DB produced by Oracle for key-value data in byte arrays. It's not a relational database, like LevelDB and RocksDB.
* **Firebird SQL**: A relational database offering many ANSI SQL standard features with excellent concurrency, high performance, and powerful language support for stored procedures and triggers.

**Comparison table:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **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\Surface\AppData\Local\Microsoft\Windows\INetCache\Content.Outlook\ZDWM92J7\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#, 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).

## Embedded Database Feasibility

### Pros/Cons of Embedded Database

**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 access. 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.

### Use cases

You can consider using an embedded DB in following use cases:

* When your application is stand-alone. Embedded DBs are self-contained, with no connection to other apps or databases.
* When great amount of data needs to be stored, but the data can be recreated quickly and is not critical.
* Your app needs to fetch some kind of state data or do some kind of validation really quickly.
* You’re have a white list which needs to be checked at every HTTP request in a millisecond or less.
* You’re building an ad server where read latency requirements to querying has to be really low, in the order of a few milliseconds.
* An in-memory database would be a great fit, so would be any other in memory querying system or if the data is small, even a local variable within the app.
* You want to write some non-critical data, in the quickest possible fashion that can be eventually consistent across other servers in your cluster.
* …

## 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**.

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) |
| 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.

## 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 at [section III, 1](#_Single_Access_vs.).
* **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.

### Downloading

Latest version here: <https://www.oracle.com/database/technologies/related/berkeleydb-downloads.html> (Note: You have to log in to download.)

**Warning:**

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

### Building

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

Berkeley DB can be built and run on many platforms. In this research, we choose **Linux** as it's the best option for embedded systems.

**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 # Berkeley DB on Linux needs libraries and header files in this package

**1. Build the libraries**

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

../dist/configure

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:**

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

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

* 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.

### Examples

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

Now let's lern some basic examples to get started with Berkeley DB. Their sequence diagrams are below. The language we're using is C, but note that the flows in below diagrams share similarities between different languagues.

**Open Database**

The flow of **opening database** in Berkeley DB is as below:

The flow of opening a database is as below:
    .. uml::

        @startuml
        skinparam NoteBackgroundColor #DDDDDD
        skinparam NoteBorderColor #666666

        TITLE 1. Open database
        PARTICIPANT Application AS app
        PARTICIPANT BerkeleyDB AS db

        -> app: openDb(dbName)
        ACTIVATE app

        NOTE RIGHT app #pink: Create and initialize the DB handler
        app -> db ++: db_create(...)
        RETURN dbp : //DB//

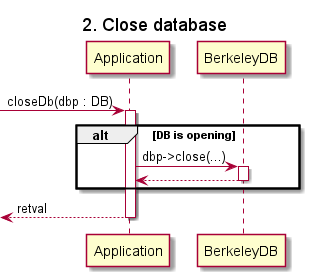
        NOTE RIGHT app #pink: Set up error handling for the DB
        app -> db ++: dbp->set_errfile(...)
        RETURN
        app -> db ++: dbp->set_errfx(...)
        RETURN

        NOTE RIGHT app #pink: Open the DB
        app -> db ++: dbp->db_open(..., dbName, ...)
        RETURN

        <-- app: retval
        DEACTIVATE app
        @enduml

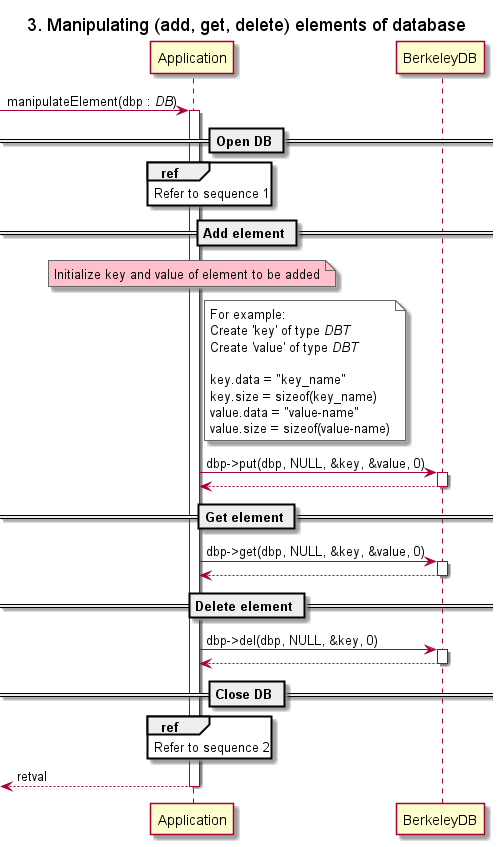
**Close Database**

The flow of **closing database** in Berkeley DB is as below:



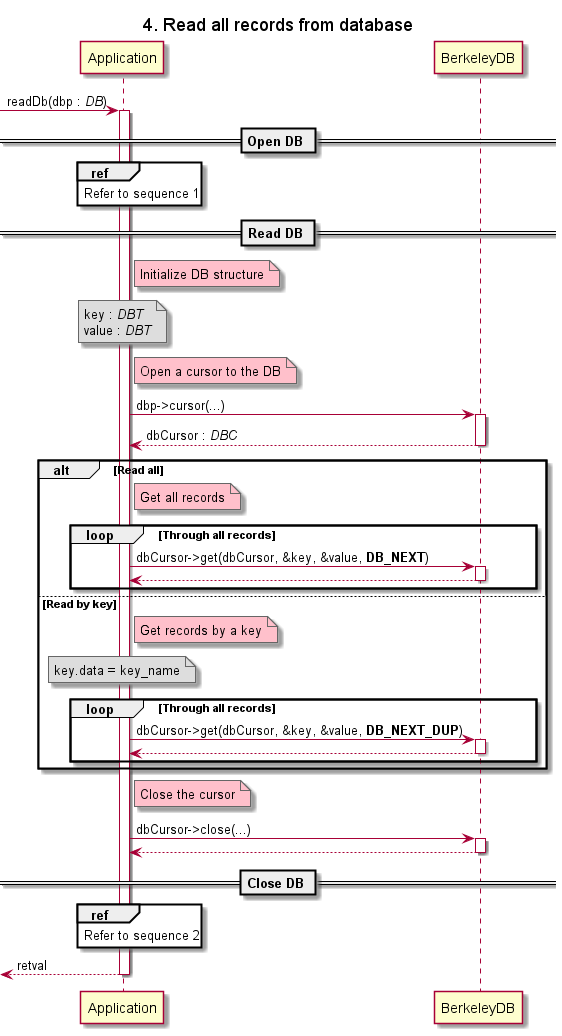
**Manipulating (Add, Get, Delete) Elements**

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



**Read All Records**

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



# GPU (Graphic Processing Unit)

## What is GPU, networking using GPU

The graphics processing unit, or GPU, has become one of the most important types of computing technology, both for personal and business computing. Designed for parallel processing, the GPU is used in a wide range of applications, including graphics and video rendering. Although they’re best known for their capabilities in gaming, GPUs are becoming more popular for use in creative production and artificial intelligence (AI).

GPUs become the best choice for many types of parallel general-purpose application from gaming, video editing to machine learning, deep learning. GPUs are widely used for accelerating parallel tasks in high-performance computing, and their architecture has been evolving to enable efficient execution of complex, general-purpose workloads. Nvidia listed up more than 200 general-purpose applications using GPU but no mention of GPU using for networking, telecom... Recent work shows that GPUs can boost power efficiency and performance for web servers, but the GPU prototype lacked an actual network implementation because GPU-native networking support does not yet exist.

## Architecture

### CPU vs GPU Architecture

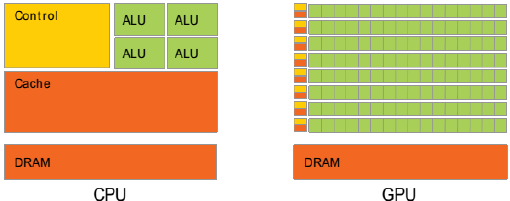


Figure 3 CPU & GPU Architecture

Difference between CPU and GPU

|  |  |
| --- | --- |
| **CPU** | **GPU** |
| Has large board instruction set, manages every input/output of a computer | Has specific instruction set, only work for its functions |
| CPU core is fast and smart | GPU core is slow and not full feature |
| Best suited for general-purpose serial applications | Best suited for repetitive and highly-parallel computing tasks |
| CPU core consists of:   * Control unit block * ALU block * Cache memory | GPU consists of:   * PF unit * INT unit |

### GPU CUDA

CUDA (Compute Unified Device Architecture) is NVIDIA’s GPU architecture featured in the GPU cards, positioning itself as a new means for general purpose computing with GPUs. CUDA gives advantage of massive computational power to the programmer. This massive parallel computational power is provided by Nvidia’s graphics cards.

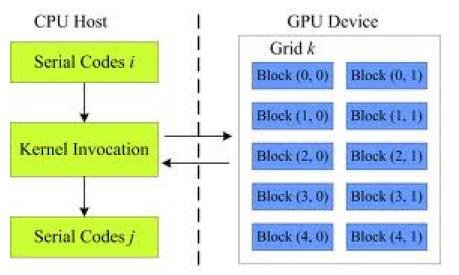


Figure 4 CPU & GPU combination

For running multithreaded applications there is no need of streaming computing in GPU, because cores can communicate also can exchange information with each other. CUDA is only well suited for highly parallel algorithms and is useful for highly parallel algorithms. If you want to increase performance of your algorithm while running on GPU then you need to have many threads. Normally a greater number of threads gives better performance. For the most of the serial algorithms, CUDA is not that useful. If the problem cannot be broken down into at least a thousand threads then using CUDA has no overall advantage.

### Basic Units of CUDA

CUDA Architecture comprises of three basic parts, which help the programmer to effectively utilize the full computational capability of the graphics card on the system. CUDA Architecture splits the device into grids, blocks and threads in a hierarchical structure as below image:

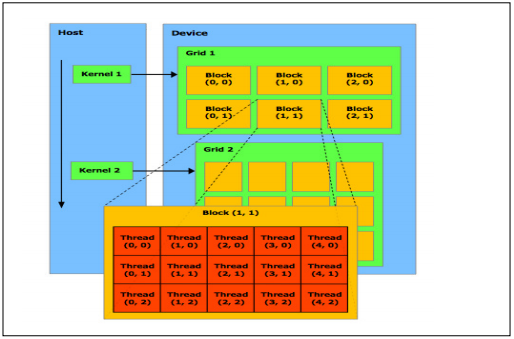


Figure 5 Basic unit of CUDA architecture

Since there are a number of threads in one block and a number of blocks in one grid and a number of grids in one GPU, the parallelism that is achieved using such a hierarchical architecture is immense.

#### 3.1 The Grid

A grid is a group of threads all running the same kernel. These threads are not synchronized. Every call to CUDA from CPU is made through one grid. Starting a grid on CPU is a synchronous operation but multiple grids can run at once. On multi-GPU systems, grids cannot be shared between GPUs because they use several grids for maximum efficiency.

#### 3.2 The Block

Grids are composed of blocks. Each block is a logical unit containing a number of coordinating threads, a certain amount of shared memory. Just as grids are not shared between GPUs, blocks are not shared between multiprocessors. All blocks in a grid use the same program. A built-in variable "blockIdx" can be used to identify the current block. Block IDs can be 1D or 2D (based on grid dimension). Usually there are 65,535 blocks in a GPU.

#### 3.3 The Thread

Blocks are composed of threads. Threads are run on the individual cores of the multiprocessors, but unlike grids and blocks, they are not restricted to a single core. Like blocks, each thread has an ID (threadIdx). Thread IDs can be 1D, 2D or 3D (based on block dimension). The thread id is relative to the block it is in. Threads have a certain amount of register memory. Usually there can be 512 threads per block.

### CUDA Memory types

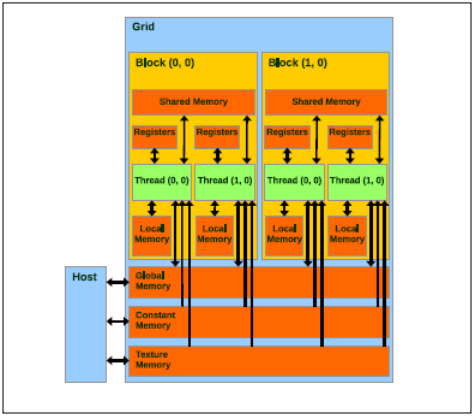


Figure 6 CUDA memory type structure

#### 4.1 Global memory

It is a read and write memory. It is slow & un-cached and requires sequential & aligned 16 byte reads and writes to be fast (coalesced read/write).

#### 4.2 Texture memory

It is a read only memory. Its cache optimized for 2D spatial access pattern.

#### 4.3 Constant memory

This is where constants and kernel arguments are stored. It is slow, but with cache.

#### 4.4 Shared memory

All threads in a block can use shared memory for read or write operations. It is common for all threads in a block and its size is smaller than global memory. The number of threads that can be executed simultaneously in a block is determined the shared memory that is specified and it denotes the occupancy of that block

#### 4.5 Local memory

It is generally used for whatever does not fit into registers. It is slow & uncached, but allows automatic coalesced reads and writes.

#### 4.6 Registers

This is likely the fastest memory available. One set of register memory I given to each thread and it uses them for fast storage and retrieval of data like counters, which are frequently used by a thread.

## GPU System Application

### GPU as an accelerated device (CPU-GPU combination)

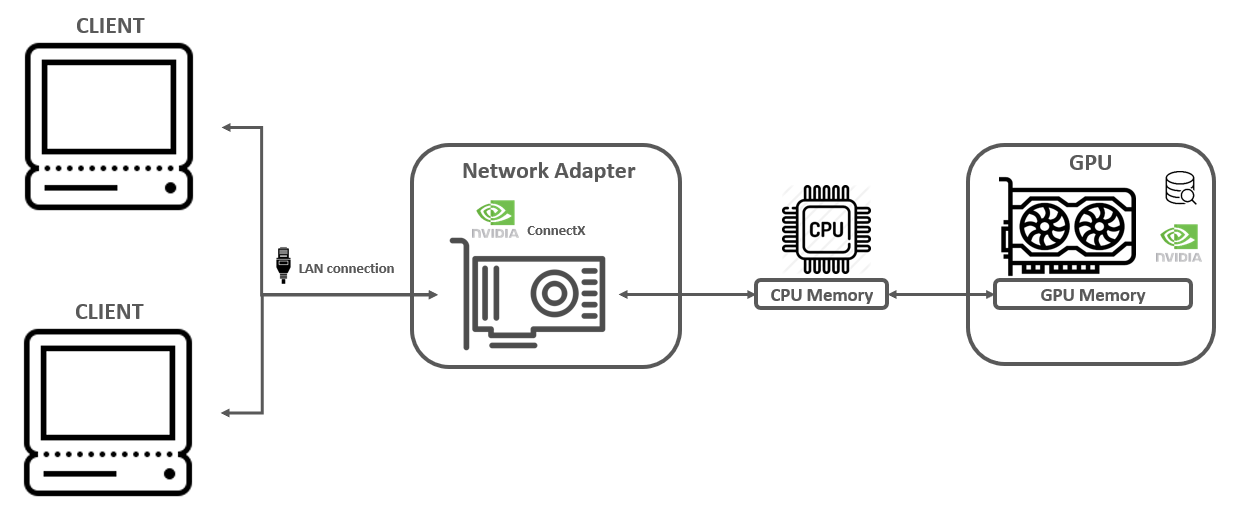


Figure 7 GPU as an accelerated device structure

### GPU as a control server (only GPU)

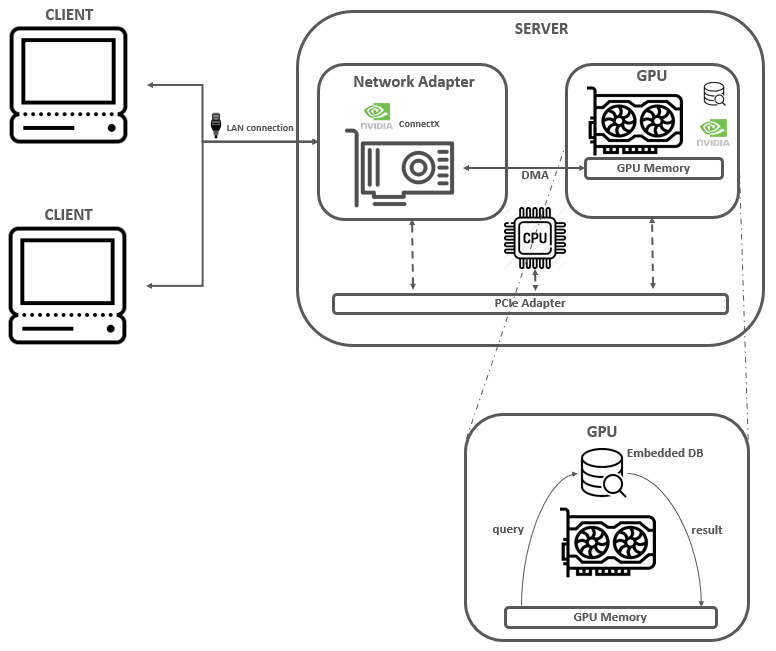


Figure 8 GPU as a control server

# Embedded Database on GPU

## Current Embedded Database Engine

### OmniSciDB

(Formerly named '[MapD](https://docs.omnisci.com/v3.2.0/)')

Official website: <https://www.omnisci.com/platform/omniscidb>

#### Features

**Open-Source Code**

OmniSciDB is an open-source SQL engine and [available on GitHub](https://github.com/mapd/mapd-core) under the Apache 2.0 license.

**APIs**

CLI (via omnisql), Java (via JDBC), C/C++ (via ODBC), Thrift, Python (via pymapd), VGA, R (via RJDBC).

Refs: <https://docs-new.omnisci.com/apis-and-interfaces>

**Advanced Memory Management**

OmniSciDB keeps hot data in GPU memory for the fastest access possible. Other [GPU database](https://www.omnisci.com/technical-glossary/gpu-database) systems store the data in CPU memory, only moving it to GPU at query time, trading the gains they receive from GPU parallelism with transfer overheads over the PCIe bus.

OmniSciDB avoids this inefficiency by **caching recently touched data in High Bandwidth Memory on the GPU**, which offers up to 10x the bandwidth of CPU DRAM and far lower latency.

OmniSciDB is also designed to exploit efficient inter-GPU communication infrastructure such as NVIDIA NVLink when available.

**Native SQL Engine**

OmniSciDB **natively supports industry-standard SQL**. Thus, users can reuse their existing SQL querying data.

Besides, it can operate as a standalone SQL engine using the command line tool [mapdql](https://docs.omnisci.com/v4.1.0/3_mapdql.html), or the SQL editor (which is part of the OmniSci Immerse visual analytics interface).

**JIT Query Compilation**

OmniSciDB takes advantage of **the JIT (Just-In-Time) compilation framework** built on LLVM (Low-level Virtual Machine). By pre-generating compiled code for the query, it avoids many memory bandwidth and cache-space inefficiencies of traditional VMs or transpiler approaches.

Using LLVM, compilation times are much quicker – generally under 30 milliseconds for entirely new SQL queries. Furthermore, the system can cache templated versions of compiled query plans for reuse. This is important in situations where users are leveraging OmniSci Immerse to cross-filter billions of rows over multiple correlated visualizations.

**Hybrid Execution**

OmniSciDB can be run on **hybrid CPU/GPU systems**, as well as on **CPU-only systems** featuring X86, Power, and ARM (experimental support) architectures.

**Distributed Architecture**

**When a query is launched, each GPU processes a slice of data independently from other GPUs**. Even though multiple GPUs reside within a single machine, the data is fanned out from CPU to multiple GPUs and then gathered back together onto the CPU.

A distributed architecture also provides faster data load times. Import times speed up linearly with the number of nodes because loading can be done concurrently across multiple nodes. Reads from disk also benefit from similar acceleration in a scale-out configuration.

#### Downloads and Installation

#### Requirements

* OS: Linux (Ubuntu or CentOS)
* GPU:

#### Installation Methods

One of following ways:

##### From Pre-Built Binaries

Guide: <https://docs.omnisci.com/installation-and-configuration/installation>

##### From Source Code

Guide: <https://omnisci.github.io/omniscidb/>

#### Running

**1. Start OmniSci server**

$ sudo systemctl start omnisci\_server

# Another way:

# $ sudo ./opt/omnisci/startomnisci

The unity omnisci's commands are documented [here](https://docs-new.omnisci.com/apis-and-interfaces/omnisql).

**2. Connect to DB**

To connect to the default DB 'omnisci', run:

/opt/omnisci/bin/omnisql omnisci

password: ••••••••••••••••

Note:

* The default username is "admin" and password is "HyperInteractive". More [details](https://docs.omnisci.com/v5.1.1/5_usersanddatabases.html).
* The defaul TCP ports are 6274 and 6278.

**3. Test connection**

Run any valid SQL query. If there is valid return, the connection is established successfully.

For example:

omnisql> SELECT origin\_city AS "Origin", dest\_city AS "Destination", AVG(airtime) AS "Average Airtime" FROM flights\_2008\_10k WHERE distance < 175 GROUP BY origin\_city, dest\_city;

The results should be similar to below:

Origin|Destination|Average Airtime

Austin|Houston|33.055556

Norfolk|Baltimore|36.071429

Ft. Myers|Orlando|28.666667

Orlando|Ft. Myers|32.583333

…

#### APIs - Thrift

[OmniSciDB Developer Documentation — OmniSciDB documentation](https://omnisci.github.io/omniscidb/)

OmniSciDB uses Apache's Thrift framework for all internal communication between the processes within OmniSciDB and external client communication. The full list of thrift API methods can be found in the *omnisci.thrift* file in the root of the OmniSciDB source directory.

##### Installation (Ubuntu)

###### Install Boost library

Thrift needs Boost library to work with.

Run following command:

$ sudo apt-get install libboost-all-dev

# To make sure Boost is installed, run:

# whereis boost

# Typically it's located in the /usr/include/boost

Another way:

Guide: <https://thelinuxcluster.com/2013/05/07/compiling-and-installing-boost-1-53/>

Download Boost src 1.63.0: <https://boostorg.jfrog.io/artifactory/main/release/1.63.0/source/>

###### Install Other Packages

Thrift needs these packages to work with.

Install runtime libraries for lex and yacc

$ sudo apt-get install -y bison flex

Install libtool:

$ sudo apt-get install -y libtool

Install libssl-dev:

$ sudo apt-get install -y libssl-dev

Install pkg-config autoconf macros (pkg.m4):

$ sudo apt-get install -y pkg-config

###### Thrift

Clone GitHub src: <https://github.com/apache/thrift>

Build and install Thrift: <https://thrift.apache.org/docs/BuildingFromSource.html>

Note: The whold building process can takes about 1-2 hours.

Common errors:

1. While building src with the "make" command, Thrift might not find Boost static libs and following issue occurs:

Cannot find libboost\_unit\_test\_framework.a, libboost\_system.a, libboost\_thread.a, libboost\_filesystem.a, libboost\_chrono.a, etc.

Solution: Create symbolic links to help Thrift finds above libs.

<https://stackoverflow.com/a/42394281>

<https://stackoverflow.com/a/11803653>

##### Generate OmnisciDB C++ Code from Thrift

In OmnisciDB top dir, run:

$ thrift -r --gen cpp omnisci.thrift

This will create a folder named "gen-cpp" with following files:

common\_types.cpp

completion\_hints\_types.h

OmniSci.cpp

omnisci\_types.cpp

serialized\_result\_set\_types.h

common\_types.h

extension\_functions\_types.cpp

OmniSci.h

omnisci\_types.h

completion\_hints\_types.cpp

extension\_functions\_types.h

OmniSci\_server.skeleton.cpp

serialized\_result\_set\_types.cpp

Now, you can write and build C++ code for your OmnisciDB project with Thrift.

Ref: thrift\lib\cpp\README.md

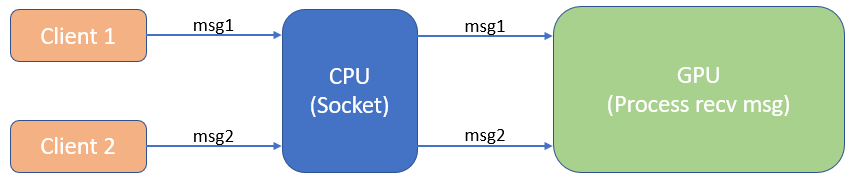
##### C++ Code to Query Data in OmnisciDB

## Comparison

# Demonstration

## Demonstration overview

## Basic Design



## Measure performance

## Evaluation result

# Conclusion

# Reference