Université Libre de Bruxelles

Advanced Databases

Project

Perst embedded database

Author:
Pierre-Alexandre Bourdais
Sofiane Madrane

Supervisor:
Prof. Esteban Zimanyi

December 19, 2017

Contents

1 Introduction

Nowadays, the usefulness of databases in any computer application is well established. But for some complex projects, the structuring of data becomes more complex. It is often necessary to store complex data and access it as quickly as possible. Thus, this application project is to prove the limitation of a relational database in the world of the embedded, in the case mobile applications, compared to the object-oriented model. This is what will be demonstrated during this document using SqLite and Perst.

2 Object Database

A Database Management System is a tool for managing a collection of interdependent data, shared among multiple users or applications, persistently stored and independently of the programs that use them.

A database management system must perform three main functions:

- A description function to describe the data or entities handled.
- A manipulation function that must allow users (or application programs) to create, retrieve, modify or delete data in the database.
- A control function that must guarantee the integrity, security and confidentiality of the data in the database.

NB: I would talk about the ODMG standard in the general sense of the term. Because Perst does not implement it.

2.1 Relational DBMS issues

The relational model is very successful and is very suitable for traditional database applications (management). His model is relatively simple and his approach formally defined by normalization and algebra is effective.

But because of its poor semantics, RDBMS is much less suitable for new, more complex applications such as:

- DB of images and graphics
- Geographic Databases (GIS: Geographic Information Systems)
- Multimedia DB (sound, picture, text, etc. combined)

These new applications have different characteristics from traditional management applications and introduce new needs, including:

- More complex object structures,
- Transactions of longer duration and unsuitable for new applications,
- New types of data for storing images or large text documents
- The ability to define non-standard operations that are specific to the applications,

Moreover, the development of an application is tedious because there are two different philosophies to respect. Two worlds forced to communicate (conversions to perform - tedious for developers). ODBMS are an attempt to answer these new needs.

2.2 Oriented-Object approach

An object database first assumes the existence of an object language to represent the data stored in the database. We will study here the specific properties that this approach must present. It is a set of methodologies and tools for designing and producing structured and reusable software, by composition of independent elements. Which aims the productivity of the programmers and a means of reuse of each element. As essential concepts encapsulation, interface, inheritance, etc. These concepts are used in OO programming languages such as Java, C ++, C#, ...

All these fundamental concepts of the object-oriented are implemented in a ODBMS, making it possible to carry out processing on the data, in particular:

- Identifies an object:
  - Promotes data sharing
  - Supports typed pointers
- Encapsulation of data:
  - Allows data isolation of operations
  - Facilitate the evolution of data structures
- Structure operation inheritances:
  - Facilitates data type reuse
  - Allows programs to be customized to the needs of the application
- Polymorphism:
  - Increases developer productivity in DB implementation

This technology brings a revolutionary approach. Allowing easy access, development and maintenance. Respecting all the SI constraints of the application, ensuring logical and physical model independence, versionning and persistence of data.

2.3 Oriented-Object DBMS

2.3.1 Bases of OO DBMS

The approach of using a SQL data manipulation language to access an object database from a programming language seems quite inefficient if the language itself is also an object language. In fact, we would like to hide as much as possible the syntax differences between objects in memory and objects on disk. A regulatory standard is problematic, created by many industry leaders. The ODMG standard. But many ODBMS do not follow (including Perst).

2.3.2 Gold rules of OO DBMS

In general no matter the modeling standard. A DBMS must follow a set of rules that are inspired by OO languages:

- Identity: Any object in the database is identifiable (OID). This identification is an invariant in the life of the object. And the existence of an object is independent of its value.
- Abstraction: Ability to eliminate non-essential features of an object to create object groupings called "types" or "classes". An object is then an instance of a type or class. Abstraction also makes it possible to make dynamic links between objects.
- Encapsulation: Describe in the DBMS. A class is defined with a set of methods. The use of a method is done by sending a "message" to the DB. It will be possible to access only the services of its methods and not their implementations (access, update, manipulate).
- Reusability: An important notion is the inheritance that makes it possible to factorize between objects, to have simple, multiple, partial and selective representations of the same object
- Persistence: To have flexible data, to be able to store them. The DBMS provides tools to users. Ex: collections (SET, LIST) , Persistent Object
- Complex structure: Can define his own type of data. The SGBDO allow the extensibility of basic types. Ex: Image, Card.
- Security: The DBMS must offer confidentiality, integrity rules (trigger, constraints, etc.). And have a system of global versions.
- Non-procedural interface: Large amount of data, physical grouping techniques, indexing, query optimization, and cache management. The language also offers the power of a programming language

3 Perst - object-oriented embedded database

In December 2006, GigaSpaces Technologies has integrated McObject's Perst open source, an object-oriented all-Java embedded database for real-time

data management in its massively scaleable distributed enterprise application technology. GigaSpaces embeds Perst in version 5.1 of its software, where Perst provides persistence for applications that are deployed and optimized using GigaSpaces' highly scalable, self-managing distributed solution.

Perst's all-Java architecture was a good fit for GigaSpaces. The company's software products are developed in Java, both for maximum portability across diverse enterprise platforms, and to take advantage of advanced Java capabilities, including JavaSpaces, a simple unified mechanism for dynamic communication, coordination, and sharing of objects between Java technology-based network resources. Perst is specifically integrated within the GigaSpaces In-Memory Data Grid , as an embedded object-oriented database offered as an alternative to a relational database due to its superior performance and minimal resource (CPU cycles and memory) requirements.. "A major advantage of Perst is the efficiency provided by its ability, as an object-oriented database, to store application data as 'plain old Java objects' (POJOs), rather than requiring translation of this data to a relational format," said Guy Nirpaz, vice president for research and development of GigaSpaces. GigaSpaces implements a unique space-based architecture that incorporates aspects of grid computing and service-oriented architecture and dramatically boosts the scalability and performance of both new and existing applications. GigaSpaces' software meets the requirements of high performance, low latency and grid-based applications, and adds new possibilities and flexibility by introducing advanced application design patterns. It has been tested extensively in large financial applications with proven linear scalability and extremely high performance.

3.1 About Perst

Perst is McObject's high-performance object-oriented embedded database for Java and C#, and is tightly integrated with these programming languages. In contrast to object-relational databases, or tools that provide object-relational mapping, Perst stores data directly in Java C# objects. This eliminates the need for expensive (in performance terms) runtime conversions between representations of the data. Unlike many other object-oriented databases, Perst requires no dedicated compiler or pre-processor, yet provides a high degree of application transparency. The Perst API is convenient, flexible and easy-to-use. Perst also offers a very small footprint. The engine's core is

just 5,000 lines of code, and the run-time requires between 30K and 300K of RAM. Perst requires no end-user administration, and despite its simplicity, Perst ensures integrity via transactions that adhere to the "ACID" properties (Atomicity, Consistency, Isolation and Durability) with very fast recovery. The Perst open source software distribution also includes Perst Lite, a micro-footprint version of Perst targeting embedded systems and intelligent devices developed on the Java 2 Platform, Micro Edition (J2ME).

3.2 Perst embedded database features and benefits

Here is is a list of all the features and benefits of Perst :

- Object-oriented : Perst stores data directly in Java and .NET objects, eliminating the translation required for storage in relational and object-relational databases. This boosts run-time performance.
- Compact : Perst's core consists of only five thousand lines of code. The small footprint imposes minimal demands on system resources.
- Fast : In McObject's benchmarks, Perst displays one of its strongest features: its significant performance advantage over Java and .NET embedded database alternatives.
- Reliable : Perst supports transactions with the ACID (Atomic, Consistent, Isolated and Durable) properties, and requires no end-user administration.
- Rich in development tools : The Perst API is flexible and easy-to-use. The breadth of Perst's specialized collection classes is unparalleled. These include a classic B-Tree implementation; R-tree indexes for spatial data representation; database containers optimized for memory-only access, and much more.
- Transparent persistence : Perst is distinguished by its ease in working with Java and C# objects, and suitability for aspect-oriented programming with tools such as AspectJ and JAssist. The result is greater efficiency in coding.
- Source code available : With free, available source code, nothing in Perst is hidden, and the developer gains complete control of the application and its interaction with the database.

- Advanced capabilities : Perst's extras include garbage collection, schema evolution, a "wrapper" that provides a SQL-like database interface (SubSQL), XML import/export, database replication, support for large databases, and more.

3.3 Perst Design Principles

Perst's goal is to provide developers in Java and C# with a convenient and powerful mechanism to deal with large volumes of data. Perst's design principles include the following:

- Persistent ojects should be accessed in almost the same way as transient objects (transparent persistence)
- A database engine should be able to efficiently manage much more data than can fit in main memory
- No specialized preprocessors, enhancers, compilers, virtual machines or other tools should be required to use the database or develop applications with it.

3.4 Perst embedded database specifications

3.4.1 Supported Platforms

Supported Platforms		
Product	Platform	Language
Perst Java	J2SE 1.4 and higher J2EE 1.4 and higher Android J2ME/CDC	Java
Perst.Lite Java	J2SE 1.1 2ME MIDP 2.0/CLDC 1.1	Java
Perst.Net	.NET Framework (1.0, 2.0, 3.0, 3.5, 4.0) .NET Compact Framework (1.0, 2.0) Silverlight Windows Phone 7 (WP7) Mono	C#, J#, Managed C++, VB.NET and all other managed .NET languages

3.4.2 Persistence

Persistence	
Supported types	All primitive types Strings Arrays Enums .NET structs and embedded objects in Java GUID, decimal and DateTime types BLOBs Raw types (objects serialized using system or custom serializers) Generic (parameterized) types
Transparency	Controlled recursive loading of objects Persistence by reachability Fully transparent persistence using AOP tools such as AspectJ and JAssist Almost transparent persistence for .NET classes using generator of derived class overriding virtual properties of the class
Flexibility	User-defined class loaders User-defined memory allocators Custom serializers Custom full text search components (stemmer, parser, ...) Explicit or implicit memory allocation (garbage collection) Abstract file interface to provide specific file implementations

3.4.3 Queries

Queries	
Indexing algorithms	B-Tree T-Tree (optimized for in-memory database) R-Tree (spatial index) Patricia (prefix search) KD-Tree (multidimensional index) Time series (large number of fixed size objects with timestamp)
Collections	List Map Index (range search) Field index (extracts keys from object) Multidimensional index Set Scalable set (uses array for small set and B-Tree for large set) Spatial index Multidimensional index
Search kinds	Object-oriented using various search and iteration methods of Perst collection classes Query-by-example (including range search) implemented using multidimensional indexes JSQL - object-oriented subset of SQL Full text search: built-in full text search engine or integration with Lucene Native queries and LINQ (search predicate specified in native code)

3.4.4 Transactions

Transactions	
Implementation	Shadow objects
Features	ACID No log file Fast recovery
Locking granularity	File-level locking Database-level locking Fine grain (object-level) locking
hline Locking models	Pessimistic (resource is locked before access) Optimistic (conflicts are detected at commit stage)
Isolation levels	Cooperative transactions Repeatable reads Serializable transactions

3.4.5 Performance

Performance	
Caching	Object cache (LRU, weak, strong...) Page pool In-memory database
Performance benchmark	List Map Index (range search) Field index (extracts keys from object) Multidimensional index Set Scalable set (uses array for small set and B-Tree for large set) Spatial index Multidimensional index
Large volumes of data	Maximal number of objects: 2,000,000,000 Maximal database size: 1 terabyte
Small footprint	Library size from 250KB (Perst.Lite) to 500KB (Perst Java/.NET). Proven on mobile phones with heap size limited to less than 1MB
Scalability	High level of concurrency because of fine grain locking in pessimistic mode or use of optimistic transaction mode Can split data between several physical devices Possible to store BLOBs in separate storage locations, making object-caching more efficient
Load balancing	Master-slave replication provides read-only access to primary database by multiple replicas Access to the same database permitted from multiple processes (JVMs)

3.4.6 Reliability

Reliability	
Recovery	Automatic recovery in case of application, system or hardware failure
Data replication	Asynchronous or synchronous data replication
Backup	Online of offline backup
Data protection	Database encryption

3.4.7 Schema evolution

Schema evolution	
Change scalar field type	Automatic
Add/remove field	Automatic
Move/rename fields	By means of XML export/import
Custom transformations	Load/store object handles, database version information

3.4.8 Internationalization

Internationalization	
Default string encoding	UTF-16
Explicit specification of encoding	Available
Custom comparator	Many Perst collections allow user to specify a comparator class

3.4.9 Advanced features

Advanced features	
XML import/export	Available
Database encryption	Available
Database compression	Available
Portable database format	Available
Multiversioning	Available
Full text search	Available
Fast database upload on mobile devices	Available

4 Perst Vs SQLite

4.1 Application description

The application is a very simple benchmark measuring performance of basic database operation: inserting, searching and deleting records. It uses simple records with two primary key columns: one of 8-byte integer type and another of string type. These columns are assigned random values during database initialization.

4.1.1 Benchmark

The benchmark consists of four steps:

- Insert data into the database.
- Perform index searches for all objects using both indexes.
- Iterate through all objects using index iterators.(sequential scans).
- Locate and remove all objects one-by-one.

4.1.2 Structure

To allow comparison of Perst's performance with that of the built-in SQLite database system, this application includes a port of the benchmark for SQLite. The structure of the application is the following:

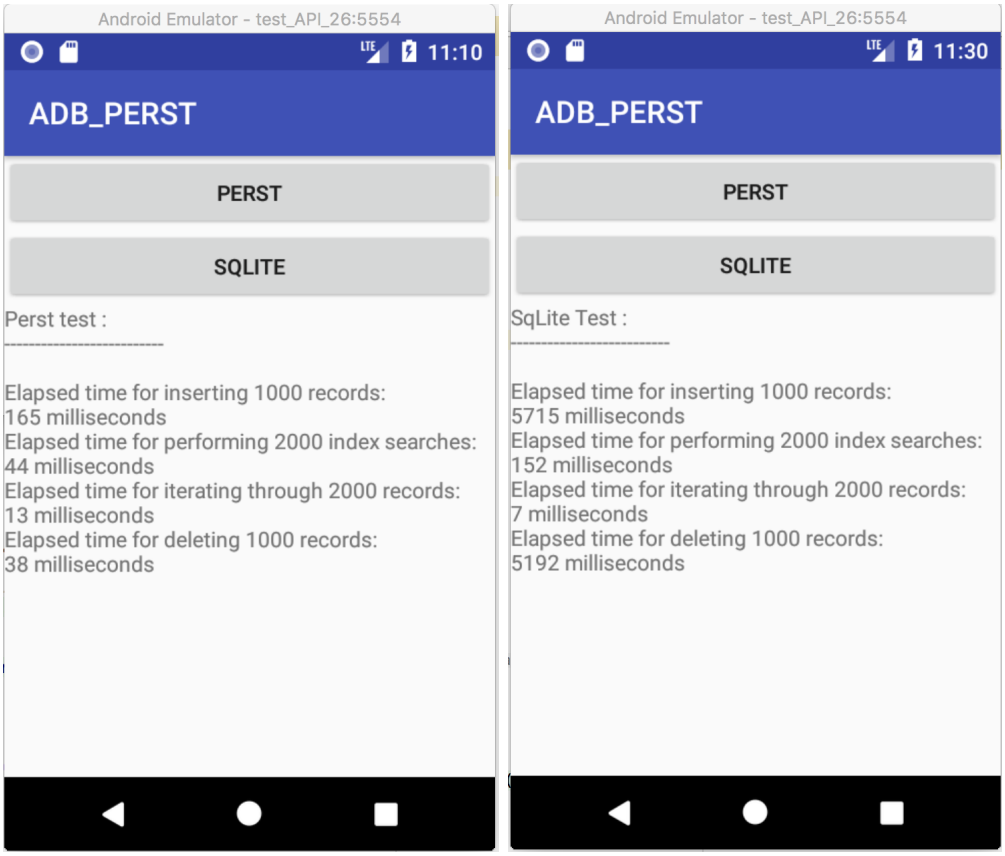
- Main Activity : makes possible to user to start Perst or SQLite benchmark.
- PerstTest : implementation of the benchmark for Perst.
- Progress Monitor : allow to print the test result on UI during test execution.
- SQliteTest : implementation of the benchmark for SQlite.
- Test : base class for all tests

4.2 Environment

- Android Studio allows to use Sqlite because it is included in Android's SDK.
- Android Smartphone running Android 8.0 Oreo
- SDK 26 Google
- API 26 Android Studio
- Perst 4.39

4.3 Monitor

In order to print out the result, a monitor is emulate by android studio. After running Perst and SQlite tests the monitor is like the following figures. All the result are in millisecond.



4.4 Results

4.4.1 Benchmark

In order to compare SQlite and Perst, it is need to run a few benchmark with different values of the number of object. The number of objects will be 1000, 10 000, 100 000.
Moreover for each of these values the benchmark will be run three benchmark in order to get different values and accurate results.

Firstly the number of object is set to 1000. The results are stock in the following table.

	Insert	Search Index	Scan	Locate and remove
Perst				
1st	165 (ms)	44	13	38
2nd	76	4	2	25
3rd	38	4	1	15
SQlite				
1st	5715	152	7	5192
2nd	4644	69	6	4045
3rd	4585	56	2	4093

According to those result it seems that the first running for Perst and SQlite are taking more time than the second and third one. So the conclusion will be drawing using the 2nd and 3rd running of Perst and SQlite test.

Perst is much more faster in order to perform insertion, also it is locating and removing each object of the DB one by one faster than SQlite. The index searches for all objects using both indices is a little bite faster for Perst. However the iteration through all objects using index iterators is almost as fast for Perst and SQlite.

Secondly the number of object is set to 10000. The results are stock in the following table.

	Insert	Search Index	Scan	Locate and remove
Perst				
1st	685 (ms)	177	50	227
2nd	304	103	37	193
3rd	174	79	32	141
SQlite				
1st	68004 (ms)	383	44	53284
2nd	31965	189	24	30607
3rd	31268	186	24	30786

According to those result it seems that the first running for Perst and SQlite are taking more time than the second and third one. So the conclusion will be drawing using the 2nd and 3rd running of Perst and SQlite test.

Once again Perst is much more faster in order to perform insertion, also it is locating and removing each object of the DB one by one faster than SQlite. Moreover the index searches for all objects using both indices is again a little bite faster for Perst. However the iteration through all objects using index iterators is this time faster for SQlite, but the results are quite close.

Thirdly the number of object is set to 100000. The results are stock in the following table.

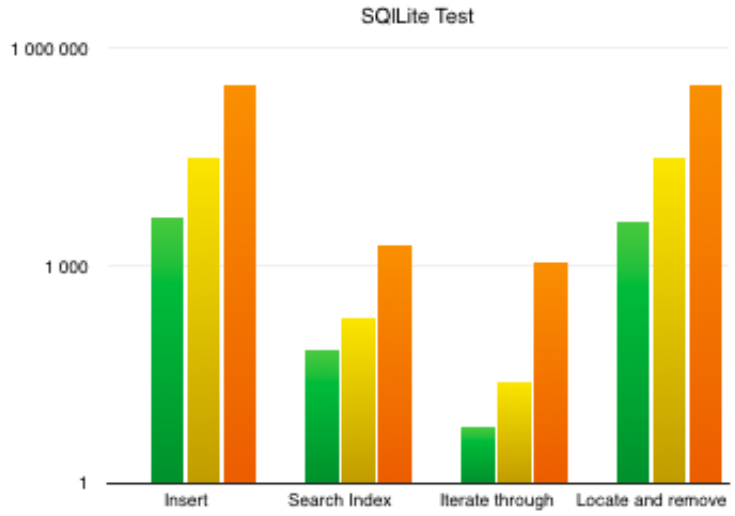
	Insert	Search Index	Scan	Locate and remove
Perst				
1st	7307	3804	3454	5841
2nd	6756	4025	3520	5834
3rd	6464	3962	3538	6320
SQlite				
1st	313885	2105	1132	301075
2nd	308158	1869	1087	308982
3rd	303026	1888	1107	313516

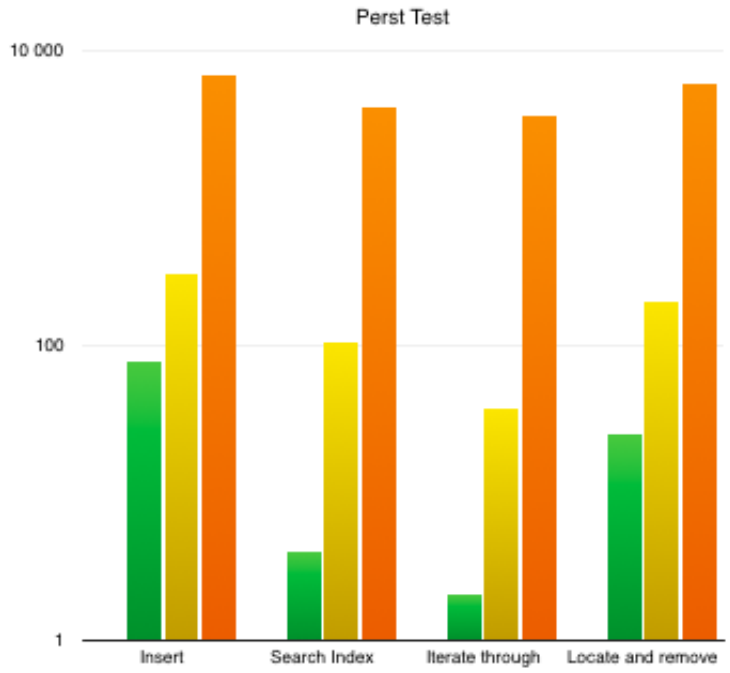
As expected Perst is once more much faster in order to perform insertion, locating and removing. This time the index searches for all objects using both indices is faster for SQlites, it is the same thing for the interation through all objects using index iterators.

4.4.2 Chart

Now that all the benchmark are ran, a column chart can be created. This will reveals the increasing of the running time for each operations while increasing the number of objects.

The two following figures are using logarithm scale in order to get a better overview. Green color correspond to the test with 1000 objects, yellow to 10 000 objects, orange 100 000 objects.





As we expected more there is object in the DB more it takes times to perform the operations over the data.

Regarding to those figures it appears that Perst is actually way better with small DB. For SQlite the running time seems to increase proportionally while increasing the number of objects.

5 Discussion

What accounts for the performance disparity? For SQlite insert and delete operations, one obvious gating factor is the lack of explicit transaction support in its Android API. Each update must be performed as a separate transaction, in autocommit mode, resulting in significant transaction processing overhead.
This also compromises the ability to maintain the logical consistency of the database contents (i.e. to define a database unit of work that consists of

updates to two or more table rows in the database).

Search time for SQlite is slower and it is explained by overhead added by using Android's Java interface to access the native C language database, and the overhead of parsing, optimizing and executing the interpreted SQL. In contrast, Perst's interface works directly with database objects no interpretation of an intermediate language is needed.

SQlite's advantage in scan operations probably stems from the test's simple tabular data layout. As a relational database, SQlite organizes data in rows, and these rows are physically close to one another in storage, like rows of a spreadsheet. This proximity lends an edge in sequentially fetching the rows. In contrast, in Perst, everything is an object, including index pages, and objects are interleaved in the storage. The following factor also affect these databases' performance and their "fit" with mobile applications, and is worth considering:

Object-oriented vs. relational database system. Choice of database model is often viewed as hinging on the user's programming philosophy or style. But developers targeting resource-constrained devices should bear in mind the run-time efficiency gained by pairing an object-oriented database with an object-oriented language such as Java. In Java, developers work with Java objects; an object-oriented database stores these as Java objects, eliminating the translation required for storage in a relational or object-relational database.

This boosts run-time performance. It also permits the developer to stay within the O-O paradigm, whereas use of a relational database with an SQL interface requires shifting back and forth between O-O and a set-based, declarative query language.

6 Conclusion

In order to conclude this project it's needed to draw a conclusion according to the content of the report and the test results. Perst is a specific ODBMS as it is not following the ODMG standard.
Perst is definitely more easy to use than SQLite because it is using the O-O paradigm. So there is no need to wrapped the object we are working with. Perst is definitely more efficient than SQLite regarding to the results of the

benchmark.

Mobile devices such as smartphones manage increasingly large data sets, to meet widely varying application needs. Enterprise and business-oriented database systems typically outstrip such devices' CPU and memory resources. However, several small-footprint embedded DBMS's have emerged, providing developers with the luxury of choosing between data models, APIs, index types and other database features for applications that will run on Android, BlackBerry, Windows Mobile, Symbian and other mobile devices. With this choice comes the ability to optimize data management based on application function and performance needs.

7 References

http://www.mcobject.com
https://developer.android.com
http://www.drdobbs.com/database/kernel-mode-databases/207401567
https://en.wikipedia.org/wiki/Database_transaction
https://en.wikipedia.org/wiki/Object_database
https://www.thoughtco.com/the-acid-model-1019731
https://en.wikipedia.org/wiki/Embedded_database
https://en.wikipedia.org/wiki/Object_Data_Management_Group
http://www.odbms.org/introduction-to-odbms/definition/