STARCOUNTER

STARCOUNTER VMDBMS TECHNOLOGY

PATENTED TECHNOLOGY

Background

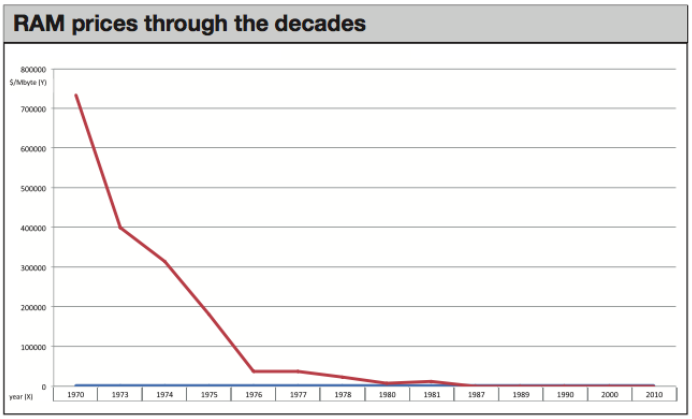
New generations of enterprise databases and other data engines will play an important part of our future. The fundamental principles of database technology have roots from the early 1970s. This is good and bad. The good thing is that database technology has become a science and is well researched mathematically. The bad thing is that many of the underlying assumptions are no longer true.

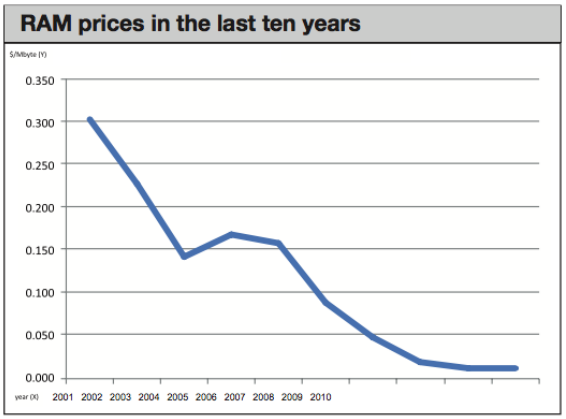
The ongoing RAM revolution

Databases use computer memory to store data. Historically, memory has been divided into primary memory and secondary memory. Primary memory uses RAM. The main media for secondary memory over the last 40 years have been the disk drive. Accessing data in RAM is thousands of times faster than accessing data on disk. The main reason databases have traditionally focused on being efficient using secondary memory is often believed to be that disks are persistent, whereas RAM loses its memory when power is cut. But as disks can also fail, thus multiple redundant storage are needed anyway. The main reason why RAM is not the primary operating medium of databases has always been cost and availability. When the SQL database was born, one megabyte of RAM was over 160,000 US dollars. Now the same amount RAM is less than 1 cent. Securing data to redundant storage can be done efficiently, even using old mechanical disks (since sequential writes do not require the mechanical arm of the disk drive to move, multiple disks can be written to in parallel). Securing data to disk can be done faster than the rate of change in a RAM database.

The first fix - big database caches

What do you do as a database vendor when RAM is suddenly cheap and a 64-bit CPU can address it in terabytes? As you cannot reinvent and rewrite your product in an instant, the easiest option is to increase the size of your caches. Today, many traditional databases operate entirely in RAM if enough of it is available. While speed is much faster than if the RAM was not present, the entire architecture of the DBMS has not changed. The overhead that used to consume a fraction of the execution time when the cache was small now consumes almost all the time.





The first generation RAM databases

While increasing the cache speeds things up, it will not match the speed of a true RAM database. If you adapt your database engine and assume that the data is always in RAM, you can make a database engine much faster than a traditional database. A RAM database is typically 10 times faster than a big-cache database.

The second generation RAM databases

The modern computer is nowadays a complex cluster of CPUs and memory. Taking advantage of the NUMA architectures where all memory is not equal can radically improve the performance of the first generation RAM databases. A CPU cache is a hundred times faster than normal RAM memory. Taking advantage of this and fully exploring the physical differences between RAM and Disk, a second generation RAM database can be 10 times faster than a first generation database and a hundred times faster than a big-cache database.

The third generation RAM database

If a second generation RAM database can be a hundred to a thousand times faster than a traditional database, a new opportunity opens up. If the database is as fast as your object heap in your computer language, why would you move data between the database and the heap when you are operating on your objects, rows or tuples? When we measure database operations in nanoseconds instead of milliseconds, this is a valid question to ask. One reason why you would like to move data from the database to the object heap of languages such as Java or C# is that you have a snapshot copy of the data. Nobody will see your changes until you decide to save them or serialize them. The downside is that query languages such as

SQL will not be able to sort or query your local changes together with the database data until they are saved, making coding on your business objects more difficult. But what if your computer language used the database as the heap? Transaction isolation would make the changes invisible to others until you want to make them public, this is the A and I in ACID atomicity and isolation. You keep the effect so that you can work on a transaction isolated from other transactions, but you gain the possibility to see your changes in your SQL queries. If you are adding a line item to a purchase order with existing items and have not yet saved your changes, you can still use SQL to sort all your rows, no matter if they are new and old. While in an atomic change, different threads of Java or C# object code would see the same object in different states at the same time.
The need for moving data back and forth using serialization and deserialization would be redundant. The solution would be super easy for the developer, and much faster than the second generation RAM databases. Such a merger between the object heap of the virtual machine (VM) and the transactional control by the database management system (DBMS) we will refer to as a VMDBMS.

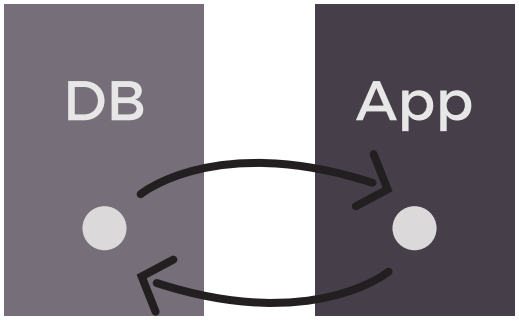
The VMDBMS defined

The bottleneck apparent in second generation RAM databases is the transferring of data between database and application code. Because data is physically stored only in one place, the VMDBMS eliminates data transfers between application and the database as well as transformation between different data formats. This is possible because the application can access data in the database as fast as its internal temporary data.

The data is never moved, the data resides in the database all the time, and the application directly accesses the data managed by the database management system. There is no copy whatsoever of the data local to the application. The “local” state of change in progress is no longer a product of a separate Java or C# object on the heap, but rather a consequence of the isolation and atomicity of the ACID engine managing the joint heap/database. As long as the log(s) are secured on disk, the database image can calmly be hibernated to disk in its own pace using asynchronous writes. Check points and recovery works the same way as in your legacy database.

```
[Database]
public class Person
{
    public string Name { get; set; }
}

[Database]
public class Quote
{
    public Person Author { get; set; }
    public string Message { get; set; }
}

class Program
{
    static void Main()
    {
        Db.Transact(() =>
        {
            var einstein = new Person() { Name = "Einstein" };
            new Quote()
            {
                Author = einstein,
                Message = "Make things as simple as possible, " +
                      "but not simpler"
            };
        });

        var quotes = Db.SQL<Quote>(
            "SELECT q FROM Quote q WHERE q.Author.Name = ?",
            "Einstein");

        Console.Write(quotes.First().Message);
    }
}
```

Optimistic concurrency control does not compromise of concurrency control, but it benefits from the fact that if conflicts are unlikely, it is better to deal with them only if they occur. This is difficult to do if the code running the transaction is separate from the database engine. In the VMDBMs the code can declare a transaction scope (see above) and the transaction can automatically be restarted. In a third generation database, transaction running time is around a thousand times faster than using Java or C# on a remote database. This means that pessimistic concurrency control, which is based on locking and slows the database down even if there is no conflict can be avoided.



A bad performance spiral

The fuel of a space rocket is burned to produce thrust to accelerate mass of the rocket out of the gravitational force of the Earth. The more energy that is needed, the more fuel it takes, and the more fuel it takes, the more energy is needed. This means that you are put in a bad performance spiral. Transactions can be said to work in a similar way, the slower the transaction, the more you need to rely on pessimistic concurrency control (locking) as conflicts are more likely. The more the engine relies on pessimistic, the slower the transactions, etc.

A good performance spiral

If the database is fast enough, it can be used as a plug in heap manager, meaning that the data of the object is not copied to the memory of Java or C#. This means that transactions become much faster and reduces concurrency conflicts. Also, as the language and the DBMS can cooperate, the DBMS can automatically restart transactions, such as lock free optimistic concurrency schemes can be used. This in turn also means faster transactions.
The faster the transaction, the less likely the conflict. The less likely the conflict, the more you can rely on optimistic concurrency control, meaning faster transactions. This puts you in a good self feeding performance spiral.

VMDBMS queries

A VMDBMS uses an enterprise database engine as a heap for languages such as Java and C#. As such, you can expect to be able to run queries on your objects. Your class is a table and your class instances are your rows. Inheritance work in the way you expect them to, as opposed to environments where the database is remote where your queries will have very little overhead and you can run millions of them per second on a single node. Also inheritance and path expressions (person.City.Country.Name) can be natively supported instead of having to be translated to a relational database resulting in poor performance.

Lock free ACID concurrency

As the VM and DBMS are merged, the database transaction can automatically restart a transaction that is in conflict with another transaction. This is known as optimistic concurrency control.

A billion objects

There is much more to a VMDBMS than to provide persistent objects. The normal Java or C# heap lacks the capabilities to host large number of objects, already when they reach the millions, the memory manager and garbage collector begins to struggle. There is no transaction scope, so that changes are not atomic, and transactions are not isolated. Your threads need to rely on locking to be thread safe, a method that is significantly slower than optimistic concurrence control. One of the most important features however, is the possibility to query your database using query languages such as SQL, and that you have the safety features of checkpoints, or recoveries and other enterprise database features.

Where should the VMDBMS be used?

The availability of memory has increased dramatically. This can somewhat be offset by the increase in information volumes. However, when you compare the number of people, products, and sales transactions on the planet, and given the fact that the name 'Bill' still requires four character of storage, for many information systems the availability of cheap computer memory outweighs the growth in entities to store information. It is rather the real time and online exposure of the information that has changed dramatically in terms of structured storage. Unstructured information that is not transactional has grown much more rapidly with the birth of the internet than compared to the structured data involved in ACID transactions. The VMDBMS is ideal for applications such as ERP systems, online applications, finance, and other areas were structure and transactions are needed.

Einstein's general theory of relativity

Third generation RAM database entails a database access measured in nanoseconds. The implication is that physical distance between interacting nodes becomes very important. It takes the light three nanoseconds to travel one meter, and ss Einstein has showed, this is an absolute limit for electricity based or light based computers. By writing your software, be it an ERP system or a cloud service, the VMDBMS makes sure that the speed of light works in your favor.