* **Database As A Service**
* **Introduction on technology**

1. **The original VM cloud**

In the past, normally all the enterprises, universities and organizations will have their databases running on their dedicated hardware, this situation is well suited for the past because the limitation of network bandwidth, as well as for many other concerns(e.g. privacy concerns, accessibility etc.). And yet, in the past several years we saw the great improvement on network speed, virtualization technologies, and the technologies for cloud server are becoming more and more mature, big organizations are tempted to put the all their different databases into different VM, and in this way, one server will hosts many separated databases, and one DB administrator can then monitor several databases. Also, in this way, the usability of hardware improved dramatically, yet this architecture can only cater for big enterprises, and maybe wasteful for some small organizations to host such VM shared machine.

Not long ago, we witnessed the significant advancing on Cloud technologies and the enormous economic potential both for the cloud server provider and the service client themselves. The first kind of technologies immediately followed by the great advancing on virtualization technologies, of course, was the VM-cloud. In this architecture, each client's database or application are installed within a VM machine, the cloud server will provide a powerful server to host many VM from different clients, the client then no longer needed to concern the hardware, they just give their VM to the cloud service provider, and sign a contract with the cloud server provider, like "any request needed to be responded with 1 second" etc. In this way, they no longer own the physical hardware to run the database or their applications, and they don't need to hire expert database administrator to tune their database, their contract explains all, then the cloud provider provide the desire the service.

1. **Emergence of process sharing and Database as a Service**

Yet the VM-cloud suffer from a lot of limitations, one clear defeat is that VM is a hard sharing of the hardware, meaning that it would be hard to divide a VM, and so one VM must be placed in one node of the cloud server, this will limit a lot of the potential concurrency ability of cloud sharing. Also, in order to use VM to host a database, each VM will need to install an OS and DBMS, therefore wasting a lot of unneeded space. Under these premises, a new cloud service is born, in this technology, the cloud service provider contains many nodes, each node hosts a DBMS, and one DBMS provide service for many clients' databases. This design has a major difference to the VM cloud method in that a client's database can be easily be partitioned, and so the pieces of a client's database can be located on several nodes on the cloud server, and so providing some potential concurrent processing ability. As before, in this design, the client also need not worrying about the physical devices and hiring a database administrator, which can be significantly more economically beneficial.

Yet another implementation of cloud sharing is for client to share a common table, but this method abandon the flexibility requirement of client databases, what if one client decide to add one column to one of the tables shared? This is a common problem, and so this implementation is never realized.

1. **Difficulties in implementing Database as a Service**

The method that different client databases sharing one DBMS in one of the many nodes in the cloud service provider, is the implementation called "Database As A Service"(DBaaS). However, in order for this service to be popular and be accept by many clients, there are many difficulties needed to be tackled.

One obvious problem is how do we provide privacy for the client? Because the client let their database run on the cloud service, some method has to be implemented in order to convince the clients that the administrator of the cloud server(the hyper supervisor) will not be able to peek into the data within their clients' databases, and also, one client's databases is entirely separated from other clients' databases. The last problem is harden because now client's database has been partitioned, and are sharing the same DBMS, without carefully design of the system, other client's data might be exposed, leading to privacy problem. As suggested in [1], one obvious solution would be using encryption on data, and the DBMS is require to do SQL query processing on encrypted data, in this way, not only other client sharing the same DBMS cannot know the data(since they did not own the keys needed to decrypt the data), but also the hyper supervisor, the database administrator on the cloud server cannot understand the client's data. Yet this implementation ensues one immediate question, that is, how do we encrypt the data in order to let the database to do query processing on these encrypted data, also, we need to do the processing on the encrypted data in an efficient way, otherwise the benefit of sharing DBMS in the cloud would be entirely ruined. Under the suggestions in [1], it has been showed that using homomorphic encryption(HOM) can enable the DBMS to perform operations on encrypted data. Also, the DBMS might employ some kind of an "onion" design on the decrypted data, that is, for each row, we use different keys to encrypt the data from several times, each layer of encryption will protect data from performing some kind of operation(for example, the outer most layer may be protecting from basic accessing of the data, then the second layer protecting update and delete, then the last layer protecting joins), only the client will have the keys for decrypting in each layer of the onion, and the client knows what kind of operations they are performing, and so, the client send only the necessary keys to the server to enable the server to do the necessary steps. This design is good in that the DBMS do not knows the keys of the encrypted data, and just doing operations on encrypted data, and different "layer" of operations are all protected and can be control by the client, providing a feasible privacy.

Another problem is that how do we partition the database, this question might not be first encountered in the database world, because even in the old days when the database are only located on the client's own dedicated hardware, sometime the database is so large that cannot be hosted within machine, and when that happens, the database administrator will need to consider how to partition the database and locate the pieces on different machine. But the problem occurs in DBaaS world is much more trickier, that is because the databases are handled automatically, and when in the old days the database administrator of their company's knows well how to partition the database will be optimal, now in the cloud, the hyper supervisor who are tuning all these databases without the ability to know any transaction details about it, will be especially difficult to know the best way to partition the database. But when one node in the cloud server cannot host the entire database, this database must be partitioned. And also, since we are using many nodes anyway, we want a way to partition the database to make some kind of queries can be fulfilled by different nodes, and so getting the concurrent benefit of the cloud, in some cases, these concurrent ability might let the query processing even faster than using dedicated hardware, since the dedicated hardware is unlikely to consist of many nodes and providing good concurrency. On the other hand, researches, as in [1], has shown that it is unwise to partition the database in a way that many transaction will span to many nodes, that is, the data that are frequently being request together should not be placed on different places, the reason is that the overhead in order to combine the data and locking on different nodes are a major bottleneck. Having this observation, [1] suggested a way to partition the database, they implemented a thread to periodically logging the set of records that are accessed together within the same operation or transaction, and they use a graph partition method to partition the database as follow: whenever data are being accessing from different nodes of the cloud server within one transaction, and edge is created, and a weight of the edge will be generated, basically representing the frequency of them being accessing together, and so, they will have a graph generated representing the potential wrong partition of a database. According to the graph, they run a graph partition algorithm on it, and alarms the system when some frequently fetch data is separated on different nodes. This ways on partition is innovated in that they no longer require the premise that the database administrator to know the database well, such as foreign key constraint and the nature of the transaction performed, and are well suited on cloud server because of its automatic partition ability.

1. **Methods for partitioning the databases and deciding the good packing of a node**

Yet another problem concerns how do we relocate the pieces of databases and how do we know a "packing" of pieces of databases from different clients can be efficiently working together within a node. At the first part of this problem, assumes now we already know that in some node the packing has exceeded the capability of a single node, and we must perform the operation of migrating a piece of one client's database to another node, the main concerns here is that, we need a method to migrate the piece of database to another place without any downtime, and possibly reduce the increased latency during the migration as much as we can. Although the cloud server's nodes and usually connected with very high bandwidth network cable, given that the size of the migration piece might be large(from the discussion on the previous problem we know that we might migrate a large piece of database if all the data inside it are frequently accessed within a single transaction), the migration operation takes at least several minutes to hours, therefore we must strike for a way to do the migration "on-the-fly", but then because transferring a large piece of data will undoubtedly consume some or considerable available network bandwidth of each node, and so migration operation done in a not careful fashion will degrade the whole system's performance. And so, when we need to decide which piece of data need to be migrated, we must also take into the account of the effect it has on the whole system, and see whether it is worth to migrate at the expense of the degrade of the system's performance.

The other part of this problem is to decide whether a "packing" of pieces of clients' databases is appropriate(i.e. not exceeding the node's capacity). Let's divide this problem into two part, that is, knowing the capacity of each node, and another part is obviously, knowing the consumption of different resource of each located database piece. The former one seems obvious, the hyper supervisor will normally know a node's RAM size, CPU rate, network bandwidth and disk size etc. But we need to be more careful about this, could these physical parameter encapsulate one node's total capacity? Will there be any other concerning parameter that is not known until the run-time, and therefore blindfolding the hyper supervisor? In fact, and buffer pool can be hard to predict, also, and the total Disk I/O maybe tricky, we need to take into account whether the I/O is on a random basis or sequential basis. And so, for the cloud service architect, it is important that the system takes care of all these run-time details automatically, and let the hyper supervisor to concern only about the physical capacity, which is well suggested in [3]. This approach also having the advantage that it is no longer necessary for the hyper supervisor to have an inside into one client's database workload, because enabling the outsider to peek into the workload may give the opportunity for the outsider to guest which client's database piece it is, or what kind of operation this database is doing, and performing some malicious operations.

As have been suggested in the previous paragraph, another side of this problem is that we need to determine the total resource consumption of each of the pieces of clients' databases, which has been proven to be an non-trivial task. In the VM-cloud architecture, however, this task is considerably easier, that is because of the nature that VM machine can be hard to split and run on different nodes, and so normally the VM will be located within one single node, and also, because the VM is like the sealed box of the client's data, it is much easier for us the deduce that total resource consumption once we hack into the VM controller, the only hard part would be that since several VM are still sharing the same hard drive, and so the disk I/O might be a little bit harder to predict. In the Database As a Service architecture, the situation becomes much more tricky by the fact that now the client's databases are commonly sharing the DBMS, and the DMBS's buffered pool are also shared, the client's operations are now no longer "sealed inside a box", the sharing pattern is much more intricate. We also do not wish to achieve the monitoring of a client's database without introducing a significant change to the currently available DBMS, because it pushes too much pressure on the DBMS implementation and so becomes un-attractive to DBMS implementer, and also it result in a less flexible architecture. And so, the modern Database As a Service provider will construct their monitoring component above the available DBMS, and calculate various resource consumption profile either directly extracted from the DBMS itself, or do some estimation based on the available data.

As mentioned in the previous paragraph, monitoring each client database's resource consumption profile has proven to be non-trivial. This is mainly result from the fact that current DBMS do not provide enough useful details of resource consumption, or even if the DBMS provides the data, it is not on a shared per-client basis. [3] has shown that based on the OS profile and some plugin or enhancement of some DBMS, some of the required information can be directly extracted from, these resource data might includes CPU usage, network bandwidth usage, the number of lock acquired on a per table basis and RAM, buffered pool consumption. And yet for some other kind of information, it is hard to be extracted directly from neither the OS nor the DBMS, and require a more sophisticated method. In [3], it has been suggested that estimating disk write is an non-trivial task, we obverse that there are basically two kind of disk write operations performed on a normal database, one is the log write, which provide the undo and redo ability, another kind is the disk write of dirty pages. Since the fact that log write are mostly sequential and are linearly related to the number of transactions executed, it would be easy to deduce the number of log write once we hack into the log documented by the DBMS. Yet the disk write for the dirty pages is much more tricky, because it tend to be random writes, and the trace of the write operations are scatter in different places. The authors in [3] make a important observation that at the time the system is in a stable state, the speed of all the pages are getting dirty are the same with the speed of the pages that are flushed into the disk. Since tracking the rate of pages being dirtied is trivial, we have a approximation of the disk write for data. Although this method neglect some facts such as when dealing with databases which number of disk write are highly fluctuated, it makes a good approximation on general databases.

Another solution of estimating the resource usage profile has been given in [2], in the resource metric system they devise, they have also discovered that some sub-set of the required information can be directly extract from the OS or the DBMS, the information used in this scheme includes the Write Percent( the percentage of write operation perform against all operations), the Average Operation Complexity( the average number of pages touched in one transaction), the Percent Cache Hit (as the name suggested, is the percentage of cache hit, the paper, however, does not specific whether it is L1 cache or other kind of cache), the Buffer Pool Size, the OS Page Cache Size, the Database Size, the Throughput (the average number of transaction completed by a single client in one second). The main metric that they used to calculate the resource consumption pattern for each client is expected disk IOPS, throughput, and operation complexity. They use all the information extracted above to calculate each of these metric. Oddly enough, they have also discovered that the disk I/O rate cannot be non-trivially extracted from neither OS nor DBMS directly, and instead of making the assumption on the page dirtied rate and page flushing rate being the same at the steady state as in [3], the author in [2] devise an estimation equation for approximating the disk I/O per second. Let P(A) denote the buffer pool miss probability, that is, on a page request, the page cannot find in the buffered pool, let P(B) denote the cache miss probability. The greatest assumption on this equation is that we assume event A and event B are independent to each other, and so the expected disk I/O per second is given as P(A ∩ B) = P(A)P(B). This equation will be further expanded, eventually it will be able to calculated provided that we have extracted the aforementioned information from the OS or the DBMS.

1. **The detail usage of the user's resource usage profile extracted**

Once we have generate the detail resource consumption profile for each of the tenant in one node, many further operation becomes viable, the most noticeably viable operation now is to decide in run-time, whether the "packing" of different tenant in one node has been exceeded the maximum capacity of that node, and when this situation happens, we take the migration operation. The run time property can only be done based on the resource profile extracted above is that, cloud server normally contains a large number of tenant, and the hyper supervisor has little inside into the tenant query's nature. Requiring the manually adjustment of the placement of pieces of databases in these many nodes would be nearly impossible. And so an automated solution is required. In [2], the author has provided us with an automated solution. As mentioned in the previous paragraph, the three main metric used in [2] is expected disk IOPS, throughput, and operation complexity, and for each of these three metric, we classify the recourse consumption of one tenant into high, medium and low group, and so one possible profile of a tenant might be "high disk IOPS; high throughput; medium operation complexity". Also, for each node, it can be in one of the three states:{under utilized, good utilized, over utilized}. We use a vector representing the number of tenant in each of the resource usage profile group, for example, the vector [2,0,3] may represent there are 2 tenant with "high disk IOPS, high throughput and medium operation complexity" and 3 tenant with low in all these three metric. Once the node's resource consumption has exceeded some threshold on either disk IOPS, throughput or operation complexity, the node will be label "over utilized" are will the subject to migrating tenant from. The thresholds of these three metric required the administrator of the cloud service provider to define, yet given that now the hyper supervisor only concerns the overall aspect of one node, and the administrator is suppose to know these three metric given that they are comparatively "visual" to the administrator, and so this requirement could be easily met. Once an over utilized node has been detected, a search algorithm is perform, to expedite the finding of the recipient, [2] suggested that we should use the hill-climbing algorithm, and consider all the immediate neighbors, the algorithm will result with a destination node with the greatest improvement on the overall performance of the system. If no destination could be found, it simply suggested that the administrator should make plan for acquiring more nodes for the system.

The estimation on the resource profile on every tenant, not only provide us with the way to proper replacement and migration, as suggested in [3], it could be also innovate the way the cloud service provider changes its client and provide an exact and detail bill. The VM cloud, given that it is a hard isolation model, the resource usage can be scaled to a linear model, and so the billing is comparatively easy. On the other hand, in the Database As a Service model, the fact that many tenant are sharing the same DBMS make the resource usage model much more complicated, and it is usually not linear, for example, a tenant with 100 I/O per sec and another tenant with 40 I/O per sec normally do not add up to a total of 140 I/O per second for the whole system, the true I/O is somewhat slightly large given that fact that they are sharing a buffered pool and the stealing of pages might occur. The advent of the more accurate resource usage profile provided above solve this problem by often their client with the detail throughput and I/O consumption etc., and so the client can pay on per use basis of the cloud service, or throughput basis, which will undoubtedly make the client feel fairer in the paying scheme.

Also, suggested in [3], a more accurate resource usage profile enable the administrator in the cloud service provider to make more beneficial choice on tuning the cloud databases. Although we have previously mentioned that the workload of each tenant's database should not be visible to the administrator, we could provide the overall usage profile to the administrator in a more accurate way. Also, we could blindfold the administrator to let him know only a small necessary portion of a client's resource usage profile, which make the tuning work become more accurate and beneficial.

1. **Conclusion**

In conclusion, Database As a Service innovates in the ways that it discards the old VM cloud method which waste much of the space and efficiency, and achieve the sharing of different client's databases in each of the node's DBMS. It determine the best way to partition a client's database when it is necessary, and perform automatic tenant relocation and migration base on a more accurate resource usage profile extracted from OS and DBMS. It also guarantees that each of the database is encrypted and are isolated from other tenant in the system and also the administrator in the cloud service provider.

* **Evaluation on the technology**

The advent of Database as a Service technology changes the original usage pattern of the database dramatically. Without having the economic burden of owning a dedicated hardware and requiring some professional database administrator for maintenance, the database client now have the opportunity of simply signing a contract with the cloud service provider, and the service provider guarantee the service objective like the minimum respond time, by doing this, the client could save significant amount capital.

We have already witnessing the tremendous growth in the Database as a Service business, much more customers are willing to cut down the capital invested in owning all the hardware and professional administrators, and with the growth in network bandwidth, the respond time are more easily be achievable.

As more and more enterprise and organization moving toward Internet, changes are the number of client in need for a database to host their information will increase drastically, for small organization, or for example, an small department in an University, it would be a waste to owning the hardware and professional administrator, their databases are tent to be quite small to use a dedicated server to host. In the dawn of Database As a service, they can therefore cut down these unnecessary expenses but owning a reasonable fast database service.

The database cloud technology, therefore, would be a good fit for the kind of database client that owning a relatively small database and would be otherwise much wasteful if using dedicated hardware and professional admins, they should also be able to tolerant some latency in their transaction execution and the database itself would not contains some highly confidential data.

The first reason has well explained in the previous paragraphs, the latter two reasons are result from the underlining mechanism of the database cloud. Latency when using Database as a Service is undoubtedly higher than using dedicated hardware, because firstly the database are no longer located within the organization and connected with the upper application with a high speed network cable, now the database is located in the remote cloud service provider, although the overall network bandwidth has improved rapidly over the years, the latency tend to become higher. Also the latency might be result from the fact that on the database cloud server, sometime the system has decided that some piece of our database should be migrated into other node in the system. Given that the piece to be migrated could be of a large size, the whole process might take several minutes or even hours to complete, during which time, even though some cloud service provider provides migration without any downtime of the database, the time during the migration will be undoubtedly slower than normal, the client might find it weird that a normal operation that usually finished within one minute is suddenly taking 3 minutes to complete, if the migration time happens to conflict with the time when the system has a large number of request on the migrating data, the transaction throughput will drop drastically, the client, therefore, should be well prepared for such situation, and so it would not be appropriate for some real time application to use Database as a Service to host their database.

As for the privacy issue, it is probability unwise to use Database as a Service to host their database provided that the database contains some very confidential data. Although the cloud service provider always claims that their system has well encrypted the client's database, and the DBMS on the server will execute directly on the encrypted data without knowing the client's key to decrypt, it is not sure that whether the cloud service provider or just some malicious users in the service provider company has any method to decrypt all the data submitted by the user, after all, the cloud database system is built on top of the normal DBMS by the cloud service provider, they can easily chip in some of the malicious code to detect the user's data. Also, even if the data is not decrypted, because the system need the resource usage profile to accurately compute how to partition a user's database and decide where to relocate some piece of user's database, this metric are the detail resource consumption pattern of the user, and based on this profile, we can easily detect what kind of operations a database is performing, and from which we might easily deduce which company this database belonged to. Some malicious user in the cloud database provider company may take advantage of these and do some damage of the database, resulting in large economic lose in that company. Although the Database as a service is aiming to provide to the administrator in the cloud database company only the overall resource usage profile in a node but not on some specific database, yet since the detail metric of a user's profile is hard core measured in the system, it would be hard to imagine there is no way for any employee in the cloud service company to access these profiles.

Given the previous discussion, it may seem like only a small part of databases should be located on the database cloud service. However, we are neglecting an important fact, that is, currently not only the public database cloud service is thriving, but also the private database cloud is witnessing a drastic growth. For some large enterprise, they might consist on a considerable number of different databases, and requiring administrator for tuning all the different databases is a costly action, and so, for these enterprises, they may make use of the database cloud technology, and build a database center for their enterprise, now many databases can share a common hardware in the cloud server, and the personnel needed to administrate these databases would be cut down drastically, resulting in a large saving. The latency issue is much lessened here, since within an enterprise, it is common to have very high speed network cable connecting all the departments. The privacy issue might be of a little bit concern, yet since we are within the context of the same enterprise, these concerns would be much less severe as in public database cloud. Currently, owning the cloud service software might still be an expensive purchase, and so not many small company and organizations will be able to afford owning their own database cloud, and so they will need to resort to the public database cloud solution, and when latency and privacy is of high concern, they will not able to use these public cloud. Yet we assume that as the technology become more and more mature, the pricing for owning a cloud service software would eventually be affordable for many clients, and so it will enable much more databases to be put into the cloud service.

All being said, although as the software becoming cheaper, more and more databases will put into the cloud, some specific database which are highly intolerant to latency(i.e. databases for real time applications) or containing highly confidential data will likely remains in their dedicated hardware.

Obviously, given that fact that more and more databases are emerging, and the fact the financial gain of using the public database cloud service or deploying the database cloud service for the enterprise it own, it would be worthwhile to invest into this newly and thriving technology. And given the aforementioned limitation of Database as a Service technology, if we were to invest into its development, we have two options, one options is to invest into the enhancement of privacy and migration without downtime technique, with the advancement on these two aspects, public database cloud would be much more attractive to a large set of clients, and it is the main focus of the current trend of research. We have since several major products using the public database cloud technology, and will become more popular once we have increase the privacy issue and reduce the latency issue.

Yet when choosing to invest into this new technology, we might also want to focus on decreasing the price of the cloud service software. The current cloud service software are built upon the basis of normal DBMS, and perform automated administration, which is still of too much a price for a small organization or company, once we could reduce the price of the whole automated system, given that the latency and security issue is lessen when an enterprise use its own database cloud, more and more organizations and companies will start to build their own database cloud.

In conclusion, Database as a service, although some limitations would be hard to overcome shortly either due to price or technology, it is still a very attractive technology and will thrive in the future.

* **Column-stores**
* **Introduction of the technology**

1. **Overview of column-stores and the basic comparison to row-stores**

For a long time, the only kind of storage organization for table are row-stores, in this architecture, one table is stored within one file(provided that this table is not enormously large and need partitioning), the data of each row is stored together, in a row by row fashion. This architecture has the advantage that normally our application will assume using the data in a row basis, and so storing the data in row fashion let the result of all operations performed still remains in the row fashion, and will be easy for the application to retrieve the data row by row.

Yet the speed of CPU and RAM increase drastically, whereas the speed of disk increase in a comparatively very slow, as we have entered the 21th century, the slow transferring of disk I/O has become a major bottleneck of the whole system. Also, the explosion in Internet applications also inspire database table to be a fat table with hundreds of columns, and most of the columns will not be used in a single query, yet since we are storing all the data in a row fashion, every time we extracted a row from the database, we will need to read all the columns, be if useful or not. Considering that the gap between the speed of CPU and hard drive is becoming increasingly widening, and so using row-stored for these fat tables becomes a major bottleneck for the system. Also, it may be useful that we can add or drop a column to a big table with millions of rows without downtime. The advancement of the Internet Social Network requires us to do exactly this. Suppose now Facebook wants to add a new attribute to every user, say now they can support user to fill in their "spouse name", considering the amount of user using Facebook, the rows can be count in billions, if we were using the old fashion row-stores, adding a column to a table require a big lock on the whole table, and the processing of adding a column might takes days. Yet since the Internet is keep developing, the add column operation might happen quite frequently, like adding linked in account etc.

Column-stores method solve the two problems above nicely, in column-stores, the data are no long stored in a row fashion, instead, each column is stored in one file, and one table consists of several files, each representing one column. This design solves the fat table problem in that the system can now just read in the necessary column without reading other useless column data, therefore saving a considerable amount of disk bandwidth and I/O, since the disk I/O is the major bottleneck, this design will speed up significantly when answering those queries. In the adding column problem, we could now just add a new column file to the table, without requiring a big lock on the whole table, and so adding a column requires no down-time, extremely suitable for modern Social Networks, and we have seen Facebook has abandon using MySQL and start using NOSQL databases for this reason.

1. **The key technologies in Column-stores**

In the simplest form of column-stores, it is not much of a fundamental revolution in the database storage organization, that is because only a small set of situation is more efficient in using column-stores. These kind of operations usually having some common features, such as using only a small number of columns in a fat table, and also within these necessary columns, they use a large number of rows' value, and so we can run our sql query towards blocks of column data in a tight loop, realizing good utilization of cache and CPU. But when we are under the OLTP situation, and many time we just modify or read in no more than a few number of rows, in this situation, because of the tuple reconstruction overhead and the fact that we might need to read in from several different files in order to get our result, the performance would be worse than using row-stores. And so, in order for column-stores to be attractive, several importance improvement on column-stores has been developed, they includes Block-oriented and vectorized processing, Late materialization, Column-specific compression, direct operation on compressed data, and Database cracking and adaptive indexing [4].

* **Block-oriented and vectorized processing**

Vectorized processing is not improvement specific to just column-stores. Originally, our query optimizer will only consider left-deep plans, and generate the results in the pipelining fashion. That is, no intermediate result will need to be materialized and written to disk, when the system need to generate another result, say the application layer JDBC has called a "iterator.next()" function call, the DBMS will generate the result on demand, meaning that just retrieving one tuple and do the necessary operations and return just that result tuple. Yet because of the left-deep plans decision tree might get too deep that the instruction in the lowest lever is evicted outside the cache when the "next()" operation is called, then the system will have a thrashing bottleneck, and so in row-stores, we solve this problem by bring in blocks of rows into cache, and performing operations on one block other than just one tuple, and so even the instruction has been evicted, and cost of bring back the instruction is amortized.

In column-stores, we can make the similar improvement on the query execute component by not just bringing in one value of a column, we bring in a block of values of a column into the cache, we call each block a vector. Similarly, when the application layer calls the "next()" operations, we operates on a vector on tuples instead of operate on just one tuple.

The advantage of this technology is that ① as in the situation of row-stores, it amortized the cost of bringing in instructions into the L1 instruction cache; ② the number of function calls invoked by the interpreter reduced by a factor equals to the size of the vector[4]; ③ the vectorized processing will use the size of L1 cache as the vector size, and so we will not experience cache thrashing, and for many operations, bring in a vector of column value exhibit high cache locality; ④ in the one-tuple-a-time method, we might need to check for some condition like output cache overflow conditions for every tuple, as with the vectorized method, since we can do an aggregate check, checking whether there is enough space of a whole block, so the number of checking required reduced drastically. [4]

The aforementioned advantages are just a sub-set of benefit vectorized processing can bring to the system. By the column specific compression, we have the opportunity to represent each column in an array like fix length format, and so using vectorized processing would drastically improve the overall performance in that the operation can done on blocks of values in a tight loop, reducing disk I/O.

* **Column-specific compression**

One of the most important advantages column-stores had compared to row-stores in that the compression ratio of column-stores is much larger than if the same data was compressed in row-stores. The main reason is that in column-stores, we stores all the data of a column in a file, and since the data within a column usually have many similarities, and so we can make our compression algorithm based on the locality of data. Intuitively, data that are more similar to each other(for example like the length of data, the format of the data etc.) will likely to be compressed in a more compact form. Row-stores do not have this advantage in that all the data in a row are stored together, and given that the types and format of different columns, the compression ratio will not likely to be high.

Using the operations on compressed data technology, we can achieve very good performance in column-stores. The high compression ratio not only reducing the overall size of the data, but also decreasing much of the CPU time on query operations. More compact data means that data could be located more compact on hard drive, and might be able to put in a single track, therefore, the seek time of the magnetic head could reduce thanks to the reduced size, increasing the overall transferring time. Also, even without the ability of performing operations on compressed data, since the speed of CPU is increasing drastically whereas the speed of disk I/O increases very slow, we might consider read in few data but take more time to compute the decompressed data. The high compression ratio on column-stores therefore enable the DBMS to transfer much less data compared to row-stores, overcoming the major bottleneck of low disk I/O.

When compressing such relatively similar data, we are therefore tempted to use light-weight compression scheme to generate fix-length data, making the compressed column becomes an array, instead of using high-weight compression scheme such as Huffman Code. Huffman Code do not generate fix-length data, yet if we are able to sacrifice some compression ratio to trade for an fix-length array, we would be able to perform many operations on these array in a tight loop, and we do not need to probe for the length on each value, increasing the overall performance compared to high-weight compression scheme.

There are several compression methods available [4], if the data was sorted, using Run-length Encoding is the best option, that is, we detects the "runs" of a repeated value, and replace each of these runs with a fix-length tuple (value, start position, runLength). Obviously, this compression scheme is not efficient if the data is not sorted and the same value do not compact to a runs. Instead, if the data is sorted, this scheme yield the optimal compression ratio and the compressed data generated are fix-length.

In Bit-Vector Encoding, we represent each distinct value in a column by a "vector" of bits, and replace the actual value in the column with this bit vector. This method is a little similar to Huffman Code, but we would make each vector the same length. Obviously, this method would be much better if the distinct value number in a column is small, otherwise, the vector for representing each value could be large, results in a very low compression ratio.

Dictionary encoding works as its name suggested, it constructs a "translation table" for every distinct value in a column, and replace each value in the column with the smaller translated value. Of course, we could make this translated value the same length. It is similar to the Bit-vector Encoding scheme, and are most effective if the number of distinct value in a column is low.

Yet another encoding scheme is the Frame Of Reference scheme, this scheme could only be used when the values in a column has exhibit high locality, which means that the data that are near also have their value close to each other. In this situation, we could represent a block of value by only storing the starting value of each block, than each following value is just the difference to the starting value.

* **Operating Directly On Compressed Data**

The ability that column-stores can operate directly on compressed data increase the system performance in the greatest form. Intuitively, by choosing the appropriate compression scheme, we could make certain kinds of operations operating directly on data. As showed in [4], Run-length Encoding enable many of the aggregation method operates in a extremely efficiently way. SUM, AVG operations, specifically, when using Run-length encoding, could directly perform on the compressed data.

Yet as suggested in [4], to enable such ability, we would need to change the query interpreter dramatically, to enable the interpreter to understand what kind of compression scheme we are using, and perform different operations on different compressed data. This method is not scalable in that, every time we add a new kind of compression method, we would need to modify the query interpreter.

And so, a better method would be that we organized the compressed data into blocks, and for each blocks, regardless of their compression method, we maintain a set of metadata, and provide a set of API for the query interpreter to call. The example metadata might be minValue, maxValue etc. We need to consider all possible operations on the block of data, and implement on the necessary APIs. In this way, the query interpreter are no longer require to understand the underlining compressing method, and simply a little extension to the original query interpreter is needed. As suggested in [4], by conducting the operations directly on the compression data using this kind of extension, we could achieve an order of magnitude improvement in performance.

* **Late Materialization**

In short, late materialization technology means that we will not retrieve the exact data until we are absolutely necessary to do so. What it means is that we do operations on a list of positions whereas a list of actual data as long as we are able to do so. Not until we are doing some operations that require us to retrieve the actual data do we materialize. For example, when the where clause contains predicates like R.a<50, when we will defer the materialization process until we encounter this predicate, and we would materialize only the data in column a. Join operations are also done on position list whenever possible, and so after the join, some additional sorting on the position list might be needed.

Using late materialization bring us many performance boost [4], for example, ①in many situations, selection and aggregation operations will result in materializing some tuples that are later found useless, in late materialization, these situations would be largely eliminates; ② some compression scheme, like the Run-length encoding scheme, when materializing, required us to decompress the data, deferring the actual materialization, since it might eliminate some unnecessary materialization, could also eliminate some of these decompress operations; ③ in column-stores, when the data are brought into cache, the cache will not be polluted with other unrelated data, since the cache is one very valuable data, it increase the overall performance.

* **Adaptive Indexing and Database Cracking**

Database Cracking is the technology that building an index on a table "on-the-fly", that is, the database administrator do not need to consider whether there is a need to construct an index on a column beforehand, the DBMS will slowly and adaptively build an index on a column if the workload of the database exhibit such necessity.

The way database cracking works is that, given a normal column a, suppose some query issue a query with where clause containing "a<10", than when we are processing such query, we partition the column file into 2 parts, one part with all the values smaller than 10, and another part with all the value bigger than or equals to 10 as a side-effect. The decision made here is based on the fact that a query done on a column would likely be needed in the future, and when another query with a>80 executing, we further partition the column file into 3 parts by separating the second part into one part with values bigger than or equals to 10 and smaller than or equals to 80, and another part with values bigger than 80. As more queries are performed on the column file, the file is partitioned in a way that would be optimal to all query, and so we have slowly build an index on this column in an amortized way.

The benefit of Database cracking is important, it no longer require the long processing time to build an index, and it do not require the database administrator to understand the workload of a column before deciding whether we should build an index on it. These two points are becoming more and more important as big data processing might exhibit no constant workload pattern and a lot of applications now do not have time to put in a sand box to evaluate the actual workload it would get [4]. The DBMS just build the index when necessary and without any additional knowledge.

1. **Simulating Column-stores and the coexistence of both column-stores and row-stores**

Given that row-stores and column-stores both have their own set of advantages, we might want the coexistence of both column-stores and row-stores. The simplest way to achieve this would be using two separate copy of tables for each table, one for column-stores, one for row-stores, yet this method not only waste large amount of space, but also incur overheads when doing update, insert operations.

In [4], the authors have provided two ways for a row-stores DBMS to simulate column-stores, in these way, the row-stores DBMS is free to do the operations in the old fashion ways and could use some the benefit of column-stores when necessary.

The first method is vertical partitioning, in this method, we divide each table into many tables, each table represent one column in the original table, and in order to do the tuple reconstruction, each sub-table would consists of two column, one for the position of the value, one for the data in the column. This method is simple and do not require much modification to the row-stores DBMS, yet the overhead in joining all these different sub-tables in order to just reconstructing the original table is unbearable.

The second method is creating an index on every column, one immediate overhead is that maintaining the indexes for each record might be a large overhead, and that a more important problem is that when in index, the order of data is unlikely to be the same as in the original raw data table. When this happens, during the joining operations, it might result in the situations that both position list are unsorted, and so complicating the joining operations.

In [5], Microsoft SQL Server achieves a way of coexisting both row-stores and column-stores by introducing a new kind of index, the Column-wise Index. Without creating this index on a table, the table is in the normal row-stores form. Once creating this index on a table, each column is then separated into column segments, and a segment directory(basically a tree) is created, associating all the column segments. Essentially, this is similar to creating an separate file to store a column version of the original table, but user now have a choice on whether a table need to use the column-stores ability. All the column-stores ability could be available to tables once this index is created, the Batch Mode Processing in [5] is essentially the vertorized processing in [4]. On any query operations, the query optimizer would make a decision on whether to use the normal row-stores method or using column-methods, and so utilizing the advantage on both architecture. One obvious overhead is also the cost on maintaining two separate copies, yet since only a small number of table need the Column-wise Index, this overhead is acceptable.

1. **Conclusion**

The emergence of column-stores is based on the fact that the major bottleneck in the current system is the big speed mismatch between hard drive and CPU, and because column-stores and its associate improvements are sometime a well fit for this current hardware, it is the optimal solution in some situations. The coexistence of both row-stores and column-stores is also viable, utilizing both advantages.

* **Evaluation on the technology**

1. **Details comparison between row-stores and column-stores**

In order to indentify the kind of user that would be mostly appeal to column-stores, we would need a details comparison of different performance of each architecture on different situations, and most importantly, we might want to compare the performance difference when executing column-stores in the worst case.

And so the most obvious question is what is the worst case situation in column-stores. According to [4], we all know that column-stores is in best case when only one entire column is needed, in this situation, only one column file is scanned, reducing both the disk I/O and the CPU time. But then when we are required to scan the whole table, in the normal row-stores, this operation is trivial, and all we need to do is to locate the starting point of a table, and read in data row by row. Yet when this situation happens in column-stores, much more complicated operations needed to be performed, for one thing, because we will eventually reconstruct the tuple in the row form, and so we are required to read in all the column files, and since now the column files are scattered in different places in disk, the overhead of accessing multiple locations of disk and processing the reconstruction operations might be very large.

But, shown in [4], because of the fact that the CPU speed increase much faster than the hard drive bandwidth increases, to which was added the fact that currently many OS have focused on doing read optimizations, for example, pre-fetching possible data, the worst case situation of column-stores might be more competitive against row-stores. According to a research done in 2006, in the worst-case situation where all column are accessed, column-stores is only 20%-30% slower than row-stores. Also, another research has shown that when using Flash SSDs other than magnetic hard drive, the worst-case situation in column-stores in as fast as row-stores performance. And so, the authors of [4] predict that the worst-case performance of column-stores will eventually be the same as in row-stores, and in some cases, when scanning the whole table, and there is a predicates in the where clause having high selectivity, column-stores could even be faster than row-stores.

Yet the previous discussion has neglect one important operations, the insert and update operations. Column-stores cannot do insert and update operations as fast as in row-stores, which is very obvious. In row-stores, whenever we need to update or insert one record, all we need is just one disk I/O, yet when using column-stores, all the columns of one single table is in different places, and so the update or insert operations on one tuple might ends up incurring many disk I/O, and these disk I/O are mostly random I/O, degrading the overall system performance even more.When doing a batch of updates or insert, the situation becomes even worst, in row-stores, these disk I/O might be able to pack together, but in column-stores, more random I/O are incurred, the performance is poor.

Another major weakness for column-stores when performing updates is that, because as mentioned before, one of the major benefit of column-stores is that it has high compression ratio thanks to the column specific compression, and because it can operate directly on compressed data. But when doing update operations, this advantage becomes a disadvantage. Many compression scheme require us to first decompress the original data, storing the intermediate results(when Run-length Encoding is used), and do the updates, and then compress the data once again and finally write back all the data. This makes the update operation very complicated, and require much more disk I/O than in row-stores.

One solution, shown in [4], and used by C-Store and MonetDB, is that they no longer storing the data in one place. Instead, they divide the data into two zones, one is the "read-store", which is the original storage in column-stores, and the other is the "write-store", which stores the newly added data or updated data. Update operations could also be implemented as insertion followed by a deletion. All the operations should now not only considering the data in read-store, but also consider the data in write-store, and periodically, the system will merge the two stores together, limiting the size of write-store, otherwise the cost in normal operations in merging the two in run time would be high.