Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 58 (2015) 622 – 634

Procedia Computer Science

Second International Symposium on Computer Vision and the Internet(VisionNet'15)

Architectures for scalable databases in cloud – and application specifications

Pankaj Deep Kaur[a], Gitanjali Sharma[a,*]

[a] *Department of Computer Science and Engineering, Guru Nanak Dev University, Regional Campus-Jalandhar, 144001, India*

Abstract

Despite cloud abstractions, data is critical to applications hosted on cloud. Thus, overwhelming data flow also requires database tier to be scaled efficiently and dynamically besides other tiers in cloud platform. However, this is not easy since databases maintain tightly coupled state while promising transactional support. Objective of this paper is to survey scalable DBMS architectures deployed for hosting data-centric applications that are gaining momentum in analytical processing, scientific researches, personalized searches and corporate dealings. This paper reviews scalable architectures and their suitability to host specific classes of applications. Finally it addresses issues in current architectures and presents some future suggestions.

Peer-review under responsibility of organizing committee of the Second International Symposium on Computer Vision and the Internet (VisionNet'15)

Keywords: Atomic transactions; HDFS; MPI-I/O; Multi-tenancy; NoSQL; Scalability; IaaS; PaaS; SaaS;

1. Main text

Over the past decades, cloud computing has emerged as a ubiquitous and successful computing paradigm. Despite of cloud abstractions, data is critical to applications hosted on cloud. Since DBMSs physically store and maintain application specific data, they constitute a central component of cloud software store. However, scalable distributed DBMSs deployed over cloud and hosting diverse applications face numerous challenges due to different schemas, different API/query interfaces, workload fluctuations, design decisions, etc. As the workload increases, system scales and poses numerous challenges. Some of the mission critical tasks in scalable distributed DBMSs include

* Corresponding author. Tel.: +91-947-884-9950.
E-mail address: gitanjali1992sharma@gmail.com

Peer-review under responsibility of organizing committee of the Second International Symposium on Computer Vision and the Internet (VisionNet'15)
doi:10.1016/j.procs.2015.08.081

balancing load, supporting multiple tenants with minimum operating cost, reducing flow of data over network, parallel computing, interoperability among heterogeneous data stores and many more.

The objective of this paper is to distill primary concepts and discuss their implementations and scope. Like: applications that access large portions of historic data for analysis (like OLAP), have to deal with multiple dimensions of data. Applications like these require parallel computing solutions where system scales by adding processors for processing dimensions. Similarly, scientific parallel applications have been deployed over shared network file systems till date. These are characterized by dominant read operations, thereby causing excessive movement of data fragments to local nodes.

Besides many diverse applications are being hosted on cloud and each has a unique architecture. As a result there is a need for interoperability and global transaction management across heterogeneous stores. Further, many existing architectural solutions offer services by trading off between scalability and consistency. Supporting multiple tenants across underlying scalable distributed databases is another issue in this domain. Also, to support interoperability and multi-tenancy over distributed cloud databases, one needs to be aware of the interoperability challenges involved among different cloud service providers like SaaS, PaaS and IaaS.

1.1. Structure

Rest of the paper reviews various architectural implementations of key concepts in context of scalable database management in cloud. Section 2 presents architectural design deployed using parallel computing for processing multi-dimensional data. Section 3 discusses scalable design which exploits local data access. In section 4, a multi-tier architecture has been discussed to achieve interoperability among heterogeneous data stores. Scalable architecture to support multiple tenants while minimizing operational cost is discussed in section 5. Further, section 6 discusses scope or applicability of afore mentioned architectures. Section 7 discusses interoperability challenges among cloud service providers and presents certain recommendations. In section 8, other related DBMS issues like: partitioning, replication, consistency and automatic self-management are discussed which are necessary to achieve efficient scalability of databases over cloud. Section 9 reviews related work in this domain. Finally, section 10 concludes the paper by addressing the issues in exiting architectures and presents future suggestions for a scalable architecture capable of addressing the mission critical challenges.

2. Parallel computing

Parallel computing architecture deploys multiple processors, say (m+1), where each processor is capable of executing k independent parallel processes [6]. Like other high performance database architectures, each processor has its own main memory to store data. This architecture has the ability of scaling m processors dynamically by deploying additional processors. As a result, it promises that both processing and storage (memory) capability will scale dynamically with increase in database size. Architecture of parallel computing is designed such that, out of m+1 processors, one is master and the rest m are workers. Master is responsible for receiving insert/query requests from clients and responding them with a memory reference to the location where worker processor has posted the results. Hence, high degree of parallelism is achieved: distributed execution of multiple insert/query requests over multiple processors and further parallel execution of concurrent operations by each worker. Architectures like these are further supported by distributed multi-dimensional tree data structures like PDCR trees [6], PDR trees [6] which assist in querying, processing, inserting and aggregating data in real-time. Master stores one tree and every worker stores multiple trees. These trees are implemented using array logic where new nodes are appended and tree links are represented via an integer references rather than memory pointers [6]. These tree structures are leaf oriented [6] i.e. data is stored in leaves while intermediate nodes have routing information. PDCR trees and alike are thus, designed such that they scale dynamically and can be efficiently compressed, split off and propagated among processors through message passing [6].

For a given data store with table T and a set of S dimensions $\{S_1, S_2, S_3,..., S_s\}$, such that for all S_i, $1<=i<=S$ there is a hierarchy H_i consisting of attributes corresponding to each hierarchy level as $H_i = \{H_{i1}, H_{i2}, ..., H_{il}\}$.

A three level hierarchy for each dimension is shown in figure 1 (a).

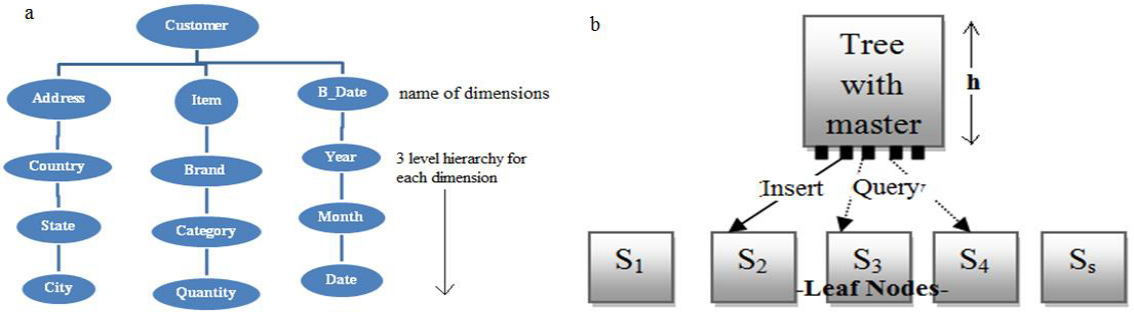

Fig. 1. (a) A 3-dimensional data warehouse; (b) Illustration of PDCR tree.

Initially, this architecture begins with one master processor storing one PDCR alike tree and an empty database (i.e. m=0). In the beginning, tree stores complete database and is tuned to a depth of, say h as shown in figure 1 (b), after which remaining levels form sub-trees and are meant to be stored with n workers (n>>m) thereby dynamically creating a worker processor. Hence, it can be inferred that data is present in workers whereas routing information and aggregation information is present with workers. Since, master keeps track of sub-tree allocation to all workers and creation of new workers, it is responsible for load balancing. It periodically shuffles n>>m [6] sub-trees to ensure that parallel query threads are distributed evenly among workers. Hence, load balancing is achieved with ongoing dynamic scalability so that system does not compromise on performance.

3. Local data access

Unlike network shared file systems with local disks, distributed file systems over nodes with locally attached disks have capability of scaling well with dynamic workload. For this purpose Hadoop Distributed File Systems (HDFS) [7] are deployed which in turn support processing of MapReduce [22] type workloads [10]. The main motive behind implementation of MapReduce is that it is efficient and fast for propagating computations to stored database fragments rather than moving data fragments over network towards computations. Hence, this architecture shifts from compute-centric processing to data-centric processing, thereby exploiting data locality and avoiding excessive data movement over the network especially in case of read operations.

All the incoming parallel jobs are submitted to a PD_Scheduler which is responsible for mapping processes to data, thereby reducing the amount of data that is being pulled across the network. Computations now need not wait for nearby processing nodes to get free and data fragments to become available from stored memory rather they are directly sent to data fragments. This reduces input-output latency. PD _Scheduler identifies particular fragment assigned to each node and determines the location of every unassigned data fragment. It is assisted by Fragment Position Monitor [10] in determining the position of fragments. Fragment Position Monitor works at the backend to periodically update PD_Scheduler with status of unassigned fragments. Scheduler invokes monitor to determine location of fragment and hence, monitor is implemented between PD-Scheduler and HDFS. PD_Scheduler then routes the computation to fragment location as shown in figure 2.

HDFS storage is partitioned into chunks/fragments of data. It maintains more than one copy of each data fragment so as to avoid deadlock and balance the load. Applications interface with HDFS via HDFSNameNode [10] using libHDFS [10]. They query NameNode for determining fragment location which in turn responds with a list of nodes that physically store the requested fragments. On the basis of such information, applications decide as to which node would execute a given computation. However several discrepancies are faced while executing MPI based [10] applications on HDFS [10]. Firstly, HDFS processes its own I/O interface whose constituent programs for read and write are different from earlier MPI or POSIX-I/O which either ran on shared Network File System or UNIX File System. Also other feature differentiating HDFS I/O from MPI-I/O is that former only allows append writes while latter supports concurrent distributed writes. As a result, there arises an incompatibility between HDFS and traditional MPI [12] based programs which need to be resolved. To address this issue, an I/O translation layer

must be deployed. I/O translation layer transparently transforms high level I/O computations to specific I/O HDFS routines.

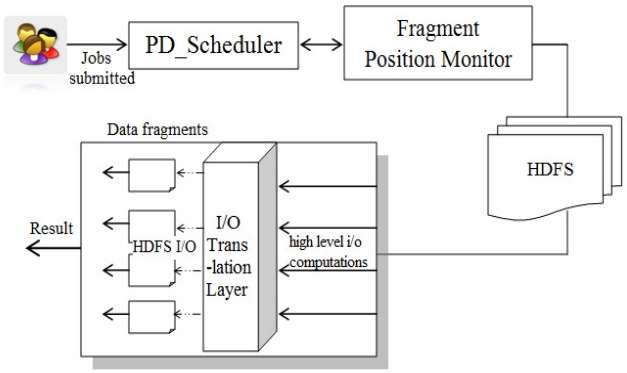

Fig. 2: Architecture for local data access.

This I/O layer interfaces with HDFS server via hdfsConnect() and deploys HDFS as a local directory [10] for computation nodes, thereby triggering corresponding HDFS I/O routines. Moreover, concurrent writes are handled by maintaining an output directory in HDFS and output is written to each physical file in the output directory.

4. Interoperability among data stores

As discussed in [8], [11], existing NOSQL data stores are heterogeneous in terms of API/query interface, consistency and partitioning guarantees, etc. As a consequence, these support different applications with different service level objectives. In order to achieve inter operability among heterogeneous NoSQL data stores and support atomicity of transactions across these data stores a multi-tiered architecture with inter transaction support is needed. The tiered architecture is formed by segregating each involved physical entity into a particular tier i.e. client, center-tier and distributed database [14] as shown in figure 3.

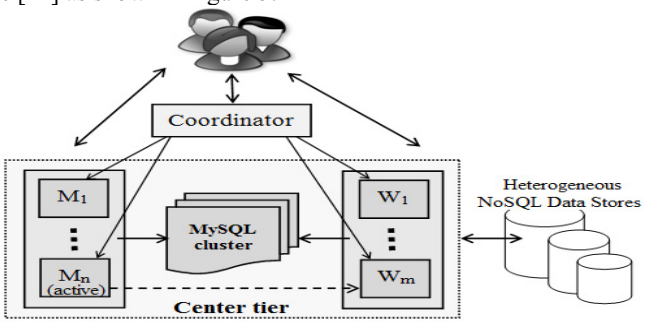

Fig. 3: Multi-tiered architecture.

Client tier consists of multiple client nodes. User interfaces with whole architecture through the services provided by client node. It is leveraged with the responsibility of consolidating transaction requests received from the user and sending it to respective Inter Transaction Manager (ITM) [14] whose address is achieved by requesting master node of respective ITM.

Atomicity of transactions and scalability is ensured by center-tier which is further classified into worker nodes, master nodes, ITM under center-tier is responsible for guaranteeing atomicity/serializability of all transactions. It is supported by client port [14] at the backend through which client connects to ITM and this enables the architecture

to maintain status of transactions. Each ITM maintains metadata [14] to keep record of transactions as they proceed. Hence, each ITM position and its allocated transaction can be easily identified by metadata [14] and can be useful for recovering a transaction in the event of a failure. Metadata structure maintains information about status, timestamp, transaction_Ids, read/write locks or conflicts, etc. It is stored using distributed MySQL-cluster [13, 14] in place of HDFS, since former is more SQL compatible, scalable, reliable and also provides easy and faster access.

Further ITM runs over a worker_node. These worker nodes ensure scalability and are controlled by master node. When a new worker_node arrives it contacts coordinator to determine the location of active master in order to register itself to master. There are multiple Master nodes present out of which only one is active at a particular instant of time. Various algorithms are used to replace a failed master node. It guarantees load balancing and recovery from failures through balancer and cleanser processes respectively. Coordinator is a service promising high performance, synchronization, master maintenance and FIFO execution [14].

5. Multi-tenant support

Owing to the 80/20 rule [24] there is need to device database architecture based on the logic of loading as per the query requirements. Cost efficient multi-tenant architectural approach only maintains those tables in DBMS that are being or will be queried in near future. Besides that, all the other tables are maintained outside DBMS in a single table. Thus, when a table required by a query is not present in DBMS, new table is created and then loaded with data after which the execution of query begins. If an ideal table is present in DBMS, all its updates are backed up into shared table and table is finally dropped. Similarly, when a query finishes its execution, the corresponding table is dropped from DBMS. As a result, only working tables are maintained in database instead of maintaining every private table of each tenant.

This architectural approach thus, implements two schemas: Shared_table schema [24] and Active_table schema. There is only one shared table which is used to integrate data from all tenants into one place. However, there are a number of active tables under Active_table schema which hosts data required for processing and responding queries. All columns of tenant's logical table are consolidated and stored in a common column under Shared_table schema which is used for restoring tables when needed. Each Active_table consist of two additional columns ID_columns and Status_column. ID_columns identify ID of logical tenant table's row so as to map it to the corresponding row in the shared table while flushing the updates to Shared_table schema. Status_column on the other hand, is used to determine whether a row has been updated and if any insertions or deletions are carried over a particular row. Initially all the values of the Status_column are set to 0 and is modified to 1 if insertion/updations is carried on a row or is modified to 2 if it is deleted.

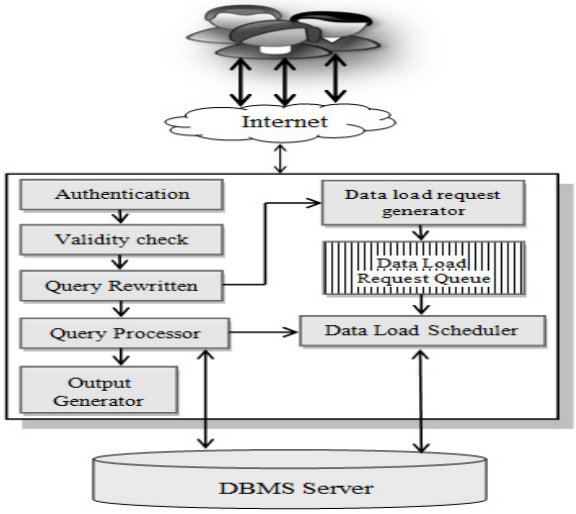

Fig. 4: Distributed in-memory architecture.

Further, depending upon how the Shared_table schema or tenant tables (other than active tables) are stored and what service level objectives are supported by tenants, two implementations of this architecture are possible: on disk or in memory based RDBMS models [24]. Former one ensures acceptable performance and high consolidation whereas latter yields high performance. Disk-based RDBMS is applicable where high throughput is not a priority and abundant memory resources are not available. It is a centralized approach and hence is not suitable for fault tolerance and load balancing. On the contrary, In-Memory-based RDBMS is applicable where high throughput is preferred and abundant memory resources are available. This employs a distributed approach as query processing and data storage is carried out in different database instances (unlike centralized Disk-based RDBMS).

In centralized Disk-based RDBMS architecture, as shown in figure 4, query requested by client is passed onto application server which first authenticates the client and then validates the query. Only after validity check the query is rewritten to be executed by query processor which finally generates results and transfers them to output transmitter. To handle the events when required tenant table is not in DBMS, a data load request generator generates a load request and request is passed into the data load request queue which in turn invokes data load scheduler.

Distributed In-Memory RDBMS architecture, as shown in figure 5, has one maser node and multiple slave nodes. Master is leveraged with the responsibility of authenticating the client, monitoring the slaves, balancing and distributing the load among slaves. Slave nodes are responsible for validity checks, query rewriting and processing, result generation and loading requested data or backing the excessive load. It stores all data in a distributed data store deployed over distributed file system, HDFS, which automatically replicates data blocks over multiple nodes, thereby guaranteeing fault tolerance. Thus, data is stored in data stores implemented on HDFS while query processing is carried by in-memory RDBMS.

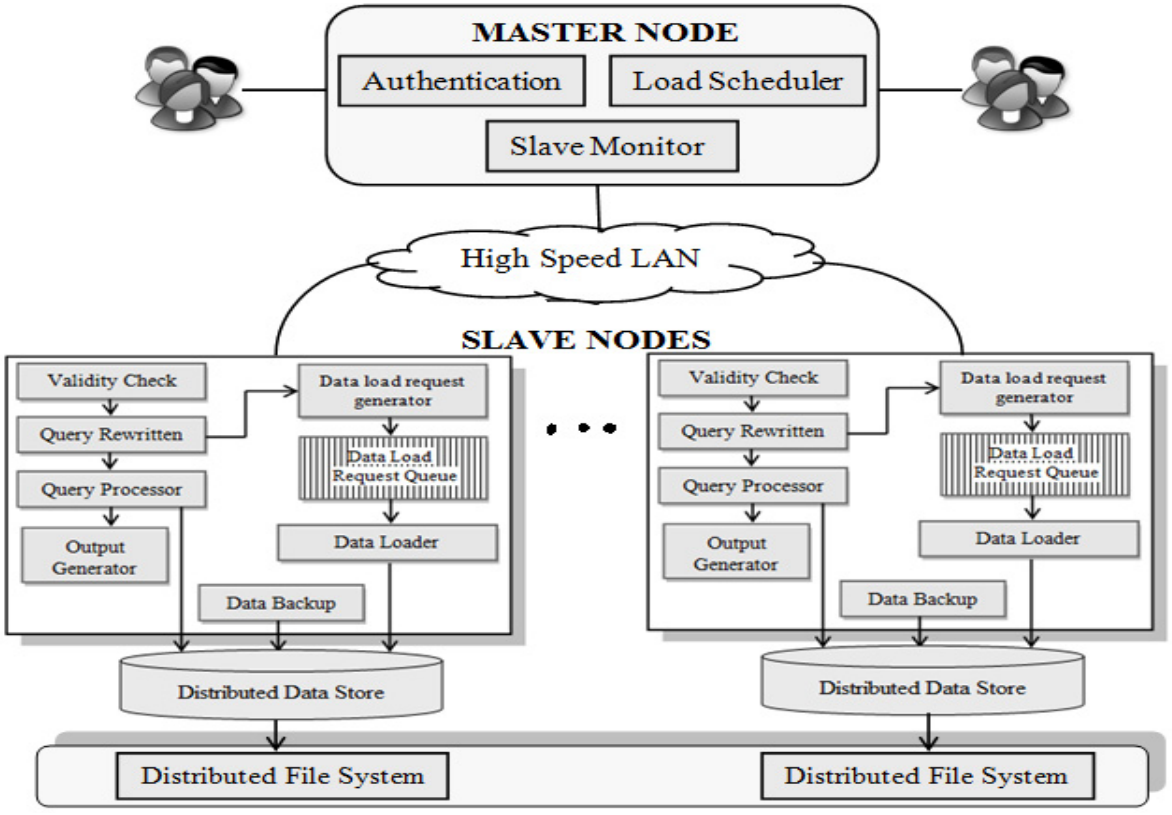

Fig. 5. Centralized disk based RDBMS architecture.

6. Scope

6.1. Analytical processes

Parallel computing architecture discussed in section 2 is useful for applications that access/aggregate large portions of data like online analytical processing (OLAP) which process historic data for analysis. Such applications need up-to-date information for effective analysis, thereby calling in for dynamic scalable real time updates and multi dimensional data handling. Multi-dimensional tree data structures used in this architecture has proved very useful and are capable of dynamically scaling processors with increase in database size.

6.2. Scientific parallel applications

Scalable architecture with local data access capability as discussed in section 3 is suitable for applications handling excessive read operations like scientific parallel application. These applications are characterized by excessive movement of data from one cluster node to another. With the increasing data set size, the processing and propagating of data fragments becomes very costly and also causes delays. Thus, local data access by adopting a data-centric approach can considerably reduce the delay as well as cost involved.

6.3. E-Commerce and E-Banking

Numerous applications hosted on cloud implement different NoSQL data stores which use different API/query interfaces, partitioning and replication mechanisms, consistency levels and concurrency guarantees. There is a lack of common standard among these heterogeneous data stores. Data and transactions are not portable or reusable. Further, there is a need of inter database transaction support or global atomic transaction management over heterogeneous NoSQL data stores [14] while guaranteeing fault tolerance and accuracy. Applications such as e-commerce, e-banking require multi-tier interoperable architecture discussed in section 4.

6.4. Data Market Applications

There are applications where tenants are provided data as a service. A user submits a query over the data and is charged as per the results generated by query processor. Although these users tend to read or analyze the results at the time of generation itself, but service providers have to maintain query results for future references. As a result, they have to maintain tables which may not be accessed by a tenant over a long period of time. It is cumbersome for service providers to load all the tables for each tenant every time they access the database, since a lot of resources and storage capacity is wasted. Moreover, it generates unnecessary workload over the network. Hence, cost efficient multi tenant support architecture, discussed in section 5, is suitable for such applications.

7. Interoperability among cloud service providers

In context of cloud computing, interoperability can be viewed as the ability of public or private clouds and various enterprise systems to understand one another's services and application interfaces, settings, security parameters, etc. in order to interoperate and cooperate with each other [25]. The vital components of interactions are those belonging to cloud service client [25] which communicates with cloud service provider components. Hence, prescribed API's and interfaces are typical when we talk of interoperability. However, there is no standardization API or interface. As a result different service providers use diverse API's and interfaces. But still three most common interfaces are:-

- Functional Interfaces: Deals with major functionalities provided by cloud services.
- Administrative Interfaces: Deals with management, authorization and monitoring of services and their characteristics.
- Business Interfaces: Includes billing, invoice, subscription, information etc.

Degree of interoperability varies from one cloud service to another. IaaS provides highest interoperability since it has many de-facto or formal standard interfaces as well as equivalent functionality. PaaS has comparatively lower interoperability since few standard interfaces or open source projects have been devised so far. However, SaaS has the least interoperability as very few standard interfaces. It involves the transformation of on interface to another. Therefore, PaaS and SaaS present a greater challenge in achieving interoperability.

Further, application code in the cloud service is managed by client in IaaS and PaaS whereas it is under the control of providers in SaaS. It is controlled and implemented by the API's. Moreover, client operations use these API's for migrating, linking or utilizing the cloud services, thereby, challenging the interoperability.

As discussed in [25], few cases can be considered to understand the interoperability challenges and considerations. These include: -

- Client Migrates between the cloud service providers.
- Client uses cloud service from providers.
- Client links particular service to a different service.

7.1 Case 1

A client migrates from one cloud service provider to another in order to avail the benefits of wide market of cloud services and to eliminate the risk of lock-ins [25]. Table 1 discusses the various considerations while migrating between different service provider combinations and suggests their corresponding recommendations.

Table1.Client migrates between cloud service providers.

Considerations	Recommendations
From SaaS to IaaS or PaaS No portability of application code. Application odes may be completely different between two service providers. Hence, APIs not interoperable. Incompatible interfaces. Transformation of APIs which wastes efforts.	For SaaS, focus on standard and well defined APIs, data formats, interfaces and protocols.
From PaaS to IaaS: No portability of application code (rarely happens). Varies among different service providers.	For PaaS, make sure that open source projects and standards are used to avoid the need of transformation.
From IaaS to PaaS: Rarely challenging.	For IaaS, make sure that all service providers use standard and open source formats like OVF, etc.

7.2 Case 2

A client might use services from multiple service providers in order to avail the benefits of their unique abilities. Depending upon equivalent or different functionality between the two service providers, interoperability varies between their interfaces. Table 2 reviews the considerations while using services from different combinations of providers and suggests the corresponding recommendations.

Table 2.Client uses cloud services from different providers.

Considerations	Recommendations

Equivalent functionality	
Case 1: Interoperable interfaces and components	Case 1: Common interface or component can be used
Case 2: No interoperable interfaces.	Case 2: Client components will need to be updated to cooperate with both interfaces.
Differential Functionality	Implement an Enterprise Service Bus (ESB) [] to transform interfaces, data formats and protocols among service providers. Use a middle layer to identify and resolve such issues.
No interoperable interfaces though may be designed over common technology.	

Rest, recommendations for IaaS, PaaS and SaaS are same as discussed in case 1.

7.3 Case 3

Client can associate cloud services together for a particular application to create a more effective capability. To associate cloud services together, the second service provider must possess an API which the first service provider can use remotely [25]. Hence, type of first and second service provider affect the link between the two.

Table 3.Client links particular service to another service.

Considerations	Recommendations
When any one service provider is IaaS or PaaS then application code is under the control of client	API of IaaS or PaaS can be used by others.
When any one service provider is SaaS then application code is under the control of service provider	Requires pre-configuration of interfaces or services for use by second provider.
	Use standard or de-facto API.
	Make sure to use an intermediate layer to identify and resolve the interoperability issues.
	Focus on using SOA design [25] concepts

Consider the recommendations of case 2 also.

As discussed in [25], some standards have been devised to overcome the challenge of interoperability like: Open Cloud Computing Interface (OCCI), Cloud Application Management for Platforms (CAMP), Cloud Data Management Interfaces (CDMI), Open Virtualization Format (OVF), etc. Certain open source projects have also been designed like: OpenStack in IaaS and Docker, Heroku, etc. in PaaS.

8. Other related DBMS issues

Owing to the fact that many applications in cloud are extensively data driven, data management systems hosting these applications embody a vital component in cloud software store. However, maintaining performance of database read/write operations under fluctuating workloads, both regionally and globally, is quite challenging. In this context, distributed scalable data stores in cloud have promised high performance and reliable read/write intensive applications through rapid partitioning, replication, consistency measures and automatic managers for self-management of database. This section briefly reviews afore-mentioned concepts of partitioning, replication, consistency and automated manager.

8.1. Partitioning

Partitioning helps in determining the location of data on the different geographically distributed servers in cloud. Thus, it is one of the important features in deciding the read, write or data storage scalability [11] of the system. Partitioning of data can be achieved either through vertical partitioning or horizontal partitioning (also called sharding).

Vertical Partitioning is achieved by splitting columns (i.e. attributes) of database table into new subsets of columns such that there is a mapping between subset of columns and application functionality. As discussed in [29], the multi-table 'join' operations will now be carried within application code [29] and not over the relational schema; therefore it is essential to choose appropriate table and columns in order to create a right partition. Hence, column family data stores are capable of providing both vertical partitioning and horizontal partitioning.

Horizontal Partitioning or Sharding is achieved by splitting rows (i.e. tuples) of data base table into different tables with less number of tuples. Partitioning is achieved on the basis of shard keys which can either be directory-based, range-based or hash-based [29]. Based on the mapping of key and server mode, write requests are propagated to the appropriate server. Sharding based on later two keys is most commonly implemented by many major scalable data stores. Thus, two common techniques of sharding are: Range Partitioning or Consistent Hashing.

8.2. Replication

Replication is the mechanism of storing multiple copies of same data over different servers in order to execute read/write requests by distributing queries over replicas. Besides being an important feature in determining scalability, it is also an essential feature in guaranteeing fault-tolerance and affecting consistency level.

According to [11], the main approaches to replication can be differentiated as: master-slave, multi-master or masterless replication.

In master-slave replication, one node is designated as master and rest as slaves. Only master is responsible for processing the write requests and propagating data to slaves. Thus, the direction of propagation is always from master to slaves. Some of the data stores implementing master-slaves replications are: HBase, Redis, and Berkley DB.

In multi-master replication, any number of nodes can process the write requests and updates are then propagated to every other node. Thus, here propagation can be in any direction. Some of the data stores implementing multi-master replication are: Couchbase Server and Couch DB. Master-less replication is similar to multi-master approach except the fact that in former, all the nodes play same role in replication system [11]. Examples of data stores using master-less replication are: Cassandra, Voldemort and Riak. NewSQL data store achieve replications through Paxos state machine algorithm like in Google Spanner [26] or through transaction/session manager [11] like in Volt DB and Clustrix, etc.

Further, read and write scalability of data store is affected by the choice of replication approach. Master-slave provides read scalability but not write. However, multi-master and master-less replication provide both read and write scalability owing to the fact that all nodes are allowed to handle both read and write requests.

As discussed in [29], web applications are basically deployed over a multi–tier cloud architecture where every single tier is responsible for handling the functionalities, coordinating with other tiers and providing desired services to the clients. Therefore, replicating a single tier is not an effective solution to achieve desired scalability. Besides read or write requests, there are compute intensive and data intensive operations. Former needs more resources or scalability at application/logic tier and latter needs the same at data/persistence layer [29]. Moreover, in case of failures, interdependencies among tiers must not result in multiple execution of same workload at both application and database tier. Hence, considering above arguments, vertical and horizontal replication patterns are classified in [29].

Vertical replication pattern integrates one application and one database server into single replication unit which can be replicated vertically to achieve higher scalability. Here, replication logic is transparent to the replication unit, thereby enabling seamless working of the unit. However, it demands effective partitioning of application and its corresponding data to achieve expected scalability. Such systems are rarely used.

Horizontal replication pattern allows each tier to replicate independently and a replication awareness scheme [29] runs in between for coordinating the tiers. Thus, it provides flexibility to scale each tier independently but for effective performance the awareness mechanism is must, such systems are used almost everywhere.

In order to support the above two, various architectural decisions need to be considered like multi-tier coordination, determining the exact unit containing replica, dynamic partitioning, etc.

8.3. Consistency

ACID consistency implies that transaction drives database from one consistent state to another. However, BASE [31] captures the reasoning of CAP [27], [28] very closely. CAP consistency refers to how data seems among different server nodes after update has been performed. BASE states that for a system which is partitioned functionally, operations can be broken down and pipelined to generate asynchronous updates over different replicas. This ensures that clients wishing to achieve read access can be instantly serviced without letting them wait for completion. Such highly pipelined [29] systems with instant servicing raise various consistency issues thereby, evolving the concept of relaxed consistencies. Hence, consistency models can be categorized into strong/immediate consistency and eventual consistency.

Strong consistency makes sure that after writes are performed, same data is visible to all incoming reads thereafter. As discussed earlier, synchronous replication provides strong consistency but owing to latency introduced, it is not suitable for NoSQL data stores which demand high levels of performance. HBase is the only NoSQL database which provides strong consistency. Some other NoSQL databases like MongoDB achieves strong consistency by either setting read-only from master or setting write concern parameter to "Replica Acknowledgement" [11].

Eventual consistency [30] makes sure that user need not wait for data even though system is working at backend to achieve consistency. Updates are eventually propagated to all replicas over a period of time. Again as discussed earlier, asynchronous replication provides eventual consistency. Almost all NoSQL stores provide eventual consistency since they are asynchronously replicated.

However, many of NoSQL databases configure consistency models to provide strong consistency using various consistency guarantee mechanisms. As presented in [29], different forms of consistency can be combined to ensure consistency at client side as shown in table 4.

Table 4.Types of eventual consistency.

Types of Eventual Consistency	Functionality
Casual	If casual dependency exists between two processes then update committed by one will be rejected by another.
Read-your-Write	Makes sure that after update, subsequent reads will access that newly updated value.
Session	Ensures that Read-your-Write consistency is provided until the session expires.
Monotonic Read	Makes sure that once a read accesses specific value of data item then no subsequent read will access earlier committed value.
Monotonic Write	Ensures serialization of writes by the same transaction.

8.4. Automatic management

Managing large database management systems presents remarkable challenges in system monitoring, operations and management. Automatic manager is accountable for following:

- Monitoring system behavior.
- Tuning system performance.
- Elastic scaling.
- Load balancing on the basis of dynamic consumption patterns.
- Modelling system characteristics in order to predict the workload spikes.
- Taking effective control measures to deal with such spikes.

Further, in order to guarantee efficient multi-tenant performance and service level agreements (SLAs) [32], the manager must configure dynamic behavior and resource requirements of various tenants for elastic scaling. Also migration costs in context of decisions regarding where to migrate, when to migrate and which tenant to migrate must be predicted. Automatic manager comprises of two logical components: static and dynamic.

Static component configures characteristics of tenants and their resource utilization to determine their placement and identify co-located tenants with supplementary resource requirements. This configuration presumes that once tenant characteristics are modeled and their placement is identified, system will retain its behavior and is thus called static component.

Dynamic component configures entire system's characteristics to govern appropriate moment for elastic load balancing. It guarantees minimum changes in tenant positioning and rebalances load through live database migration. Dynamic component identifies dynamic transformations in load and resource usage characteristics.

9. Related work

Like Hadoop [21], there are related projects like HadoopDB [1] and Hive [3]. However, unlike parallel computing architecture discussed in section II, these perform batch processing instead of real time OLTP operations. Distributed B-Trees [5] are also used for cloud computing as in [20]. However, they are capable of processing only one-dimensional data which is not suitable for OLAP applications. Efforts have been made to design multi-dimensional distributed data structures like R-Trees [15] as in [9], [23] but they don't possess the ability to meet OLAP requirements.

MPI-based [12] parallel applications are compute-centric i.e. data fragments are moved to computations. For example: mpiBLAST [2] which is BLAST's [4], [18, 19] parallel implementation. Like architecture discussed in section III, it also makes use of scheduler but it eliminates overhead of data movement from shared network storage to computations by using local disk caches. This fails when database increases in size. Hadoop-BLAST [16], [24] also implements parallel Map-Reduce [22] for BLAST but does not include an I/O controller.

As discussed in [8], [11], existing NoSQL data stores differ on the basis of data model used, partitioning and replication mechanism used, level of consistency guaranteed, concurrency model used, query/API interface used and mechanisms to ensure network security.

SaaS adopted three schemas to implement multi-tenant DBaaS: Shared Machine, Shared Process and Shared Table [17], [24]. However owing to some discrepancies, they are no more suitable for multi-tenant cloud platform. In Shared Machine scheme, memory consumption increases with increase in number of tenants [24]. Shared Process scheme, on the contrary, has less memory consumption per tenant and provides independent logical tables for every tenant [17]. On the other hand, in Shared Table scheme, memory consumption remains constant even with increase in tenants and it provides better utilization of storage than other schemes. Even then it is suitable only when each tenant has a similar schema instance.

10. Conclusion

Architectures discussed in this paper implement either a scheduler or master node for workload distribution and scheduling. However, this centralized approach can result in a bottleneck for performance and dynamic scaling. A decentralized approach must be devised such that loading and assigning depends upon the frequency or usage patterns of client. This can be achieved by maintaining and monitoring logs for each tenant. Further, an intelligent module needs to be incorporated in these architectures which can determine between in-memory and on-disk DBMS implementations as per the tenant's service level objectives.

These architectures also lack in replication mechanism which is required for fault tolerance and efficient run-time load balancing. Similarly, a proper partitioning mechanism must be chosen such that there is transactional support for distributed operations among all scattered DBMS instances. Moreover, a flexible I/O interface must be designed which is capable of sustaining atomic and serializable transactions among heterogeneous databases to make them interoperable.

References

1 Abouzeid A, Pawlikowski KB, Abadi D, Silberschatz A and Rasin A. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. [Online] Accessed 28 Feb, 2015. http://www.vldb.org/pvldb/2/vldb09-861.pdf

2 Darling AE, Carey L and FengW-C. The design, implementation, and evaluation of mpiBLAST. [Online] Accessed 5April,2015 http://salsahpc.indiana.edu/csci-b649-2011/mpiBLAST.pdf
3 Thusoo A, Sarma JS, Jain N, Shao Z, Chakka P, Anthony S, Liu H, Wyckoff P and Murthy R. Hive-a warehousing solution over a Map-Reduce framework. [Online] http://www.vldb.org/pvldb/2/vldb09-938.pdf Accessed 5 April, 2015.
4 BLAST. http://en.wikipedia.org/wiki/BLAST [Online] Accessed 1 April 2015.
5 B-tree. http://en.wikipedia.org/wiki/B-tree [Online] Accessed 31 Jan 2015.
6 Dehne F, Kong Q, Chaplin AR, Zaboli H and Zhou R (2014) Scalable real-time OLAP on cloud architectures. In Elsevier 2014.
7 HDFS. [Online] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html Accessed 31 Jan 2015.
8 Hu H, Wen Y, Chua T-S and Li X (2014) Toward scalable systems for Big Data analytics: a technology tutorial. In IEEE 2014.
9 Wang J, Wu S, Geo H, Li J and Ooi BC. Indexing multi-dimensional data in a cloud system. [Online] Accessed 5 April, 2015. http://www.comp.nus.edu.sg/~ooibc/sigmod10rtcan.pdf
10 Yin J, Zhang J, Wang J and Feng WC (2014) SDAFT: A novel scalable data access framework for parallel BLAST. In Elsevier 2014.
11 Grolinger K, Higashino WA, Tiwari A and Capretz A AM (2013) Data management in cloud environments: NoSQL and NewSQL data stores. In Journal of cloud computing advances, system and applications.
12 MPI. [Online] http://computing.llnl.gov/tutorials/mpi/ Accessed 1 April, 2015.
13 MySQL Cluster. [Online] http://eprints.sics.se/237/1/elansary-singlespaced.pdf Accessed 12 April 2015.
14 Dharvath R and Kumar C (2014) A scalable genric transaction model scenario for distributed NoSQL databases. In elsevier 2014. Pg. 43-58.
15 R-tree. [Online] http://en.wikipedia.org/wiki/R-tree Accessed 31 Jan 2015.
16 Running Hadoop-Blast in distributed Hadoop. [Online] http://salsahpc.indiana.edu/tutorial/hadoopblastex3.html Accessed 12 April 2015.
17 Das S (2011). Scalable and elastic transactional data stores for cloud computing platforms. A dissertation.
18 Altschul SF, Madden TL, Schaffer AA, Zhang J, Zhang Z, Miller W and Lipman DJ. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. In Nucleic Acids Res. 1997 Sep 1; 25(17): 3389–3402.
19 Altschul SF, Gish W, Miller W, Myers EW and Lipman DJ. Basic local alignment search tool. [Online] http://myerslab.mpi-cbg.de/wp-content/uploads/2014/06/blast.pdf Accessed 20 April, 2015.
20 Wu S, Jiang D, Ooi BC and Wu K-L. Efficient B-tree based indexing for cloud data processing. [Online] Accessed 20 April, 2015. http://www.comp.nus.edu.sg/~ooibc/vldb10-cgindex.pdf
21 What is Hadoop? [Online] Accessed 31 Jan, 2015. http://www-01.ibm.com/software/data/infosphere/hadoop/
22 What is MapReduce? [Online] Accessed 31 Jan, 2015. http://www-01.ibm.com/software/data/infosphere/hadoop/mapreduce/
23 Zhang X, Ai J, Wang Z, Lu J and Meng X. An efficient multi-dimensional index for cloud data management [Online] http://idke.ruc.cn/publications/2009/X.Zhang-CloudDB-09.pdf Accessed 20 April, 2015
24 Luo Y, Zhou S and Guan J (2014) LAYER: a cost-efficient mechanism to support multi-tenant database as a service in cloud. In Elsevier (2014) pg. 86-96.
25 Interoperability and portability for cloud computing: a guide. (2014) A whitepaper. Copyright @ cloud standards customer council.
26 Corbett JC, Dean J, Epstein M, Fikes A, Frost C, Furman JJ, Ghemawat S, Gubarev A, Heiser C, Hochschild P, Hsieh W, Kanthak S, Kogan E, Li H, Lloyd A, Melnik S, Mwaura D, Nagle D, Quinlan S, Rao R, Rolig L, Saito Y, Szymaniak M, Taylor C, Wang R, Woodford D. (2012) Spanner: Google's globally-distributed database. Published in the proceedings of OSDI
27 Diack BW, Ndiaye S and Slimani Y (2013) CAP Theorem between claims and misunderstandings: what is too be scarificed? In IJAST vol. 56, July, 2013.
28 Gilbert S and Lynch N. Perspectives on the CAP theorem.
29 Kamal JMM and Murshed M. (2014) Chapter2 Distributed database management systems: architectural design choices for the cloud. Under Springer International publishing- Mahmood-ed, cloud computing, computer communications and networks.
30 Vogels W (2009) Eventually consistent. In ACM vol. 52 No.1 Pages 40-44
31 Pritchett D (2008) BASE: an ACID alternative. In ACM QUEUE 1542-7738. Pages 48-55
32 Agrawal D, Abadi AE, Das S and Elmore AJ. Database scalability, elasticity and autonomy in the cloud [Extended abstract] Technical report UCSB CS