ijecs open access www.ijecs.in
International Journal Of Engineering And Computer Science ISSN:2319-7242
Volume 2 Issue 1 Jan 2013 Page No. 229-233

A Survey on Cloud Database Management

Ms.V.Srimathi, Ms.N.Sathyabhama and Ms.D.Hemalatha

[1]Department of MCA, SNS College of Technology, Coimbatore, 641035, India
[2]Department of MCA, SNS College of Technology, Coimbatore, 641035, India
[3]Department of MCA.SNS College of Technology, Coimbatore, 641035, India

Abstract

Recently the cloud computing has been gaining significant importance in almost all the fields. In this survey we discuss the limitations and opportunities of deploying data management issues on cloud computing platforms. A cloud database management system is a distributed database that delivers computing as a service. The various issues related to security that has to be offered by the Database as a service is studied. Various operations such as Scaling, provisioning, performance tuning, privacy and backup. This study also reveals about consistency rationing and various adaptive policies.

Keywords Relational Cloud, Consistency, Privacy, DBMS

1. Introduction

DBMS which is an integral and indispensable component is outsourced is attractive because energy, hardware and DBaas (Database as a Service) are minimized. This survey deals with determining workload for multi tenancy environment, elastic scalability and an adjustable security scheme to run over encrypted data. This survey also studies about the efficient and scalable ACID transactions in the cloud by decomposing functions of a database storage engine into transactional component and Data Component.

2. Data Management in the Cloud

Cloud computing is elastic in the face of changing conditions. The payment is only for one needs and so increased resources can be obtained to handle spikes in load and then it can be released. It is desirable that a complete software stack with a restricted API so that software developers are forced to write programs that can run in a shared-nothing environment and thus facilitate elastic scaling [1].

2.1 Data management applications in the cloud

The two largest components of the data management market into the cloud are transactional data management and analytical data management. The transactional data management does not use a shared-nothing architecture. The main benefit of a shared-nothing architecture is its scalability [2]. It is hard to maintain ACID in the face of data replication over large geographic distances**.** Applications like airline reservations and employee data systems have been the bread and butter of database industry - but these applications need an RDMS and transaction processing capabilities. If these applications are to be deployed over the Cloud, it is essential to come up with a scalable consistency model over a scalable data store for the Cloud that supports referential integrity and a transaction mix with a majority of updates. This boils down to a problem of deploying databases over the Cloud without compromising on any existing features [3]. A good DBaaS must support database and workloads of different sizes. A DBaaS must therefore support scale-out, where the responsibility for query processing and the corresponding data is partitioned amongst multiple nodes to achieve higher throughput. The workload-aware partitioner uses graph partitioning to automatically analyze complex query workloads and map data items to nodes to minimize the number of multi-node transactions/statements.[4]. The system design of a the Relational cloud uses existing unmodified DBMS engine as the back-end query processing and query nodes. Applications communicate with Relational Cloud using a standard connectivity layer such as JDBC. Relational Cloud runs the databases with the same database server. A special driver is used to communicate with the front end for maintaining the privacy of the data. The database has one or more tables and an associated workload, defined as the set of queries and transactions issued to it. The database is partitioned into one or more pieces when the load exceeds the capacity of a single machine[5].

2.2 Database Partitioning

Relational Cloud uses database partitioning for two purposes: (1) To scale a single database to multiple nodes, useful when the load exceeds the capacity of a single ma-

chine (2) to enable more granular placement and load balance on the back-end machines compared to placing entire databases. The main notion of this partitioning is to minimize the number of multi-node transactions. Relational Cloud uses graph partitioning [6] to find ` balanced logical partitions, while minimizing the total weight of the cut edges. This minimization corresponds to find a partitioning of the database tuples that minimizes the number of distributed transactions. The output of the partitioner is an assignment of individual tuples to logical partitions. Relational Cloud now has to come up with a succinct representation of these partitions, because the front-end's router needs a compact way to determine where to dispatch a given SQL statement. Relational Cloud solves this problem by finding a set of predicates on the tuple attributes. It is natural to formulate this problem as a classification problem, where we are given a set of tuples (the tuple attributes are features), and a partition label for each tuple (the classification attribute). The system extracts a set of candidate attributes from the predicates used in the trace. The attribute values are fed into a decision tree algorithm together with the partitioning labels. If the decision tree successfully generalizes the partitioning with few simple predicates, a good explanation for the graph partitioning is found. If no predicate-based explanation is found (e.g., if thousands of predicates are generated), the system falls back to lookup tables to represent the partitioning scheme.[4].

2.3 Resource Allocation

Resource allocation is a major challenge when designing a scalable, multi-tenant service like Relational Cloud. Problems include: (i) monitoring the resource requirements of each workload, (ii) predicting the load multiple workloads will generate when run together on a server, (iii) assigning workloads to physical servers, and (iv) migrating them between physical nodes. The monitoring and consolidation engine includes Resource Monitor which captures a number of DBMS and OS statistics from a running database., a Combined Load Predictor which includes a developed a model of CPU, RAM, and disk that allows Kairos to predict the combined resource requirements when multiple workloads are consolidated onto a single physical server, a Consolidation Engine to: (1) minimize the number of machines required to support a given workload mix, and (2) balance load across the back-end machines, while not exceeding machine capacities.[4].

2.4 Privacy

CryptDB, To implement adjustable security, our idea is to encrypt each value of each row independently into an onion: each value in the table is dressed in layers of increasingly stronger encryption, as shown in Figure 2. Each integer value is stored three times: twice encrypted as an onion to allow queries and once encrypted with homomo phic encryption for integers; each string type is stored once, encrypted in an onion that allows equalities and word searches and has an associated token allowing inequalities.[4].

3. Replication –Fault Tolerance

A replica refers to a complete copy of the data. Fault-tolerance and availability is ensured by replicating (or caching) frequently used data across multiple locations. The downside of this is that co-ordination of the various replicas is a cause for overhead, possibly reducing system availability. This survey deals with elastic and scalable transaction management when databases are deployed as a service over the Cloud, without any loss of functionality. For DaaS (Database as a Service), a scalable transaction management paradigm is necessary; one which would work well even when the majority of the transactions involve updates.[3].

3.1 Architectural Study

A **node** (or a replica) is a virtual machine instance. Each node consists of 1. **Data Objects**: Data Objects are simply tables stored as .db files on Amazon S3. Each data file contains a single. A table has many attributes and rows (like a matrix). Each table row has a unique row number and contains the timestamp of the transaction which performed the last update. Data Objects are fully replicated over the network. Each row in the table has an entry for the last update time (or the value of the global time stamp of the transaction which last updated this row). 2. **Update Queue**: The Update Queue is the recovery mechanism present at each node. The transaction requests from the Transaction Managers (which are server-side virtual machine instances) are added to the Update Queue at each replica. 3. **Queue Manager**: The Queue Manager handles the Update Queue and runs the transaction with the smallest time-stamp. After successful commit, this transaction is dequeued. Other components of the system include: **Interest Group (IG)**: An IG is a group of replicas in which all (or a majority of replicas) have the most recent global timestamp. **Transaction Managers (TMs)**: The TM is a server-side virtual machine instance. A TM is very closely associated with the replicas. One TM is designated as the **Master TM (MTM)**. Read/write requests from the client are received by the TMs.[3]

3.2 Consistency Algorithm and Recovery Algorithm

The client must submit an estimate of the transaction workload initially. This is used to determine the number of TMs and number of nodes (total number of virtual machine instances) to be launched on each hypervisor [7]. After this initial startup phase, the system scales with the transaction workload following the pay-as-you-go model. Initially, all the replicas start with the same version number- Version 0. All the IG maps are reset. The replicas are randomly grouped together into an IG by the MTM. The starting size

of the IG is specified by the application developer. The MTM then sends its IG map to other TMs for synchroniz ation. A client request can go to any TM, thus ensuring that no TM is overwhelmed by more requests than it can service (load balancing). The TM which receives the client write request multi-casts the request to all the nodes in the IG. The TM is required to wait for acknowledgment from only the current replicas in the IG. The nodes in the IG forward the write requests to the other replicas after sending their acknowledgements to the TM. Inside the IG, the updates are applied in parallel to the replicas. Hence, the amount of time the client has to wait is bounded by the slowest replica in the IG. This makes the write latency of the IG equal to the write latency of the slowest replica in the IG currently. Write latency of a replica for a TM is the difference between the time at which a TM sends an update to this replica and the time at which this TM receives the acknowledgement. Write latency is variable due to the dynamic nature of the IG. Suppose the IG has m nodes and the write latency of ith node is ti and the write latency of the IG is TIG. Then,
TIG = *max* (t1, t2, t3, …, tm).
The updates are performed on those replicas which are currently not 'in use'. These updates go into an Update Queue at each replica. In each replica, the Queue Manager dequeues the update with the smallest timestamp from the Update Queue. If at least one of the rows affected by this update is ‘in use’ then the update must wait in the Update Queue. This ensures that the updates at all replicas are applied consistently and in order. Each queue acts as a log. Unlike the write operation, there is no multi-cast needed for a read operation. The data can be read from any current replica in the IG. Since there are two types of operations - read and write, this leads to following depedencies of operations on a single table row: *Read after Write*: A read coming into the IG when a write oper tion is being performed. The read can be issued on any replica on which this write has been performed and committed. *Read after Read*: A read coming into the IG when a read operation is being performed. The read can be issued on any replica. The TM sends a read request to only one replica. *Write after Read*: A write coming into the IG when a read operation is being performed. The writes can be issued on all the replicas except on which this read is being performed. The write on this replica is issued after this read is done. *Write after Write*: A write coming into the IG when a write operation is being performed. The writes can be issued on all the replicas except on which this write is being performed. The write on this replica (or replicas) is issued after this write is committed. TMs put each read/write request into the Update Queue at each replica in the IG. The Queue Manager at each replica must keep track of these dependencies while issuing the transactions from the Update Queue. This algorithm ensures that whenever a client issues a read operation, a consistent and correct value is read from the data store.

A study on RECOVERY ALGORITHM states that a failure is defined as a state in which a replica cannotservice any client request at all. Partial failures are not considered. When a node (inside or outside the IG) fails, a check is run by any one TM on the network and on the IG. This checks if the IG continues to have the minimum number of nodes and establishes if more nodes can be added to the IG if it is at less than the maximum capacity. This is called **reevaluation of the IG** and is performed by the MTM. MTM sends the new contents of the IG map to the other TMs.[3]

More than one node can be added to the IG after reevaluation. The re-evaluation of IG does not mean that the system is down. While a node is getting added to an existing IG, a client operation can still be performed on the remaining nodes in the IG. If operation was a write, it can be performed on the new node added to the IG only when its addition to the IG is complete. The algorithm presented in section III allows for a recovery scheme to handle failures. This is elaborated by considering the following possible events. When node(s) inside the IG fails: The IG is reevaluated. If no node can be added to the IG and the IG has less than the minimum number of nodes, then the system is unavailable and does not service any client request until a new IG is formed. When node(s) outside the IG fails: The IG is reevaluated. When node(s) comes up: The IG (either existing or see if a new one can be formed) is re-evaluated. This node can become a part of the existing IG or forms a new IG. The above scenarios can be thought of as interrupts to the system. Each of which will lead to re-evaluation of the IG. Queue Manager: Each replica has a Queue Manager. As discussed earlier, the updates at each replica are first added to a queue, before applying to the respective table row. This queue called the Update Queue which ensures that the updates are applied in order of time. The queue also serves as a log of transactions which can be used when the replica comes up after failure. Let there be two replicas A and B. The version of the data file at B is V1 and that at A is V. If V1 is the most current version of the file, then B's update queue would be empty. If it is not the most current version of the file and suppose V is the most current version of the file then V1 would need, say, n updates to reach V or simply V = V1 + n. Now, suppose B fails and is down for m units of time. In this time, the version at A would advance from V to a new version V2. Say there were n1 updates done during this time. Now the current version at this time is V2 and V needs to have (n + n1) updates to become V2 when B comes up. Now, the question to be answered is: From where would B get the updates after it comes up? To go to the version V1, it simply needs to de-queue all the updates in its update queue when it failed. Then to go to V2 it needs to ask A to send it the modified tuples or all the tuples. This being an operation conducted infrequently, should not affect the performance of the system greatly. If this is done, then B does not need to use its update queue at all. The updates from the TM(s) would still be coming to the replica B and would be added to its update queue. So if B comes up at time t, then it asks any replica in the IG for the most updated data. Until the file at B comes to the current version at the time t, the Queue Manager does not ap-

ply the updates to the table at B. When the IG fails (which implies that no node outside the IG can be added to the IG after IG re-evaluation), there is no need for the transmission of updates from other nodes as the IG nodes would already be current when they come up. A more general form of this is when the entire system fails; each replica which is not current has to do what B does in the above example. But these types of failures would be unlikely. When a node(s) executing a data operation fails: Reads can be forwarded to another node with the current version of the replica. This is done by the TM, when it does not get any acknowledgement from this node. In case of writes, the time stamp of the transaction is read from the front of the update queue and a local undo operation is executed on the corresponding row in the table. When a TM fails: Each Cloud node has one TM as a virtual machine instance and any number of virtual machine instances as nodes or replicas. When a TM fails, a new virtual machine instance is started and it has the same state as the failed TM. It continues processing from the time the TM had failed. So when the system is down when either all replicas are down or there is no functional IG in the network (There might be current replicas active even without an IG in the network. This is a tradeoff between what the minimum size of an IG should be and also the maximum size. It is application dependent and it is best if the developer selects it.) The algorithm is dynamic and reflects the current state of the network. The size of the IG is a function of the number of working nodes. As long as the IG is up and running, the client finds the system available and reading consistent values. The client latency depends on the current size of the IG currently. Since the size of the IG is dynamically dependent on the number of working replicas it is safe to say that the client latency also depends on the number of working replicas. Clients can get concurrent reads and writes which happen in parallel. When a replica in the IG fails, the IG is re-evaluated and possibly more replicas are brought into the IG hence the performance does not suffer for each IG replica failure. The replicas outside the IG (which are up and running) also keep on applying the updates but the TM does not wait for acknowledgments from them. But they are not left behind as the updates are sent to them as well. The system keeps evolving and ensures immediate consistency for the IG and eventual consistency for replicas outside the IG.[3].

4. Consistency Rationing

Consistency rationing as a new transaction paradigm, which not only allows to define the consistency guarantees on the data instead of transaction level, but also allows to automatically switch consistency guarantees at run-time. That is, consistency rationing provides the framework to manage the trade-off between consistency and costs in a fine-grained way. only two levels of consistency (session consistency, and serializability) and divide the data into three categories. Session consistency has been identified as the minimum consistency level in a distributed setting that does not result in excessive complexity for the application developer. Consistency Rationing in analogy to Inventory Rationing [9]. Inventory rationing is a strategy for inventory control where inventories are monitored with varying precision depending on the value of the items. Following this idea, the data is divided into three categories(A, B, and C), and treat each category differently depending on the consis tency level provided. The A category contains data for which a consistency violation would result in large penalty costs. The C category contains data for which (temporary) inconsistency is acceptable (i.e., no or low penalty cost exists; or no real inconsistencies occur). The B category comprises all the data where the consistency requirements vary over time depending on, for example, the actual availability of an item. This is typically data that is modified concurrently by many users and that often is a bottleneck in the system. [11]

4.1 Adaptive Policies

The Time policy switches between guarantee levels based on time, typically running at session consistency until a given point in time and then switching to serializability. These two first policies can be applied to any data item, regardless of its type. For the very common case of numeric values (e.g., prices, inventories, supply reserves), we consider three additional policies. The Fixed threshold policy switches guarantee levels depending on the absolute value of the data item. Since this policy depends on a fixed threshold that might be difficult to define, the remaining two policies use more flexible thresholds. The Demarcation policy considers relative values with respect to a global threshold while the Dynamic policy adapts the idea of the General policy for numerical data by both analyzing the update frequency and the actual values of items.[11]

5. Conclusions

Database Management Systems as a cloud service are engineered to run as a scalable, elastic service available on a cloud infrastructure. CloudDBMSs will have an impact for vendors desiring a less expensive platform for development. In this paper, we presented the idea of DBMS in the cloud, the possibilities to be offered as one of the services offered by promising capability of cloud computing, that is to be a DBMS as a Service

References

1. Daniel J.Abadi and New Haven .Data Management in the Cloud: Limitations and Opportunities
2. S.Madden,D.DeWitt, and M.Stonebraker. Database parallelism choices greatly impact scalability.

3. Database column blog. http://www.databasecolumn.com
4. Rohan G. Tiwari, Shamkant B. Navathe and Gaurav J. Kulkarni, TOWARDS TRANSACTIONAL DATA MANAGEMENT OVER THE CLOUD
5. Carlo Curino,Evan P.C.Jones ,Relational Cloud: A Database-as-a-Service for the Cloud
6. F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows,T.CChandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In OSDI, 2006.
7. G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput., 20(1), 1998
8. "Hypervisor", December 2004. [Online]. Available:http://en.wikipedia.org/wiki/Hypervisor. [Accessed: Apr. 23, 2010
9. Andrew S. Tanenbaum and Maarten Van Steen. Distributed Systems: Principles and Paradigms. Prentice Hall, 2 edition, 2006
10. Edward A. Silver, David F. Pyke, and Rein Peterson. Inventory Management and Production Planning and Scheduling. Wiley, 3 edition, 1998
11. Building Database Applications in the Cloud A dissertation submitted to the SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH by TIM KRASKA
12. BuyyaR, BrobergJ andGoscinskiA, "Cloud computing Principles and Paradigms", A Jon Wiley & Sons, Inc. Publication, **(2011).**
13. FeinbergD, "DBMS as a Cloud Service", **(2010)**,Gartner, Inc. and/or its Affiliates.
14. KelloggD, "DBMS in the Cloud: Amazon SimpleDB", http://kellblog.com/2007/12/18/dbms-in-the-cloud-amazon-simpledb/