Exploiting the Social Structure of Online Media to Face Transient Heavy Workload

Ibrahima Gueye and Idrissa Sarr
LID laboratory
Université Cheikh Anta Diop
Dakar, Senegal
(ibrahima82.gueye, idrissa.sarr)@ucad.edu.sn

Hubert Naacke
LIP6 Laboratory
Sorbonne Universités, UPMC Univ Paris 06
Paris, France
hubert.naacke@lip6.fr

***Abstract*—A challenging issue many online social media have to deal with is facing egocentric workloads that are very frequent. Such a situation is generally due to the simultaneous access of several users to a small piece of data owned by a user or a few ones. A key example is the number of comments posted on the Manchester United Facebook page after the manager announced his retirement (more than 1 billion comments on the related subjects). Since egocentric workloads are transient, two dimensions must be taken into account to deal with them: (1) the rapidity to react to the peak load and, (2) the lightness of the solution or its low cost. Therefore, the first goal of this paper is to exploit the underlying social structure of online social media to localize from which the peaks take place and to face them in their early stage. The second goal is to combine an elastic approach with a load balancing process to sustain the overall performances while minimizing the required resources. Our solution is evaluated through simulation with SimJava. The obtained results show the soundness of the approach as well as it feasibility.**

Keywords–Transaction, Social workload, Load balancing, Elasticity.

I. INTRODUCTION

Social media applications are characterized by online collaborative actions such as chatting, tagging and content sharing. The user experience is more and more guided by her social context or social position, i.e., a user with many connections tends to be involved in frequent online interactions. As reported in [1], the data belonging to the most popular users are the most frequently accessed. Furthermore, when a popular user acts in response to another user's action, this can causes other users to respond subsequently, generating a so called net effect. As a result, users may simultaneously access the same piece of data for a short period of time. We say that we face a set of *egocentric workloads* that are characterized by a socially dependent and fluctuating access pattern. The reason is that the overall workload derives from few users and their close contacts based on the status or role of users. To face egocentric workloads, a challenging issue is to deliver fast, scalable and cheap data access, using a reduced amount of resources.

A. Motivations and Problem Statement

The interactions between users as well as the actions (comment, tag, etc.) made by a user on the items owned by others shape the well-known social structure. This structure is generally represented as a graph of a set of vertices with edges between them. Vertices are users or their items while edges are interactions or links between users. The number of neighbors or edges of a user is called *centrality degree*. A node with a high number of neighbors is called a popular or important node and has therefore a high centrality degree value. Less important nodes are called peripheral nodes. Figure 1 depicts a social network where big rings represent popular users and small rings designate peripheral users.

It is obvious that popular users are involved more frequently than peripheral ones in online interactions. That is, paramount of the workload derives essentialy from popular users and is the main reason we characterize the social workload as a set of egocentric workloads. Furthermore, an egocentric workload is transient since users behaviors are event-dependent and old events attract less attention leading to a disappearence of the related workload.

However, based on the interactions of users or their similarity, nodes can form groups for which the network connections are dense, but between which they are sparse. Such groups are called communities [2][3] or circles as in Google+. For instance, Figure 1 shows different groups: users within a group have a similar color. Moreover, users interact more with their neighbors within a circle than with others belonging to another circle. Thus, it is worth-noting that the overall workload is biaised since the social position of a user as well as the size of its group impacts the number of interactions within the circle. Whatsoever the particularity of the social workload, it is made of by read and write intensive operations since (1) the overall number of users is very important and (2) almost every user action causes data read, insert and update. That is, even though the actions or interactions done by users are socially dependent, the generated workload is quite the same as the workload of classical applications (*i.e.,* set of read and write operations). Therefore, egocentric peak load observed from social applications can be handled by using and adapting traditional techniques such as data partition and replication. The main issue to address therefore is how such techniques can be used for facing peak load while including social features. To face this issue, three problems may be formulated as follows

- How to detect data causing transient workload in its early stage within a social network?
- How to partition such data while ensuring fast interactions between a user and its contacts?
- How to forecast data that will cause a peak and to anticipate it based on the social structure?

Here are the set of problems we unveil and that we want to deal with through this paper. It is worth noting that even though the peak load is transient, it lasts thousand times longer

 ISBN: 978-1-61208-334-6

Figure 1: A social network graph.

than the time to execute a transaction. That is, partition and replication done for facing a peak load are cost for value.

B. Contributions and paper organization

Our goal in this paper is to face the previous problems and the key novelties of our approach can be summarized as follows.

- A fine-grain identification of a peak load. Actually, we propose 1) a naive approach that relies on the number of transactions accessing a partition, and, 2) a social-based approach that uses interactions between users. The last approach is in fact a peak prediction model and it is designed using the homophily principle, which states that the flow of information from person to person is a declining function of distance in Blau space [4]. That is, it is possible to locate the scope of interactions initiated by a user, and to assess whether such interactions may lead to a peak.
  Moreover, since social network is composed by a set of communities, peak origins are located in those communities. Such a mechanism has the edge to isolate the peak origin and to face it locally.

- A lightweight data migration method that moves only relevant data, on a pull-on-demand basis, with minimal disruption on transactions processing. The data migration method is coupled with an elastic load balancing mechanism that is optimized for reducing resource usage while maintaining bounded response time.

The rest of this paper is structured as follows: in Section II, we present the the social workloads, basic concepts and global architecture of our system. In Section III, we show how we detect peak load as well as the prediction model we use to anticipate their appearance. In Section IV, we present the management of transient heavy workload. In Section V, we present the validation of our approaches and we highlight, in Section VI, a few related works before we conclude in Section VII.

II. Basic Concepts and Global Architecture

In this section, we describe the global architecture we use and the social workloads we plan to face.

A. Social workload

The workload is made of user actions. A user action is a sequence of transactions and we assume that each transaction reads and writes data owned by a single user. A user may share data with other users and grants consequently read and/or write permissions on them.
Actually, with a social networking website as Facebook or Google+, users have a various sort of data that may concern distinct subsets of their contacts. In other words, the user belongs to several circles. For instance, users may have professional circles that contain their items related to their professional activities and that will attract more their colleagues and collaborators. They can share a private circle with only their close friends and relatives. Therefore, the items of one user may be seen as a set of cohesive data that are more attached to a specific circle.
Furthermore, the workload looms from users with various popularity levels. Thus, a peak load may be observed on so called popular data belonging to popular users. Since all popular users are not active at the same time, therefore, the overall workload is not distributed uniformelly over circles as well as over popular users (say we face a non-uniform distribution of the workload). In other words, the workload of the group i can be light while the one of the group j is heavy. With this insigth, it is trivial to identify groups with peaks or those underloaded. Getting this kind of information has the edge to apply a selective mechanism to face peak load within a group while minimizing the cost and required time.

B. Architecture

We devise an architecture using two layers: the routing layer and the datastore layer (see Figure 2). Our solution is a middleware that serves as an interface with the data manipulation procedures of applications. The routing layer is made of a set of nodes called client nodes (CN) and routers while the datastore layer contains database nodes (DB) that store data and execute queries. Data are stored on DB nodes by using community or cricle configuration in such a way that all related data of one group are on the same DB node. This is possible since the number of users within a circle is generally limited and hence, the related data can be hold in one single DB node. Transactions are sent by CN to any router, which afterwards forwards them to the right DB based on their access classes for execution. Note that each router stores a part of the global index, which allows them to locate data among database nodes. Transactions accessing the same data are routed in a serial way and the DBs guarantee consistent execution of transactions without locking.

Moreover, the routing layer includes a special and useful node called *Controller node (CtlN)*. It monitors the database layer for detecting whether a DB becomes a bottleneck or tends to be underloaded. In this respect, every DB sends periodically its load to the CtlN in order to permit overload detection based on a threshold. We mention that once a DB is found as overloaded, a migration process consisting of moving part of its data to a less loaded DB or a new one is initialized.

 ISBN: 978-1-61208-334-6

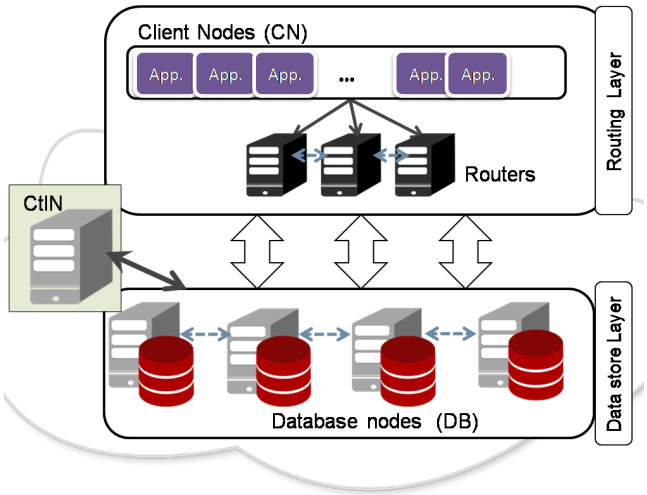

Figure 2: System architecture.

Reversely, an underloaded DB will bring out the merge of its data with another DB that has not enough load.

Furthermore, DB nodes are able to communicate between them for ensuring data migration from one to another in a consistent manner. To figure out the part of data to migrate from one to another DB, we rely on metadata hold via a data structure called *trace*. The *trace* records information about transactions such as their identifiers, their arrival dates, their waiting times.

III. Peak load detection

A. Definitions

We consider a set of nodes $\mathcal{N}$. Each node $N_k \in \mathcal{N}$ is a (virtual) machine managing the database DB_k. Each DB_k node stores a set of partitions, p_i^k denoting the partition i of database k. During operation, a DB node executes the incoming transactions in sequence. Let ω denote the most recent observation window, expressed in second. Each DB node logs the incoming transactions requests: $T_\omega(p_i^k)$ denotes the set of transactions requesting the partition p_i^k, which either terminated during ω or are not currently terminated (i.e., pending or running transactions). The log informs about the current execution time and waiting time of each transaction. To quantify the node load, we aggregate recent log information, let $RT_\omega(p_i^k)$ denote sum of the execution and waiting times of all transactions in $T_\omega(p_i^k)$.

We define $load(p_i^k)$ as the mean load of p_i^k within ω as:

$$load(p_i^k) = \frac{RT_\omega(p_i^k)}{\mid \omega \mid} \quad (1)$$

Since a DB node may store many partitions based on its storage capacity, we define the load of a DB node as the sum of the loads of all partitions under its control. Formally, the load of a DB_k holding n partition is:

$$load(DB_k) = \sum_i load(p_i^k) \mid p_i^k \in DB_k \quad (2)$$

Let τ_k be the standalone transaction processing time at node DB_k. Let rt_k be the observed transaction response time (including the waiting time). The $load(DB_k)$ can be considered as a penality factor impacting rt_k as follows: $rt_k = \tau_k.load(DB_k)$

B. Detecting peak load

We define the stability conditions of every DB node as the conditions under which it is neither overloaded nor under-loaded. More precisely, we expect every transaction to be executed in bounded time. Let T_{max} denote the maximum expected response time of a transaction. For each DB node, we expect $rt_k \leq T_{max}$, that is, the following condition must hold:

$$load(DB_k) \leq \frac{T_{max}}{\tau_k} \quad (3)$$

Reversely, a node is considered under-loaded if it remains idle (i.e., no transaction execution). Thus any DB node must satisfy the following condition:

$$load(DB_k) > 0 \quad (4)$$

A node is detected as overloaded (resp. idle) if the condition (3) (resp. (4)) does not hold for a given amount of time $\omega_{overload}$ (resp. ω_{idle}). It is worth noting that T_{max} as well as the size of the time windows $\omega_{overload}$ and ω_{idle}, are key performance indicators. T_{max} can be set based on the SLA of the of cloud provider, while ω values are tuned in order to make accurate decisions.

C. Identifying peak origins

A peak load occurs at a DB node if one or many partitions are overloaded, resulting in slow response time. With this respect, finding the origins of a peak can be summarized intuitively as identifying the sufficient set of partitions, with the highest load, that correspond to the extra load Δ_{load} defined as:

$$\Delta load_k = load(DB_k) - \frac{T_{max}}{\tau_k} \quad (5)$$

For each overloaded DB_k node, we sort its set of partitions $\{p_k^i\}$ in descending order of $load(p_i^k)$. Then, we determine a subset M_k of $\{p_i^k\}$ such that:

$$\sum_{p_i^k \in M_k} load(p_i^k) \geq \Delta load_k \quad (6)$$

Notice that the size of M_k is minimal since M_k is a prefix of the ordered set $\{p_i^k\}$. This aim to further reduce the number of partitions to move. Next, we will move iteratively all partitions in M_k, to restore DB_k in a normal load status.

D. Predicting peak origins

The peak origin is identified previously based on the load of the partition. That is, we gather some statistics during a set of time windows before being able to detect peak. Briefly, we detect peak after their arrivals. However, we should be able to detect whether a pic will arrive soon based on the social interactions. We aim in this section to predict when a peak can happen based on data social characteristics or the social graph.
In fact, since each interaction corresponds to a transaction, thus, it is possible to estimate the number of transactions that are related to a user u_i and his/her related data. Therefore,

 ISBN: 978-1-61208-334-6

the partition that stores data of u_i can be continually checked whether it is overloaded or not.
To reach our goal, we rely entirely on the homophily principle, which states that the flow of information from person to person is a declining function of distance. Thus, using the degree centrality of nodes or its neigbhorhood can help to locate the scope of interactions initiated by a user, and to assess whether such interactions may lead to a peak.

We start from the point that each neighbor of u_i may either partake to u_i's activities or not. Let p_m be the probability that m neighbors are involved in a given activity a_i of u_i. Thus, the number of transactions associated with a_i is approximatively equal to m when considering that each neighbor of u_i reacts only one time to a_i. Therefore, it is trivial to express the social load $L_s(p_i^k)$ of the partition p_i^k storing u_i's data by using Equation 1. For sake of presentation, we assume $L_s(p_i^k) = \chi.m$, where χ is the mapping function that expresses the load in terms of response time. Once $L_s(p_i^k)$ is obtained, we consider two possible states for p_i^k: acceptable (A) and overloaded (O). The partition is overloaded if $L_s(p_i^k) \geq T_{max}$. Given an activity a_i of u_i, its neigbhors can either partake to it or not, thus we can define a set of Bernoulli random variables $X = X_1 + X_2 + ... + X_n$ where X_l is representing a situation in which actor l have participated to a_i. Hence, the sum of such independent variables S follows the *binomial distribution* $\sim B(n, p)$, from which we derive the probability p_m of having m actors during an activity.

$$p_m = p(X = m) = \binom{n}{m} p^m (1-p)^{n-m}, \tag{7}$$

where p is the probability that an actor participates to an activity and $(1 - p)$ the probability it misses it. Moreover, the probability of having m neighbors interacting during N activities is

$$\prod_i^N p_m. \tag{8}$$

Once this probability defined, we set a threshold γ beyond which the likelihood of having $L_s(p_i^k) = \chi.m$ is very high, i.e., the pic load prediction is more accurate. Formally,

$$\prod_i^N p_m \geq \gamma \tag{9}$$

with γ a threshold based on the average participation rate of users. This approach has the advantage to take into account interactions of social network and therefore helps to forsee a peak once some users start interacting.

IV. Facing transient workload

The main idea of facing transient workload is to migrate data of overloaded partitions to a less loaded DB. To this end, we proceed by selective fragmentation and migration that directly takes the overloaded partitions and distribute them to less loaded databases. The problems we face are twofold: 1) identify the database candidate that will receive the extra load and, 2) process the migration mechanism.

A. Naive Identication of DB candidates

The basic and naive approach consists of using the less under-loaded DB as candidates to receive extra loads. Basically, when a database is chosen to receive a load from another database, it must remain not overloaded. The naive algorithm of facing a peak of DB_k works step by step as follows:

- For each p_i^k in M_k (see section III-C) that is overloaded, evaluate its load;
- Find all DB candidates that are not overloaded and able to receive $load(p_i^k)$ without being overloaded afterwards. In fact, DB_d is a candidate destination to receive $load(p_i^k)$ if:

$$load(DB_d)) \leq \frac{T_{max}}{\tau_d} - load(p_i^k) \tag{10}$$

- If there is no database able to receive p_i^k, then the condition (10) is checked for p_{i+1}^k.
- After each p_i^k migration, M_k is updated. If M_k still not empty and if no database is able to receive its content then we start a new DB instance and the M_k's content is allocated to it.

This approach has the edge to be simple and easy to be implemented and works well for rather regular, slowly fluctuating, workloads. However, in case of stressed workloads generating frequent peaks, this may lead to cascading migrations, i.e., migrating data from a DB that just previously received data from another DB. We need to better anticipate workload fluctuations in order to avoid overloading a DB, which receives an extra load recently. To this end, we propose to take into account the social characteristics of the data. We will exclude a DB candidate that stores data of important users, since it has a high probability to become overloaded in a near future.

B. Social-based identification of DB candidates

As mentioned before, important users hold data that are usually the peak origins. An intuitive approach can be to stave off gathering data of several important users in the same database. To this end, we rely on the user interactions graph when migrating data. That is, before moving data of DB_1 to DB_2 we check if the latter does not have important users and if neighbors of such users are likely to participate to activities. In fact, we replace step 2 of the naive approach by an identification method based on the prediction model that uses graph interactions. A DB is a candidate if the data of users it holds respect the model. The other steps are kept unchanged.

C. Migration process

We use a similar migration mechanism as in Relational Cloud [5] and ElasTraS [6]. Data is lazily fetched from the source as needed to support transactions on the destination. In-flight transactions are redirected to the destination. Once the data to be migrated and their destination DB are identified, the migration process starts; it consists of two steps:

- First, the initialization step. The source DB informs its associated router and the destination DB. The router spreads this information to the other routers. The indexes, which locate the data partitions are updated. From now, the future transactions on that data are

 ISBN: 978-1-61208-334-6

redirected to the new destination DB. This initiates load balancing. However, at this stage, no data is transmitted yet. Only the indexes are updated and the destination DB is ready for inserting new data when needed.

- Once the initialization step is done, all the transactions accessing the migrated data are routed to the destination DB. When the destination DB receives a transaction on the migrated data, it pulls the pieces of the data that the transaction intends to access, from the source and write it to the DB destination. And so forth, the pieces of the migrated data are transmitted to the destination DB on demand until completion. This pull on demand migration process has the advantages (i) to allow the source DB to alternate data transfer and transaction processing, which reduce the overhead (waiting time) due to the migration; and (ii) to migrate just the needed part of the data. In fact, even if a data have to be migrated, only actually accessed pieces of it are transmitted from the source DB to the destination DB. This aims to reduce the amount of transmitted data, saving communication resource.

• *Preventing continuous migration of a partition* We assume that if a partition p_i^k of database DB_k is migrated to another newly initialized DB_z then the maximum load of that partition p_i^k could not exceed the DB_z capacities. That means this partition won't be migrated anymore for overload reasons.

D. Example

Given the database $\{p_1..p_{10000}\}$. Each p_i represents one users' data. The data are distributed among three DB nodes DB_1 to DB_3 of various processing capacity. The standalone processing time of a single transaction at DB_1, DB_2, and DB_3 is $\tau_1 = 20ms$, $\tau_2 = 10ms$, $\tau_3 = 8ms$ respectively. The users require a maximum response time T_{max} equals to 100ms. Thus, the maximum supported load at DB_1 is $\frac{T_{max}}{\tau_1} = 5$. Respectively, DB_2 and DB_3 support a maximum load of 10 and 12.5.
During operation, each DB node process incoming transactions in sequence, queuing pending transactions. We see in Figure 3, two (red colored) transactions pending at node DB_1: they will wait too long and exceed T_{max}; such case requires data migration. To this end, during the last period ω $(10s)$, we measure at node DB_1, the following workload values: $load(p_1^1) = 1.4$, $load(p_7^1) = 2.1$ and $load(p_8^1) = 3.5$, which sums to $load(DB_1) = 7$. The amount of extra load at DB_1 is:

$$\Delta_{load_1} = 7 - \frac{T_{max}}{\tau_1} = 2$$

The smaller $load(p_i^1)$ value greater than Δ_{load_1} is $load(p_7^1) = 2.1$. Thus, migrating p_7^1 will cause to drop 2.1 of extra load, and to return under T_{max} response time.
Finally, to find a destination candidate, we measure the amount of $\Delta load$ that non-overloaded DBs could accept: $\Delta load$ values are 1 for DB_2, which is too small compared to our need, and 2.5 for DB_3, which suits our need. The migration operation give the result we can see in Figure 4.
Note that while estimating the availability a DB node (i.e., the maximum load it may accept), we take into account the predicted load of its important users so to prevent cascading migrations.

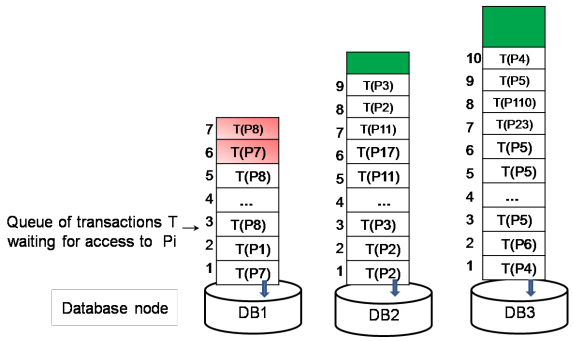

Figure 3: Example: State before migration.

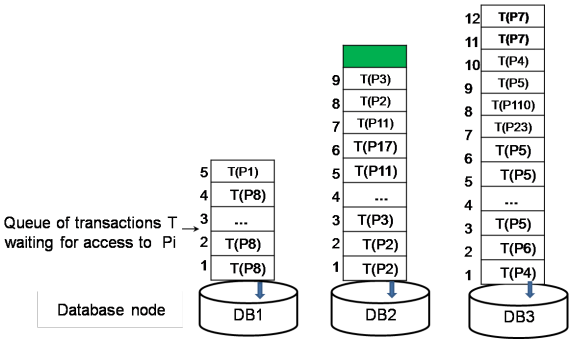

Figure 4: Example: State after migration.

V. Validation

In this section, we evaluate the overall performances of our approach and we answer the following questions: (i) how long does it take the system to respond to an overload? (ii) What is the cost of facing to an overload? (iii) What is the impact of the reconfiguration on transaction latency? (iv) Does the system can ensure a response time below a given thresholdd for more than 90 % of cases? (v) How does the number of DB evolve, depending on the workload?
We begin by describing the environment and the tools used during the experiments. Afterwards, we describe the experiments and their results while answering the unveiled questions.

A. Experimental setup

We use Simjava [7] to simulate our approach and to evaluate it. Simjava is a toolkit (API Java) for modeling complex systems. It is based on discrete events simulation and includes facilities for representing simulation objects. We implement each of entities: *Client nodes (CN), Routeurs, Database nodes (DB) and the Controller node (CtrlN)*. We use during the simulations, some data structures to represent the storage layer. Each entity is embedded into a thread and exchanges with others through events. During simulation experiments, we focus on the reaction of our system against an heavy and transient workload, and the evolution of the DB nodes. The experiments were conducted on an Intel duo core with 2 GB of RAM and 2.66 GHz running under Ubuntu LTS 12.04.

In the experiments, we implemented the social workload using the algorithm described below. In all experiments, we

 ISBN: 978-1-61208-334-6

started with a small size system (relatively to the number of running machines): one Router and two DB nodes, which store all the data. All transactions are read and write accesses.

1) Algorithm for workload generation

We aim to generate the load peaks in a controlled way, ensuring that the load at each partition (i.e., $load(p_i^k)$) corresponds to a given value. Indeed, the load values at each partition characterizes the workload pattern. We propose an algorithm to generate the workload, which follows the biased (e.g., power-law) pattern of social networks, as described in [1], i.e., the data of users with a high centrality degree receive a higher load. Accordingly, we control how many users are concurrently accessing each partition. Without loss of generality, we assume that a user involved into a peak load is accessing only one partition, as long as the peak occurs. Notice that a user may still access several partitions if her session lasts longer than the peak duration.

More precisely, we design an algorithm to assign users to partitions. We also change this assignment dynamically such that the overall data access frequency follows the same distribution as the underlying centrality degree distribution. Our algorithm takes as input the number m of partitions, the number U of users, the peak duration D in seconds, a distribution function f (*e.g.* the zipfian function), and the amount of work to do in a run expressed as the number N of transactions to process.

- We assess the distribution of the N transactions to the m partitions. We use f to get the number of accesses A_i^k to each partition p_i^k, such that the sum of all the A_i^k values equals N.
- We distribute the U users to a subset of the partitions such that there is A_i^k users per partition, and the sum of the A_i^k of the partitions in that subset equals U.
- For D seconds, every user submits a sequence of transactions to its assigned partition.
- Then we redistribute the users to another subset of the partitions; then we continue the run for D more seconds, redistribute again, and so on until all the partitions have been accessed by A_i^k users.

B. Experiments

1) Reaction to unexpected overload

The objective of this experiment is to assess how our system reacts against a higher load and how fast this is done. The results of the experiment are depicted on Figures 5 and 6. Data are horizontally partitioned and each DB node stores 50% of the overall data. We consider a heavy load on a small range of data that does not change during the experiment. Since we use a zipfian distribution to generate this range of hot data, most of them are for the first hundred users and are stored on DB0. We set a threshold of one second for the response time. We observe that the system handles the overload situation a few seconds after it starts: hot data are identified and the load is balanced between DB0 and DB1. Actually, after a peak arrival, one can see it does not take a long time to our system to deal with the peak and to stabilize the latency. The difference between results obtained on DB0 and DB1 is mainly due by

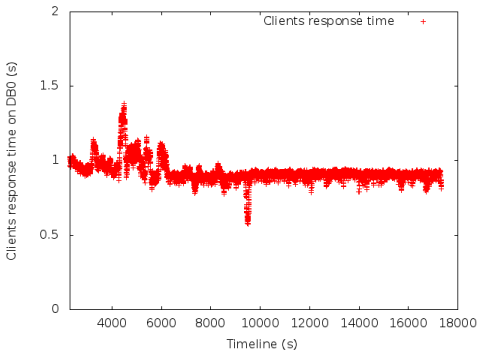

Figure 5: Reaction to a high and subite load at DB0.

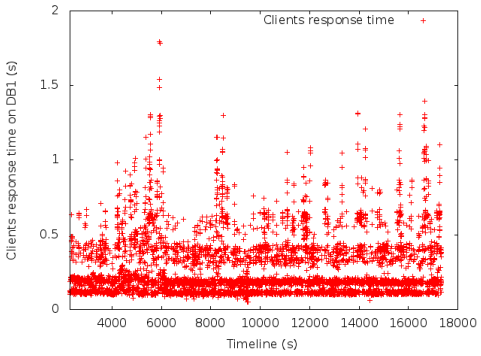

Figure 6: Reaction to a high and subite load at DB1.

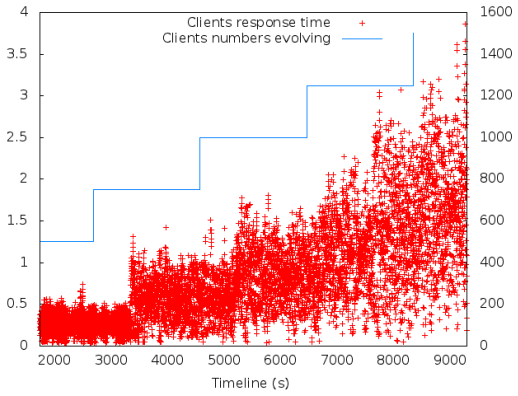

Figure 7: Not controlled: Response times on DB0 increases rapidly while the number of users increases (peaks are on DB0).

the fact that data are moved from DB0 to DB1. That is, it will take more time to DB1 to store arriving data and to process afterwards transactions on such data. Moreover, the workload on DB0 is slightly more dense than the one on DB1.

2) Not controlled vs Controlled system

This experiment highlights the impact of controlling the peak load on the performances. In this respect, we compare our system with another one without a peak load management. Results of such comparison are shown on Figures 7 and 8. To this end, the load is gradually increasing from 500 to 1500 users at regular intervals. As expected, the response time increases drastically with a non-controlled system as pointed out by Figure 7. Meanwhile, the response time increases in a logarithmic fashion with our system (see Figure 8).

 ISBN: 978-1-61208-334-6

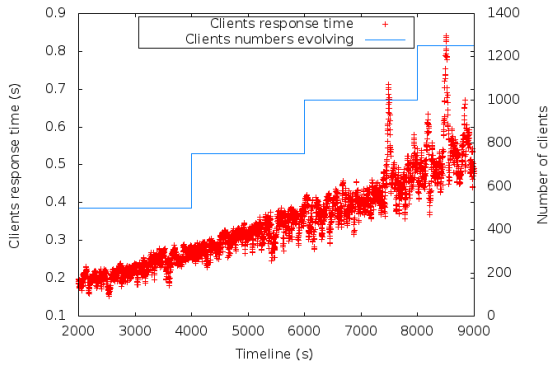

Figure 8: Controlled: Response times on DB0 increases slowly.

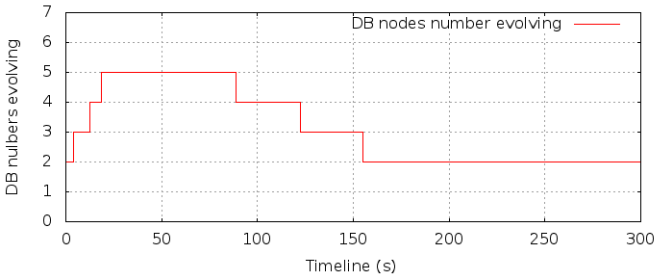

Figure 9: number of DB vs. workload variation.

3) *Number of resources used*

In this experiment, we aim at computing the number of DB used when the load varies. The main goal is to check whether the total number of DB decreases or increases when the workload changes. The load varies from 150 to 300 concurrent users. After a while (i.e., one minute), we reduce the number of clients till 95% of them are off. We set the latency threshold to one second. As one can see on Figure 9, the number of DB nodes grows from 2 to 5 when the workload is increasing. When the workload becomes lighter, the number of DB nodes decreases till we get the initial configuration. Our load balancing algorithm achieves to save computing resources.

C. *Futher experiments and prototype*

We still conducting our experiments in a real-world cloud environment: the Amazon EC2. In this thorough experiments, we use the Oracle NoSQL Database for the storage layer and each of entities (i.e., *Client nodes (CN), Routeurs, Database nodes (DB)*) is embeded in a virtual machine.

The first obtained results are promising and confirm what we got with SimJava. We will publish this results in an extended subsequent version of this work as soon as possible.

VI. Related work

Elasticity and load balancing are essential features to optimize the operation cost in data management systems deployed on a pay-per-use cloud infrastructure, particulary when those data concerns some social media applications. They permit to cope with transient and unpredictable peak loads in the involved systsem, if they are automated. In this context, several works have been proposed to address the problem of elasticity [6][8][9][10][11][12]. Most of these studies have adopted the principle of a partitioned database [6][11] and live migration to distribute the load. In particular, Carlo Curino et al. in [8], ElasTras [6] and Albatross [12] that tackle the problem of minimizing the operating costs of Database systems serving multitenant cloud platforms by efficient resource sharing.

In the literature, most of the work that focused on promoting elasticity in databases are accompanied by migration techniques, and some studies have simply focused on the migration itself. We can mention among them the work done in Slacker [9], which uses hot Back-Up tools to copy the database while allowing service continue during this phase. The migration method is based on the available processing capacity on the node source node in order to bound the response time. It is a solution that prevents interruption of the execution during the migration, but it is based on a Back-Up solutions that are database dependent. However, the authors argue that the Back-Up solution is not a problem since their migration is in middleware. Another problem we raise is, as in [10][11][12], the choice of moving a partition (a tenant in this case) without specifying which one. The idea of moving an entire partition is risky if we do not identify which one to move. With the solution such as Zephyr [10] and Albatros [12], the authors, after identifying the destination, propose a lightweight migration method to move data to their new destination. This migration technique uses an on-demand copy during transaction processing. Thus, they prevent interruptions during transaction processing. The main differences between these solutions and ours are twofold: i) we identify the data to move in order to face directly the source of the bottleneck; and ii) we migrate only required data. More recently, Jan Schaffner et al. propose RTP: *Robust Tenant Placement for Elastic In-Memory Database Clusters* [13]. This work tackles the minimization of operational costs by proposing algorithms that elastically adapt the system size depending on the tenant behavior. This work differs from ours in the way that they consider a read-intensive workload and use replication on their system.

More generally, there are some data management systems recently produced (less than a decade) [14][15][16][17][18][19][20][21]. These systems are usually oriented to the management of web data (NoSQL, key/value, document, etc..) or transactional data [14]. Some of them provide elasticity mechanisms [15][16][17][18], but their elasticity is rather related to the amount of current data. When it becomes too large, they add a new node and place there a part of the data. This assumes that load is evenly distributed on all the data. This does not fit our context where the load is biased.

Furthermore, elasticity is often coupled with load balancing to minimize resources. In addition, such a minimization of resources has an indirect effect that is the gain in energy consumption. This specific objective called green computing, is reached through the effective use of available resources. In this context, many works was done or are being done to develop new techniques for load balancing [22]. The purpose of load balancing [23][24][25][26][27][28] is an efficient use of resources by redistributing dynamically load through all nodes in the system. Our algorithm is based on such principles for for more effectiveness.

 ISBN: 978-1-61208-334-6

Moreover, some works are conducted for load-balancing while avoiding distributed transactions across multiple partitions [29][30]. These approaches balance data based on current load level on that data. They use graph (hyper-graph in Sword) partitioning algorithm to find a replacement that prevents data distributed transactions. The main objective of these works is to provide an improved throughput while providing fault tolerance and scalability for distributed OLTP data management systems. They do not reflect the economy of resources used as we do.

VII. Conclusion and future work

We propose to exploit the social structure of online media to face transient heavy workload. Our solution monitors the load level within the database layer and identifies the hot data, which are the sources of peaks load when overload happens. After identifying the origins of peaks load, we proceed by migrating parts of the hot data among the database nodes, with the goal of keeping the transactions response time under a given value. In order to fully make the identification of the sources of peaks load, and choose the right destinations for migration, we have developed fine-grain identification model. Furthermore, we leverage on the social user network to anticipate the load of popular data mostly owned by users with high centrality degree. This allows for early data migration while preventing cascading migration. We validate our approach through experimentation with a synthetic dataset. These experiments show promising performances in terms of resources saved and response time guaranty. Ongoing works are conducted to evaluate our solution on a real-world database workloads for social applications. To this end, we are experimenting on top of Amazon EC2 cloud, using Oracle KVLite [31] for data storage and access at each DB node. We are using the data and the workload from the LinkBench [1] benchmark.

References

[1] T. G. Amstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, "Linkbench: a database benchmark based on the facebook social graph," in Intl Conf. on Management of Data (SIGMOD), 2013, pp. 1185–1196.

[2] M. E. J. Newman, "Fast algorithm for detecting community structure in networks," Physical Review E, vol. 69, no. 6, Jun 2004, pp. 1–5.

[3] S. Fortunato, "Community detection in graphs," Physics Reports, vol. 486, no. 3-5, February 2010, pp. 75–174.

[4] J. M. McPherson and J. R. Ranger-Moore, "Evolution on a dancing landscape: Organizations and networks in dynamic blau space," Social Forces, vol. 70, 1991, pp. 19–42.

[5] C. Curino, E. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich, "Relational cloud: A database service for the cloud," in Biennial Conf. on Innovative Data Systems Research, 2011, pp. 235–240.

[6] S. Das, D. Agrawal, and A. El Abbadi, "Elastras: An elastic, scalable, and self-managing transactional database for the cloud," ACM Trans. Database Syst., vol. 38, no. 1, 2013, pp. 5:1–5:45.

[7] F. Howell and R. Mcnab, "simjava: A discrete event simulation library for java," in Intl Conf. on Web-Based Modeling and Simulation, 1998, pp. 51–56.

[8] C. Curino, E. P. Jones, S. Madden, and H. Balakrishnan, "Workload-aware database monitoring and consolidation," in Intl Conf. on Management of Data (SIGMOD), 2011, pp. 313–324.

[9] B. Sean Kenneth, C. Yun, M. Hyun Jin, H. Hakan, and J. S. Prashant, ""cut me some slack": latency-aware live migration for databases," in Intl Conf. on Extending Database Technology (EDBT), 2012, pp. 432–443.

[10] A. J. Elmore, S. Das, D. Agrawal, and A. E. Abbadi, "Zephyr: Live migration in shared nothing database for elastic cloud platforms," Intl Conf. on Management of Data (SIGMOD), 2011, pp. 301–312.

[11] M. Umar Farooq, L. Rui, A. Ashraf, S. Kenneth, N. Jonathan, and R. Sean, "Elastic scale-out for partition-based database systems," in IEEE International Conference on Data Engineering Workshops (ICDEW), 2012, pp. 281–288.

[12] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi, "Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration," Proc. VLDB Endow., vol. 4, no. 8, 2011, pp. 494–505.

[13] J. Schaffner, T. Januschowski, M. Kercher, T. Kraska, H. Plattner, M. J. Franklin, and D. Jacobs, "Rtp: robust tenant placement for elastic in-memory database clusters," in Intl Conf. on Management of Data (SIGMOD). ACM, 2013, pp. 773–784.

[14] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-Store: a high-performance, distributed main memory transaction processing system," Proc. VLDB Endow., vol. 1, no. 2, 2008, pp. 1496–1499.

[15] I. VoltDB, "Voltdb," Retrieved on March 2014. [Online]. Available: http://voltdb.com

[16] I. Basho Technologies, "Riak," Retrieved on March 2014. [Online]. Available: http://docs.basho.com

[17] F. Apache Software, "Apache hbase," Retrieved on March 2014. [Online]. Available: http://hbase.apache.org/

[18] I. CouchBase, "Apache couchdb," Retrieved on March 2014. [Online]. Available: http://www.couchbase.com/

[19] F. Apache Software, "Apache couchdb," Retrieved on March 2014. [Online]. Available: http://couchdb.apache.org/

[20] m. Inc., "mongodb," Retrieved on March 2014. [Online]. Available: http://www.mongodb.org/

[21] I. Amazon Web Services, "Amazon dynamodb," Retrieved on March 2014. [Online]. Available: http://aws.amazon.com/fr/dynamodb/

[22] I. C. Nidhi Jain Kansal, "Cloud load balancing techniques: A step towards green computing," International Journal of Computer Science Issues (IJCSI), vol. Vol. 9, Issue 1, No 1, 2012, pp. 238–246.

[23] A. M. Nakai, E. Madeira, and L. E. Buzato, "Load balancing for internet distributed services using limited redirection rates," IEEE Latin-American Symposium on Dependable Computing (LADC), 2011, pp. 156–165.

[24] Y. Lua, Q. Xiea, G. Kliotb, A. Gellerb, J. R. Larusb, and A. Greenber, "Join-idle-queue: A novel load balancing algorithm for dynamically scalable web services," Intl Journal on Performance evaluation, 2011, pp. 1056–1071.

[25] H. Mehta, P. Kanungo, and M. Chandwani, "Decentralized content aware load balancing algorithm for distributed computing environments," in Intl Conf. Workshop on Emerging Trends in Technology (ICWET), 2011, pp. 370–375.

[26] B. Jasma and R. Nedunchezhian, "A hybrid policy for fault tolerant load balancing in grid computing environments," Journal of Network and Computer Applications, 2012, pp. 412 – 422.

[27] G. You, S. Hwang, and N. Jain, "Scalable load balancing in cluster storage systems," in Middleware, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011, vol. 7049, pp. 101–122.

[28] H. T. Vo, C. Chen, and B. C. Ooi, "Towards elastic transactional cloud storage with range query support," VLDB Endow., 2010, pp. 506–514.

[29] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning," VLDB Endow., vol. 3, no. 1-2, 2010, pp. 48–57.

[30] Q. Abdul, K. Kumar, and A. Deshpande, "Sword: Scalable workload-aware data placement for transactional workloads," in Intl Conf. on Extending Database Technology (EDBT), 2013, pp. 430–441.

[31] C. Oracle, "Oracle nosql database," Retrieved on November 2013. [Online]. Available: http://www.oracle.com/technetwork/products/nosqldb/overview/index.html

 ISBN: 978-1-61208-334-6