Rapport de stage
Evolutions du système de réplication de bases de données GlobeTP

Thomas Aynaud
supervisé par Guillaume Pierre et Paolo Costa

Stage de mars à août 2007
Vrije Universiteit Amsterdam, Pays-bas

Résumé préliminaire en français

La majeure partie de ce rapport a été rédigée en anglais, d'une part car Paolo Costa n'est pas francophone et d'autre part car mes encadrants aimeraient en tirer une publication.

Mon stage s'est déroulé du 15 mars au 15 août à la Vrije Universiteit d'Amsterdam, aux Pays-bas dans l'équipe de *computer system*. Celle-ci travaille sur divers sujets tels que les systèmes distribués, la programmation parallèle, des applications avancées de l'Internet. Cette équipe, dirigée par le professeur Andrew Tanenbaum, est composée d'une dizaine de professeurs ou assistants professeurs, d'une vingtaine de doctorants et de quelques post-doctorants.

Je travaillais plus précisément avec Guillaume Pierre et Paolo Costa, qui étudient les méthodes d'hébergement de sites webs. Leurs travaux de recherches portent sur les techniques de cache et de réplication afin d'obtenir des serveurs webs (ou plus précisément l'ensemble constitué des serveurs traitant les requêtes http, les serveurs d'application et les serveurs de bases de données) capables de répondre plus rapidement à plus de clients simultanément.

Mon travail se fondait sur les recherches précédentes de G. Pierre et son équipe sur la réplication de bases de données. L'objectif est de faire fonctionner plusieurs serveurs de bases de données ensemble dans le but d'une part de répartir la charge de calcul entre les machines, et donc en supporter une plus importante, et d'autre part apporter une tolérance aux défaillances d'une ou plusieurs machines. Nous nous sommes focalisés sur les performances du système, et dans cette optique la réplication pose certains problèmes. Pour maintenir les différents serveurs dans le même état, chaque écriture doit être traitée par tous les serveurs. Chaque lecture peut être traitée par un seul serveur. Seulement, quand le nombre de serveurs augmente, la proportion de requêtes en écriture reçues par un serveur particulier augmente (par exemple, pour 5 serveurs, chacun reçoit environ un cinquième des lectures, mais toutes les écritures). A partir d'un certain nombre de clients, les écritures seules suffisent à surcharger les serveurs, et en ajouter n'apporte rien. Il faut donc pouvoir répartir les écritures entre les serveurs, et une solution est de ne pas dupliquer toute la base sur chaque serveur. Ainsi, chaque serveur ne reçoit que la

fraction des requêtes en écriture correspondant à la fraction de la base qu'il contient. On appelle cette approche la réplication partielle, et le système *GlobeTP* [1] utilise cette idée. Vu qu'aucun serveur ne doit contenir toute la base de données, on ne peut pas traiter toutes les requêtes SQL possibles, car faire travailler plusieurs serveurs sur la même requête en partageant des données reste très problèmatique. Par conséquent, après une analyse du code de l'application du site web, on en extrait différents modèles de requètes, et on en tire tous les sous-ensembles de la base de données qui doivent être regroupés. Les travaux précédents [1] proposaient un algorithme de placement de ces parties de base de données, et étudiaient comment le système se comportait.

A partir de ces travaux, je devais étudier comment on peut faire évoluer un système GlobeTP en fonctionnement. L'intérêt pourrait être par exemple de rajouter un nouveau serveur, afin de répondre à une augmentation de l'audience. Arrêter le système pendant la reconfiguration est en général impossible, on veut pouvoir continuer à répondre tout le temps. Plusieurs problèmes se posent :

- Comment copier une table d'un serveur à un autre, sans les arrêter, alors qu'elle est modifiée.
- Dans quel ordre effectuer ces copies.

On a donc proposé un protocole pour copier une table (ou un groupe de tables) d'un serveur à un autre. Le protocole est en deux phases. Premièrement, le serveur destination commence une transaction en lecture seule sur le serveur source, et récupère toutes les données s'y trouvant au moment du début de la transaction. Grâce aux propriétés des bases de données, les serveurs source et destination continuent de fonctionner. Un fois ces données récupérées, le serveur destination construit une table équivalente à celle sur le serveur source au début de la transaction. Commence alors la deuxième phase, où le serveur destination rattrape le serveur source en effectuant toutes les requêtes qui sont arrivées entre temps.

On a ensuite implémenté ce protocole, et on l'a testé sur plusieurs cas. On a étudié ce qui se passait quand les deux serveurs étaient chargés, quand seule la source était chargée, et finalement si un serveur pouvait être la source de plusieurs copies en parallèle. On a été assez surpris de l'efficacité de la méthode, les copies étant quasiment transparentes pour l'utilisateur, alors qu'on s'attendait à un impact beaucoup plus notable.

Etant donné qu'une fois une table copiée, on peut l'utiliser pour répondre à des requêtes, l'ordre dans lequel on fait ces copies a une certaine importance. Répliquer tôt une table à laquelle on accéde peu n'aura pas beaucoup d'impact sur les performances du système. On a donc essayé d'optimiser les performances durant les copies en choisissant un ordonnancement adéquat. Avant d'ordonnancer, il faut établir la liste des tâches à effectuer. On a utilisé l'algorithme proposé dans [1] pour calculer le placement des tables. Néanmoins, on obtient une liste de groupements de tables, et il faut les affecter à chaque serveur, ceux-ci contenant déjà une partie des données. Affecter le groupement table A + table B à un serveur contenant la table C alors qu'un autre contient déjà ces deux tables serait absurde. L'affectation des groupements de tables aux serveurs se fait à l'aide de l'algorithme dit *hongrois*.

Pour calculer un ordonnancement, nous avons tout d'abord modélisé la charge du système. Nous avions des statistiques sur les requêtes soumises au

système, à savoir une estimation du temps pris pour y répondre et une estimation de leur fréquence d'apparition. A partir de cela, on définit le coût d'une requête comme suit :

- Pour une requête en écriture : $frequence.temps$
- Pour une requête en lecture : $\frac{frequence.temps}{\text{nombre de serveurs pouvant la traiter}}$

En effet, les requêtes en écritures doivent être traitées par tous les serveurs concernés, tandis que les requêtes en lecture ne sont traitées que par un seul des serveurs parmis ceux capables de le faire. Au final, la charge d'un serveur est la somme des coûts des requêtes qu'il peut traiter, et la charge du système global est le maximum des charges des serveurs. On peut donc évaluer la charge du système après chaque copie. En considérant qu'une copie dure un temps proportionnel à la taille de la table copiée et a un effet constant sur le système, on peut tracer le graphe présentant la charge en fonction du temps. On cherche à minimiser l'aire sous cette courbe. Ne trouvant pas de meilleur moyen qu'une étude exhaustive pour calculer l'ordonnancement optimal, nous nous sommes intéressés à une approximation par un algorithme glouton. Quand cela était possible, on a comparé les résultats donnés par l'approche gloutonne et l'approche exhaustive, et les résultats étaient encourageants. Le pire résultat obtenu n'était que 60% de l'optimal. De plus, optimiser tôt la charge pouvait être efficace en pratique. En effet, les copies allaient s'effectuer sur des systèmes déjà très chargés, et réduire cette charge vite pouvait libérer du temps de calcul pour les copies suivantes.

Nous avons ensuite testé ces techniques sur une application de benchmark, RUBBoS, représentant un site de nouvelles. Nous avons aussi essayé avec une autre application (TPC-W [2]), mais le temps a manqué pour obtenir des résultats exploitables. Les tests ont été effectués sur la grappe de serveurs DAS-3, hébergée en partie par la Vrije Universiteit. Les requêtes à soumettre au système ont été pré-générées en émulant le comportement d'un visiteur, ceci afin d'isoler la partie base de données des serveurs d'application et des serveurs http.

Concernant les résultats sur le benchmark RUBBoS, le système arrive effectivement à migrer d'une configuration à n serveurs vers une configuration à $n+1$. L'ordonnancement glouton se comporte relativement bien, mais les gains ne sont pas spectaculaires. En effet, avec plusieurs serveurs, la surcharge de l'un d'entre eux n'entraîne pas une surcharge de tout le système, et c'est donc pour le passage de 1 à 2 serveurs surtout que l'ordonnancement montre son efficacité. L'étape suivante serait d'étudier si on peut effectivement faire plusieurs copies en parallèle, et comment les ordonnancer.

1 Introduction

In the past few years, the World Wide Web has become a more and more important communication medium for many companies and public services. Instead of simply delivering static HTML files, they often use specific software, called web applications, to propose news, contents, products and other things. A typical web application dynamically generates content according to users requests. Pages are made of some business logic running in an application server, which is invoked each time an HTTP request is received. This business logic, in turn, may issue any number of SQL queries to a database.

The growth of the number of users and the growth of their needs, like personnalization for example, are making these applications more and more complex. Thus, the needs for scalable hosting capable of supporting high load have increased. In practice, the database usually is the performance bottleneck for the whole system.

Background

Database replication techniques can be used to improve the throughput of the database. The approach used by most databases is full data replication where the entire data set is duplicated to multiple servers. Read queries can thus be dispatched across different servers. However queries that modify the application state (Update, Delete, Insert in SQL, here after refered as UDI queries) must be processed by every server to assure consistency of the data between servers. One possible optimization is to perform UDI queries on one master server and other servers just process logs of the differences, which is faster than processing the query entirely. However, the issue is that the master server still has to process all UDI queries, and thus becomes the performance bottleneck of the system. The more servers there are, the higher the percentage of UDI per server is, and this is also increased by using cache system that can only cache read queries. So, full replication does not scale well, and better techniques are needed to improve the database throughput.

In a previous work [1], GlobeTP, a system allowing partial replication of the data, has been presented. Partial replication supposes that each server has only a part of the data, so it receives only a part of the UDI queries, and is finally less loaded. Deciding which server should have which piece of data can however be difficult, because a query may require data items that are not all on the same server. The main idea of GlobeTP is that typical web applications use only a few different query templates. The database is divided in tables and each query template require only a subset of the tables to be processed. Thus, for each read query template, we need one server that contains all the tables required by the query template to process it. If there exists a read query template that requires a group of tables that is not entirely on one server, the placement of the tables is invalid. Write queries still must be processed by every server that contains related tables but as no server contains the whole database, servers do not receive all the UDI. An algorithm to compute a good distribution of the table has been proposed and this architecture appears to be really effective compared with full replication, as the throughput was increased by 57% to 150% compared to full replication.

Problem statement

However, the needs of a web application will certainly evolve. The load of the application may grow with the number of users, so we need to adapt the system capacity, for example by adding some servers. Another case can be a modification of the application that requires a new organisation of the tables. To achieve this, some tables must be copied from one node to another while some others must be deleted on certain servers. The problem is that in many cases stopping the system to allow a reconfiguration is simply impossible. If a web application does not respond, clients will search another solution, and may never come back. So we must allow the system to evolve without stopping it. However, the copied tables keep on being modified during the copy process. The migration process will also certainly occur on a cluster which is heavily loaded. This drives the need for an efficient migration process that has a minimal effect on the system performance. The difficulty is to remain able to process each possible query during the migration without breaking consistency. The first issue will thus be how to copy a table from one server to another without stopping them and keeping consistency, as the data may be modified during the process.

Secondly, if multiple such operations are required to perform a reconfiguration, then the order in which they are applied can have an influence on the system performances during reconfiguration. Indeed, as soon as a first subset is copied, we can immediately start using it to process some queries. Thus, the global system is, during the migration, in intermediate states that may perform differently. For example, replication of a small table that is only readed and never updated may have a more important impact during the migration than starting with a huge table that is hardly accessed.

Contribution

The goal of this thesis is first to propose an algorithm to copy a table from one database server to another one without stopping any of them, while maintaining consistency. The second objective is to optimise the migration process of multiple tables, by choosing the right scheduling for migrations. Therefore, we first propose an algorithm to copy a table and implement it for practical testing. Then, we study what would be an optimal scheduling, and how we can compute a good one. All experiments are based on one benchmark application that aim at representing a real world application.

This thesis is structured as follows : Section 2 presents related work on database replication. Section 3 presents our algorithm to copy a table from one server to another without stopping them. Section 4 explains how we reduced the effect of a reconfiguration of the whole system, by minimising the number of copies and by optimising the order of these copies. Section 5 presents the experimental results we obtain with our implementation and discusses them. Finally, section 6 concludes the thesis.

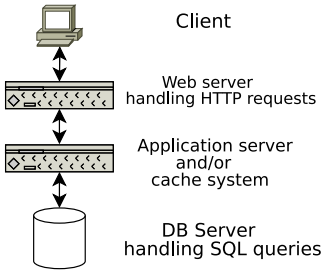

Figure 1: Web application architecture

2 Related work

2.1 Web application hosting

Typical web applications are implemented in a multi-tiers architecture as described on Figure 1. They are often deployed on a content delivery network using an edge-server architecture as depicted in Figure 2. Client requests are issued to edge servers located accross the Internet. Each of them contains the application code but no database. The database is centralized in only one site, and receives all queries from the edge servers. This implies a wide area network latency for the queries, so edge servers often contain local database cache system. Thus, only cache misses and UDI are sent to the origin database, where another cache system can be deployed to reduce the load on the database.

Several techniques have been proposed to cache the results of database queries [3, 4, 5, 6]. Consistency of the cache must be maintained with the

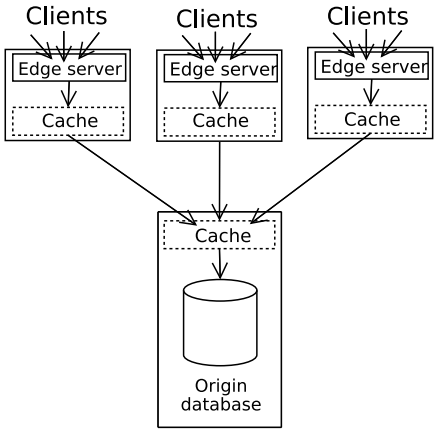

Figure 2: Typical Edge-Server architecture

origin database. This is usually done by requesting that the application programmers specifiy query templates and their relationships. When a UDI query arrive to the origin server, this one can identify which conflicting query template results are modified, and thus invalidate the different caches. Many queries can be thus answered locally, which reduces the latency and improves the total throughput by reducing the number of queries that reach the origin database. However, database caching systems have good hit ratio only if the database queries exhibit high temporal locality and contain relatively few updates.

Web applications often also cache fragments of the generated pages at the edge servers. This technique performs well, but has the same limitations as database caching. If the queries contain many UDI or do not exhibit temporal locality, the number of cache hits is limited. Caching techniques allow to improve the overall system performance. However, they are not sufficient to obtain arbitrary scalability, as their scalability bottleneck is eventually the capacity of the origin database.

A common technique used to improve the origin database capacity is replication. The motivation is twofold : availability and scalability. Replicating data can improve the database system tolerance to failure. Also, by distributing queries accross different database servers, the database throughput can be increased.

2.2 Database transactions

Before discuting more about replication techniques, it is important to understand database properties regarding transactions. A transaction is a group of queries addressed to the database. At its end, a transaction can be committed (applyied) or rolled back (finally discarded). A database transaction must conform to *ACID* properties :

- A : Atomicity, a transaction is a series of database operations which either all occur, or all do not occur at all.
- C : Consistency, refers that a transaction must respect the integrity constraints of the database.
- I : Isolation, refers to the ability of the application to make operations in a transaction that appear isolated from all other operations.
- D : Durability, refers to the guarantee that once the user has been notified of success, the transaction will persist, and not be undone, even in case of failure after the notification.

There are multiple levels of isolation between transactions. The strictest one, *serializable*, specifies that all transactions occur in a completely isolated fashion, as if all transactions in the system had executed serially. To implement this, database systems mostly use locks or multiversion concurrency control. From the client's point of view, this last one provides each user connected to the database with a snapshot of the database. The user works with this snapshot, and changes made will not be seen by other users of the database until the transaction is committed. This allows multiple clients to work on the database in parallel.

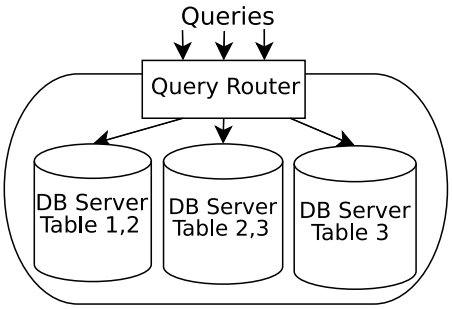

Figure 3: Globe TP architecture

2.3 Database replication

It is more difficult to respect ACID properties when data are duplicated accross several servers. Existing replication protocols can be divided into *eager* and *lazy* schemes. Lazy replication allows copies to have different values and emphasises efficiency whereas eager replication focuses on the consistency of the different database servers. As eager replication was considered as not practical [7], many lazy replication protocols have been proposed [8, 9]. They however lead to conflicts between UDI that often must be solved by the client application. This requires technical understanding by the programmers and therefore are often considered unsatisfactory.

More recent works [10] tend to prove that eager protocols, like *two phase locking*, can be practical. Eager replication is implemented by mature systems such as mySQL [11]. However, the prime goal is reliability more than scalability. Eager protocols must conform to *1-copy-serializability*. This means that the resulting schedule of the database must be equivalent to a serial schedule on a single database. Another approach proposed in [12] was to use internally lazy replication while providing 1-copy-serializability to clients. The main issue is that one server still has to process all the UDI and thus becomes the bottleneck of the database system.

2.4 GlobeTP

The approach at the basis of this thesis is that of GlobeTP [1]. GlobeTP exploits partial replication to reduce the number of UDI that each server needs to process. As described in Figure 3, the system is composed of a query router and several database servers. Each database server contains only a subset of the database tables. The query router knows which server contains each tables and can route each query to a server that contains all the tables needed. This is possible if we make the assumption that there is a finite number of query templates. This is often true with typical web applications [6, 4, 5, 1]. Read queries can be processed by any server that contains the tables needed while UDI queries must be processed by every server that contains the concerned data. So, a table that is often updated and rarely read should not be replicated on a lot of server whereas a table that is read often must be replicated on a lot of

servers. [1] proposes an algorithm to optimize the table placement.

2.5 Evolution

However, in such a system, without a single master server, evolutions of the system are more difficult. The placement also allows some optimisations on the scheduling of the migration. This works can be compared with [13, 14]. These papers proposed systems based on a pool of database servers to handle multiple applications. Data of one application are replicated on several servers, but one server can contain the data from several applications. All the servers that contain the data of one application are not always active. Some of them are just kept almost updated in order to be added quickly to the set of active servers in case of load spike. They have also proposed a way to predict when one application will be overloaded. The main difference is that they are more emphasing on a reactive system, capable of improving fastly its throughput to react to a spike of load, rather than the evolution of a system in a long term.

3 Duplication mechanisms

3.1 Mechanisms

We will now see how we can copy one table (or a group of tables) from a 'source' database to 'destination' database. The table may be big, so the copy may take minutes at least. During this time, UDI queries continue to arrive to the source and the destination. Only the source is receiving UDIs concerning the considered table. We want to achieve a point where we can say that the table on the source and the destination are consistent i.e., they both contain the same data, and no UDI is being processed. We assume that the database can provide serializable isolation level and uses multiversion model [15].

The copy is realized in two phases. During the first one, the destination reaches the state of the source at the begining of the migration whereas during the second one, the destination catch up the source and process the queries that are arrived during the first phase.

More precisely, we first wait for a time t where the source has no UDI in its queue. At the time t, the destination begins a transaction (with serializable transaction level) with the source and receive all the data it contains at this point. During the process, all the UDI related with the table being copied are dispatched to the nodes which contain the table before the begining of the copy and logged by the query router. The source is still able to handle queries concurrently since the transaction is read only and thanks to the multiversion model. When all the data have been received by the destination, this one starts building the table with them. At the ends of this process, the destination is in the state the source was at the time t. The second phase begins and the destination starts processing the logged UDI to catch up with the other nodes. It is possible because we must assume that one node can process the UDI faster than they arrived, or the system already experiences big performance issues. When the log at the query router is empty, we consider the destination consistent, and the destination is added in the routing table.

As in the GlobeTP implementation, we make the assumption that read and

write queries are independant operations and we do not support transactions. One possible workaround would be to process the logs of the transactions since the time t rather than the logged queries. We have not done this because it would have required more work on our database system internals than desired, and we have prefered to stay at a higher level.

3.2 Micro-benchmarks

To evaluate our copy method, we implemented it with the Postgres database [16]. Queries are numbered by order of arrival to the query router. For each group of 5000 queries, the 90^{th} percentile of the query response time has been calculated. This means that 90% of the queries are processed faster than the duration plotted. The first $100,000$ queries are ignored and correspond to a warmup time for the system. All the experiments have been done on a loaded system. We were able to select the number of clients by step of 30, and experiments have been done at the last step where 90% of the queries are processed in less than $100ms$.

The first experiment aims at showing the global effect of the copy when both source and destination are loaded.

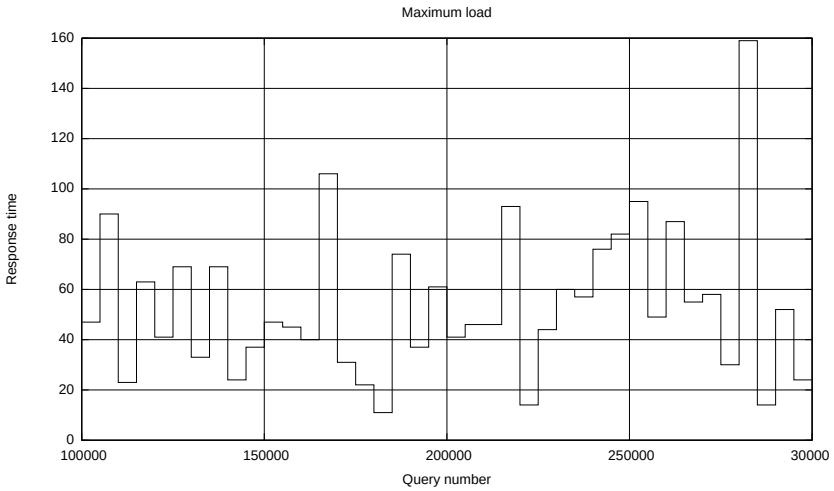

Figure 4: Response time of the whole system during the copy of one table

Figure 4 presents the response time of a system composed of two database servers. One table starts being copied from the first one to the second at the query number $200,000$ and ends at the query number $262,000$. The first server contains initially the whole database, while the second one contains the whole database except the copied table. The copy of the data from the source ends at query $204,501$ and the creation of the table ends at query $261,448$. Surprisingly, the overhead caused by the copy mechanism is much smaller than expected as it is almost imperceptible.

The second experiment aims to show the effect of the copy on the source, and the duration of the building of the table by the destination. If it appears to be effective, we will not have to take care of the source during the scheduling of the different copies.

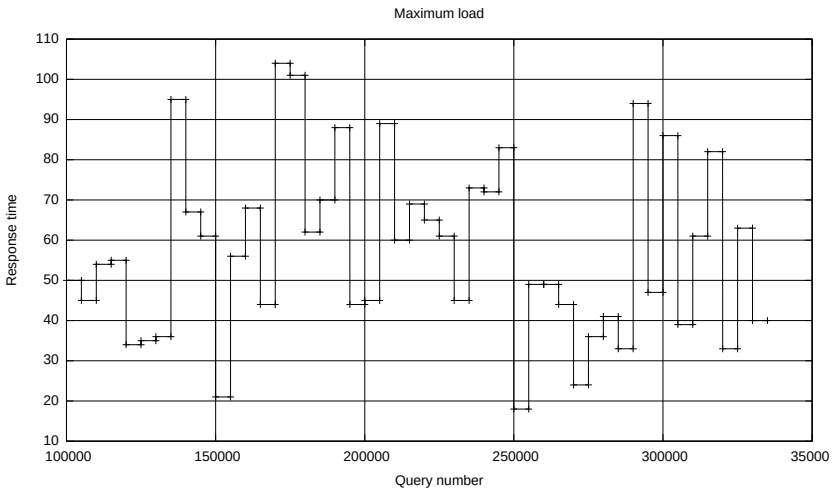

Figure 5: Response time of the source node during the copy of one table

Figure 5 represents the copy of one big table which is not often updated. The copy starts at the query number $200,000$. At query $203,575$ the destination has finished to collect data from the source and starts building the table. This ends at query $239,294$, which corresponds to the end of the first phase. As the table copied receives few UDI, the second phase is really fast and finishes immediately after. Clearly, this first phase has little impact on the source database. Thus, the source will not be a concern during the scheduling of a migration. However, the building of the table by the destination is quite slow because the data are transmitted as a group of INSERT statements. Commercial database servers often propose faster query replay mechanisms but we decided to keep the current mechanisms as it can be applied without modifying the internals of the database.

We have finally studied what would happen if one server was the source of several copies at one time to see if it is feasible.

Figure 6 represents the 90^{th} percentile of the response time of one server which is the source of six table copies executed in parallel. There, the percentile has been calculated by blocks of 3000 queries as the process is really fast. The copies start at the query number $200,000$ and finish at the query number $225,000$. The different destinations just get the data and do not build the tables. The copy of the data appears to be really fast if the source and the destination are connected by a good network, and has a very limited effect on the source. This means that we will be able to copy multiple table from only one source, which may facilitate the design of a protocol allowing multiple copies in parallel.

4 Migration Planing

Now, we will study the scheduling of the migration of the different copy and deletion. We will first see how we compute the list of the copies needed, and try to reduce the total process length and then how to schedule the different copies.

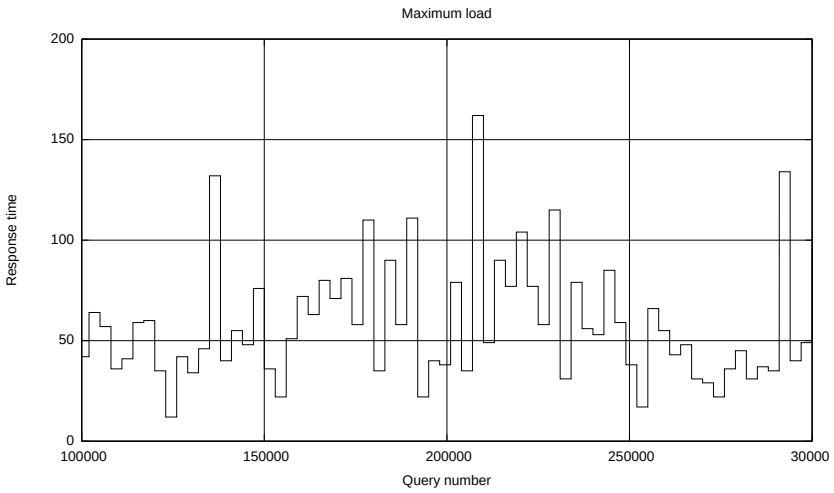

Figure 6: Response time of the source during the copy of multiple tables

4.1 Minimizing the number of copies

Let us assume we have two lists of tables that will be the lists of tables on one server before and after the migration. We assume that a copy takes a time proportionnal to the size of the table. Table deletion only requires a "DROP TABLE table (CASCADE)" SQL statement, and is really fast. We define the cost of the migration as the sum of the size of the tables that are on the server after the migration and not before the migration.

Next we have several servers, and for each of them the list of the tables they contains. We have also the repartition of tables that we want to achieve after the replication. For example, we want to add one server to the cluster :

	Server 1	Server 2	Server 3
Before	table 1,2	table 2,3	empty
After	table 1,3	table 1	table 1,2

We consider that which server contains which part does not matter, as they are similars, and the placement algorithm does not take into account physical differences such as memory or cpu. The matching here is clearly suboptimal. If we consider that all the table are of size 1, the total cost is 4 (copy 3 to server 1, copy 1 to server 2 and copy 1 and 2 to server 3). With the following matching, the cost is only 2 :

	Server 1	Server 2	Server 3
Before	table 1,2	table 2,3	empty
After	table 1,2	table 1,3	table 1

This issue can be reduced to the *Assignement problem*. Formally :

Definition. *Given two sets, X and Y, of equal size, together with a weight function $w : X \times Y \to \Re$, find a bijection $f : X \to Y$ such as the cost function :*

$$\sum_{a \in X} w(a, f(a))$$

is minimized.

Here, X and Y are the sets of servers. w is the cost function described precedently. This problem is solved by the *Hungarian algorithm* in polynomial time. For simplicity, we do not describe this intricate algorithm here and refer the interested readed to [17].

Now that we can list the copies and the deletion, and minimize its size, we can discuss the different ways to obtain the destination placement. The heuristic proposed in [1] starts with an initial valid placement and improves it at each iteration. We focus here on the evolution of the system from n to $n+1$ database servers. There are two logical possibilites :

- Use the algorithm without specific intialisation, to compute the best placement possible regardless of the existing placement
- Use the algorithm initialized with the current placement plus an empty server, hoping that it will produce a good placement, and that the system will not change too much.

We have tested both possibilities. Both produce almost equivalent placements in terms of load and in terms of cost of migration. Figure 7 and 8 show the number of copy that will be required and the expected load with both possibilities. It appears that they are equivalent. We therfore decided to use the first one, as it is simpler.

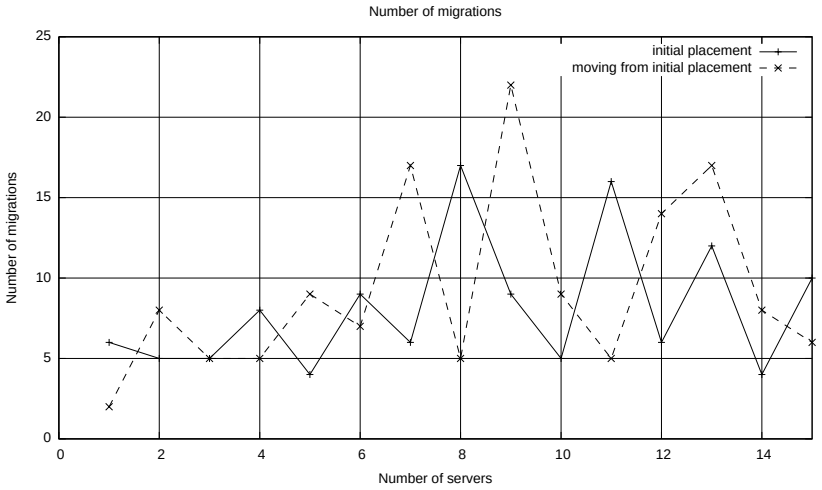

Figure 7: Number of migration to pass from n servers to $n+1$

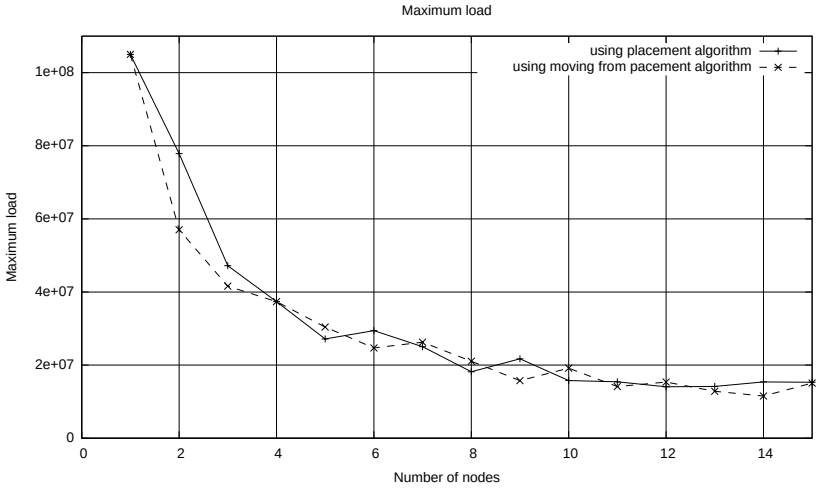

Figure 8: Expected load with the placements proposed by the two approachs

4.2 Scheduling of the copies

Now that we can obtain the list of the copies we have to do, we need to decide in which order we should apply them. The optimization criterion here is that the system should have the best possible performance during the reconfiguration. We therefore aim at minimizing the load of the most loaded server of the cluster. We exploit the fact that all the tables are not used identically. Some of them are mostly read while others receive significant number of UDIs. Replicating early a table updated often will not improve the throughput and reducing the load of a server that is not really loaded will have no effect. Figure 9 presents what we would expect of a"good" scheduling. It presents the load of the system during

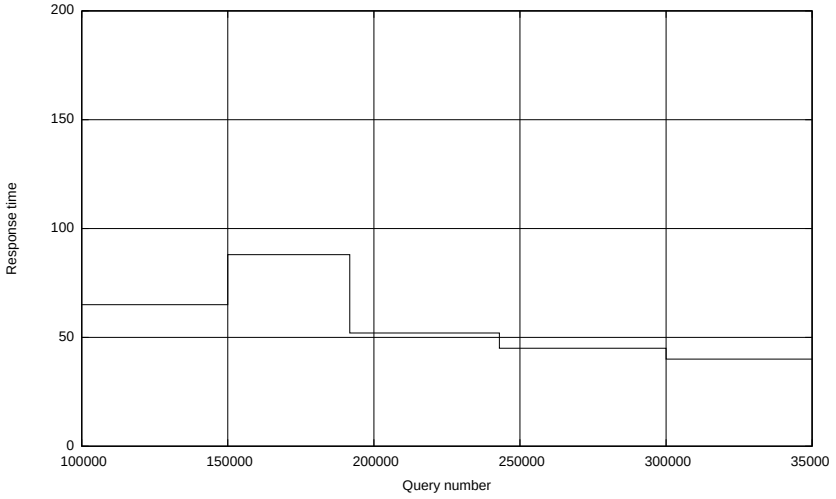

Figure 9: Expected load with a good scheduling

several copies. The first copy starts at the query number 150,000 and the load raises. After the first copy (at query around 190,000), the load improves and continue during the other one. During a good scheduling, load would improve fast and the system would work quickly better than before the migration.

At the opposite, Figure 10 presents what would be a"bad" scheduling. Load do not improve until the end of the migration. Even though both schedules have the same performance before and after reconfiguration, the first one is clearly preferable.

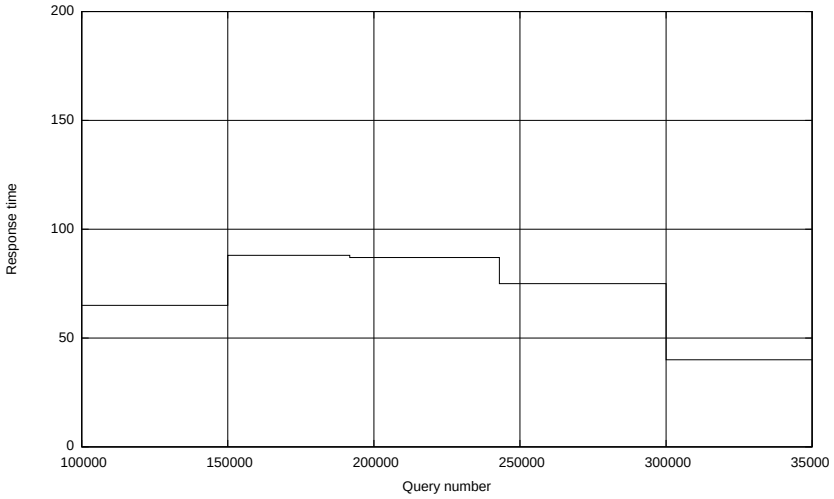

Figure 10: Expected load with a bad scheduling

To compute a good schedule, we have to make some assumptions. First, we consider that the duration of one copy depends only on the size of the table copied. Next, we consider that the negative effect of one migration on the load of a server is constant. It does not depend of the table, the source server or the destination server.

As in [1], we have statistics on the queries that arrived. We know the average duration of each query template and the average number of apparition of each query template. Read queries are processed by only one node and UDI by every node that contain related data. Thus, we can define a metric that will represent the response time of our cluster. Each query template has a cost of :

- If it is a UDI query :

$$cost = \text{query frequency} \times \text{average duration}$$

- If it is a read query :

$$cost = \frac{\text{query frequency} \times \text{average duration}}{\text{number of server that can process it}}$$

For each server, we define its load as the sum of the cost of the queries it can process, and the load of the full cluster is the maximum of the load of the servers it contains.

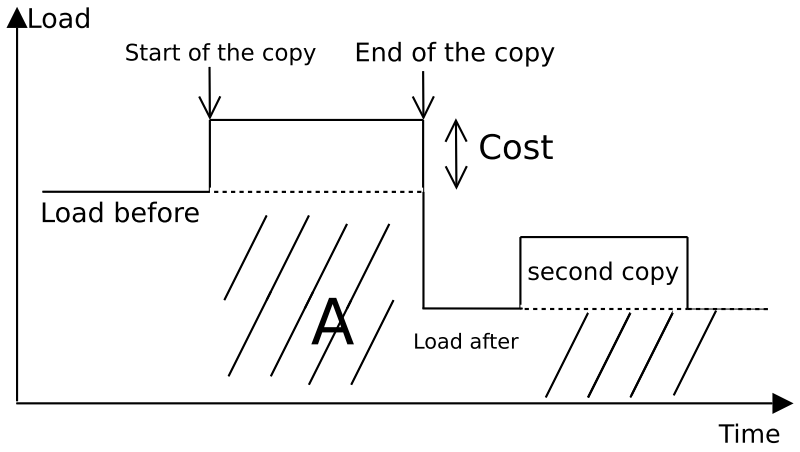

Figure 11: Expected response time of the system

Thus, we can now evaluate the load of our system at each point of the replication. We model the replication as described in Figure 11.

The system has a given response time before reconfiguration. At the begining of the first copy, the response time raises of a constant value (the cost) during the copy. The copy duration depends only on the size of the table. After the copy, the system reaches another response time level and stay at this level until the next copy. In practice, we will start the next copy just after the end of the precedent one. As the negative effect of one migration is constant, and the duration only depends on the size of the table, the response time raise caused by the copy does not depend on the order of the copies. Thus, we will ignore it. To evaluate our scheduling, we will compute the hatched area $\mathcal{A}$. Let S be our system, c a copy or a deletion and $T(c)$ the duration of c (0 if c is a deletion, the size of the table copied if c is a copy). One scheduling will be a bijection $\sigma : [1 \dots n] \rightarrow C$ where C is the set of copies and deletions to process and such as $\sigma(i)$ is the i^{th} copy. Let's call f the function that compute the load of a system and S_i^σ the system S after the i first copies/deletions of the scheduling σ (an ∞ value if the system can not handle every queries), we have that :

$$\mathcal{A}(\sigma) = f(S).T(\sigma(1)) + \sum_{i=1}^{n-1}(f(S_i^\sigma)).T(\sigma(i+1)))$$

We want to minimize this area. $f(S_i^\sigma)$ depends not only on $\sigma(i)$ but also on $\sigma(1), \sigma(2), \dots, \sigma(i-1)$ because we may, for example, first add a table $t1$ and then a table $t2$ that will allow to process a read query template that requires $t1$ and $t2$. To minimize $\mathcal{A}$, we would need to iterate through every σ which will certainly be too long ($n!$ if n is the size of C). We did not find a faster way to compute the optimal σ. Thus, we approximated the best σ with a simple greedy algorithm. Let us consider the area $\mathcal{B} = f(S).\sum_{i=1}^{n-1} T(\sigma(i))$. $\mathcal{B}$ is the area of the graph if we do no migrations. We have that :

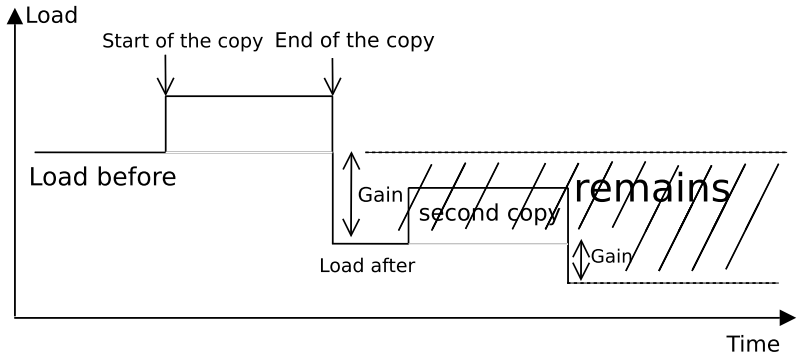

Figure 12: Expected response time of the system

$$\mathcal{B} = \mathcal{A}(\sigma) + remains$$

And *remains* can be calculated as :

$$remains = \sum_{i=1}^{n-1} \{(f(S_i^\sigma) - f(S_{i-1}^\sigma)).\sum_{j=i+1}^{n} T(\sigma(j)).\}$$

Instead of computing the area under the graph, we compute the area upside (the hatched area of the Figure 12). As $\mathcal{B}$ does not depend on σ, minimizing $\mathcal{A}(\sigma)$ is equivalent to maximizing the *remains*. The final algorithm is algorithm 1, where $c(S)$ is the system S after the copy or deletion c.

Algorithm 1 Greedy algorithm to compute a good scheduling

$C \leftarrow \{copies/deletions\}$
for $i = 1$ to $|C|$ **do**
  find c which maximizes $(f(\mathrm{c(S)}) - f(S)).\sum_{c' \in C-c} T(c')$
  $C \leftarrow C - \{c\}$
  $S \leftarrow$ S after the application of c
  $\sigma(i) \leftarrow c$
end for

We tested this greedy heuristic and compared it with cases where we were able to compute the optimal scheduling by brute force. It appears that the greedy algorithm performs very well. In 14 cases out of 22, greedy found the optimal scheduling. In 4 cases it found a scheduling whose metric is greater than 90% of the best. With the 4 remaining tests, it never finds a scheduling whose metrics is less than 60% of the best metric. We have no proof about the solution quality except these tests.

Greedy also has the advantage to try to optimise the load of the system as early as possible. As the database servers are loaded at the begining, reducing the load early may leave some process time for the other copies. We therefore expected it to perform well in real situations. We present such experimental results on real applications in the next section.

5 Experimental evaluation

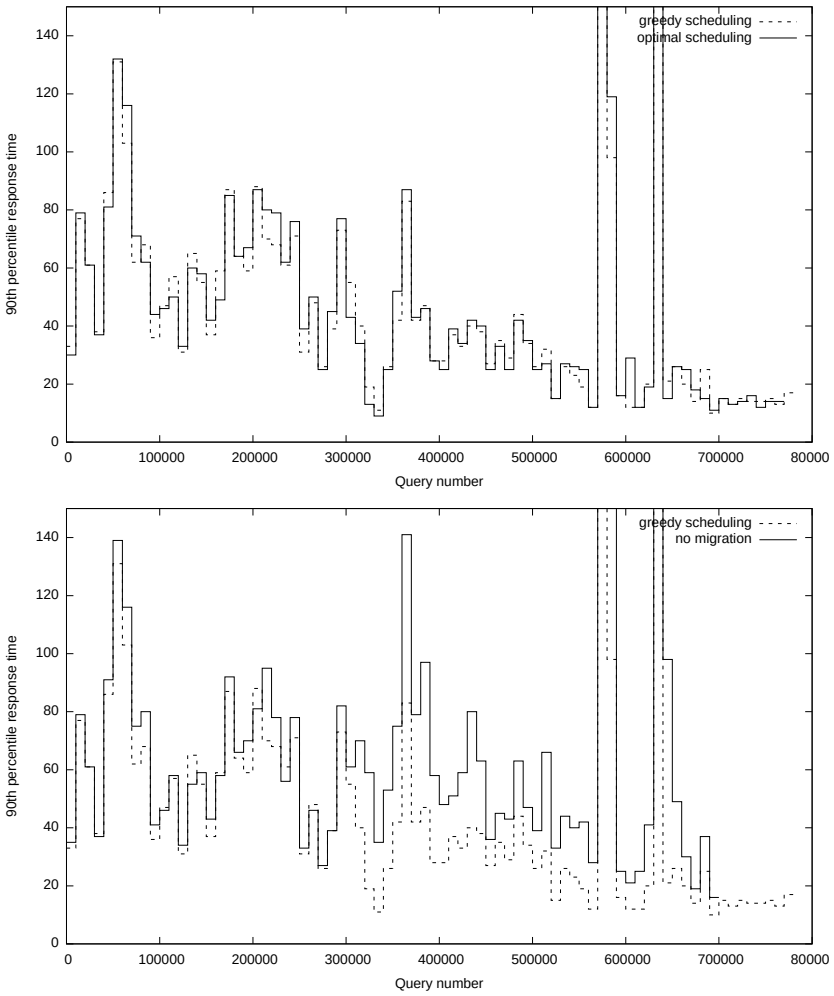

Figure 13: Response time of the system during the migration from 1 to 2 database servers with RUBBoS (300 EBs)

To test our algorithm, we first implemented them. The implementation is composed of several pieces of software.

First, one Java application implements the algorithms to compute the optimal and the approximated scheduling and output them in files readed by the query router to apply them.

Next, there is the database system. It is composed of one query router, that route a database workload generated in advance, and of several PostgreSQL version 8.1.3 servers [16]. We use only the cost-based routing algorithm presented in [1]. The query router, developped in Java, has been extended to allow it to do a migration. It reads its orders generated by the first application, starts the different copies, logs the needed queries and finally updates its routing tables.

The first phase of a copy is handled by a Postgres utility called *pg_dump* that do a transaction with a selected Postgres to collect its data and output a file that contains all the SQL statement to generate the selected data. It is generally used for database backup, but it does exactly what we need for the first phase of the table replication. The query router works with the help of a server, written in C, which runs on every node that contains a database server, to do some local processing for the migration that the database or the query router cannot handle. For example, it has to run the utility *pg_dump*, to apply the dump to the local database and to inform the query router of its progress.

We used one well-known benchmark applications to test our system. RUBBoS [18] is a web application that simulate a bulletin-board like slashdot.org. In a typical news forum like RUBBoS, users will frequently access the more recent news while queries that require several tables are quite rare.

The RUBBoS database is composed of five tables, which are queried by 36 read and 8 UDI templates. The database is initially filled with 500,000 users and approximately 200,000 comments, resulting in a database of more than 700 MB. We used the same workloads as in [1]. They have been generated by the execution of Emulated Browsers (EBs). Each EB acts as a markov chain, where pages are the nodes, and transitions are the links between pages. The different transition probabilities are generated to model the behavior of a standard user. Each benchmark is run several times under low load (30EBs), and the corresponding queries have been collected. To generate workload representating more users, we merge these small workloads. For exemple, a workload of 300 EBs is the result of the merge of ten workloads of 30 EBs. Replaying a query workload allow us to focus on the performances of the database tier alone with no interference from the application itself.

All the experiments have been performed on the DAS-3 cluster [19]. Each server, running Linux, has two AMD Opteron dual-core processors running at 2.4 Ghz with 4GB of memory. They are connected to each other with gigabit LAN, so the network latency between the server is negligible.

We conducted several experiments that implement a reconfiguration from a cluster of k database servers to a cluster of $k+1$. The queries have been numbered and packed in group. We have plotted the 90^{th} percentile of the response time for each groups. That means that 90% of the queries are answered faster than the duration plotted. The size of one group depends on the experiment. All experiments were made on a relatively loaded system. For each experiment, we selected the highest multiple of 30 EBs such that 90% of the queries were returned within 100ms. The experiments start with a warmup time of $100,000$ queries, to allow the database servers to load all their data. Next, we plot $100,000$ queries without any table copy to see the normal behavior of the system. The migration starts at the query $200,000$. At the end of the migrations, we still process $100,000$ queries to observe the behavior of the system after the migration.

Figures 13 and 14 show the system performance during a reconfiguration of RUBBoS from 1 to 2 database servers, and from 4 to 5. The scheduling of reconfiguration actions here are slightly different than those described in section 4. We forced all the table deletions to the end, which reduces the number of tasks to schedule and allows to compute the optimal schedule as a reference. We have selected the migration from 1 to 2 and from 4 to 5 because, in these cases, the greedy algorithm did not find the optimal scheduling. We can thus

compare the response time during a migration with a greedy scheduling, an optimal scheduling (as computed with the metric of Section 4), and without any migration. It appears that the greedy scheduling is really close to the optimal one. There are some load spikes, but they also appears without migration, and are just consequences of peaks in the workload itself. Moreover, if we compare the greedy scheduling response time and the response time without migration, we see that the greedy one is almost always lower. Its response time is worse mostly at the begining and before the migration (due to imprecision in the measures). This is logical, at the very begining of the migration, both systems are identical except that one server is being dumped. The response time always stay in acceptable range (less than 100ms), except during peaks in the workload. This means that our migration process is truly practical, and can be used provided that the system is not already overloaded (where queries arrive faster than the system can process them). However, these schedulings appear to be less and less usefull as the number of servers raise. It is due to the fact that the copies are only between two servers, and their changes have no consequences for the full system.

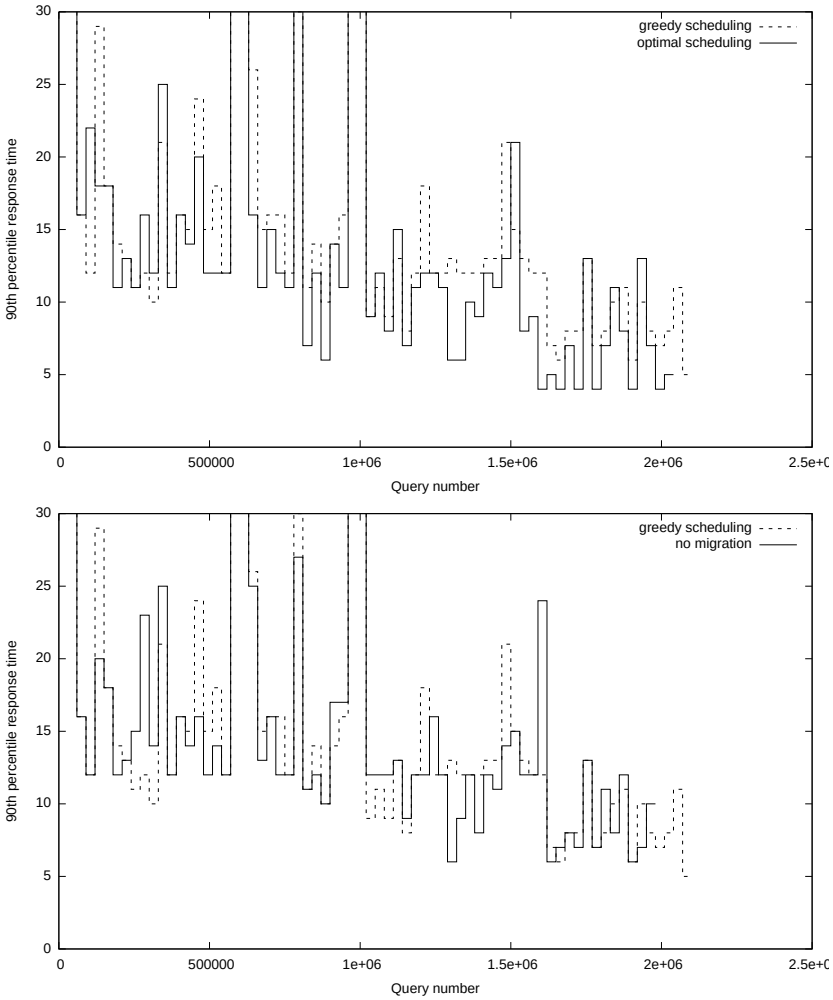

Figure 14: Response time of the system during the migration from 4 to 5 database servers with RUBBoS (660 EBs)

6 Conclusion

We have proposed a way to reconfigure a database replication system based on partial replication. This process has been optimized, first to minimize the number of copies and second to exploit the partial replication to use the new servers as soon as possible. This allows to reconfigure the system, for example, to add some servers to the cluster or to change the application code and introduce new query templates.

The migration process is practical. After a short period, the load of the system is lower than without migration, while copies are still processed. Thus, if the database is not dramatically overloaded, the reconfiguration is feasible and the queries reponse time is quickly reduced.

We have not yet studied the practical possibility to perform parallel migra-

tions. Tables are copied one after the other, but, as the effect of the copy is limited, we may speed up the process by, for example, allowing the server 1 to get a table from the server 0 while the server 3 gets a table from server 2. The system may need to do this after some first copies to be less loaded during the multiple copies. The greedy scheduling could be used for each server. Each database receives the list of the tables it will receive, schedules this list with the greedy algorithm, and does its copies sequentially.

Finally, I would like to thank my supervisors at the Vrije Universiteit, Guillaume Pierre and Paolo Costa. I am particularly gratefull to them for their guidances and constructive criticisms about my work and their understanding, even of my english. Distributed systems and web hosting was not my speciality, and I have learnt a lot of things with them.

References

[1] T. Groothuyse, S. Sivasubramanian, and G. Pierre, "Globetp: template-based database replication for scalable web applications," in *WWW '07: Proceedings of the 16th international conference on World Wide Web*, (New York, NY, USA), pp. 301–310, ACM Press, 2007.

[2] "http://www.tpc.org/tpcw/."

[3] J. Challenger, P. Dantzig, A. Iyengar, and K. Witting, "A fragment-based approach for efficiently creating dynamic web content," *ACM Trans. Inter. Tech.*, vol. 5, no. 2, pp. 359–389, 2005.

[4] C. Olston, A. Manjhi, C. Garrod, A. Ailamaki, B. Maggs, and T. Mowry, "A scalability service for dynamic web applications," 2005.

[5] S. Sivasubramanian, G. Pierre, M. van Steen, and G. Alonso, "Globecbc: Content-blind result caching for dynamic web applications," *Submitted for publication, Vrije Universiteit, June*, 2005.

[6] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan, "Dbproxy: A dynamic data cache for web applications," *icde*, vol. 00, p. 821, 2003.

[7] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 173–182, ACM Press, 1996.

[8] C. Pu and A. Leff, "Replica control in distributed systems: as asynchronous approach," in *SIGMOD '91: Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 377–386, ACM Press, 1991.

[9] D. Agrawal, G. Alonso, A. E. Abbadi, and I. Stanoi, "Exploiting atomic broadcast in replicated databases (extended abstract)," in *Euro-Par '97: Proceedings of the Third International Euro-Par Conference on Parallel Processing*, (London, UK), pp. 496–503, Springer-Verlag, 1997.

[10] B. Kemme and G. Alonso, "Don't be lazy, be consistent: Postgres-r, a new way to implement database replication," in *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, (San Francisco, CA, USA), pp. 134–143, Morgan Kaufmann Publishers Inc., 2000.

[11] M. Ronström and L. Thalmann, "Mysql cluster architecture overview," tech. rep., MySQL Technical White Paper, April 2004.

[12] C. Plattner and G. Alonso, "Ganymed: scalable replication for transactional web applications," in *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, (New York, NY, USA), pp. 155–174, Springer-Verlag New York, Inc., 2004.

[13] G. Soundararajan and C. Amza, "Reactive provisioning of backend databases in shared dynamic content server clusters," *ACM Trans. Auton. Adapt. Syst.*, vol. 1, no. 2, pp. 151–188, 2006.

[14] G. Soundararajan, C. Amza, and A. Goel, "Database replication policies for dynamic content applications," in *EuroSys '06: Proceedings of the 2006 EuroSys conference*, (New York, NY, USA), pp. 89–102, ACM Press, 2006.

[15] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[16] "http://www.postgresql.org/."

[17] J. Munkres, "Algorithms for assignment and transportation problems," *Journal of the Society for Industrial and Applied Mathematics*, vol. Volume 5, March 1957.

[18] "http://jmob.objectweb.org/rubbos.html."

[19] "www.cs.vu.nl/das3/."