Dynamic Allocation in a Self-Scaling Cluster Database

Tilmann Rabl, Marc Pfeffer, and Harald Kosch

Chair of Distributed Information Systems
University of Passau
{rabl,pfefferm,kosch}@fim.uni-passau.de

Abstract. Database systems have been vital for all forms of data processing for a long time. In recent years, the amount of processed data has been growing dramatically, even in small projects. Nevertheless, database management systems tend to be static in terms of size and performance, which makes scaling a difficult and expensive task. This is especially an acute problem in dynamic environments like grid systems.
Based on the C-JDBC project, we developed a cluster database that supports automatic scaling in the number of nodes used. To ensure good load balancing, we distribute the tablespace according to a statistical analysis of the query history. In this paper we will focus on the allocation algorithm.

1 Introduction

Nowadays databases are often accessed with prepared SQL statements, hence there are only few distinct queries. These can therefore be divided into classes, where each class holds queries which access the same set of relations. This gives a chance for optimizing the placement of the relations. If the relations referenced in one class are placed in one backend of a multidatabase, all queries of that class can be answered locally on that backend. In order to keep up with the speed of a fully replicated system, the relative amount of work of each query class has to be taken into account.

Consider the example in figure 1. We have a database with 3 relations A, B and C and three classes of queries C_1, C_2 and C_3, where queries of class C_1 reference relations A and B, queries of C_2 reference C and queries of C_3 reference all relations. The load of the database is divided as follows, C_1 produces 50% of the overall amount of work, C_2 produces 35% and C_3 the remaining 15%. For one database backend there is only one option, it has to hold all relations. Yet for two backends we can already save disk space if we do not replicate all relations on the second backend but only A and B. We still can keep both backends busy since we can schedule all queries of class C_1 on the second backend and all other queries on the first. This way both backends should have an equal workload. With three backends, we can reserve one backend fully for C_1 queries and one for queries of C_2, so we have only one instance with all relations. The workload can still be balanced, if we schedule some queries of C_1 and C_2 on the backend with

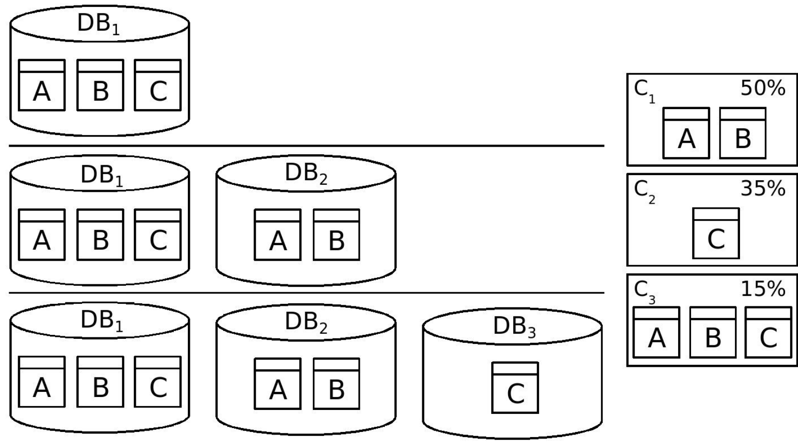

Fig. 1: Three Scaling Sizes of a Database on Basis of Three Query Classes.

all relations. Of course this leads to a substantial saving of disk space. Another advantage is that employing a new backend is less expensive, since we only need to copy some relations rather than all. The old backends in our example stay untouched; this is not possible in general, but we try to keep the overall changes minimal.

We developed an algorithm for the efficient calculation of good placements of relations in a shared nothing database cluster. Further on we have extended the Sequoia project (a continuation of the C-JDBC Project [CMZ04]) and realized a prototypical self-scaling cluster database system that makes use of this algorithm.

The remainder of the paper is structured as follows. Section 2 gives an overview of the architecture of our system. In section 3 we explain the dynamic aspects of scaling and allocation used. The allocation algorithm is explained in detail in section 4 and performance evaluations in section 5. We discuss related work in section 6 before concluding in section 7.

2 System Overview

We use Sequoia as the basis of our implementation [SEQ]. Sequoia is a middleware that allows transparent access to a cluster of database systems via JDBC [JDB]. This means that the cluster is accessed like a single database instance and that the middleware takes care of the query scheduling, update propagation and transaction management. Additionally, it offers simple load balancing, recovery and caching features. Its architecture is based on a technique called Redundant Array of Inexpensive Databases (RAIDb) [Cec]. RAIDb defines a shared nothing architecture for database clusters and uses the read-once/write-all protocol [OV99].

Three modes of operation were defined to adjust the degree of replication: partitioning (RAIDb-0), full replication (RAIDb-1) and partial replication (RAIDb-2). For our purposes, partial replication is the most interesting approach. Read requests are only executed by one database backend, while write requests are broadcasted to all backends that have allocated the appropriate relations. Figure 2 shows the architecture of a RAIDb cluster. All requests to the cluster are handled by the RAIDb controller. The data is stored in the backends, which are standalone database systems. Depending on the load of the backends and the relations referenced in the queries the controller sends read requests to one backend and write requests to all backends with the according relations. The backends process the queries and send the result to the controller. For larger clusters (> 25 backends) there can also be more than one controller.

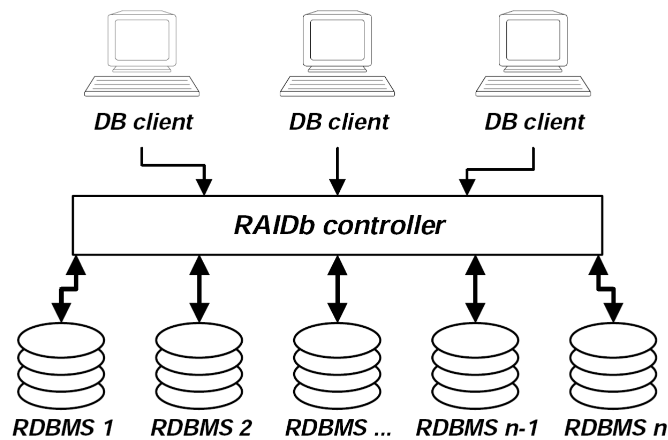

Fig. 2: RAIDb Architecture Overview (cf. [Cec]).

In this paper we present an extension of the Sequoia middleware that adds dynamic aspects to the functionality of the middleware. We added methods to adjust the allocation of the relations and degree of replication and to scale a cluster at runtime. Our software communicates via a JMX/RMI interface with the Sequoia controller. This approach makes it possible to manage a cluster consisting of more than one controller (see controller replication in [SEQ]) and to keep changes to the Sequoia code base as small as possible.

Our tool mainly consists of three components: a monitoring component, a strategy component, which implements the allocation algorithm presented in the next section, and an execution component, which executes all actions on the controller or the database backends.

Monitoring includes collecting data from the backend database systems (i.e. memory- and CPU-utilization and available disk space) and storing statistics of

requests submitted to the cluster. This information is stored in an embedded database for further analysis.

The strategy component has two main functions. First, it decides whether the cluster should be resized by adding a new database backend or by dropping an existing one. The decision is based on a scoring model and the data collected from the database backends. The second function is to calculate a data replication scheme based on the query history. This scheme balances the workload uniformly over the entire cluster and minimizes data replication.

The execution components performs the set of actions identified by the strategy component. It contains a simple task scheduler, which executes a series of actions in parallel. This may include adding a new node to or removing a node from the cluster, setting up the database system on the node and replicating or deleting data on the database backends. To improve performance, database specific tools such as bulk loaders etc. are used.

3 Allocation and Scaling Procedure

As explained our extensions do not change the functionality of the Sequoia system itself. They extend the system so that it can dynamically adjust the allocation of the relations and automatically scale itself. This way it is possible to start the system with one node and avoid further configuration.

3.1 Automatic Scaling

During runtime the monitoring component periodically measures the utilization of the single backends and the overall system. The measurements include query queue lengths, memory and disk utilization, processor load and other values. For each of the readings upper and lower thresholds are defined. A modular policy, which can be changed at runtime, defines the actions that are taken when these thresholds are exceeded. We chose a policy that automatically scales the number of backends. So if the cluster is under heavy load for a certain period of time a new backend is automatically acquired. When the load is to little the number of backends is reduced.

The first step in upscaling is the selection of a new node which will host the new backend. The set of possible hosts is held in a list, which contains each nodes IP address, username and password. The node with the fastest ping answer is chosen, this is a very simple strategy but again this policy is held modular, so it can easily be enhanced. When a node is selected the execution module establishes a SSH connection to the node and transmits the database software. Then database is started and integrated in the Sequoia cluster. To fill the database the allocation procedure is triggered.

For downscaling a node has to be removed from the cluster. The node, which will be removed, is again selected by a modular policy. We select the node with the least amount of diskspace. Before the backend can be deleted the data has to be transfered to the remaining backends. To do this we start the allocation

procedure without the selected backend. The data then is transfered according to the new allocation and the backend can be removed from the cluster. The database on the removed node is stopped and all data is removed.

3.2 Dynamic Allocation

The monitoring component stores the complete query history and the according execution times. From this history a classification of the request is calculated periodically (see section 4.1). Since the query types may vary we only use the recent queries for the classification. The interval that defines the recent queries influences the dynamic of the allocation. This depends on the application domain. To reduce effects of short-term variations it is possible to define a second interval of older queries, which are also taken into account for classification. The influence of the two intervals can be weighted.

From this classification an allocation scheme is calculated, as explained in the next section. This calculation does not take the actual configuration of the backends into account. Therefore, after the calculation the new allocation is compared with the current configuration and an optimal matching is computed (see section 4.5). To implement the matching efficiently the copying of relations is done in parallel. Unused relations are deleted after the copying. Whereever possible the whole database of a backend is transfered, since this way all indices and constraints can be saved and do not have recomputed. The detailed allocation algorithm will be described in the subsequent sections.

4 Allocation Algorithm

In this section we present an algorithm that calculates the placement scheme for the relations. Since Sequoia lacks distributed joins and relation fragmentation capabilities we have to allocate the relations in a way that all queries can be processed by a single backend. With a good allocation, the workload on all database backends of the cluster should be balanced.

4.1 Classification of Requests

In order to calculate a good allocation, a classification of all requests is needed. As any request must be processable on a single database backend, all relations referenced by this request must be available locally. Thus, all requests submitted to the cluster within a given period of time are classified based on the relations they are referencing. Because of the replication model we also need to to distinguish between read-only requests and updates. So we create two profiles, one for reads (Q) and one for updates (Q_u).

After specifying these classes, the ratio of the overall workload for each class is calculated with the cost function c_Q. The cost function measures the relative amount of the total execution time. For every request class C_i a value $a_i = c_Q(C_i)$ is calculated, where $0 \leq a_i \leq 1$ and $\sum a_i = 1$.

4.2 Problem Definition

The main goal of the presented method is to keep a balanced workload while minimizing the replication rate. The limitations of the Sequoia implementation have to be considered. To achieve this, the allocation calculated by evaluating the classification of the requests, and minimizing disk usage (another possibility is the minimization of the number of replicated relations in the cluster or minimization of the global update workload by minimizing replication of updated relations (see Q_u in 4.1) or a weighted combination of the mentioned cost functions). The problem is formalized as follows:

Let $R = \{r_1, \ldots, r_k\}$ be the global database schema. Each relation r_i has a cost $c_S(r_i)$ as evaluated by the cost function c_S. In our case $c_S(r_i)$ corresponds to the physical size of r_i. Furthermore, let $Q = \{C_1, \ldots, C_m\}$ be a request profile consisting of m request classes C_j, where $C_j \subseteq R$. A value $a_j = c_Q(C_j)$ is computed for each request class C_j for cost function c_Q. Then $\sum_{j=1}^{m} a_j = 1$ holds.

For each of the n database backends, a set of relations $R_i \subseteq R$ is computed, so that each backend gets an equal fraction of the whole workload.

The following sum has to be minimized:

$$\min \sum_{i=1}^{n} \sum_{l=1}^{k} (\text{ if } r_l \in R_i \text{ then } c_S(r_l) \text{ else } 0) \tag{1}$$

where :

$$R_i = \{\cup C_j | x_{ij} > 0\} \tag{2}$$

x_{ij} is the ratio of workload from request class C_j that is assigned to backend i.

The following constraints hold:

$$\sum_{j=1}^{m} x_{ij} a_j = \frac{\sum_{j=1}^{m} a_j}{n} = \frac{1}{n} \tag{3}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \tag{4}$$

$$0 \leq x_{ij} \leq 1 \tag{5}$$

This problem is presumably NP-hard. Therefore a heuristic method is presented that solves the problem efficiently and with adequate quality. This is done in two steps. First a greedy algorithm computes an initial solution which is then improved by simulated annealing.

4.3 Heuristic for a Initial Solution

For the efficient calculation of an initial solution to the problem stated above, we developed a greedy algorithm. Due to the nature of the problem the greedy approach is not optimal. It can however still be further improved, as will be explained later on.

The problem is similar to Bin packing or Knapsack, therefore a greedy algorithm analogously packs request classes into database bins. Each of the n backends b_i has a capacity of $1/n$. Our target is to pack all m request classes C_j into the n backends so that all capacity constraints are taken into account and that the cost function c_S is minimized. In contrast to the classical bin packing problems, the set of objects in the bins are not disjunctive. That means that for two request classes C_i and C_j mostly $C_i \cap C_j \neq \emptyset$ applies.

As a first step of the algorithm, the request profile is sorted in descending order, so that the most expensive elements in terms of cost function c_S are processed first. Then for all not completely filled backends, a precedence value is computed. For a given request class C_j, the precedence value is computed by comparing the difference between the size of the intersection and the size of the union of C_j and relations already packed in a backend by the means of cost function c_S. Then the class is put in the backend with the maximum precedence values. By this approach request classes with many relations in common are put in the same backends. An outline of the algorithm can be seen below.

```
Data: request profile Q, backends B
Result: allocation schema
sort(request profile Q) according cost function c_S descending ;
foreach C ∈ Q do
    foreach b ∈ B do
        if b.free_capacity > 0 then
            UNION ← b.relations ∪ C.relations;
            INTERSECT ← b.relations ∩ C.relations;
            diff[C, b] ← c_S(INTERSECT) − c_S(UNION);
        else
            diff[C, b] ← −∞;
    while C.rest_workload > 0 do
        b ← b ∈ B where diff[C, b] maximal;
        workload_for_b ← min(b.free_capacity, C.rest_workload);
        b.add(C, workload_for_b);
        C.rest_workload ← C.rest_workload − workload_for_b;
        diff[C, b] ← −∞;
```

Procedure `Greedy Heuristic`

The result of the algorithm can be formulated as a $n \times m$ matrix where every row corresponds to a backend and every column to a request class. Each value $0 \leq x_{ij} \leq 1$ indicates which ratio of a request class C_j is packed on a backend b_i. A database backend b_i holds all relations $r \in \bigcup C_j$ for which $x_{ij} > 0$.

4.4 Post Optimization Method

We improve the quality of the initial solution by a post optimization phase. *Simulated Annealing* [KJV83], an algorithm for a local search, is used as a meta heuristic to improve the data replication scheme.

This method starts with an initial solution and then iteratively moves to a neighbor solution. The best solution with respect to our cost function c_S is returned. Therefore, a neighborhood relation has to be specified. In our case, a neighbor solution is an allocation where one request class is swapped with another request class from another backend.

4.5 Cost Minimal Implementation

The algorithm presented above bases its computation of a new replication schema solely on the size of the cluster and on the profile of the requests. This does not take the current allocation into account. To get a cost minimal implementation of the new data replication scheme a matching method is used to compute a cost minimal perfect matching. In our case the size of the data to be transferred in the cluster is the cost.

Let $G = (V_1 + V_2, E)$ be a complete weighted bipartite graph, where $|V_1| = |V_2| = n$ holds. Each node $v \in V_1$ represents one of the n new data replication schemes for a given database backend, while each node $u \in V_2$ represents a database backend with its currently allocated relations. Let $R_v^{new} \subseteq R$ be the relations associated with a node $v \in V_1$ and $R_u^{old} \subseteq R$ be the relations already allocated on a node $u \in V_2$. For each node $v \in V_1$ and $u \in V_2$, a weighted edge e_{vu} is added to G so that G becomes a complete graph. The weight of edge is computed as follows: $w(e_{vu}) = \text{size}(R_v^{new} \setminus R_u^{old})$.

A cost minimal perfect matching can be computed efficiently ($O(n^3)$) on this graph with the *Hungarian Method* (Kuhn-Munkres algorithm) [Kuh55]. This matching determines which database backend will host which of the new data allocation schemes. According to the schemes relations are newly replicated on a backend or dropped from a backend.

5 Performance Evaluation

Our implementation favors read dominant database applications, therefore we chose a slight modification of the TPC-H benchmark to test our system and evaluate its performance. Additionally we ran a TPC-W like workload to examine performance in an OLTP like setting.

5.1 TPC-H Benchmark

The TPC Benchmark H (TPC-H) is a decision support benchmark that defines a set of queries with a high degree of complexity [PF00]. It consists of two test scenarios: A power test that defines a sequence of requests sent to the

database sequentially, surrounded by two refresh functions, and a throughput test that defines a set of query streams that are executed in parallel together with the refresh functions. At this we omit the updates because we want to test a readonly scenario. The design of the Sequoia system does not allow any speedup for the power test, since each request is always processed by a single backend. We therefore restrict our evaluation to the throughput test.

5.2 TPC-W Benchmark

The TPC Benchmark W (TPC-W) is a business oriented transactional web benchmark. It simulates a common eCommerce scenario. There are three kinds of workload profiles defined by the benchmark. They only vary in ratio of browse to buy interactions and accordingly in ratio of read to write interactions from the database's point of view. These profiles are (in descending order of read to write ratio): browsing (WIPSb), primarily shopping (WISP), and web-based ordering (WISPo).

Because we are only interested in database performance, we used a multithreaded trace player to emulate a TPC-W application server. For our performance evaluation we used a trace of 40000 transactions from the browsing mix generated with a open source implementation of the TPC-W benchmark [Mar01]. The TPC-W scaling parameter were set to 10000 items and 288000 customers. The number of emulated clients (EB's) was set to 100.

5.3 Experimental Setup

As stated above our implementation is based on Sequoia; we used a slightly modified version 2.10.6. To get significant results, we compared the performance of a normal RAIDb-1 setup to that of a RAIDb-2 setup that is based on our calculations. For both setups we used the least pending request load balancer (as proposed in [CMZ]). Result caching features of the Sequoia controller were disabled.

As database system for the backends we used Apache Derby [DER]. Derby is an open source relational database that is completely written in Java. We used the latest developer version (10.3.0.0 alpha) because there were some unresolved performance issues regarding multi-user performance in the latest stable release (10.2.2.0).

All tests were run on a Linux platform with a 2.6.19 kernel. We used SUN's Java Runtime Environment 1.6.0 for the database backends as well as for the Sequoia controller and for our tools.

We tested our system on up to ten database backends. Each backend was hosted on a PC with an Intel Pentium IV 2.8 GHz CPU and 1 GB RAM. The Sequoia controller and our tool ran on a separate machine of the same type. We used two (one) Intel Core 2 Duo E6400 machines to generate the workload for the TPC-H (TPC-W) tests. All machines were connected via a switched 100 Mbps Ethernet LAN.

5.4 Experimental Results

TPC-H Our performance analysis is based on the time needed to complete a throughput test for each configuration. The results are presented in average SQL requests per minute. To get a heavy workload on the system, we always used twice as many query streams as backends. We report the best results out of three runs for each configuration. For RAIDb-2 we executed our allocation algorithm after each run. Figure 3 shows the results of our throughput tests.

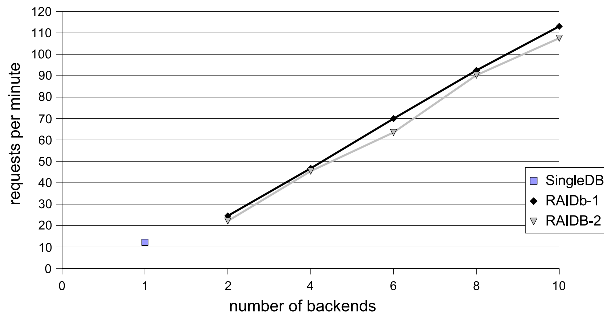

Fig. 3: Throughput RAIDb-1 vs. RAIDb-2.

The figure shows that throughput almost increases linearly with the size of the cluster in both replication modes. In this read only test setup this can be expected for a RAIDb-1 configuration. But the speedup of the RAIDb-2 configuration shows that our allocation does only slightly decrease the speed of the system.

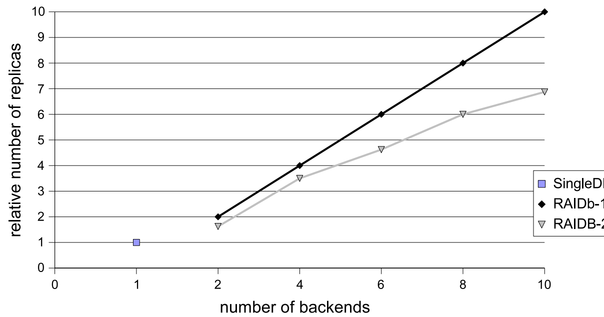

Fig. 4: Average Number of Replicas per Relation in RAIDb-1 and RAIDb-2.

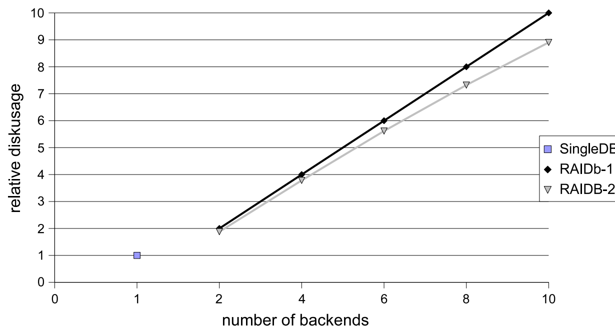

Fig. 5: Comparison of Diskspace Usage in RAIDb-1 and RAIDb-2.

Figures 4 and 5 show the relative number of replicas and usage of disk space. For RAIDb-2 the allocation algorithm tried to minimize disk space usage. If full replication is used the number of replicas and therefore the required disk space grows according to the size of the cluster. For ten backends each relation is replicated ten times and therefore ten times more disk space is used. With our partial replication strategy the number of replicas and disk space usage grows as well, but at a lower rate.

In the TPC-H benchmark the relation LINEITEM is referenced in almost every request class. Since it has three quarters of the whole data volume the volume difference is not as significant as the number of relations. Especially because for cluster sizes up to 16 nodes every node has a copy. Nevertheless, for 10 nodes we save 10 percent disk space and have already 30 percent less replicas in the system.

The results show that a cluster using the partial replication approach is competitive in performance to one using a full replication approach.

TPC-W We measured the time of running the trace of 40000 transactions for each configuration. We report the average number of transactions per second. As with TPC-H results, we report the best result out of three runs for each configuration. Because the TPC-W benchmark contains updates, we modified the cost function c_S (see 4.2) to enable our algorithm to take updates into account. c_S is now a weighted sum of two costfunction c_{S_1} and c_{S_2}, where the first one measures the relative amount of diskspace consumed by a relation ($0 \leq c_{S_1}(r_i) \leq 1$) and the other one measures the ratio of update workload associated with a relation $0 \leq c_{S_2}(r_i) \leq 1$. Both were weighted equally. Figure 6 shows the results of our throughput tests.

Contrary to the read only TPC-H benchmark, the average throughput does not scale linearly with the number of backends. This is due to the replication model used by the Sequoia controller. Using synchronous replication lacks scalability with updates because they have to be performed by every backend having

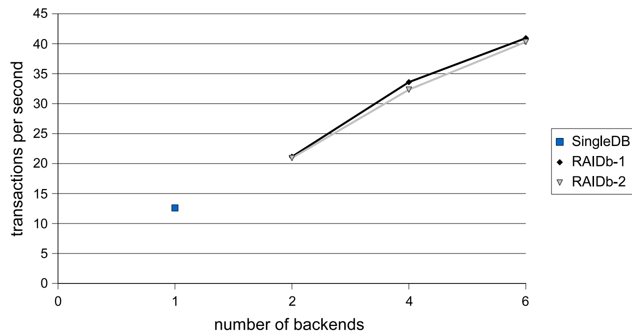

Fig. 6: Throughput RAIDb-1 and RAIDb-2.

a replica. Nevertheless there is a gain of performace when increasing the number of backends.

Figure 7 shows the relative amount of diskspace needed for full replication (RAIDb-1) and partial replication (RAIDb-2) configuration. With an increasing number of backends our allocation algorithm gives a huge gain in saving diskusage. For a six node configuration, we are already saving over 50% of diskspace compared to a RAIDb-1 configuration.

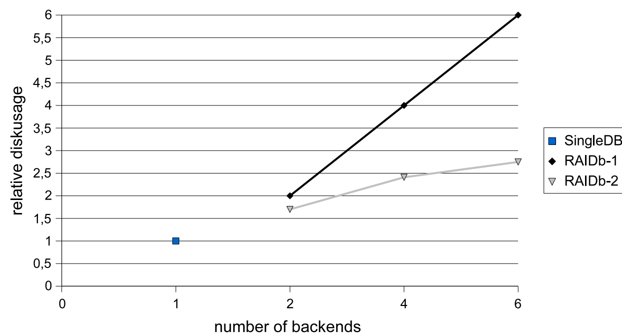

Fig. 7: Diskusage for TPC-W with RAIDb-1 and RAIDb-2

The results show that also in the context of an OLTP like workload our dynamic data allocation approach has the potential to perform equally to a full replication system while saving a great amount of resources.

6 Related Work

As stated above, our implementation is based on the Sequoia cluster database middleware. Since we did not change any of the functionality of Sequoia, our system has the same features. Sequoia has several limitations, which we addressed in this project. The major problem in Sequoia is the lack of distributed joins. This implies that if the system comprises no node containing all relations some queries cannot be processed. This can also be a problem in our system, however with the periodical allocation that is performed, all previously submitted queries will be processable after a new allocation phase. Another drawback is the scheduling in Sequoia. Usually the first backend has the highest workload, while the last is often idle. With our adapted allocation the workload is distributed automatically, since backends are tailored for certain request classes.

The Ganymed project provides a middleware layer that allows replication of database engines and transparent access [PAÖ]. The system is divided in a master and several satelite databases, where the satelites allow only read access and the write access is done only on the master. The consistency of the databases is guaranteed thanks to snapshot isolation. The goal of this project is to house multiple database instances in a single database system. The system is very fast, since it does not have to process the queries at all, but only has to differentiate between read and write requests. The major drawback of this system is that every copy of the database has to be a complete replication which implies that every satelite database uses as much disk space as the master. Also there is no dynamic aspect in this system, so the number of instances and satelites has to be set by hand.

An early approach was presented in [EK]. The architecture proposed is similar to the one in the Ganymed project. In contrast to Ganymed the replicated database backends hold only fragments of the original database and the complete system is designed for read-only access.

A commercial cluster database system is Oracle Real Application Clusters (RAC) [Pru03], it also uses snapshot isolation. In contrast to the projects presented above, RAC has a shared everything architecture. Therefore it relies on the use of specialized hardware like high speed interconnection and shared disks.

7 Conclusion

We have presented techniques for optimizing the allocation in self-scaling cluster database systems. By analyzing the request history, we classify the queries and calculate the allocation of the relations accordingly. The allocation is designed in such a manner that it leads to a balanced load distribution over the cluster and that the disk usage is minimized. We implemented our algorithms in a prototypical system based on the Sequoia project. We also reduced the shortcomings of the original Sequoia system with our dynamic approach. The test results indicate that our implementation has equal performance to the original system while reducing disk usage and improving load balance.

In future research, we plan to address the lack of distributed joins. This will enable the system to distribute the relations of small query classes on several backends and should lead to a significant reduction of the disk space usage. Another issue is the evaluation of the workload: for now we use the query processing time as a measurement, which is not appropriate in heterogeneous environments. To address this issue, we will develop benchmarking tools to estimate the relative speed of the database backends and use more fine-grained cost functions for the query classes.

References

[Cec] Emmanuel Cecchet. RAIDb: Redundant Array of Inexpensive Databases. In *Parallel and Distributed Processing and Applications, Second International-Symposium, ISPA 2004*, pages 115–125, Hong Kong, China, December 2004. LNCS 3358, Springer Verlag.

[CMZ] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. Partial Replication: Achieving Scalability in Redundant Arrays of Inexpensive Databases. In *Proceedings of the 7th International Conference on Principles of Distributed Systems (OPODIS 2003)*, pages 58–70, Martinique, December 2003. LNCS 3144, Springer Verlag.

[CMZ04] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. C-JDBC: Flexible Database Clustering Middleware. In *Proc. USENIX Annual Technical Conference, Freenix Track*, Boston, MA, USA, June 2004.

[DER] The Apache Derby Project. http://db.apache.org/derby/.

[EK] Matthieu Exbrayat and Harald Kosch. Offering Parallelism to a Sequential Database Management System on a Network of Workstations Using PVM. In *Proceedings of the 4th European PVM/MPI Users' Group Meeting*, pages 457–462, Crakow, Poland, November 1997. LNCS 1332, Springer Verlag.

[JDB] Java Database Connectivity (JDBC). http://java.sun.com/javase/technologies/database/index.jsp.

[KJV83] Scott Kirkpatrick, D. Gelatt Jr., and Mario P. Vecchi. Optimization by Simmulated Annealing. *Science*, 220(4598):671–680, 1983.

[Kuh55] H. W. Kuhn. The Hungarian Method for the Assignment Problem. *Naval Research Logistic Quarterly*, 2:83–97, 1955.

[Mar01] Morris Marden. An architectural evaluation of java tpc-w. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, page 229, Washington, DC, USA, 2001. IEEE Computer Society.

[OV99] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[PAÖ] Christian Plattner, Gustavo Alonso, and M. Tamer Özsu. DBFarm: A Scalable Cluster for Multiple Databases. In *Proceedings of the ACM/IFIP/USENIX 7th International Middleware Conference*, pages 180–200, Melbourne, Australia, November-December 2006.

[PF00] Meikel Poess and Chris Floyd. New TPC Benchmarks for Decision Support and Web Commerce. *SIGMOD Rec.*, 29(4):64–71, 2000.

[Pru03] Angelo Pruscino. Oracle RAC: Architecture and Performance. In *ACM SIGMOD Conference*, page 635, 2003.

[SEQ] The Sequoia Project. http://sequoia.continuent.org/.