Dynamic Scale-out Mechanisms for Partitioned Shared-Nothing Databases

by

Alexey Karyakin

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2011

AUTHOR'S DECLARATION

Abstract

For a database system used in pay-per-use cloud environments, elastic scaling becomes an essential feature, allowing for minimizing costs while accommodating fluctuations of load. One approach to scalability involves horizontal database partitioning and dynamic migration of partitions between servers. We define a scale-out operation as a combination of provisioning a new server followed by migration of one or more partitions to the newly-allocated server.

In this thesis we study the efficiency of different implementations of the scale-out operation in the context of online transaction processing (OLTP) workloads. We designed and implemented three migration mechanisms featuring different strategies for data transfer. The first one is based on a modification of the Xen hypervisor, Snowflock, and uses on-demand block transfers for both server provisioning and partition migration. The second one is implemented in a database management system (DBMS) and uses bulk transfers for partition migration, optimized for higher bandwidth utilization. The third one is a conventional application, using SQL commands to copy partitions between servers.

We perform an experimental comparison of those scale-out mechanisms for disk-bound and CPU-bound configurations. When comparing the mechanisms we analyze their impact on whole-system performance and on the experience of individual clients.

Acknowledgements

I would like to thank Professor Kenneth Salem for being my supervisor. I admire his patience, that made the completion of this thesis possible, and I am grateful for his thorough critical comments, that helped me to make it more consistent.

I would like to thank Professor Ashraf Aboulnaga and Professor Raouf Boutaba for being my thesis readers.

I am grateful to University of Waterloo for providing funding for my studies and being an exciting place.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Many applications naturally have highly irregular and unpredictable load over time. Interactive internet services, which have proliferated over the last few years, are great examples of such applications. Supporting a potentially fast usage growth rate as well as accommodating unpredictable load spikes are technically challenging problems. An obvious solution is to over-provision computational resources so that the quality of service is maintained at a satisfactory level most of the time. As a result of overprovisioning, the average load level of servers in datacenters is commonly accepted to be less than 30% [1] and sometimes claimed to be as low as 4% [2], causing economical as well as environmental concerns.

With the advent of commodity virtualization and, as a result, the ability to easily supply computing power in the form of virtual machines (VM), the model of cloud computing has been seen as a solution to the problem of effectively utilizing computing resources. The cloud computing model enables flexible, on-demand resource allocation to users, with a pay-as-you-go payment system. Since the cloud providers utilize big, shared pools of resources, the combination of on-demand allocation with ability of rapid provisioning can theoretically help to accommodate load spikes. This desirable property has been referred to as *elasticity* of services, a term that has been used extensively as a marketing buzzword.

However, the success of the cloud computing model, particularly, in reducing resource reservations, tends to be modest. In practice, overprovisioning has shifted to the customers of "elastic" services, who use the most powerful configurations of virtual machines and keep them running regardless of the actual load [3]. One reason for that is that the time required to adapt is not small enough to react to a spike. The other possible reason is the difficulty of dynamically scaling systems over many nodes. Hence, using the most powerful VMs helps to keep the number of nodes at minimum, and constantly running them avoids dynamic reconfigurations.

Insufficient ability to efficiently utilize dynamically-provisioned resources, or to *scale*, is attributed mostly to software components of systems, with traditional relational database systems being the particular culprit. The problem of database scalability arises because of the significant amount of state, typically located in permanent storage, that has to be moved during a reconfiguration. Additionally, the rich semantics of transactional relational operations is harder to maintain over a dynamic, distributed system.

Research in the area of database scalability and elasticity can be classified into three sub-areas:

Fragmentation. To evenly distribute the load between nodes, it has to be split into fragments that can be processed independently as much as possible. Traditionally, fragmentation is defined manually during the physical design phase. However, there are fully automatic approaches [28]. Unfortunately, finding an exact optimal solution is computationally problematic, therefore statistical approximations and heuristics are used.

Allocation policy. In order to react to the changes in the workload, there has to be a system component to decide which data should be placed onto each processing node and when it should be placed there. The ultimate goal is optimization of resource usage, and hence minimization of costs, while adhering to service level objectives [34].

Efficient reconfiguration. When a decision to redistribute data between nodes has been made, the system should perform a reconfiguration in an efficient manner [30]. We define reconfiguration as a three-step process including: *provisioning* of computing nodes, *migration* of the state between nodes, and adjusting the *routing* of clients' requests to the new locations of their data. Two common specific cases of reconfiguration include *scale-out* and *scale-in*, which correspond to increases and reductions of processing power, respectively. The current activity should be preserved as much as possible, with any possible disruption kept as short and small as possible. Since reconfiguration itself incurs significant costs, and it likely happens at the time when the system is overloaded, this problem is not trivial either. Data migration is often the largest component of those costs because of the amount of database state involved. The design choices for the reconfiguration algorithms influence not only the efficiency of reconfiguration itself, but also the performance *between* reconfigurations, which boils down to the issue of efficiency and processing costs.

The first two subproblems have drawn more attention in the research community, compared to the last one. Efficient reconfiguration has been perceived as an implementation issue, incurring a constant and inevitable cost, and not much worth investigating. This fact has motivated us to study the problem of elastic reconfiguration.

The goal of this thesis is to experimentally study the efficiency of several techniques for reconfiguration of a relational database. In particular, we focus on the *scale-out* procedure during which the number of processing nodes increases in response to increased load. We define efficiency from two, partly conflicting, points of view: the provider's and the client's. From the provider's perspective, an efficient system *as a whole* should produce more work while consuming fewer resources, such as virtual machines, which are usually paid on per-hour basis. From a client's view, a system should maintain predictable performance while causing as little service disruption as possible.

We design and evaluate three approaches for database scale-out. The first one is based on system-level, application-independent cloning of the whole virtual machine with lazy on-demand data transfers. Our implementation uses Snowflock [12] as a base for the scale-out mechanism. Snowflock has minimal "hard" downtime for both the original node and the cloned one but on-demand data transfer can incur substantial penalty in terms of both throughput and degradation of service quality. In addition to measuring performance characteristics and comparing them to other approaches, we discuss the practical issues of using a virtual machine-based approach for dynamic database scale-out.

The second approach is implemented at the database management system (DBMS) level and takes advantage of additional DBMS-specific information to better handle partitioning of the data between nodes. We have developed a prototype for this approach, based on the MySQL open source DBMS. Handling database partitions dynamically involves taking special measures to ensure consistency between nodes. Our system also implements dynamic query routing from the clients to DMBS with no intermediary router and with no modification of applications. We refer to this as the *DBMS-level* approach in this thesis.

Finally, we included for comparison a simple, application-level mechanism that uses normal SQL queries to move data partitions between nodes. This approach requires no modifications to the DBMS except adding partition locking and request routing, which are shared with the DBMS-level mechanism. We refer to this as the *application-level* approach in this thesis.

The contributions of this thesis include:

- design and practical implementations of the Snowflock-based, DBMS-level, and application-level scale-out mechanisms;
- an experimental evaluation of those scale-out mechanisms, studying their relative performance, degree of service disruption, and discussing their suitability and limitations.

Chapter 2

Scale-out Mechanisms

2.1 System Overview

In our study we consider a transaction processing system distributed over a set of storage and processing nodes in the *shared-nothing* architecture. The system contains its state in a partitioned database which may be stored in main memory or in locally-attached disks. The database conceptually consists of many partitions with each of them being dynamically allocated in one of the nodes. The partitioning is organized horizontally so all partitions share the same schema and contain different portions of the same set of tables. Each node hosts one database server process, which is responsible for a subset of database partitions.

We chose the shared-nothing architecture because we believe it provides more opportunity for optimization. Since more data are moved, the efficiency of any reorganization process becomes more important. The problem becomes even more challenging because of different types of memory (RAM and storage) involved. On the other hand, lack of slower centralized shared storage makes it possible to maximize system throughput during normal operations and to avoid the bottleneck of storage scalability.

We assume that the workload is also partitioned such that each transaction is associated with a single database partition. For example, in a Web application each remote user has their own information as a part of the server database which is accessed only when the user works with the application. The absence of inter-partition transactions greatly simplifies our design because in any possible configuration partitions are not split between nodes. As a result, the system does not have to process distributed transactions involving multiple nodes. The processing load may consist of CPU processing and I/O bandwidth usage. Although each partition is associated with a portion of load, some partitions may have more load than others, and that load may vary over time. Therefore, a node having a constant capacity may be able to handle different numbers of partitions at different times. When the load for the partitions on a node becomes too high, the system may move one or more of the node's partitions to a less loaded node along with the corresponding portion of the load, effectively implementing load balancing.

In the cloud computing scenario processing nodes can be allocated on-demand and paid per-use. Thus, by transferring some partitions to a newly allocated node the system can dynamically accommodate more load, i.e. it can *scale out*. The reverse scenario, *scale-in,* is also possible when the partitions are consolidated into smaller number of nodes and extra nodes are deallocated, reducing the operating costs. Ignoring a mechanism to start and stop servers, both scale-out and scale-in can be seen as two specific cases of load balancing. Figure 1 illustrates the components of an example system performing scale-out from one to two nodes, migrating one partition at a time.

We define two major operations which a scalability mechanism implements: *scale-out* and *scale-in*. We will use the more general term *reconfiguration* to refer to either scale-out operation or scale-in operation. The scale-out operation consists of adding (*provisioning*) new node(s) to the system, redistributing part of the database to the new nodes (*partition migration*), and adjusting client-server communication so the clients can work with the new nodes (*routing*). The reverse operation, scale-in, includes steps to consolidate the database into a smaller number of nodes, redirect the clients, and finally remove the now-inactive nodes from the cluster. The scale-out and scale-in operations apply to the whole database cluster, while migration is performed in each partition individually. All the components of scale-out and scale-in operations contribute to the performance impact on the server and client side, which should be accounted for when evaluating the mechanisms.

In addition to partitioned data, the database may contain read-only tables, which are statically replicated at each node.

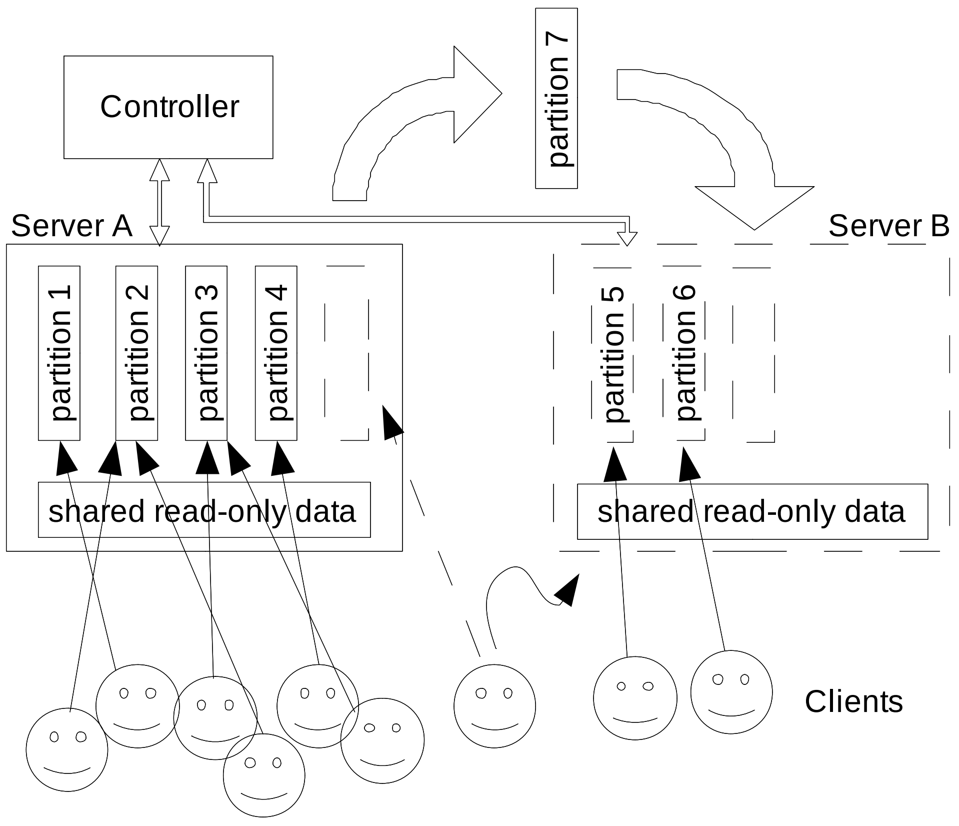


Figure 1. Scale-out scenario with two servers. Partitions 5, 6, 7 are migrating from over-loaded server A to new server B.

A complete elastic database system likely includes a *controller* which decides on when the system reconfiguration occurs and which partitions are to be transferred during this reconfiguration. Although the design of the controller is outside the scope of this work, we assume that a reasonable controller might use the combination of reactive and predictive algorithms to make its decisions. A reactive algorithm detects the overload conditions and triggers a scale-out process that would result in an additional stress to already overloaded system. A predictive approach tries to make reconfiguration decisions before the actual overload takes place, however, it may fail to do that precisely as it is not possible to predict the future.

In the following sections we provide general overviews of the three scale-out mechanisms. A deeper discussion of specific implementation details of the DBMS-level and application-level mechanisms will take place in Chapter 3.

2.2 Snowflock-based Mechanism

The Snowflock-based mechanism implements scale-out operation using Snowflock cloning. In this section we first present the description of the Snowflock architecture and its cloning process, and then explain how cloning is used to scale-out a database.

2.2.1 Snowflock Hypervisor

Snowflock [12] is a modification of the Xen [10] hypervisor, targeting instantaneous and efficient multi-way cloning of a running VM. Snowflock uses the 'vfork' cloning semantics, similar to the 'fork()' system call in UNIX operating systems, where '*v*' stands for *virtual*. In this model, a controlling application, running inside a VM, executes *vfork()* call, specifying the number of requested cloned instances. As a result of the *vfork*() call, the original VM finds itself running independently and simultaneously as the specified number of VMs in addition to the original VM. The underlying Snowflock architecture implements the vfork operation by taking a snapshot of the state of the original VM instance at a point of time and gradually transferring it to multiple cluster nodes.

The cloning process starts by creating a new VM on one of the physical nodes of the cluster, running under control of Snowflock. The allocation strategy may vary but in any case it considers the original VM parameters (number of virtual CPUs and amount of memory) and available resources in the participating nodes. The newly created VM consists of a Xen VM descriptor only and does not have the actual content of the VM's memory and disk. Instead, a special mechanism is set up in the physical node of the clone to request pages from the original node when they are used.

Snowflock then proceeds by creating a snapshot at the original node, containing both the memory and disk state of the original VM. In order to continue execution of the original VM while preserving the snapshot intact, Snowflock uses Copy-on-Write (CoW) strategy. Any modification of a memory or a disk block goes into a *differential* file in which blocks are allocated one by one for every modified block, using extra memory or disk space. In the worst case, the CoW differential file may use the same amount of state (be that memory or disk) as the original snapshot does. Because of CoW, the original VM may continue running after creating the snapshot, while the clones may later reference the VM state which existed at the time of the snapshot.

Finally, the clone VMs are allowed to run. Since they have no memory or disk state at this time, the running clones generate "virtual page faults", causing Snowflock to request the missing pages from the original VM snapshot. There is a performance impact initially as the virtual faults are frequent. However, eventually the working set of the VM gets transferred, allowing the clones to regain their normal performance.

The version of Snowflock we used does not handle open network (TCP) connections in an application-friendly way. The MAC and IP addresses of a cloned instance are updated by Snowflock when it creates a new VM. However, the higher-level state of TCP connections, which existed at the time of cloning, is not cleaned up. These connections continue to exist without any communication until a TCP timeout terminates them.

Snowflock uses a number of optimizations to achieve "instantaneous" initial VM cloning and efficient migration of state afterwards. First, only a minimal amount of data is transferred at the moment of cloning itself. Second, Snowflock uses a multicast protocol to efficiently distribute the pages among all the target nodes simultaneously, which is advantageous when many instances are involved. Third, the Snowflock's on-demand state transfer implementation avoids copying the pages of VM state that are written by the cloned VM without having been previously read. On-demand state transfer combined with the optimizations described above are potentially effective for a wide class of applications. In particular, the applications which have little shared state between the original VM and the cloned instances are able to run when only a fraction of total state has been transferred.

2.2.2 Snowflock-based Scale-out

Snowflock-based mechanism implements scale-out by cloning a database node and separating the load between new nodes so each one handles only a part of the original load. In our experiments we performed cloning during which one copy is created, but, in principle, cloning into multiple copies is possible.

In this mechanism provisioning of new processing nodes and data migration is implemented by a Snowflock cloning operation. The Snowflock cloning *logically* creates one or more exact copies of the *whole* source DBMS state which existed at the moment of cloning. Snowflock's on-demand data transfer mechanism will copy only data which is actually used at the new node. Since the new node becomes responsible only for certain partitions, only those partitions will be physically transferred. The migrated partitions still remain at the original node after cloning. However, since they are not used, they will be evicted from memory by the buffer pool replacement algorithm.

The portions of workload corresponding to the migrated partitions are also transferred to the new nodes. This can be accomplished by the same routing technique (Section 2.5), used in the DBMS-level and application-level scale-out mechanisms. However, due to the difficulty in cleaning up the state of database connections after cloning, we modified the client to disconnect temporarily from the migrating partitions for the duration of the cloning operation.

Migration of arbitrary partitions between running servers is not possible in the Snowflock-based mechanism for the following reasons. First, on-demand data migration is combined with creation of a new VM in a single Snowflock operation (cloning). Second, after cloning a node, the system cannot merge the changes made at the clone back to the parent because the lower-level hypervisor does not possess the information about which blocks of storage correspond to which logical partitions. As a result, only the scale-out operation is possible, with no scale-in or finer-grained load balancing between working nodes.

In order to avoid transferring data which do not belong to the migrating partitions, the data and run-time state of different partitions must be separated into different physical transfer units, which are memory pages and disk blocks. Physical separation of partitions on disk is ensured by the MySQL partitioning storage engine, which places partitions into separate sub-tables. However, some DBMS data structures both on disk and in the memory are shared between all partitions of one DBMS instance. The examples of such disk-based data structures are the transaction log and UNDO segments; numerous in-memory run-time data structures are also shared. Therefore, the actual size of transferred data will be larger than the size of the migrated partitions. This difference will affect the efficiency of the Snowflock mechanism.

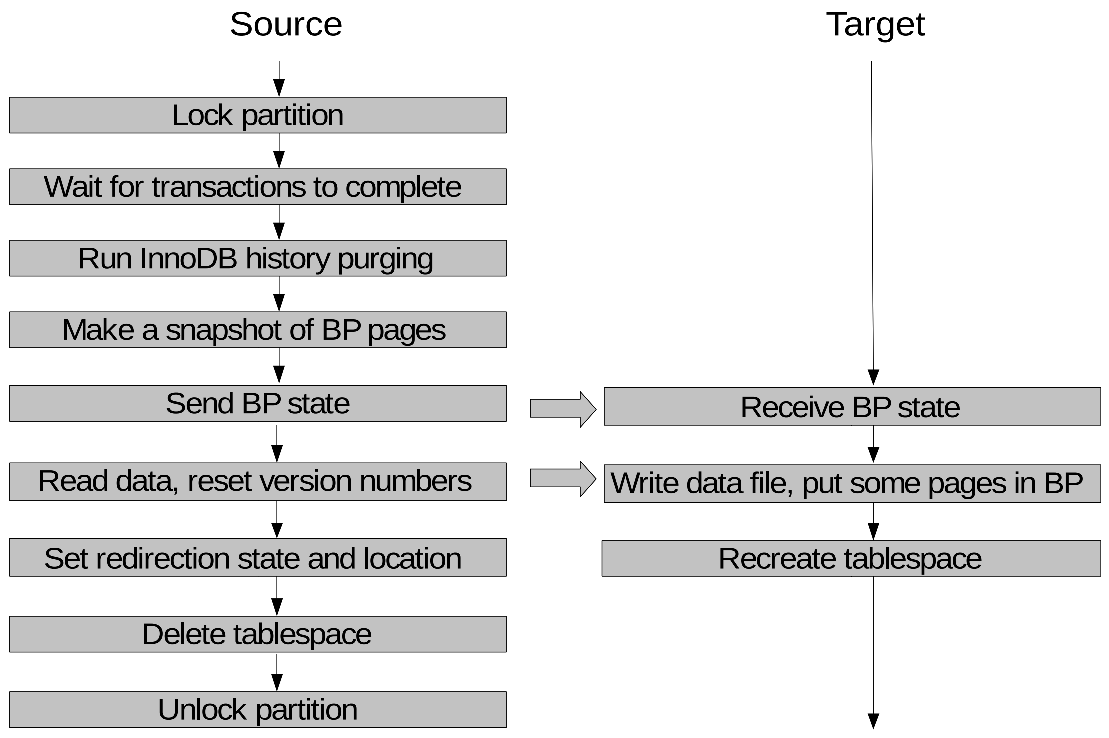
2.3 DBMS-level Mechanism

This mechanism is implemented inside a DBMS (MySQL with InnoDB storage engine in our case) and takes advantage of knowledge of the details of partitioned data physical placement. The mechanism imposes certain limitations on the workload. In particular, multi-partition transactions are not allowed and the partitions affected by the scale-out or scale-in operations are unavailable during these operations. However, by disabling activity inside the migrated partition and by using optimized bulk transfers, a high transfer rate can be achieved.

The scale-out and scale-in operations consist of adding or removing processing nodes (node provisioning) and migrating portions of database state between the affected nodes and the rest of the cluster (partition migration). The implementation of node provisioning is shared with the application-level mechanism and is discussed in Section 2.6.

The stages of the DBMS-level partition migration are shown in Figure 2. When an external application, for example, a load-balancing controller, requests a partition to be moved from one MySQL node to another, it sends a request to the current owner of the partition. The current owner (the *source node* thereafter) blocks new transactions from starting in the affecting partition, and waits until the active ones have completed. Then the source node scans the buffer pool to find the pages from the partition, to be sent to the target node. At the source node, the data is read in large blocks from the disk, and merged with any dirty pages from the buffer pool. At the target node, the data blocks are adjusted to be consistent with the running state of the database and are written to the disk, leaving some pages in the buffer pool. The source partition tablespace then is detached from the InnoDB database, its file is deleted and space is reclaimed. Meanwhile, the received partition data file is attached to the

InnoDB database. Finally, the source instance sets the redirection information for the partition and unlocks it, allowing any pending transactions to be redirected to the target.



Source
Target
Lock partition
Wait for transactions to complete
Run InnoDB history purging
Make a snapshot of BP pages
Send BP state
Receive BP state
Read data, reset version numbers
Write data file, put some pages in BP
Set redirection state and location
Recreate tablespace
Delete tablespace
Unlock partition

Figure 2. Stages of DBMS-level partition migration.

More detailed description of the DBMS-level mechanism will be presented in Section 3.2.

2.4 Application-level Mechanism

The application-level partition migration is implemented with an SQL script which is executed by an external application. The scripts make use of MySQL Federated Storage Engine to establish a remote connection between the participating nodes (the source and target) using the MySQL SQL interface. This way one of the MySQL instances can access the other's tables. The application-level scale-out mechanism uses the same node provisioning implementation as the DBMS-level mechanism.

To migrate a partition, the script reads the rows of a partition from the source node and inserts them at the target node; then it deletes the rows at the source node. Since the SQL statements are executed at one of the nodes, data is transferred directly between them, not involving the host where the script is running. The outline of the migration procedure is depicted in Figure 3.

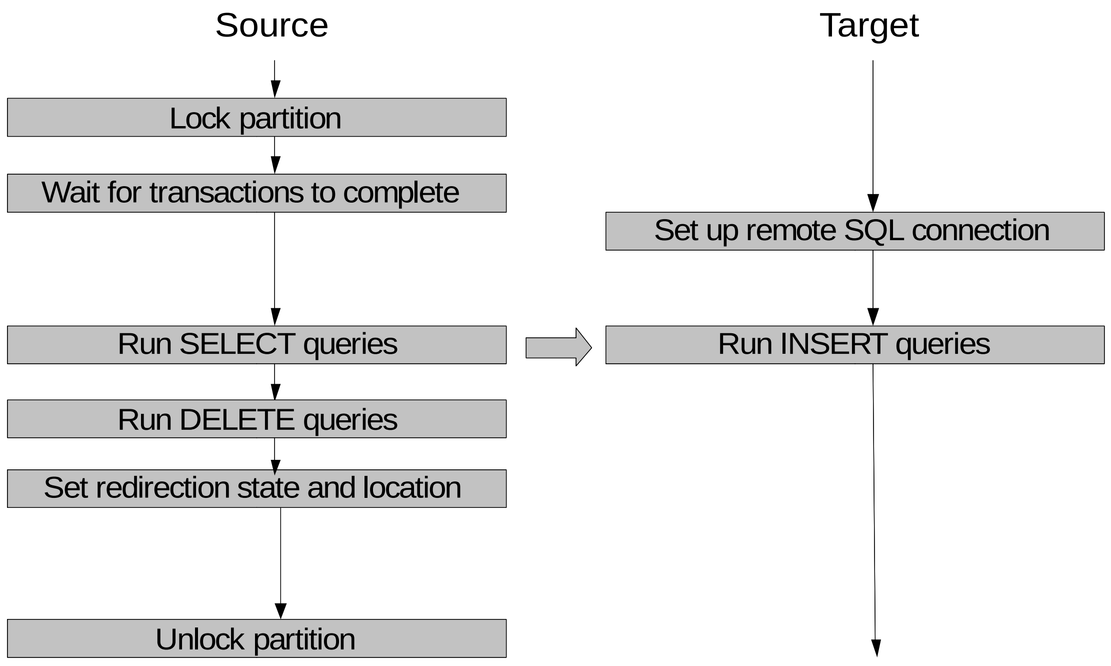


Figure 3. Stages of application-level partition migration.

2.5 Routing and Locking

Both DBMS-level and application-level mechanism share the same routing and partition locking architecture for pointing clients to the current owner of partitions and synchronizing clients' operations with the reconfiguration activities. The clients trying to access a partition that is no longer at the queried node are redirected to the new location by a special message. This approach avoids having a dedicated router, which may become a bottleneck. However, this redirection approach loses efficiency when clients often switch between partitions. Additionally, it raises certain availability issues in some scale-in scenarios.

To make sure a partition does not change while it is on the move, a simple exclusive/shared locking mechanism is used. A transaction places a shared lock on an accessed partition and the controller places an exclusive lock. Anyone trying to set an incompatible lock is forced to wait. Given an in-order policy to grant locks (shared lock requests wait for previously-issued exclusive requests), this mechanism delays the partition migration until all current transactions have completed without allowing new transactions to start. Similarly, newly issued transactions are blocked until partition migration has completed and then they are redirected to the new location.

We did not use this redirection mechanism for the Snowflock-based mechanism. Instead, we modified the workload generator to disconnect from the migrating partitions before the cloning and reconnecting immediately afterwards. The IP address of the new node is communicated to the workload generator by an external procedure which calls Snowflock cloning routine.

The details of partitioning implementation, partition locking and request routing will be further explained in Section 3.1.

2.6 Node Provisioning

In both DBMS-level and application-level scale-out mechanisms we used the same procedure to initialize and terminate nodes.

We assume that physical nodes are allocated from a larger pool of machines which can be reused for various types of applications. Thus, a newly-allocated node has only a hypervisor installed, and any other software or data (operating system, DBMS, initial database state) has to be separately loaded and initialized. Since we use VMs as database nodes, the cost of node provisioning should include the costs associated with bringing a VM into active state. These costs manifest themselves as additional delay between the time a physical machine is allocated and the time database partition migration can start, as well as additional resource usage of the machines that store the source data.

We considered the following candidate ways for provisioning new nodes:

1. Copying a VM disk image to the target physical server and cold-booting the operating system and the DBMS in the VM. The cost of this method includes the cost to transfer the disk image file, the size of which was about 1GB in our experiments, as well as the time to boot the operating system and DBMS.
2. Making a persistent snapshot of a booted but idle VM in advance, transferring the saved state to the new host, and restoring the VM. The cost of this method is determined by transferring both the disk image file and the memory state file, that would account for about 5GB in total in our test configuration, and restoring the VM. This option is similar to the previous one with the exception that the state of a *running* system is transferred. Since the snapshot does not contain data partitions, it may be created as part of an installation procedure and the cost of creating the snapshot may be ignored.
3. Using Snowflock cloning to transfer the running state of the source VM. We considered this method to be the most efficient due to Snowflock's ability to avoid transferring the data pages which are not actually used at the new VM. The main portion of state which will be used in the new node is the state of the running operating system. Since the working set of the operating system is small and most likely to be already in memory, we expected Snowflock to generate almost no disk transfers and memory transfers of a few hundred megabytes. In this method the original DBMS is under load at the time of cloning. To prevent the original workload from executing at the new node and propagating excessive data using the Snowflock on-demand mechanism, the DBMS is forcibly terminated immediately after cloning, then the database is replaced with a new copy which contains no partitions, and the DBMS is restarted.

We chose the third method of new node provisioning because it requires least amount of data to transfer.

Chapter 3
Scale-out Mechanism Implementation

In this chapter we present design decisions used in the implementation of scale-out mechanisms summarized in Chapter 2. Most information in this chapter is related to the DBMS-level mechanism because it required extensive modifications to the MySQL DBMS. The information on partitioning, partition routing and locking is more general and applies to all three mechanism.

3.1 Common Features

The features described in this section are used in the DBMS-level and application-level scale-out mechanisms implementations for managing responsibility for partitions among participating nodes. These features can also be used in the Snowflock-based mechanism as well. However, in our experiments, the responsibility of nodes for partitions was controlled by the client.

3.1.1 MySQL Partitioning

We chose MySQL as a base system for implementing a dynamic partitioning prototype. MySQL includes a Partitioning Storage Engine (MySQL PE) [25] that was useful for our implementation because it physically lays out partitions into non-overlapping blocks. The MySQL PE works as an intermediate between the upper MySQL query execution layers and any other storage engine. When a table is created using MySQL PE, the engine splits it horizontally into a predefined number of subtables, based on any of the partitioning algorithms available in MySQL PE (hash, key lists, or key ranges). Each subtable is stored as a normal table using a lower-level storage engine. In our experiments we used InnoDB, which provides full transactional capabilities, as a lower-level storage engine. InnoDB, in turn, can be configured to store each table, along with its indexes, as a separate file, called a *tablespace* internally. As a result, each partition ends up as a distinct set of disk blocks, represented as a separate data structure in MySQL, allowing us to allocate and deallocate partitions and use low-level binary copying mechanisms to efficiently transfer them between servers.

PE defines partitions for each table *individually*, creating a set of underlying subtables with names composed of the table name as a prefix and partition name as a suffix. There is no way in MySQL PE to define *logical* database partitions, i.e. groups of table partitions that should be handled together. Moreover, there is no easily-available metadata for an application to determine the partitioning configuration, for example, the list of partitions for a table. To simplify the configuration and control scripts, and to avoid maintaining a data structure to correlate logical partitions with their table counterparts, we required that all partitioned tables have equal number of partitions as well as identical partition names. Thus, the set of the lower-level subtables is a Cartesian product of partitioned tables and logical partitions. An example SQL script of a set of tables partitioned in this manner is shown in Figure 4 and the graphical diagram of the same configuration is shown in Figure 5.

In addition to partitioned tables, we support constant, non-partitioned tables that are copied in every server; however, those tables cannot be updated. For example, in TPC-C, the Item table does not scale with the number of warehouses and is not updated by transactions, so we can keep its read-only copy on each site.

3.1.2 Partition Dictionary

To correctly handle the client interaction with partitions that might migrate between nodes, a partition dictionary was added to MySQL PE. Each MySQL PE instance maintains a local copy of this dictionary. The dictionary is indexed by logical partition names and for each partition it stores a flag indicating whether the partition is locally available, its last known whereabouts if it has been migrated, and the partition lock (see Section 3.1.4). Due to the limited scope of this work, the dictionary implementation is incomplete in the following two ways. First, it is non-persistent, meaning the servers participating in an elastic cluster can only start with a predefined partition assignment. In our experiments, the start-up configuration consists of a single server, which is responsible for the whole database. Second, the dictionary is not updated transactionally during the scale-out or scale-in operations. As a result, the partition dictionaries in different nodes may become inconsistent with each other regarding the information on actual partition locations, should a failure occur.

3.1.3 Partition Detection

A distributed system must choose the node to execute each transaction. In our execution model, a transaction is permitted to access only the data contained in one partition in addition to read-only non-partitioned tables. That fact greatly simplifies the node assignment as the very first access to a partitioned table determines the "home" partition of a transaction.

Determining the server at which a transaction should be executed is a two-step process. Firstly, MySQL PE identifies the partitions which are accessed by the query using a technique called *partition pruning*. Secondly, the modified MySQL PE looks up the partition dictionary (see Section 3.1.2) to determine the node which contains the identified partition.

In order to implement automatic partition detection, we looked for the code fragments in the MySQL PE which call the underlying storage engine for each partition. In each such code fragment the partition identifier is known; therefore, we modified the code to check the referenced partition's state in the partition dictionary. If the partition is present in the server currently executing the transaction, we just proceed with the query. Otherwise, the local server had handed the partition over to other server and it knows the address of the next server that was responsible for the referenced partition after the migration from the current node. In the latter case, the MySQL PE sets an error condition which is returned to the client as a special *redirection error code* with a corresponding text message encoding location information (see Section 3.1.5 for more details).

MySQL partition pruning is an optimization technique used by MySQL PE to exclude non-referenced partitions from a query execution. We *assume* the MySQL pruning algorithm works correctly for the class of queries in our workload. However, since the partition pruning algorithm in MySQL PE is merely an optimization technique, its goal is to eliminate *most* of the accesses to unreferenced partitions, without guaranteeing to eliminate *all* of them. Therefore, our assumption may easily not hold for other workloads and our reliance on the MySQL PE partition pruning algorithm for correctness can lead to issues with our mechanism in that case.

If the pruning algorithm fails to eliminate an unreferenced partition from a query, the transaction effectively becomes multi-partition. The effect of this failure is the same as with true multi-partition transactions, which cannot be executed in our mechanism if the partitions are actually located on different nodes. In this case the client will receive a redirection message and will have to abort the transaction. If the client decides to retry redirected transactions using new location information, it will follow the redirection loop without making progress. We consider this problem as an inevitable effect of a automatic association between transactions and partitions.

An obvious way to avoid the problem would be explicit tagging of transactions with their home partition number by an application. This can be done for all the transactions or only for those containing complex queries, for which automatic partition pruning fails. We believe this method would be easy to use in real-world applications. However, we did not implement it because the automatic approach suits the experimental requirements of this thesis.

3.1.4 Partition Locking

In our simplified approach transactions may not access a partition while it is being transferred. We enforce this restriction by associating each partition with a partition lock. Partition locks reside in the partition dictionary. A partition lock can be locked in a shared (S) or exclusive (X) mode. A user transaction that accesses the partition acquires an S lock on this partition implicitly. A transaction from the controller that initiates partition migration acquires an X lock by calling a special SQL function. Any partition locks which a transaction has acquired are kept until the transaction completes.

As usual, S locks are compatible with other S locks and are incompatible with X locks. When such a conflict occurs, the requesting transaction is suspended until the blocking lock is released. The conflict of an X lock request and an existing X lock does not occur because there is only one controller which initiates scale-out or scale-in operations and does not initiate multiple operations over the same partitions simultaneously. The scheduling policy associated with partition locking does not permit a transaction with an S lock request to proceed if a transaction with an X lock request is waiting.

The partition locking implementation has the following effects:

1. A migration procedure requests an X lock on a partition and blocks until all active transactions in that partition are complete.

2. Active transactions are not interrupted and are allowed to complete before the migration procedure starts executing.
3. New transactions in that partition are blocked while there is a migration procedure in progress or waiting for existing transactions to complete.

When a transaction unblocks, the state of a partition has probably changed and the transaction may receive the redirection error code. Since requesting a lock is the only time when a transaction can receive a redirection error code, we can implement both locking and redirection detection as a single procedure that is called from every partition access point.

3.1.5 Client-Server Protocol and Dynamic Routing

Since the distributed configuration of nodes changes during dynamic scale-out and scale-in, each transaction should be able to reach the node which contains the data associated with the transaction. Typically, in similar designs applications connect to a middle-tier router ([28], [33]), which maintains the mapping information and dynamically dispatches transactions to server nodes. This approach may limit the throughput of the system due to introduction of a single component involved in processing of all clients' requests, potentially becoming a bottleneck. Therefore we made the server nodes themselves responsible for maintaining the routing information, and modified the client-server protocol to enable clients to switch between the nodes with no or little modification of applications. Although reconnection is not fully transparent to applications, the applications which abort the current transaction on receiving an unknown error condition do not need any modification.

In order to operate efficiently, the routing architecture depends on an application's temporal affinity to database partitions because there is a cost associated with the switch. That is, the transactions issued from an application using a single database connection should access the same partition, or at least the application should switch partitions infrequently. We believe this assumption holds in practice for partitioned applications; for example, a session in a web application contains multiple user interactions accessing closely-related items.

The routing mechanism operates as follows. Initially, the system starts with a predefined configuration and clients are provided with the initial IP address of the node for each partition. This way clients can execute transactions until a reconfiguration occurs. Since the reconfiguration takes time during which the partition cannot be accessed, the source node delays execution of client's requests upon its detection of the client's access to the migrating partition. The source node also naturally knows the address of the target server for partition migration and it keeps this information for future reference. Once the migration is over, the source node sends a special response message (redirection message), informing the clients of the fact that the partition moved and including the IP address of the target node. The modified MySQL client library handles the redirection message by closing the TCP connection associated with the client's session and updating the server address, stored in the session data structure. Finally, the client application receives an error condition, and aborts the current transaction. The next transaction issued by the application in the same session causes the client library to automatically reconnect, using the stored location information,

```
CREATE TABLE t1
(
      id INTEGER PRIMARY KEY,
      ...
)
PARTITION BY LIST(id) (
      PARTITION p1 VALUES IN (1),
      PARTITION p2 VALUES IN (2),
      PARTITION p3 VALUES IN (3),
      PARTITION p4 VALUES IN (4));

CREATE TABLE t2
(
      id INTEGER PRIMARY KEY,
      ...
)
PARTITION BY LIST(id) (
      PARTITION p1 VALUES IN (1),
      PARTITION p2 VALUES IN (2),
      PARTITION p3 VALUES IN (3),
      PARTITION p4 VALUES IN (4));
```

Figure 4. An example of two-table, four-partition configuration in MySQL.

which now points to the new server for the partition. The process of redirecting a client to a new node after the accessed partition has been moved is illustrated in Figure 7.

In principle, automatic reconnection and transaction retry can be performed by the MySQL client library itself, without notifying the application, because a transaction which is fully contained in a partition receives the redirection error in its first attempt to access data so no transaction state has been updated before this error has been detected. However, for the purpose of the experiments of this thesis, such transparency was not required because the benchmarking program we used already implements the proper error processing.

In its current implementation, the partition location information is stored in the main memory only and is not persistent. Consequently, the start-up configuration must have a predefined partition assignment, for example, one server containing the whole database. A practical implementation should store the location information persistently, for example, as part of a database catalog. The process of handing over the responsibility for a partition between nodes should use a distributed transaction to ensure consistency of that information in a case of a failure during the migration.

As long as the application have error processing logic that leads to a transaction abort after receiving an unexpected error code, this mechanism does not require any modifications to the application. Thus, the changes in the client-server protocol are entirely localized in the database system-provided software.

For each partition each server maintains information about whether the data is available locally, and if not, the address of the server which received the partition when it migrated.

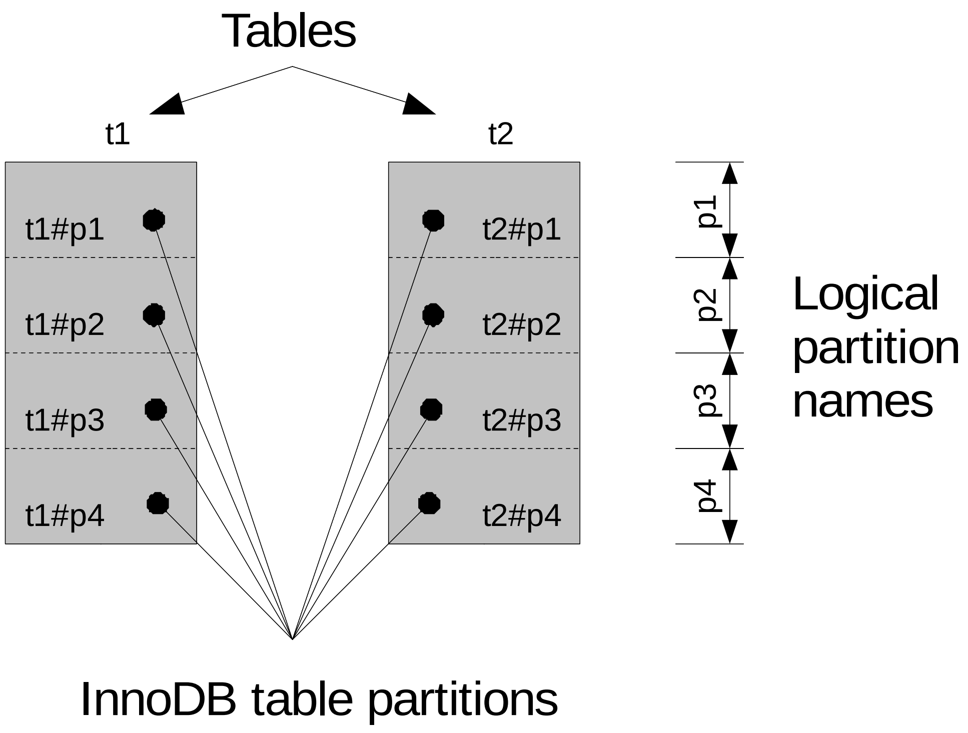


Figure 5. An illustration of the partitioning configuration in Figure 4.

Each partition may migrate multiple times, thus forming a chain of redirection pointers, starting from the initial start-up node. During each migration, only the pointer to the final destination changes, making sure the chain eventually leads to the actual server for that partition, even if one node participates in migrations multiple times with different destinations. As long as the servers in the redirection chain are not shut down, the client can ultimately reaches the actual partition site, assuming that the application can follow the redirection chain at least as fast as the partition moves.

A temporal affinity between the client connection and the home partition of its transactions is *desirable* from the performance point of view as each reconnection takes time. However, the affinity is not *required* and even in the worst case, when transactions in one connection uniformly access all partitions, the redirection protocol preserves *correctness*, although, with an impact on performance. We define correctness as the ability of the protocol to eventually route the client to the processing node of the transaction, provided that all the nodes remain operating.

However, *availability* of service can be affected if some nodes are shut down, for example, because of scale-in. Some of problems in this case are illustrated in Figure 6. Since the same physical node both provides routing information to clients and processes transactions, switching off one may create a gap in the redirection chain (Figure 6B). Thus, a client may be given a node address which is not responding. We did not address this problem during this thesis work. A simple but limited solution would be to impose a restriction on the scaling-in policy so that nodes are released in the reverse order of the migration chain. Obviously, this solution does not work if the migration chains are different for different partitions and there is no single last node to stop (see Figure 6C).

Another related issue that occurs even with no gaps in the redirection chain is that clients that do not access *any of the partitions* for too long so may encounter a node that is not operating and thus they can not get the correct address of the current partition home (Figure 6A).

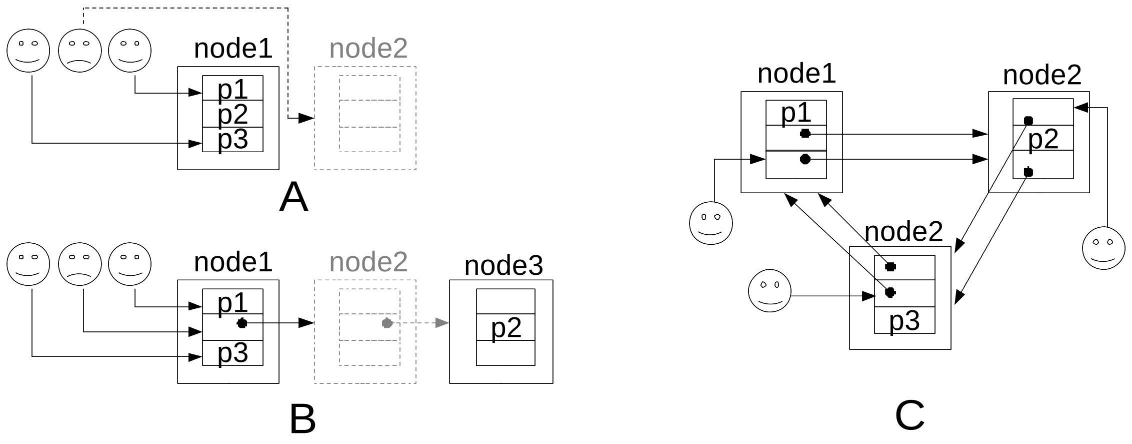


Figure 6. Reachability issues with scale-in scenario. A. A late client accesses the node that is shut down. B. Redirection chain is broken by shutting down an intermediary node. C. Shutting down any node will prevent some clients from reaching their partitions.

A more complete solution would be a distributed mapping of partitions to node addresses, shared between client and server nodes. Since migrations are relatively rare, for example, compared to transaction processing, updating the state of this distributed map would not add a significant delay to the rest of the migration process.

There may be other possible approaches to handle the routing clients' requests to the nodes responsible for the appropriate partitions. The detailed analysis and comparison of those approaches is outside the scope of this thesis.

3.2 Features Specific to the DBMS-level Mechanism

In this Section we describe the details of the scale-out mechanism implemented at the DBMS level. This mechanism takes advantage of the DBMS's awareness of the distribution of logically-defined partitions over the data blocks (pages), permitting complete transfer of a partitions between nodes for both scale-out and scale-in scenarios. The low-level block-oriented approach makes high throughput data transfers possible, leveraging the ability of disks to efficiently schedule sequential I/O operations in the presence of the random I/O due to normal load.

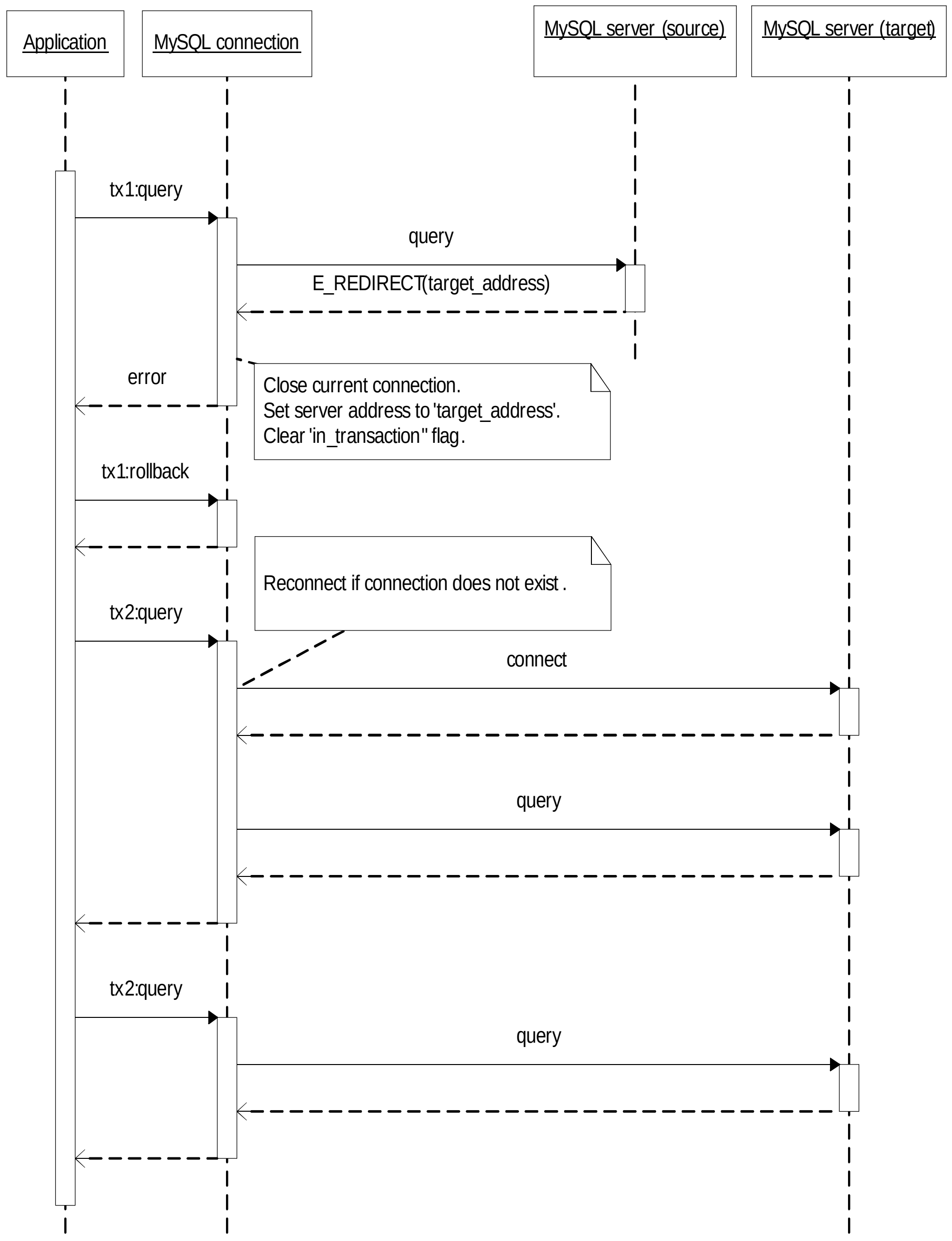

Application
MySQL connection
MySQL server (source)
MySQL server (target)
tx1:query
query
E_REDIRECT(target_address)
error
Close current connection.
Set server address to 'target_address'.
Clear 'in_transaction" flag.
tx1:rollback
Reconnect if connection does not exist .
tx2:query
connect
query
tx2:query
query

Figure 7. Redirection sequence diagram.

The DBMS-level mechanism involves both server-side and client-side modifications, as well as a slight change in the client-server protocol itself. Applications, however, need not be modified, as long as they follow reasonable error-handling practices. As expected, a DBMS-specific solution depends on particular internal DBMS features, thus limiting its portability. However, we believe this dependence is a good price for facilitating the partition placement information, thus enabling better performance and flexibility of the scale-out and scale-in operations.

3.2.1 Low-Level Partition I/O

One of the design goals of the DBMS-level scale-out mechanism was to achieve a data transfer rate that is close to the physical limits of hardware in the presence of concurrent load. We achieve this goal by using large blocks in the I/O operations for partition migration.

Given that the I/O pattern of OLTP workload is mostly random, large I/O sizes are particularly beneficial when the database resides on magnetic disks. Since the performance in that case is limited by disk seeks, injecting a small number of large I/O requests does not harm the normal load significantly. At the same time, large I/O requests for partition migration can utilize most of the disk bandwidth.

In the case of OLTP workload and magnetic disks, we believe the large block I/O is the main factor leading to high efficiency of DBMS-level scale-out.

3.2.2 Dirty Page Handling

At the time of partition migration many pages in the source system buffer pool are dirty. One approach would force the system to flush those pages to the disk so the files are up to date. Our DBMS-level mechanism avoids this additional I/O by taking dirty pages directly from the buffer pool.

This optimization is implemented in a straightforward way: as the chunks of data are read from the files, their range of page identifiers is looked up in the buffer pool to see if it includes any buffered pages and whether they are dirty. For any dirty pages found, the chunk is patched using the buffered pages and the resulting patched chunk is sent to the target system.

3.2.3 Preserving Buffer Pool State

In the important case of a database that exceeds the main memory capacity, the migration algorithm writes the data files to the target system's local disks. The conventional approach, in which data is copied by operating system commands and the DBMS is later started using the newly received data, causes a potentially long period of degraded performance due to the cold DBMS buffer pool. This problem can occur even if the DBMS uses the buffer pool of the operating system because the block access pattern in the DBMS is different from the one in the file copy operation.

By implementing the data transfer operation *inside* the DBMS we can avoid the problem of cold start (with an empty buffer pool) of the secondary system. The target DBMS inserts some of the received blocks into its buffer pool before writing them to the disk. The block access pattern within a partition does not likely change after migration as both the data and the client load are moved to the new location. This observation allows us to use the state of the buffer pool at the source DBMS to determine which data pages to keep in the target DBMS buffer pool.

The capacity of the target node's buffer pool may be less than the size of the partitions allocated to this node, which is normal for disk databases. In that case the system should decide which of the pages in the target buffer pool to replace by the pages of the newly migrated partitions. We call this problem the *buffer pool merging problem*. Solving this problem for the LRU replacement algorithm, used in InnoDB, is complicated by the fact that the state of the LRU policy only contains estimations of *relative* probabilities of eviction for pages. Therefore, the probabilities of pages from different LRU queues are incomparable, and producing the combined LRU queue, which equally well approximates the replacement preference for *all* pages, is not possible.

We took the following simple approach to merging the buffer pools. The main assumption is that the load on each partition is equal, so each partition needs the same amount of the buffer pool memory to maintain the same miss ratio. Although this assumption does not hold in the general case, it allows us to make an estimation of initial buffer pool allocation on the target node. We determine the number of pages of a partition to inject into the target system buffer pool n_{NEW} by a linear proportion:

$$n_{NEW} = n_{TOTAL} * \frac{1}{W_{CURRENT} + 1} \quad (1)$$

where n_{TOTAL} is the target system buffer pool capacity (pages), and $W_{CURRENT}$ is the number of partitions served by the target system before the migration.

During the migration procedure a snapshot of the *source* system buffer pool page identifiers is made. Those page identifiers are in the LRU order naturally so n_{NEW} page identifiers from the *hot* (most recently used) side, belonging to the migrating partition, are transferred to the target system before copying actual data pages. On the target system, when data pages are received, those with identifiers in the list are inserted to the *hot* side of the LRU list. If the buffer pool at the target has more available memory (the system is underloaded), the rest of the received pages are inserted to the *cold* side of the LRU list.

3.2.4 Consistency Issues

A single instance of the InnoDB storage engine handles all of the partitions on a server. The InnoDB storage engine maintains information which is shared between different parts of a database. This shared information includes both volatile main memory state and persistent data on the disk. Only the persistent part is kept between server restarts, however, we would like to perform partition migration while a server is online, thus the memory part is also important to consider. During normal operations, the shared state is modified along with partitions. Hence, if a partition is removed from a running system and later injected to a different one, the consistency between the shared state and the moved partition should be preserved on both systems. The consistency requirements are defined individually for the following data structures:

1. The transaction log and recovery state, including the current LSN counter.
2. Record versioning state, including UNDO segments in the system tablespace and transaction number counter.
3. Insertion buffer.

We will discuss the specific consistency requirements for each case below. Generally speaking, the migration procedure ensures the requirements by either partially limiting functionality, performing housekeeping activities before the migration, or modifying the data in the partition during the transfer.

3.2.5 Transaction Log and Recovery State

InnoDB uses the transaction log for REDO recovery after a system crash. Each modification of a page in the buffer pool causes a log record to be appended to the log. The position of a record is represented by a Log Sequence Number (LSN), which is a byte offset of the record position in a conceptually infinite log file. Each persisted page in a data file contains an LSN value which is a lower bound of the LSN of the log records that can potentially be applied to this page during recovery. Since different server instances write their own transaction logs independently, their LSN numbers are not comparable and the stored LSN value in a page would become invalid if that page were migrated.

Since active transactions are not allowed during migration and all dirty pages are included in the copy, the log records in the source transaction log related to the migrated partition can be ignored. Moreover, the migration procedure itself is synchronous, meaning the original copy of a partition is deactivated only after the target copy has been successfully written to persistent storage, therefore the data pages can potentially be recovered after a crash during migration.

In the case when the LSN numbers in the pages originate from a different system which has a separate sequence of LSN numbers and the corresponding log records, the following scenario would lead to failed recovery:

1. A page P is modified at server A and is written to disk, marked with LSN_A. This means that the log records with $LSN \geq LSN_A$ must be applied during recovery should server A crash.
2. P migrates to the server B, then is modified once more, marked with LSN_B. Let $LSN_B < LSN_A$. Upon writing modified P to disk, server B has to mark the page with its own LSN number, even if it is less than the previous one, to ensure that its log records with $LSN_B < LSN < LSN_A$ are applied on potential future recovery.
3. P migrates back to server A. Now its LSN number is less than it used to be. If A crashes, the recovery process will try to apply the log records to P which are too old and may be removed from the log during a checkpoint. Recovery cannot proceed as it thinks the log records are unavailable.

To solve the problem of invalid LSN numbers in the data pages, the target system patches all received data pages with the target's current LSN number. The target's current LSN number is read at the start of the migration process. Thus, after migration no preexisting log records can be applied at the target to the newly migrated data as their LSN is always less than the LSN of the data pages.

3.2.6 Record History State

InnoDB contains a multi-version implementation of transaction isolation, a variation of snapshot isolation. Allowing multiple versions of a record to exist in the database requires some form of garbage collection to remove the versions that are no longer relevant. InnoDB separates the most recent version of a record, which is kept in the table itself, from older versions, which accumulate in the UNDO segment. The UNDO segments are shared by all tables in the database and are additionally used for UNDO recovery in the case of a system crash or transaction rollbacks. InnoDB uses a global counter, stored persistently in the system tablespace, to assign transaction sequence numbers (timestamps) to maintain their total order. Every record in a data table is labelled by the sequence number of the transaction that created this record version, as well as a physical pointer to the previous version in the UNDO segment.

Migrating an InnoDB table between systems poses two potential issues. First, like LSN values, transaction numbers and pointers to UNDO records are not comparable across different systems and may cause confusion when tracking a record history in a partition that has been moved. Second, the garbage collection implementation requires the current version to be accessible, thus it cannot proceed if garbage from a partition remains after the partition has been moved away.

To handle these issues we rely on the absence of active transactions in the migrating partition during the migration. This means there are no transactions which may look for old versions of records in the partition. Therefore, the old versions in the UNDO segment may be safely removed. At the same time, the garbage collection procedure in InnoDB writes to the current version to remove its pointer to the history in the UNDO segment. If the partition containing the current version is moved away by migration, garbage collection process will crash. While it may be possible to modify the garbage collection algorithm to remove record history without accessing the current record, we chose a simpler approach which repeats garbage collection at the source system until there are no more versions in the UNDO segment whose transaction numbers are below the current transaction number value during the start of the migration. This forced garbage collection may significantly delay the migration start if there are long running transactions even in non-affected partitions because the current garbage collection algorithm has to purge old versions for all partitions. A long-running transaction prevents old versions in the UNDO segment from being collected, effectively stopping the collection. However, this is generally not an issue for short transactions in the OLTP workload.

Another issue may happen at the target system after migration if records keep the version numbers and old version pointers which were valid in the source system. If the target system decides to follow a migrated record's version history pointer, which is not meaningful at the target system, it will probably crash. To prevent this from happening, all records in the partition are scanned and their transaction numbers are replaced with a small number, such as one. This approach to move data consistently between MySQL servers is the same as described in [27] .

3.2.7 Insertion Buffer

InnoDB uses the Insertion Buffer as a technique to speed up random insertions to an index which is too big to fit in the buffer pool. In the normal case each insertion would lead with high probability to a buffer pool miss. InnoDB tries to avoid these misses by inserting the record in the special data structure, called Insertion Buffer. It is organized as a persistent index, located in the system tablespace, with a record page number as a key. If during the insertion to an index its leaf node is not in the buffer pool, the inserted record is instead put into the Insertion Buffer. Delayed insertions are merged into the main index either later by a background process or when the page is eventually loaded into the buffer pool due to a read miss.

When the Insertion Buffer contains records from a partition, migrating the partition to another system will cause missing records in the partition index(es). Since the Insertion Buffer is an optimization technique and is not required for correct operations, and to keep our prototype simple, we decided not to have a forced merging of Insert Buffer records to the partition before its migration. Instead, we disabled the Insertion Buffer functionality.

Chapter 4
Experimental Methodology

4.1 Overview of the Experiments

In the previous chapter we described the design of three scalability mechanisms. Now we would like to investigate their efficiency. For the purpose of this thesis we define efficiency as the ability to perform system reconfiguration (scale-out or scale-in) quickly while causing little disruption to the client workload. Therefore, we would like to answer the following questions during the experiments:

- How long does it take for the mechanism to complete the reconfiguration process?
- What is the cost of the reconfiguration for each mechanism?
- What is the effect of the reconfiguration on the clients' experience? How deep, and how long, is the disruption and how is it distributed among the clients?
- What (if any) practical issues may arise when using the scale-out mechanisms?

When addressing the performance-related questions, we pay special attention to presenting independent views of the system performance from both the provider's and client's perspectives. The provider and the clients may have conflicting optimization goals. For example, in order to maximize overall throughput, the provider may choose not to execute requests from certain clients. However, such a situation may be unacceptable from the client's point of view.

We construct the client such that its request generation rate is independent of the ability of server to execute the requests. When the server fails to fully handle the offered load, we evaluate the degree of this failure by analyzing the difference between the offered load and the actual load.

In most of the experiments we study one scale-out operation performed by different mechanisms, moving the database from one to two nodes and migrating half of the partitions to a newly started node. Before the scale-out operation we allow the system to reach steady state. We gather performance information before, during, and after scale-out to estimate its effects on the system. The workflow of the experiments is coded in the controlling script, which allocates certain fixed time intervals for each phase of the experiment.

To illustrate a more general case of load balancing, we also included a scale-in experiment. Unfortunately, the Snowflock-based mechanism does not provide the ability for migrating partitions to already running nodes, and the application-level one is not efficient so we could not perform a comparison of different approaches in this case.

Migrating database state in the *shared-nothing* architecture works over whole partitions and, in most cases, involves moving data located on the persistent storage (disk in our case). We would like to investigate how the involvement of the disk affects scale-out efficiency. Thus, we perform the experiments in two distinct scenarios: disk-bound and processor-bound. In the former scenario, the database working set size is larger than the size of the database server buffer pool and the workload execution causes page misses in the buffer pool. Additionally, intensive update activity requires the server to flush dirty pages, consuming a significant portion of disk bandwidth. As a result, the overall performance is limited by the I/O system capacity.

In the CPU-bound scenario the server is configured to store the database entirely on a RAM-disk. The database has to be small enough to fit in memory. The performance of the system, including the scale-out itself, now does not depend on disk I/O, as accessing database pages only involves memory-copying operations. Although using a RAM-disk seems to be less natural than merely configuring a buffer pool large enough to contain the whole database, our method eliminates physical I/O not only for reads but also for dirty page flushing, which would still happen otherwise. Since the TPC-C workload is update-intensive, experiments involving large buffer pools would still be I/O bound in that case.

4.2 Workload

In this thesis we focused on online transaction processing (OLTP) workloads. We believe it is more difficult to achieve good scalability for OLTP workloads as compared to analytical ones as the unit of work in OLTP is smaller and the amount of interaction between system components is potentially higher.

We chose a TPC-C-like workload for our experiments. TPC-C [35] is an established benchmark for evaluating RDBMS OLTP performance. TPC-C simulates a warehouse transaction processing system with medium-length transactions. TPC-C stresses various components of a system, including I/O and buffer pool, locking and concurrency control algorithms. Despite being old and relatively simple, TPC-C is still a challenge for database system implementers, particularly due to high I/O demand, and is widely used as a standard OLTP workload.

A TPC-C workload is defined as a mix of 5 different types of transactions: *New Order*, *Payment*, *Delivery*, *Stock Level*, and *Order Status*. Each transaction type occurs a certain percentage of the time in the mix. Throughput is expressed as the number of New Order transactions executed per minute.

We used the open-source OSDL Database Benchmark 2 (DBT2) implementation [36] which closely models the TPC-C specification. It should be noted that the TPC Council requires a formal certification of any results that claim to represent any of the Council's benchmarks. Therefore, any metrics obtained outside of the certification procedures should not be associated with the TPC-C benchmark and cannot be compared to other results.

Our experimental workload differed from the TPC-C workload in the following respects:

- Transactions that can access remote warehouses are disallowed.

- We use fixed-size databases.
- The method to control the transaction arrival rate differs significantly.

Those differences are described in detail in the following sections.

4.2.1 Strictly Partitioned Workload

Each TPC-C warehouse is a separate partition in our database. Although the TPC-C workload can be partitioned by warehouses for most of the transactions, the New Order and Payment transactions occasionally do access more than one warehouse. Since multi-partition transactions cannot be executed by the modified MySQL PE, we changed the DBT2 input data generator to use only one warehouse in all transactions.

Since all SQL queries in TPC-C include a warehouse number in their search condition, MySQL partition pruning algorithm, described in Section 3.1.3, has no difficulty correctly associating transactions with their warehouses.

4.2.2 Database Sizing

We used a fixed database size, which was primarily determined by the constraints of our hardware configuration. We tried to find the maximum size that permitted us to run the experiments reliably in all configurations (disk-bound and CPU-bound). A serious limitation was imposed by the Snowflock snapshot functionality, which requires extra space reservations to store memory and disk snapshots, effectively making available only half of the memory.

Table 2 summarizes the space parameters for the initial state of the database. In the disk-bound experiments only half of the warehouses were actually accessed by the workload; we show the numbers for the full database in parentheses. The reason for this was that our initial intention was to increase the working set size twice at scale-out time so the ratio between the total buffer pools size and the working set size did not change as a result of scale-out. That scenario would correspond to the real-world case in which new users cause extra processing and use additional data. However, we experienced difficulty running those experiments as the scale-out under those setting was too slow, so we do not include them in this thesis.

4.2.3 Controlling the Load

We conducted experiments with two types of load: flooding the server with transactions and limiting the load so that there is spare capacity to accommodate the overhead of scale-out. We refer to the former type as *maximum load* and the latter one as *controlled load*. We consider the maximum-load experiments as supplementary with their main purpose being to determine the maximum possible system throughput. In the controlled-load experiments we provide more results and analysis and believe it is more realistic because in real applications there is always some degree of overprovisioning.

In the controlled-load case, we used a significantly modified algorithm to control the rate of transaction generation at the client side. The main difference between it and the one from the

TPC-C specification (and the original implementation in DBT2) is independence from the server performance. In the TPC-C benchmark the transaction cycle executed by the client contains keying and think times added to the response time of the actual transaction execution. In this approach, a slow server would not only cause the reported response times to rise, but also increases the total length of the cycle, thus decreasing the rate at which the client issues transactions.

In contrast, we used a open-loop approach, in which the response times do not affect the transaction generation rate. This has several advantages. First, when evaluating the quality of service as perceived by clients, it is more natural to assume the clients have their own idea about the desired work intensity. Knowing the transaction start times in advance makes it easier to assess a possible disruption of service by comparing them to the actual transaction execution. Second, our approach allows us to set a desired performance level in a straightforward manner in the experiments in which the server is not fully loaded.

The workload generator uses multiple threads, which issue transactions independently from each other. Each thread emulates a TPC-C terminal. Each terminal is attached to one warehouse for the duration of the experiment and maintains a separate database connection. Each warehouse has a possibly variable number of terminals *TPW(t)*, which was 4 for CPU-bound experiments and 2 for the disk-bound ones at the experiment start. These numbers were chosen to be different from the TPC-C default value (10) in order to avoid having too many threads in the server and limit possible contention, as MySQL uses thread-per-connection model.

Each terminal continuously executes transaction cycles consisting of the following phases.

- Choosing the transaction type, calculating start time, generating transaction parameters.
- Deciding whether the transaction is too late to start. If it is not, waiting until the execution is due.
- Sending the transaction to the server.
- Waiting for the server's response.
- Receiving the response and logging the transaction result.

We designed the workload generation algorithm as follows. Let $TPM_{REQ}(t)$ be the target total offered transaction load, measured as a number of New Order transactions per minute per warehouse. For the experiments with variable offered load, the change in the load level is done by changing the number of terminals, $TPW(t)$, dynamically starting and stopping terminal threads in the workload generator. The load generated by each terminal is fixed for the whole experiment and is determined by:

$$TPM_{REQ}^{TERM} = \frac{TPM_{REQ}(0)}{TPW(0)} \quad (2)$$

Here, the load is assumed to be distributed equally between the terminals. The initial values for $TPM_{REQ}(0)$ and $TPW(0)$ are passed to the workload generator as parameters.

Conceptually, each terminal generates a separate schedule of start times for each transaction type. The full schedule for a terminal is the superposition of the per-type schedules. Since the scheduled transaction start times do not depend on the server responses, generating and storing the whole schedule in advance is not necessary. Instead, each terminal keeps a vector of absolute start times, one value for each type of transaction, and chooses the type with the smallest value for execution. After the execution of a transaction, the next absolute time is calculated for the same transaction type. In this calculation, only absolute times (since the start of the experiment) are used, therefore, the schedule does not depend on server response times nor does it drift due to the overhead of client-side computations and transaction parameter generation. The actual clock time is used to control the delay before the next transaction issue and to determine whether the transaction is late.

Since the full schedule is a superposition of the schedules for each transaction type, the individual schedules are computed first for each transaction type $i \in \{NewOrder, Payment, Delivery, StockLevel, OrderStatus\}$. Mean transaction cycle duration $\bar{T}_i$, in seconds, is computed using the requested throughput, TPM_{REQ}^{TERM}, and the transaction mix ratios using:

$$\bar{T}_i = \frac{60}{TPM_{REQ}^{TERM}} * \frac{N_{NewOrder}}{N_i} \tag{3}$$

where N_i is the ratio of *i*-type transaction in the mix. N_i is taken from the TPC-C specification [35] and its values are shown in Table 1.

Transaction type, *i*	**Relative ratio in the mix,** N_i
New Order	0.45
Payment	0.43
Delivery	0.04
Stock Level	0.04
Order Status	0.04

Table 1. Mix ratios for transaction types

Actual cycle durations are chosen randomly with negative exponential distribution limited by ten means.

$$T_i = \bar{T}_i * min(-\ln r, 10) \tag{4}$$

where *r* is a uniformly distributed random variable taken from the range (0, 1).

	CPU-bound	**Disk-bound**
W, Warehouses	8	30 (60)
Total data, GB	1.1	~4.5 (8.4)
Partitioned data, GB	1	~3.9 (7.8)
Non-partitioned data, InnoDB system tablespace, metadata, GB	0.1	0.58
InnoDB transaction logs, GB	2 x 0.75	2 x 0.5
InnoDB buffer pool size, GB	1.5	1

Table 2. Database size parameters (total values for the loaded database in parentheses)

The time lines of the standard TPC-C transaction generation cycles and our transaction generation cycles are presented in the Figure 8. In contrast to TPC-C, with its separate keying and think times, there is one client-imposed delay between transactions used to enforce the total cycle duration.

At this point each client terminal has a schedule of start times for every transaction type. We allow a transaction to start slightly later than its scheduled start time if the previous transaction has not completed before the next one is supposed to start. The reasons for allowing delayed starts are explained in the next paragraph. If the transaction start time is delayed too long, the transaction is skipped and reported as *cancelled*. Once a transaction is submitted to the server, it is allowed to complete regardless of its actual execution duration. Each terminal uses a single connection to server and since the connection only supports synchronous transaction execution, there is at most one transaction in progress for each terminal.

As random start times are calculated for different transaction types independently, it is possible that start times of different transaction types fall into a time interval small enough that the resulting transaction executions would overlap even under normal load. Additionally, the exponential distribution may produce two consecutive start times of the same transaction type within a short interval. In other words, the workload generation algorithm may produce overlapping executions even when the server response time is normal. Those overlaps are random and their probability decreases quickly for longer delays. In contrast, the server inability to handle the requested load would lead to systematic violations and accumulation of the delay. Therefore, in order to distinguish workload-generation-imposed from server-imposed delays, we allow transaction start times to be delayed no longer than some small time interval (tolerance) $T_{TIMEOUT}$. In our experiments, $T_{TIMEOUT}$ was chosen as 1 second for CPU-bound and 5 seconds for disk-bound workloads, which is sufficiently small in comparison to the duration of the experiments so it does not lead to significant accumulation of late transactions when the server cannot handle them. At the same time, the chosen tolerance values are big enough so that we did not observe cancelled transactions when the server is under normal load. Possible timing scenarios in workload generation are illustrated in Figure 9.

4.3 Metrics

In our experiments we model a service provider environment. In this kind of environment the provider and its clients have their own views and objectives regarding the system performance. The provider's goal is to maximize profit, which we will assume is related to the amount of work actually done for the clients minus the associated costs. The clients, in turn, wish to receive the promised services with good quality. The exact definition of service quality varies; however, it typically includes the amount of the service actually received, service availability and service responsiveness. An important aspect of the client-centred measurement stems from the clients' independence, meaning that each client has its own valuable experience, which will not be captured by simple aggregating metrics computed over multiple clients.

To represent the interests of both the producer and the clients, we use two groups of metrics. The first group corresponds to the provider's view and includes system throughput and the cost of operations. Assuming that in a cloud environment services are paid per use, the provider should maximize amount of work done using the smallest number of servers. Also, assuming that the provider does not have limits in scalability and can adjust the number of servers quickly in response to a change in clients' demand, the load level of servers determines the provider's economic efficiency. For a given workload, the load level can be expressed in terms of throughput per server. In the extreme case, targeting solely the provider goal, the provider would load each server to its saturation. However, increasing the load level of servers would lead to deterioration of the client experience due to fluctuations in the clients' demand and the lack of spare server capacity to accommodate those fluctuations. That is why another group of metrics is needed.

For the second group of measurements, we focus on service interruption by calculating the amount of time during which the service is not available. We also measure throughput separately for different groups of clients. In particular, we differentiate between the clients that access migrating partitions and the clients that access partitions that remain on the original host.

4.3.1 Cost and Throughput Metrics

The main provider-centered metric is the total throughput $TPM(t)$, measured in transactions per minute, normalized per warehouse, defined in each small time interval t during the experiment:

$$TPM(t) = \frac{\sum_{w=1}^{W} N_{NewOrder}(w, t+\Delta t) - \sum_{w=1}^{W} N_{NewOrder}(w, t)}{W * \Delta t} \qquad (5)$$

where

> $N_{NewOrder}(w,t)$ is the number of New Order transactions executed in warehouse *w* from the start of the experiment until the time *t*;
>
> Δt is the length of the interval, for which the throughput is reported. Δt is assumed to be a constant for an experiment.

We define the cost of scale-out as the amount of work that was not performed by a server ("lost work") because of the performance effects of the scale-out procedure. The server's inability to process the offered load is manifested as cancelled transactions due to scheduling timeouts. Since the client tries to maintain the load independently of the server performance, the number of cancelled transactions directly corresponds to the amount of "lost work".

We compute the number of cancelled transaction $N_{CANCELLED}$ as a difference between the offered load and the actual execution rate, assuming that the offered load does not change during the interval $t_{start} \leq t < t_{end}$:

$$N_{CANCELLED} = TPM_{REQ}(t_{start}) * (t_{end} - t_{start}) * W - \sum_{w=1}^{W} (N_{NewOrder}(w, t_{end}) - N_{NewOrder}(w, t_{start})) \quad (6)$$

where

> W is the number of warehouses;
>
> $TPM_{REQ}(t)$ is the offered load from each client (warehouse), in New Order transactions per minute;
>
> $N_{NewOrder}(w,t)$ is the actual count of executed transactions since the start of the experiment for the warehouse *w* until the time *t*;
>
> t_{end}, t_{start} are the start and end of the time interval for which the number of cancelled transaction is calculated.

We convert the number of cancelled transactions in all clients, $N_{CANCELLED}$, to the time-domain metric T_{LOST}, measured in minutes, by dividing this number by the normal offered load.

$$T_{LOST} = \frac{N_{CANCELLED}}{TPM_{REQ}(t_{start}) * (t_{end} - t_{start}) * W} \quad (7)$$

Intuitively, T_{LOST} measures the number of minutes of server time required to compensate for the loss of productivity due to the impact of scale-out, assuming the server load is maintained at the $TPM_{REQ}(t_{start})$ level.

4.4 Experimental Environment

The experiments are conducted on the *Muscat* cluster [37] at the University of Waterloo. We use two identical cluster nodes, each of which is an IBM BladeCenter LS-21 [38] blade

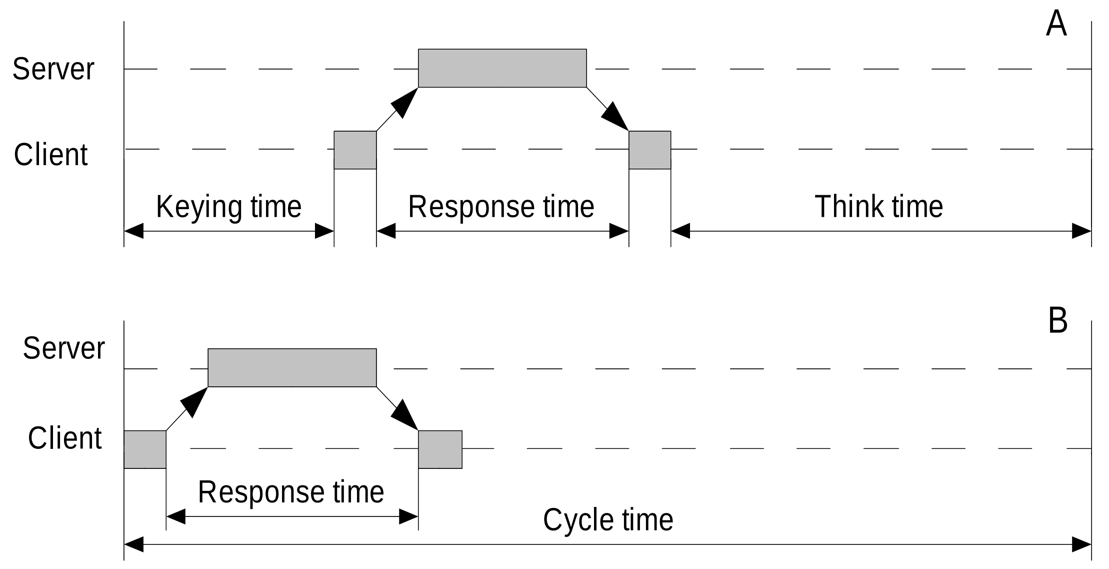


Figure 8. Time lines of the transaction cycle: TPC-C (A) and our modified TPC-C (B). The shaded blocks correspond to processing time in the server and client.

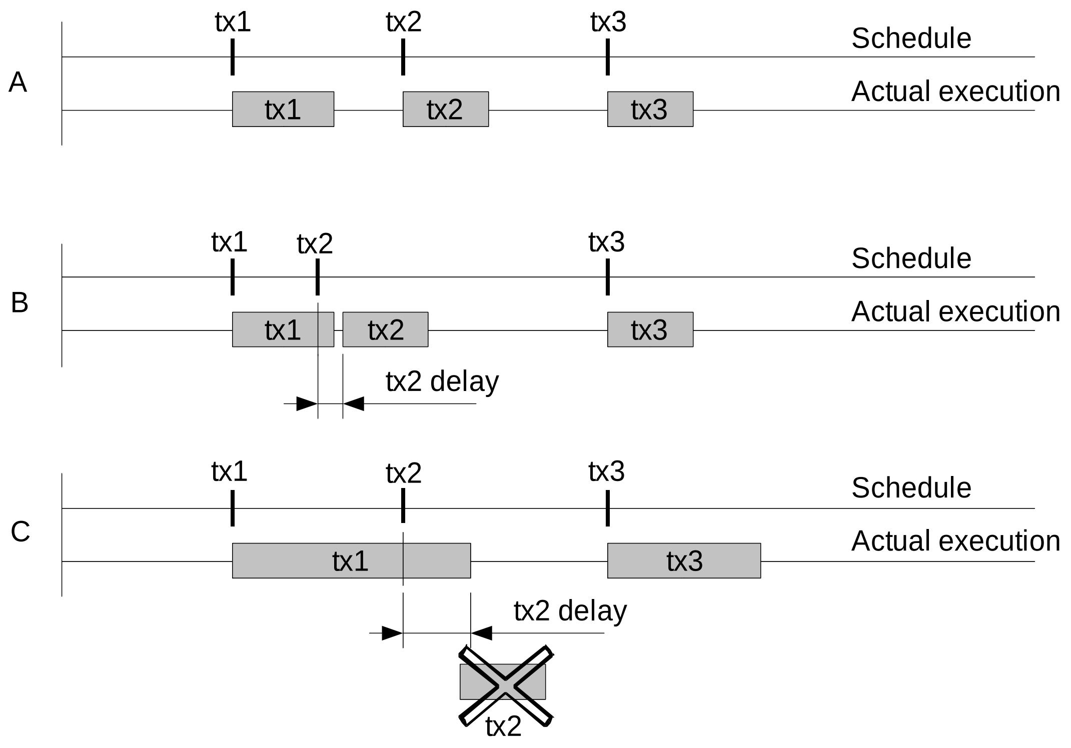


Figure 9. Transaction rate regulation. Normal execution (A). Small delay due to scheduled conflict, no cancellation (B). Big delay due to previous transaction (tx1) being too slow, transaction tx2 is cancelled (C).

server equipped with two dual-core AMD 2212 HE CPUs, clocked at 2.0 GHz, 10 GB of RAM, and a locally-attached Seagate Savvio ST936701 10K rpm 36 GB SCSI disk [39]. The connectivity includes two 1 Gbps network adaptors, only one of which was actually used.

The blades are booted into Snowflock Xen VMM version 3.4 with Debian Linux 5.04 installed in both dom0 and domU in the virtual machines. The Linux operating system used a Snowflock-specific Xen-enabled kernel based on version 2.6.18. Each VM is allocated with 1 virtual CPU and 4 GB of memory. In the disk-bound configurations the database is located in the same virtual disk as the rest of the filesystem.

The Snowflock resource allocator chooses the location of a newly cloned VM copy based on the available resources of the cluster physical nodes. Our goal was to study scale-out between physical nodes, therefore, we configured the Snowflock available resource limits on each node to match closely the specification of the VM. Thus, a new cloned copy would always be placed on the second physical node of the cluster.

The load generator program, DBT2 driver, was located at the same node on which the original VM was running. Since the VM only was allowed to use a single core, while there are four total cores in the physical server, there was no CPU contention between the benchmark client and the database server.

DBT2 driver was configured to use native MySQL connectivity, implementing each transaction as server-side stored procedure. By doing this we minimized the overhead of the client processing and the communication. The number of terminals attached to each warehouse *during normal load* was 2 for the disk-bound and 4 for the CPU-bound setups. Combined with the number of warehouses, this totals in 60 and 32 simultaneous connections to the MySQL server. Given the *thread-per-connection* model of MySQL, the degree of parallelism inside the server was equal to these numbers.

To keep the experiments short, we used prepopulated databases. Before each experiment, a file image of the database directory was copied to the MySQL database directories in the main VM. For each one of two database sizes we generated the database in two steps, saving each step's results. First, we created the schema and loaded the non-partitioned table (Item). The result of this step was used to initialize the MySQL instance at the new node for the DBMS-level and application-level mechanisms. Next, we populated all partitioned tables and kept the resulting image for the main node initialization.

Several system metrics were collected during the experiments. In particular, we used standard Linux monitoring tools to gather CPU usage, context switch counts, and I/O statistics for the virtual nodes and the hosts, as well as the MySQL processes (pidstat). The network utilization was collected with the sar utility on all VMs and the hosts. Additionally, the Xen-related metrics were logged with the xentop utility on the hosts. The sampling interval for all statistics was 1 second, however, the data were aggregated into larger intervals for graph plotting.

Chapter 5
Experiments and Results

5.1 Overview

In this section we present the experiments and their results. We examined the scale-out mechanisms in two use cases which differ by the primary bottleneck involved: disk-bound and CPU-bound. The former case represents a realistic scenario of a shared-nothing database application, while the latter case is included in order to show how the presence of disk I/O bottleneck affects the scale-out process for the mechanisms in question. As was noted in the Section 4.2.3, we ran the experiments with two types of load generation: maximum load and controlled load.

All experiments were conducted on two physical servers. The scale-out operation in all experiments expanded the system from one to two nodes. The timeline of most experiments consisted of the following steps:

1. Starting a server on the initial node, containing all the partitions.
2. A period with the initial number of client terminals. During this period, the system warms up and reaches steady state.
3. The number of client terminals is doubled. For the maximum load experiments, no change in the throughput metric, compared to the previous period, would show the server was actually fully loaded. For the controlled load experiments, the offered load is similarly doubled, and the initial configuration becomes overloaded.
4. Scale-out from one to two nodes is initiated.
5. A period for scale-out operation to proceed. Eventually the system reaches another steady state, now with two nodes.

The disk-bound experiments with the DBMS-level mechanism also included the following additional steps, demonstrating the scale-in process:

6. The number of terminals dropped twice to return to the initial number. This decreases the load twice and the system becomes lightly loaded.
7. Scale-in operation initiated to move the system from two nodes to one node.
8. A period for the scale-in operation to proceed so that the system reaches steady state with one node. The system and load configuration is now the same as it was initially.

The timeline of the experiments is illustrated in Figure 10. The lengths of intervals T_i, $1 \leq i \leq 5$ are determined by control script parameters and vary between disk-bound and CPU-bound experiments. The variables t_i, $1 \leq i \leq 5$ indicate the start times of the corresponding periods.

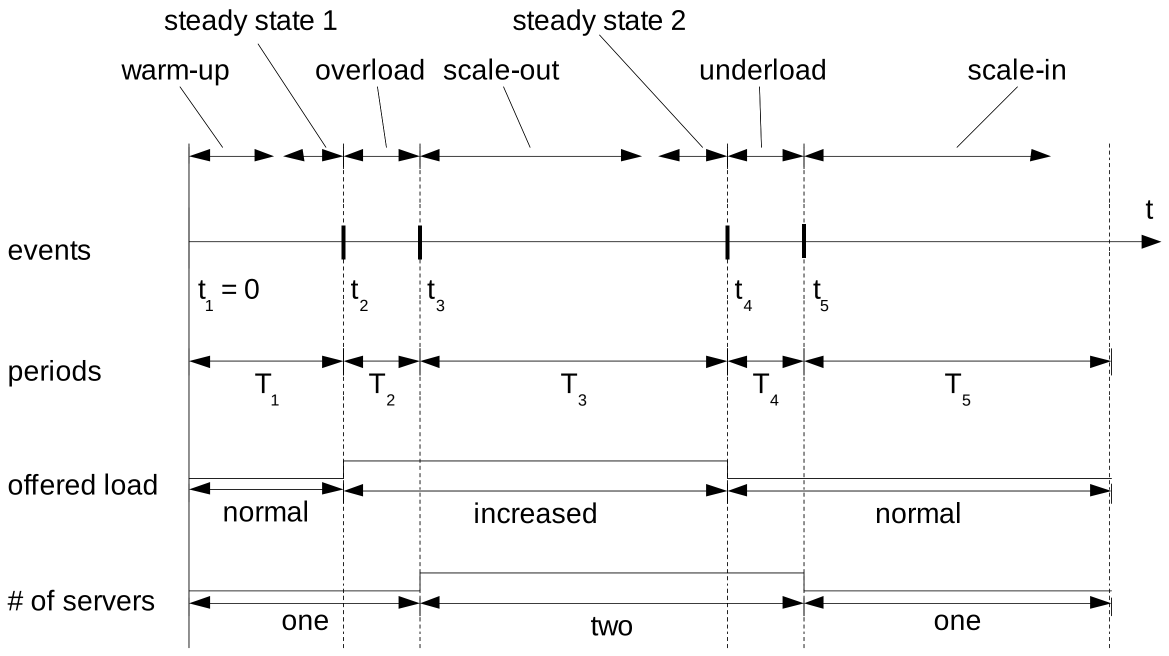


Figure 10. Timeline of the experiments.

According to out experimental methodology, we present a provider's and client's views on system behavior during scale-out. Those views are presented in the *Throughput Analysis* and *Unavailability Analysis* subsections, respectively.

5.2 Disk-Bound Experiments

In this series of experiments we model a realistic scale-out scenario in which the total size of the data partitions is larger than the available memory. We used the TPC-C dataset consisting of W = 30 warehouses, which takes about 4.5GB of storage space. The database was placed in the root filesystem of the VM image file. The MySQL buffer pool was configured to 1GB so that about 1/4 of all data fits there. The MySQL instance was also configured to use Linux Direct IO so data is not cached by the operating system file cache. In this settings we expect significant random disk read and write activity on behalf of the MySQL server that would affect the duration of partition migration and service disruption.

The timing parameters of the controlling script were set according to Table 3. During the overload period we increase the number of terminal threads two-fold, thus doubling the rate of transaction requests. However, the new terminals are connected to the same set of warehouses as the existing ones. Thus, we keep the size of the working set the same.

In addition to the scale-out we included in the experiment the scale-in process, in which the system migrates the partitions from the second node back to the original one. In a real-life case the scale-in mechanism would be triggered by a control module in response to the insufficient load. The scale-in is not possible under the Snowlock-based mechanism as there is no way to merge the states of VMs which will have diverged since the cloning.

Period	**Duration, minutes**	**Start time, minutes**	**Comment**
Initial warm-up period with one node	$T_1 = 30$	$t_1 = 0$	
Overload	$T_2 = 15$	$t_2 = 30$	
Scale-out to two nodes	$T_3 = 60$	$t_3 = 45$	
Reduced load	$T_4 = 15$	$t_4 = 105$	
Scale-in to one node	$T_5 = 60$	$t_5 = 120$	Only for DBMS-level mechanism

Table 3. Disk-bound experiment time parameters

The application-level mechanism performed very poorly in this experiment, taking more than one hour to completely migrate the half of the database in one direction. Therefore we decided it is impractical to include it in the comparison. Instead we focused on the Snowflock-based and DBMS-level mechanisms.

During the first runs we observed severe performance degradation of Snowflock copy-on-write disks after a new snapshot is created. In Snowflock, a new snapshot is created upon starting the virtual cluster and each time a new VM clone is created. Although taking a snapshot at the cluster start is not technically required, it is useful for conducting multiple tests with the same VM image, keeping the changes separately and discarding them after the run. In Snowflock, snapshots are implemented by the Copy On Write (CoW) technique, in which write block requests are redirected to a separate file, called a CoW slice, while keeping the original VM disk image intact. Immediately after a new snapshot is created, the virtual disk could sustain only a few operations per second, with a throughput of about few hundred kilobytes per second, even when the CoW slice and the original VM disk image were on the same machine. This poor performance, which is two orders of magnitude worse than normal, can be easily noticed, for example, when copying files to a VM. As I/O activity continues, the performance improves and eventually it stabilizes at the normal level.

After a little experimentation we found that the CoW disks were slow when the CoW slices was stored as Linux sparse files and had normal performance when stored in preallocated files. The exact mechanics of this slowdown are not clear, but we suspect this is due to the high overhead of dynamically allocating new blocks in a sparse file in the Linux ext2fs filesystem, causing many synchronous physical disk writes for each logical write.

Despite the fact that there certainly must exist some I/O performance cost of updating a CoW disk, we believe the observed degree of the degradation in the current Snowflock implementation is excessive and caused by inefficient implementation rather than the CoW concept. Therefore, in effort to obtain more realistic data, we modified Snowflock scripts to avoid this overhead. Originally, Snowflock created a temporary directory with a random name for each cluster run and placed the CoW slices into this directory. With our modifications it uses fixed configured paths to the locations of CoW slices. Before the experiments we created the files in those locations with the size equal to the original image size. This way, Snowflock picks up existing files to use as CoW slices and no random block allocation takes place.

Conceptually, these Snowflock modifications defeat the main goal of Snowflock, cheap and fast VM cloning, as it becomes impossible to instantly allocate a large file. Moreover, the space consumption becomes very high. In our opinion, both strategies of dealing with CoW file allocation represent the extreme cases, therefore, we included both in our experiment settings. The efficiency of the original (too slow) and the modified (too fast) configurations corresponds to the upper and lower bounds, respectively, of the duration of scale-out and the degree of service disruption.

The maximum throughput of the server was determined by first running the experiment in the maximum load setting. The achieved throughput was approximately 60 tpmC per warehouse at 30 warehouses (corresponding to 1800 tpmC total for the server). Based on this number, we chose 50 tpmC as a client-offered load level for our controlled load experiment, which is about 85 percent of the maximum.

Cost and Throughput Analysis

The graphs for total transaction throughput and network utilization are shown in Figure 11 and Figure 12 respectively. The system first achieves steady state approximately at t=15 minutes. During the excessive load period $30 \leq t < 45$ when the number of clients and the offered load both doubled, the system becomes limited by server capability, therefore it can accommodate only small performance increase. At the time t=45 minutes scale-out is initiated.

The original Snowflock implementation experiences a deep throughput drop and restores very slowly, taking about 25 minutes until the original level and about 40 minutes until the maximum level is reached. However, it still had not finally reached the offered load level, staying about 10% short of it. Later, from Figure 16 and Error: Reference source not found we will see that the original node performance is the culprit while the migrating one is restoring smoothly, albeit slowly, and finally reaches the offered level of 100 tpmC per warehouse. Thus we can conclude that in the case of the original Snowflock implementation the latency of remote block reads is not the main cause of the performance impact but the degradation of physical disk I/O performance at the original node due to CoW block allocations is. When the factor of slow CoW block allocation is removed from the equation with preallocated CoW slices, the system scales out much quicker even though the latency of remote reads still affects it.

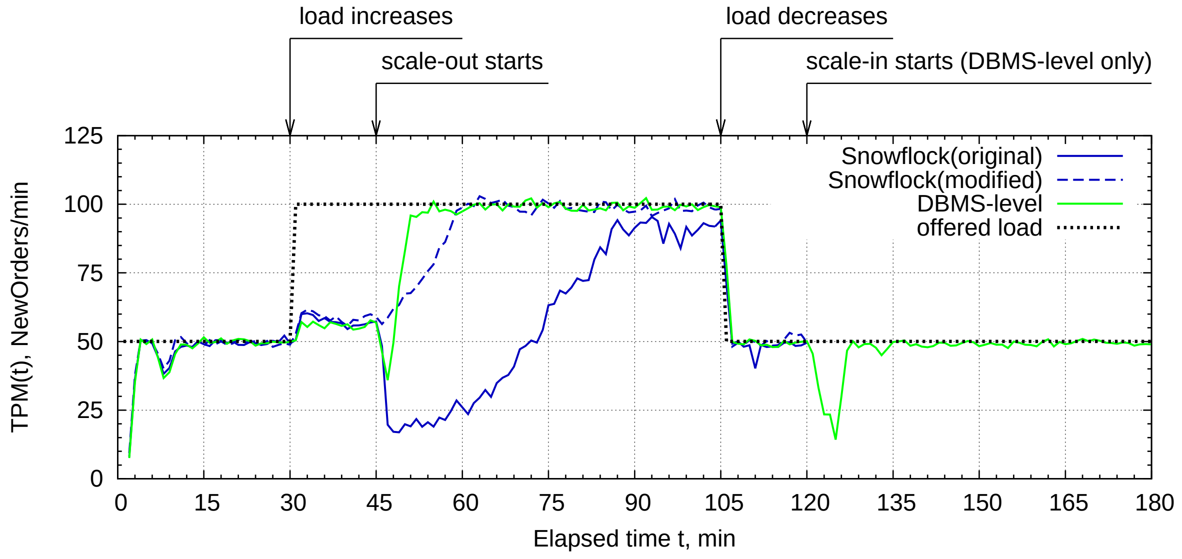


Figure 11. Average transaction throughput per warehouse.

The modified Snowflock mechanism takes about 12 minutes to fully scale out and, interestingly, shows no overall degradation during the transition. There is degradation for the partitions that are migrating, but the almost instantaneous acceleration of the non-migrating ones compensates this.

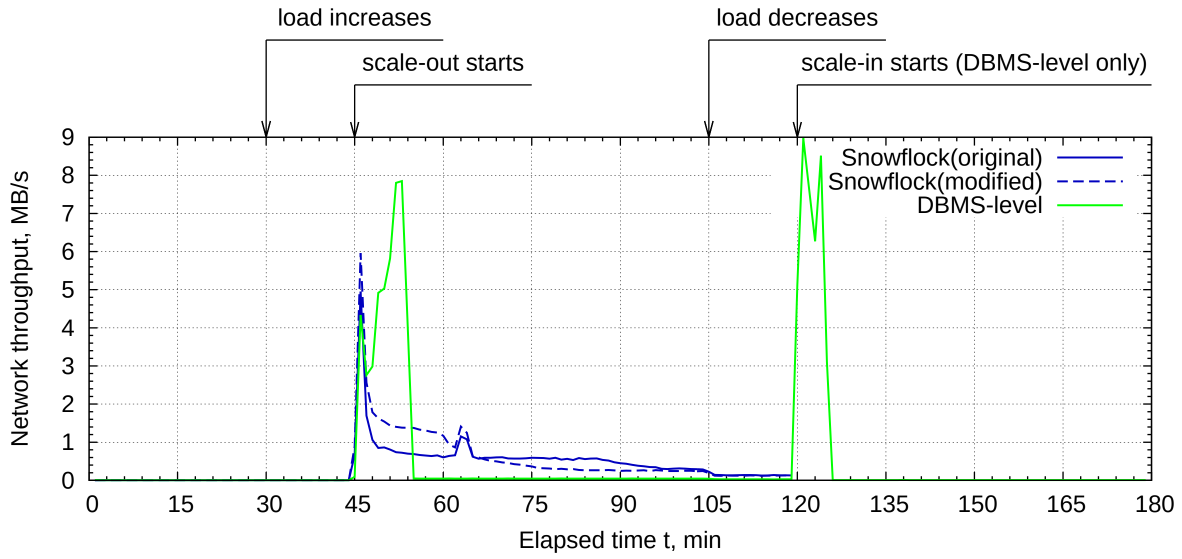


Figure 12. Network throughput between physical nodes

The DBMS-level implementation is the quickest to scale fully, taking approximately 6 minutes, but has a short degradation initially that has its minimum at about t = (45 + 2) minutes. To investigate the reasons of this degradation we examined the logs and analyzed the durations of the components of the scale-out process. The detailed development of throughput and network traffic is shown in Figure 13. The logs showed that the VM cloning took 29 seconds, reinitializing the database with new data files took 92 seconds, and starting a new MySQL instance took 28 seconds. During database reinitialization, deleting old files at the target took most time, followed by copying new files. Cloning, file deletion, and file copying steps affected the source node performance as its physical node was also used as a data source for database files and the VM image. Both steps combined took almost exactly 2 minutes. Then a new MySQL process started initializing locally, easing the load on the original node and allowing it to recover partially. The partition migration itself started soon after that and was accompanied by fast performance recovery. From this timing information we can conclude that the cost of bringing up a new node is the cause of the temporary performance degradation. The whole DBMS-level scale-out took about 8.5 minutes, which

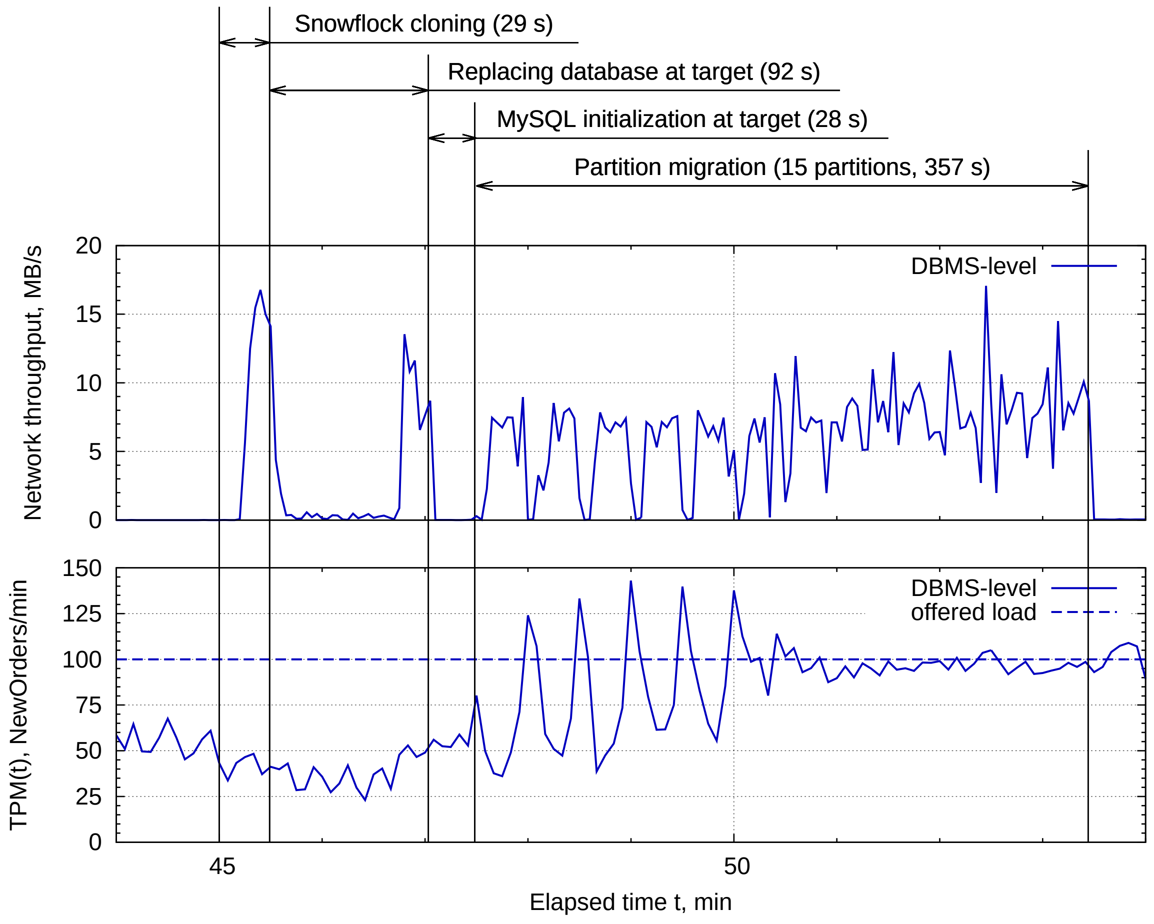


Figure 13. Detailed transaction and network throughput during DBMS-level scale-out.

consisted of about 2.5 minutes to initialize a new MySQL instance and 6 minutes to move the partitions, i.e. 24 seconds per partition on average. Interestingly, the full performance recovery was accomplished when only half of the migrating partitions had been actually moved and enough capacity had been freed at the source node.

The results of network traffic measurements are presented in Table 4. Although Snowflock-based scale-out takes much more time than DBMS-level one, the amount of traffic (to the point when system throughput has been mostly recovered) by Snowflock-based mechanism is less. We can conclude that on-demand transfer approach is efficient in minimizing the amount of transferred data. Therefore, slowness of the Snowflock-based mechanism is determined by transfer latencies and not by bandwidth limitations.

Mechanism	**Approx. time to gain 90% of the final throughput, minutes**	**OS cloning and database reinitialization duration, minutes**	**Data migration duration, minutes**	**OS cloning and database reinitialization traffic, MB**	**Partition data traffic, MB**	**Total migration traffic, MB**
Original Snowflock	40	N/A	N/A	~1830 (until 90% of final throughput is reached)		
Modified Snowflock	12	N/A	N/A	~1850 (until 90% of final throughput is reached)		
DBMS-level	2	2.5	8.4	439	2195	2634

Table 4. Data transfer measurements.

We performed the analysis of the lost work during the scale-out and scale-in periods according to Section 4.3.1. Table 5 shows the amount of lost work during *scale-out* with the following parameters: $TPM_{REQ}=100\,min^{-1}$, $t_{start}=45\,min$, $t_{end}=105\,min$.

Mechanism	**Total number of not executed New Order transactions since scale-out begins $N_{CANCELLED}$**	**Total amount of lost server working time caused by scale-out T_{LOST}, minutes**
Snowflock original	73202	24.4
Snowflock pre-allocated	10214	3.4
DBMS-level	5818	1.9

Table 5. Results of the lost work analysis (scale-out)

Table 6 shows the results of the lost work analysis during *scale-in* with the following parameters: $TPM_{REQ}=50\,min^{-1}$, $t_{start}=120\,min$, $t_{end}=180\,min$.

Mechanism	**Total number of not executed New Order transactions since scale-in begins $N_{CANCELLED}$**	**Total amount of lost server working time caused by scale-in T_{LOST}, minutes**
DBMS-level	4603	1.5

Table 6. Results of the lost work analysis (scale-in)

Quality-of-Service Analysis

We analyzed service disruption for each warehouse by calculating the number of 5-second intervals during which there are no transactions executed at that warehouse. Although this metric has some degree of subjectivity as it does not capture periods with partially (but still significantly) degraded performance, we believe it can illustrate the degree of complete service blackout. A 5-second interval corresponds to approximately 10 transactions of all types for the normal warehouse load (50 tpmC). We only analyzed the time period 45 ≤ t < 105 minutes (the scale-out period). The result is shown in the Figure 14. Due to very high original Snowflock numbers, a logarithmic scale is used in *y* axis.

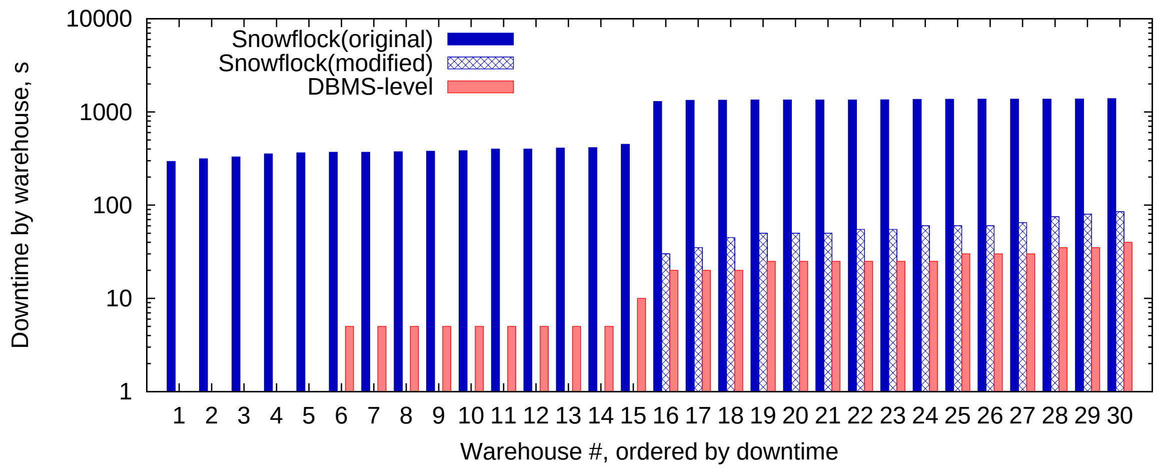


Figure 14. Downtime distribution for warehouses (scale-out, 45 ≤ t < 105 minutes)

The results show that the very costly original Snowflock mechanism causes substantial downtime for every warehouse, with half of them experiencing more disruption than the other half. The downtime for both preallocated Snowflock and DBMS-level mechanisms are similar and are experienced by the migrating warehouses only. In the case of the modified Snowflock-based mechanism, the absence of downtime for the half of the warehouses is caused by the fact that the synchronous part of Snowflock cloning takes about 2 seconds and

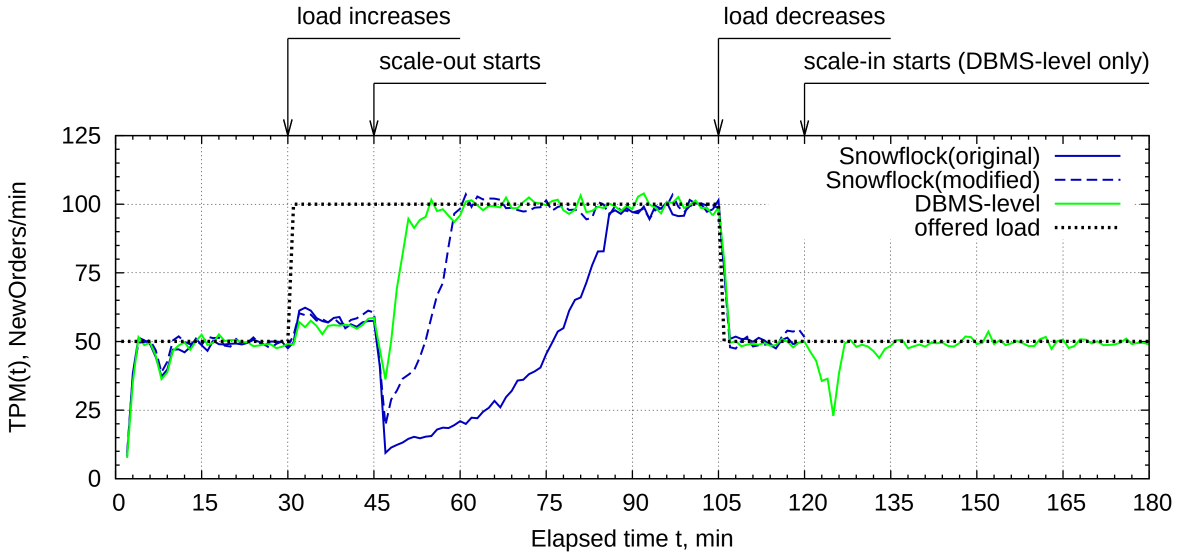


Figure 15. Average throughput in the migrating partitions, per warehouse.

is not counted as "downtime" in our test. For the migrating half, the downtime is determined by the initial slow execution period at the secondary node when there are too many "virtual page faults" (modified Snowflock) and by the one partition copy time (DBMS-level).

The throughput graphs for the migrating and non-migrating warehouses are shown in Figure 15 and Figure 16, respectively. Although the warehouse unavailability times for the DBMS-

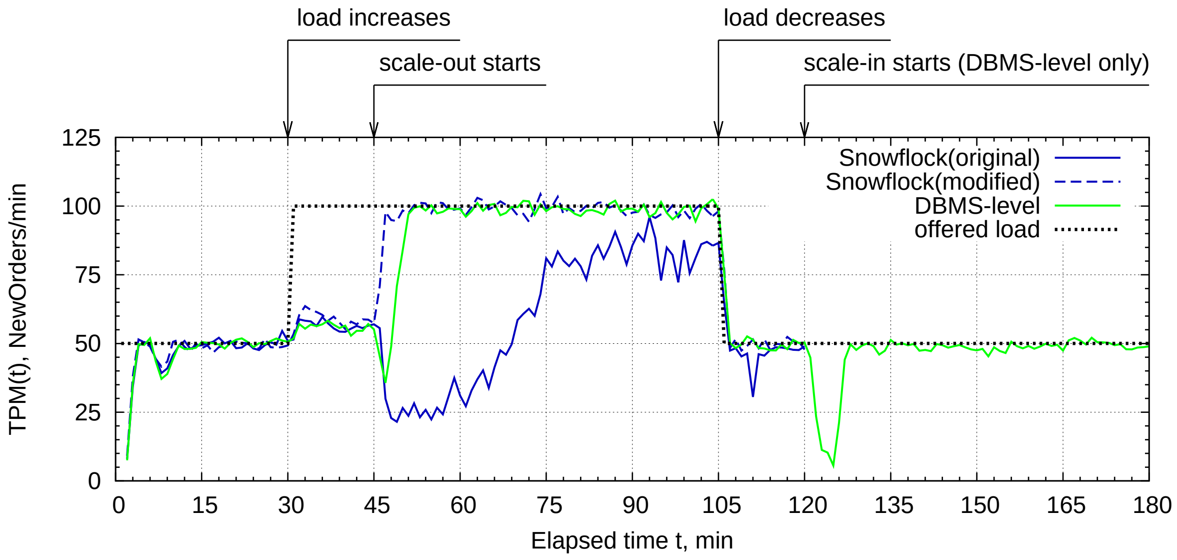


Figure 16. Average throughput in the non-migrating partitions, per warehouse.

level and the modified Snowflock mechanism are similar, the overall throughput of migrating and non-migrating warehouses is different under the two approaches. In the DBMS-level mechanism both migrating and non-migrating show approximately the same performance evolution during scale-out.

In the Snowflock mechanism, the migrating part accelerates smoothly until it reaches the target throughput level, faster for the preallocated implementation and slower for the original one. The performance of the non-migrating part, however, is drastically different for the original and modified Snowflock mechanisms. In the original one, performance improves slowly and in a jagged way, never reaching the target performance during the 1 hour time interval. On the contrary, the modified Snowflock mechanism restores the performance of the non-migrating part almost instantly, even faster than the DBMS-level mechanism.

Based on these observations we can conclude that the original Snowflock performance during scale-out depends on the factors of network latency and snapshot disk latency, the latter being the major one. However, in the modified Snowflock mechanism, snapshot disk latency no longer affects the performance.

5.3 CPU-Bound Experiments

The goal of the second set of experiments is to study how the scale-out behaviour changes without the impact of disk I/O. In this case the scale-out process is competing with the normal load for CPU cycles and memory transfers. We compare all three scalability mechanisms here: Snowflock, DBMS-level, and application-level. For all three mechanisms the system configuration is exactly the same, allowing us to compare the absolute numbers obtained from measurements. The database server runs in a VM having 4GB of memory.

To fit the database into memory, we chose the TPC-C scale factor W = 8. The initial size of the database data files is approximately 1.2 GB. From 4GB of total VM memory we allocate 1.5GB to InnoDB buffer pool so InnoDB does not experience read misses. To avoid physical writes of dirty pages we also put the database into a RAM-disk using a ramfs filesystem. In Linux, ramfs filesystems use memory to store files without any disk backing. Although this configuration is redundant in terms of memory usage, it completely eliminates physical I/O as well as the overhead of memory copying from the RAM disk to the buffer pool. In total, the data files and logs on disk and in the buffer pool consume about 3GB of memory, leaving the rest for the operational overhead of both MySQL and operating system, and for data growth.

Each warehouse in this experiment had 4 terminals connected initially, therefore, the total number of terminals was 32. During the period of increased load, the number of terminals increased to 8 per warehouse, amounting to 64 in total.

The experiment scenario follows the same timeline as the disk-bound one with timing parameters specified in Table 7.

Period	Duration, minutes	Start time, minutes
Initial warm-up period with one node	$T_1 = 5$	$t_1 = 0$
Overload	$T_2 = 5$	$t_2 = 5$
Scale-out to two nodes	$T_3 = 15$	$t_3 = 10$

Table 7. CPU-bound experiment time parameters

5.3.1 CPU-Bound, Maximum Load

The goal of the first experiment was to determine the maximum load level that one server can handle. The load generator was configured to have no delays between transactions so the next transaction is issued immediately after receiving the response for the completed one. This mode of load generation, as well as running multiple threads in the server (32 threads in the initial period), ensured that the server was flooded with transaction requests.

During the maximum load runs, we observed some random oscillation of throughput during the initial period (before scale-out) between two visibly stable levels of approximately 1000 and 1100 tpmC per warehouse for all types of scale-out mechanisms. The switch between levels occurred rarely so some runs maintained one of the levels for the entire duration of the experiment while others switched once or twice. In total we performed 12 runs for three mechanism types. To illustrate the oscillation behavior, we produced a histogram of average throughput values over 1-minute intervals for of all runs, with $1 \leq t < 10$ minutes (initial steady state). The histogram is shown in Figure 17. The distribution exhibits two clusters around of 1000 and 1130 tpmC per warehouse, with equal probability of throughput falling into either of those clusters.

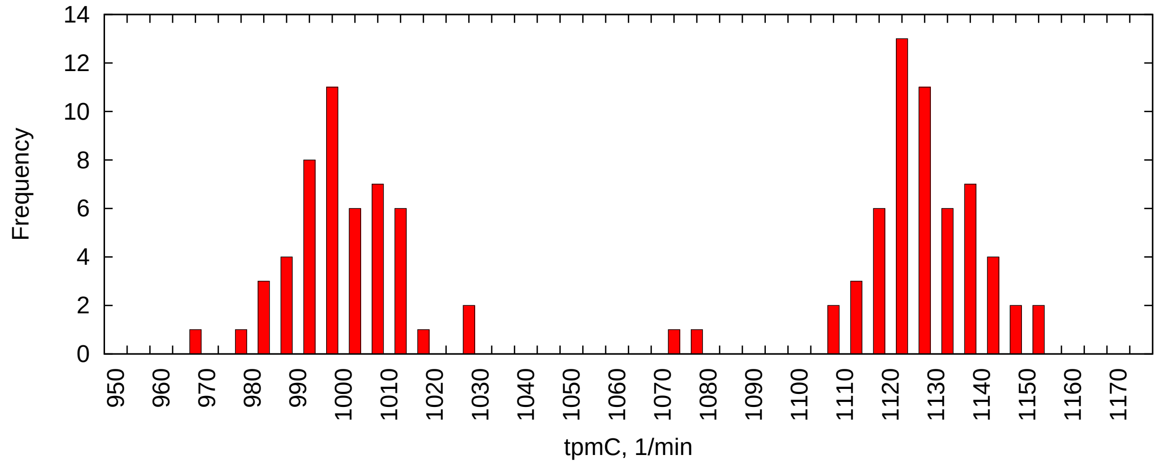


Figure 17. Distribution of throughput values of 1-minute intervals during the initial steady state period in all 12 maximum-load CPU-bound experimental runs.

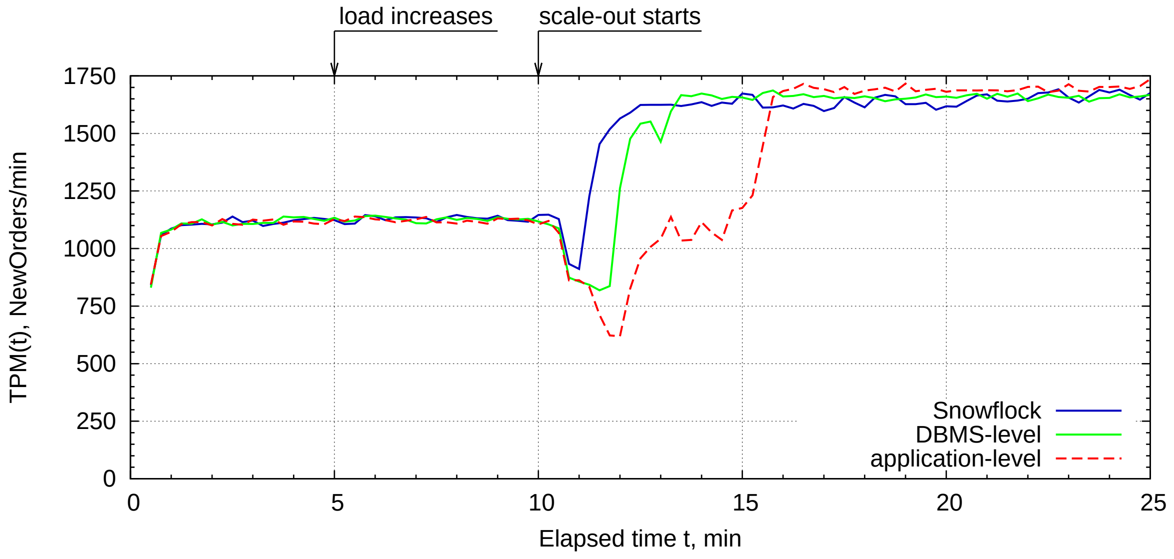


Figure 18. Comparative throughput graph for the CPU-bound experiment.

The exact cause of this variation was not discovered. In order to minimize its effect, we picked runs which had throughput equal to 1130 tpmC during the initial steady state. We also performed the same selection of runs in the controlled load experiments, based on the throughput level during the overload period, when the throughput is limited by the server.

The complete throughput graph is shown in the Figure 18. From the visual observations, we chose the first two minutes as the warm-up time, the next three minutes as the first steady state and the last five minutes as the second steady state. Although we might expect a small instability of throughput due to contention in the engine during the time interval 5 to 10 minutes, when the number of threads is being increased, it remained almost equally steady, thus the contention factor is insignificant for those experiments. There is also no change in

Mechanism	**Steady state 1, t=2..5 minutes**	**Steady state 2, t=20..25 minutes**
Snowflock	1119	1662
DBMS-level	1129	1658
Application-level	1116	1696
Average	1121	1672
Standard deviation	6.6	21

Table 8. Average total throughput during the initial and the final steady state periods (normalized per warehouse), maximum load, tpmC.

the average throughput value, meaning the performance is fully determined by the server side and it is at its maximum.

The average throughput numbers for the corresponding steady state are summarized in the Table 8, we will use those number as a base for the controlled load level in the later experiments.

5.3.2 CPU-Bound, Controlled Load

In the next experiment we model a more realistic scenario by maintaining the workload at a configured level. The initial client-offered load is lower than the server can potentially handle, thus leaving some spare capacity to simulate overprovisioning of computing resources. This experiment has the same settings as the previous experiment, with the exception of the load generation.

We set the scheduled rate of client-generated transactions at 750 tpmC per warehouse, which is about one third less than the peak rate measure in the previous "maximum load" experiment. During the excessive load period, we are running twice as many clients without changing the database size, effectively increasing the offered load to 1500 tpmC per warehouse.

Cost and Throughput Analysis

The transaction throughput and average response graphs are shown in Figure 19 and Figure 20 respectively. At $t = 5$ minutes the offered load level is doubled, however, the system throughput increases only by the amount of overprovisioned capacity. The response times rise significantly, indicating that the offered load exceeds the server's capacity.

The most interesting part of the graphs are at 10 minutes and beyond, when the system undergoes a transition period. We observe here that the Snowflock scalability mechanism can recover most of the maximum throughput in about 1 minute, followed by the DBMS-level mechanism (about 2 minutes) and the application-level mechanism (5 minutes).

For a reason that has not been determined, in both DBMS-level and application-level test runs the Snowflock cloning process took longer than in the Snowflock test runs, with approximately 25 second delay between issuing the clone command by the control script and actual observable network and performance effects. This delay did not occur in the Snowflock scale-out process.

Mechanism	$N_{CANCELLED}$	T_{LOST} **, seconds**
Snowflock	8913	5.2
DBMS-level	10005	13.3
Application-level	22623	30.2

Table 9. Results of the lost work analysis, scale-out

To evaluate the cost of scale-out for a provider, we aggregated the total number of New Order transaction cancellations $N_{CANCELLED}$ starting at t=10 minutes, when scale-out starts, and converted this number to time units T_{LOST}, normalizing by the initial offered load, according to (7). The resulting time quantities represent the *lost opportunity*, the working time of the server that could have been saved if scale-out was "free". In the environments where server time has a monetary cost, as in cloud platforms, those numbers can directly translate into the monetary cost of scale-out. Table 9 shows the results of the computations using the following parameters: $TPM_{REQ}=1500\,min^{-1}$, $t_{start}=10\,min$, $t_{end}=25\,min$.

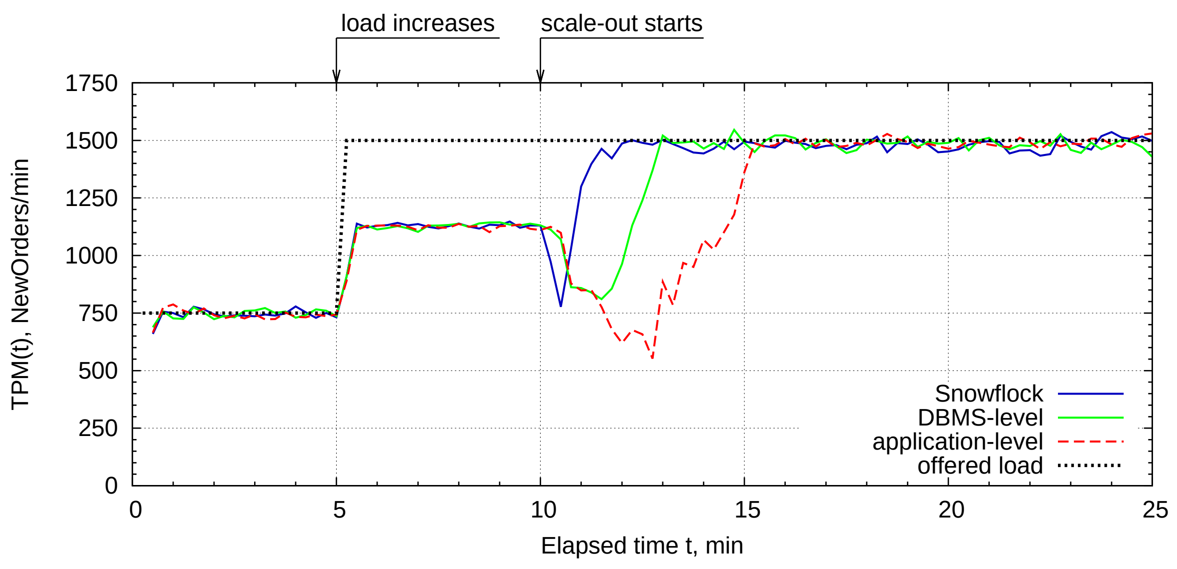


Figure 19. Average transaction throughput in the CPU-bound experiment, per warehouse.

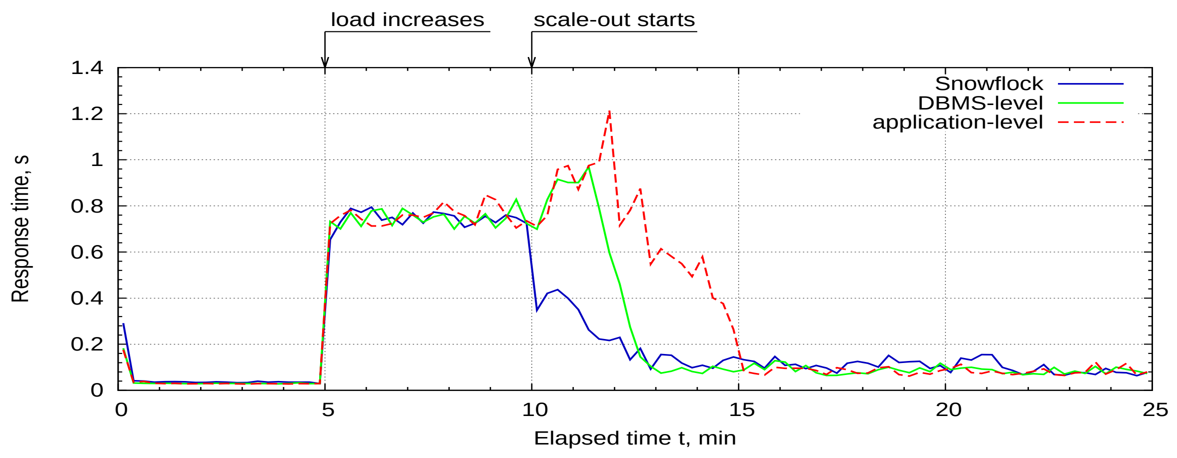


Figure 20. Average response times during the CPU-bound experiment

To study the efficiency of database state transfer we analyze the network traffic between the physical hosts during the experiments. The network throughput graph is shown on the Figure 21. The total traffic includes both the migrated data as well as the client-server protocol messages and there is no simple way to precisely isolate the traffic related to partition migration. However, we expect that most of the throughput is used by the migration traffic.

Figure 21 illustrates the network throughput of the different scale-out mechanisms. The Snowflock mechanism has an initial surge of data with an exponential decay that, however, does not fade completely but instead stabilizes after approximately 2 minutes with much slower further decline. It is interesting to note that the total performance (in Figure 19) of the system is mostly recovered within these 2 minutes so we can conclude that the residual network transfer at a moderate rate (~2MB/s) does not affect the performance.

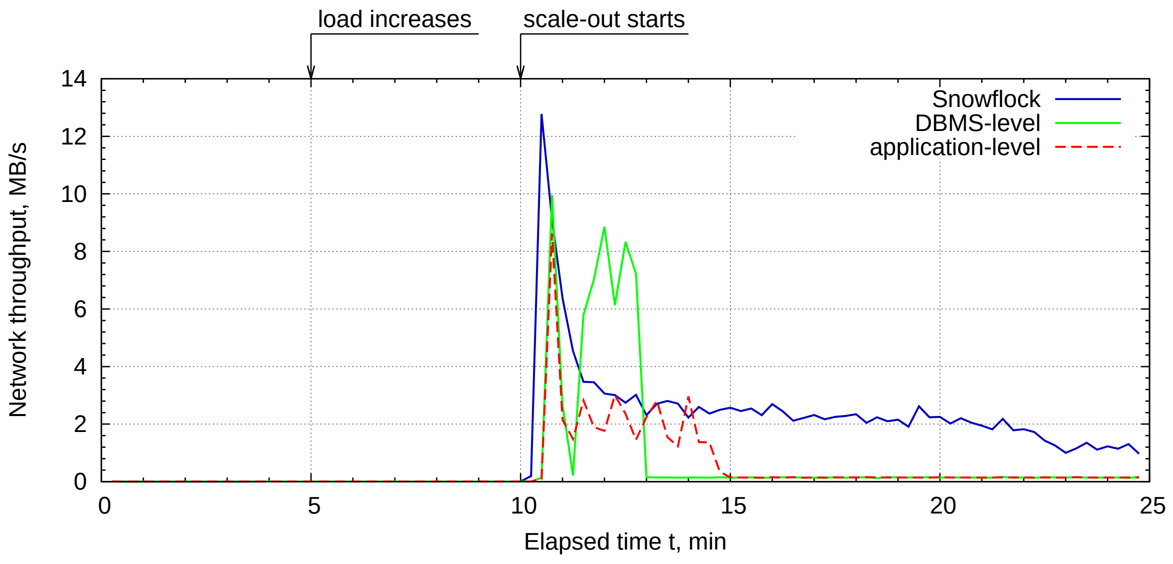


Figure 21. Network throughput. For DBMS-level and application-level mechanisms, the first peak corresponds to Snowflock-based OS state and empty database transfer and the further high utilization levels are for the partition transfers. During the gap after the first peak the MySQL is initializing at the new node.

Both DBMS-level and application-level scale-out mechanisms show a similar (but smaller) surge due to the Snowflock-implemented transfer of operating system state, followed by a short pause when the secondary system starts up, followed by the partition migration traffic.

The results of measuring the total amount of network traffic are shown in the table Table 10. For the Snowflock-based mechanism, the amount of data was measured up to the time reported in the second column, corresponding to 90% performance level. The data transfer in this run continued through all 25 minutes of the test, exceeding 2GB in total, and has not completed by then. For the DBMS-level and application-level mechanism the amounts in the table were measured separately during the OS state and partition data transfers periods. Each period was marked by timestamps in the log files.

From this table it can be seen that the Snowflock-based mechanism has the smallest amount of data transferred. We can attribute this result to its on-demand copying strategy, and non-uniform page access distribution in the TPC-C workload.

Having the migration duration and the amount of data transfer from Table 10, we can calculate the average network utilization during this time. The DBMS-level mechanism achieves the highest network utilization, reaching 6.2 MB/s. The application-level mechanism only used 1.6 MB/s on average, which indicates much higher internal costs of processing the transferred data in MySQL. On the other hand, the application-level used SQL-based messages for data transfer, which were about 40% more compact than binary representation of InnoDB data pages, used by the DBMS-level mechanism.

Mechanism	**Approx. time to gain 90% of the final throughput, seconds**	**OS cloning and database reinitialization duration, seconds**	**Data migration duration, seconds**	**OS cloning and database reinitialization traffic, MB**	**Partition data traffic, MB**	**Total migration traffic, MB**
Snowflock	60	N/A	N/A	~400 (until 90% throughput is reached)		
DBMS-level	150	79 *	101	188	627	815
application-level	280	72 *	233	179	380	560

* Including approximately 25 second delay before actual Snowflock cloning starts.

Table 10. Data transfer measurements in the CPU-bound experiment.

Quality-of-Service Analysis

First, we present the distribution of downtime over warehouses in Figure 22. We obtain this result similarly to the disk-bound experiments, using the time interval of 1 second. Therefore, we counted the number of such intervals during which no transaction executed as the length of downtime. Here the application-level mechanism imposes significant downtime for warehouses, including those that are not migrating. In contrast, the Snowflock and DBMS-level mechanisms cause unavailability only for the migrating warehouse.

To further study the impact of scale-out we plotted the *worst-case* throughput graphs in Figure 23. The graph shows the throughput of the *worst-performing* warehouse in every 10-second time slot. The graph shows that all mechanisms make some warehouses completely unavailable with the application-level one having the longest periods of unavailability taking almost all scale-out time. The DBMS-level one has shorter but repeating downtime periods.

Finally, the Snowflock-based mechanism demonstrates the shortest downtime caused by the synchronous phase of its cloning operation, taking only a few seconds.

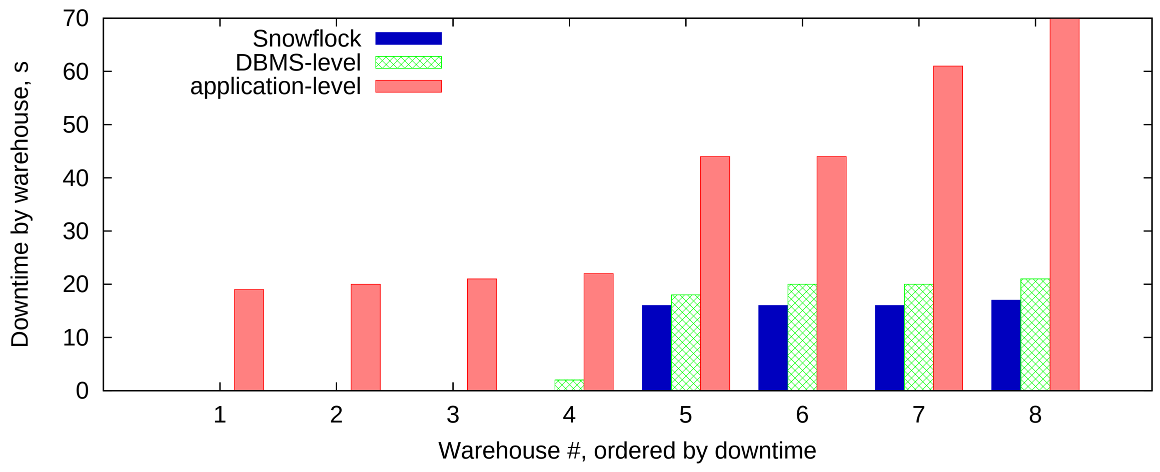


Figure 22. Number of one-second time intervals when a warehouse is not available.

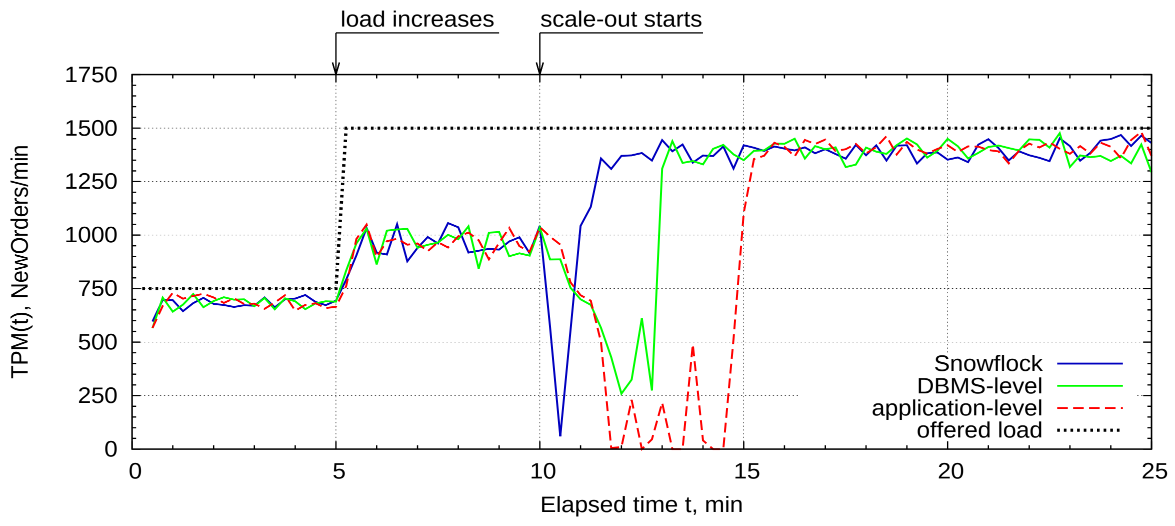


Figure 23. Worst-case throughput in the CPU-bound experiment.

Summarizing, we can conclude that the costly application-level mechanism cause significantly more transaction processing disruption than the DBMS-level and Snowflock-based ones. The latter two have comparable downtime but it is distributed differently between warehouses. The DBMS-level mechanism disables warehouses one by one, taking more time in total, and the Snowflock-based mechanism stops all warehouses whose data is being migrated simultaneously, but for a short time.

Chapter 6
Related Work

The work described in the thesis is related to research in the areas of distributed databases, virtualization, cloud computing, and dynamic resource management. In this section we provide a short review of those areas as well as alternative approaches to our problem or other similar problems.

6.1 Cloud Computing

This thesis was motivated by the problem of efficient reconfiguration of database services in a cloud computing environment in response to changes in users' demand. Cloud computing promises to provide both flexibility to users in consuming computing services and more efficient resource utilization for service providers.

A variety of online services, which can be advertised as cloud services, are offered on the market. One example is Amazon AWS [4], which stands for Amazon Web Services. Amazon has a wide offering of services, spanning multiple layers. The noteworthy examples include IaaS-level Xen-based virtual machines dubbed Elastic Computing Cloud (EC2), persistent Elastic Block Store (EBS) and Simple Storage Services (S3), and higher-level SimpleDB key-value store and Relational Database Service (RDS). The latter [6] is the most relevant for our study as it offers a locked-down modified MySQL database server, running on shared hardware (most likely virtual), and utilizing a persistent distributed shared storage, similar to EBS. The elasticity of RDS is limited to ability to scale-up and scale-down, migrating a database instance between several offered "instance classes" of different computational and storage capacities. According to RDS documentation [7], switching the instance type incurs a "short downtime". Our approach, in which nodes are added or removed in the shared-nothing architecture, is potentially more scalable because it is not limited by the capacity of a single node.

6.2 Virtualization

An important type of cloud services, called infrastructure-as-a-service, depends on the concept of a virtual machine (VM) as a means to provision computational resources. Despite being known for decades, for example in IBM's series of mainframes [8], the widespread use of VMs has begun with the introduction of hardware virtualization technologies to the x86 processor architecture [9]. With hardware virtualization it has become possible to efficiently run multiple instances of an operating system as an isolated virtual machine (VM) with no or little modification on commodity servers.

Each VM is controlled by a *hypervisor.* Xen [10] is an open-source hypervisor implementation, widely used in production and often chosen as a base system for experimental research. A Xen extension [11] enables *live migration* of virtual machines with the disruption of service limited to a few dozen or hundred milliseconds. Clark et al [11] discusses in detail the strategies to copy the memory state, including *stop-and-copy, pre-copy,* and *pure on-demand* memory migration strategies. The pre-copy strategy transfers most of the VM state concurrently with the running VM and suspends only for a short time to complete the migration. The iterative pre-copy strategy provides the shortest downtime with balanced total migration time and overhead during the initial stage of the migration. In addition to management purposes, the live migration mechanism may be used to scale a system or for load balancing within a cluster of virtual machines. Short downtime is only possible when only a small amount of state, usually contained in the main memory, is transferred. The implementation of migration at the very low level of the Xen hypervisor, while isolating an application and OS from the migration process, has a VM as a minimal unit of resource allocation. In hypervisor-based approaches, making the partitioning unit smaller than a VM is not possible as the only way to ensure VM consistency during migration, without knowing the details of the higher-level application state, is to take a snapshot of a whole VM. The consequence of that coarsely granular service partitioning is extra redundancy and higher associated overhead. On the other hand, the discussed live migration strategies [11] can potentially be applied to upper levels of the software stack, such as to a DBMS.

The ability of a hypervisor to manipulate the state of VMs transparently to the applications running inside them can be used to implement a coarse form of scalability. When a VM needs more (or less) processing resources, the state of the VM is saved, transferred to another, more (or less) powerful host, and the VM is restored. For example, Amazon calls this ability *Auto Scaling* in their EC2 platform [5]. However, more advanced techniques are needed for more transparent and efficient scalability.

Snowflock [12] is a virtual machine cloning mechanism based on the Xen hypervisor. In contrast to the basic feature of hypervisors, Xen included, to save and restore a running VM, Snowflock does *multi-way* live VM cloning in an efficient manner. The efficiency is defined as the ability to *instantly* make clones without blocking copying of the whole VM state. The resulting VMs have information about the CPU context but no memory or disk pages. The remaining parts are transferred *on-demand*, as the clones try to run and cause *virtual page faults* by touching memory pages, or perform disk I/O. To gain further efficiency Snowflock employs multicast network transfers and several optimizations to avoid transferring parts of memory that will not be used at the clone. Snowflock is best suited for ad-hoc computational tasks that share little state between the original VM copy and the clones.

In this thesis we extensively used the concept of VMs. However, we explored ways to manipulate partitions smaller than a VM. One of the studied mechanisms uses the Snowflock hypervisor as a sole means to gradually migrate the database state between nodes in a cluster. The other two mechanisms only provision empty VMs with Snowflock and migrate the state as a separate step.

6.3 Parallel and Distributed Databases

Traditionally, the problem of database scalability has been addressed by using parallel and distributed systems. Horizontal partitioning is a long-known set of techniques to improve performance of database systems [20]. The degree of parallelism varied between different approaches to partitioning. Distributing the database files between multiple disks within one machine allows for parallel I/O while keeping the query processing centralized. Another approach is increasing the number of processing nodes and distributing the query processing among them. In that case the architectures of data distribution can be further classified into *shared-disk* and *shared-nothing* [21].

The shared-disk architecture includes a single high-throughput I/O subsystem shared between nodes in a coordinated manner [23]. The complexity of the shared storage subsystem results in the higher data access latencies and the required coordination may impose a scalability limit on increasing the number of processing nodes. On the other hand, a shared storage system can more easily balance the load between the processing nodes as no persistent data movement is required.

In the shared-nothing architecture each processing node has its locally-attached permanent storage. This architecture tends to have a lower cost as there is no need for the specialized high-throughput shared storage subsystem and it can be composed of components, combining CPU(s), local memory, and disk(s) in one standardized package. The shared-nothing approach is hypothesized to have better potential scalability [21]. However, this approach has the problem of efficient distribution of the data between nodes and the need for complex algorithms to optimize query execution.

Traditionally, in shared-nothing architectures the partitioning scheme was defined statically during the physical design phase and changing it required a manual intervention and a costly data reorganization. Examples of static distributed partitioning implementations include DB2 Parallel Edition [22] and federated servers (distributed partitioned views) in Microsoft SQL Server 2005 [24].

MySQL open-source DBMS contains two implementations of data partitioning, both accessible with a single SQL interface. The first one is a feature of the MySQL Cluster storage engine and the second one is a separate partitioning storage engine, handling partitioning atop of other storage engines [25]. The flexibility of MySQL to provide various data organization options stems from its pluggable storage engine architecture, logically separating the upper query-processing layer from lower levels via a data access interface. There a number of storage engines available for MySQL, including some developed internally and some developed by third parties.

MySQL Cluster originated as an Ericsson project [29] aiming to create a fast, reliable, replicated main-memory DBMS for the telecommunication industry. The project supports both horizontal and vertical table partitioning, with horizontal partitioning using distributed linear hashing and distributed B+tree as fragmentation algorithms. The choice of either one apparently depends on the user's requirement to have an order-preserving indexing. Both algorithms permit dynamic splitting and merging of fragments in response to node failures, node addition, or load balancing due to a management request. During fragment splitting or merging, tuples are transferred one by one to prevent large-scale fragment locking. Any active transactions are allowed to execute at the source node and tuple modifications are sent to the target node along with the stored tuples in a block. Despite only having in-memory data, fragment migration is referred to as costly operations, requiring care when planning the cluster node configuration.

MySQL Partitioning Engine (MySQL PE) [25] was developed to mimic the MySQL Cluster partitioning interface, with the ability to use any other MySQL storage engine as an underlying data storage. MySQL PE partitioning criteria are based on hashing, ranges, and lists of attribute values. MySQL PE interacts with the MySQL query optimizer to *prune* the partitions that are not used by the query. The current implementation of MySQL PE supports dynamic reorganization of partitions using administrative SQL DDL interface, however, those reorganizations involve costly data movements between the underlying tables. The major shortcoming of MySQL PE is that, despite the fact MySQL includes a storage engine to access remote servers, it is not possible to implement a *distributed* partitioning using this storage engine. Another potential deficiency is a lack of optimization when the number of partitions is large, due to use of O(n) algorithms in MySQL PE, related to the number of partitions, and significant overhead of processing inside MySQL PE itself [26].

In this thesis we used the shared-nothing approach for distributing the database between cluster nodes in the DBMS-level and application-level scale-out mechanisms. We took MySQL PE as a tool to implement partitioning and extended it with ability to dynamically and efficiently move partitions between working servers.

An optimized and feature-enhanced version of MySQL, XtraDB, was developed by Percona [27]. XtraDB is proposed as a better MySQL alternative for application hosting providers implementing SaaS scenarios. Besides general performance improvements, XtraDB is claimed to be better suited for multi-tenant configurations due to flexible schema management, advanced monitoring tools, and the ability to transfer tables between running nodes. The latter feature is particularly relevant for our work and our database-level scale-out mechanism used some ideas of InnoDB table data modification to allow their compatibility between different servers. However, XtraDB does not include tools for doing scale-out itself, which has to be implemented by users. The partitioning design and query routing in XtraDB should also be provided by applications.

Hauglid et al [34] proposed an algorithm for dynamic database fragmentation and for finding an optimal placement of fragments over a set of cluster nodes. The fragment placement decisions are based on a simple cost model using a statistical analysis of row access frequencies. The cost model takes into account the frequency of remote data accesses, including the cost of fragment migration, as a cost metric for minimization. Client queries are presumed to be simple row operations (read, write) that originate from the same nodes where the database is located. Depending on the relative frequencies of reads and writes, the system may choose to create multiple replicas for a fragment. A read request has the cost of transferring a row from the closest node, which is a local node in the best case. A write request has the cost of one or more transfers to each of the replicas. The costs of read and writes are assumed to be constant for every local and remote operation, which does not account for effects of caching and row placement locality. An important issue is the low-cost online cost estimation algorithm that uses access statistics from all sites for the ranges of fragmentation attribute to make decisions on optimal fragment placement.

6.4 Scalable Non-Relational Data Services

The elasticity and scalability of traditional relational DBMS are usually perceived as limited. Regardless of whether this public perception reflects the inherent architectural limitations or just practical experiences with implementations that are often aged and inflexible, a number of attempts have been made to design better data-management systems from scratch. These newer distributed data stores emphasize elasticity, scalability, and fault-tolerance as primary goals, while offering simple programming models, such as a key-value structured dictionaries or *key-value stores*. Typical examples are BigTable [17], HBase [19], Dynamo [18], and Cassandra [16]. Those systems are typically intended to provide only weak consistency guarantees, however, some of them allow a user to adjust the consistency level to some extent. As a rule, distributed key-values stores can incrementally add and remove computing nodes, either for planned maintenance or due to a node failure. Adding or removing a node possibly causes a query to visit additional nodes while the background reconfiguration is in progress. However, the overhead of repartitioning is considered small since nodes are expected to arrive or depart in small numbers so only a small part of the system is affected at a time. We focus on scaling out a relational database, which provides richer programming model to applications compared to key-value stores. However, their internal mechanisms for data distribution, request routing, and elastic reconfiguration may be of interest for DBMS scalability mechanisms.

6.5 Multi-tenant databases

As an alternative to addressing the scalability problem for an arbitrary application by means of improving the underlying system software, some opportunities emerged with the approach of service a collection of unrelated applications using a single system. The term *multi-tenant* DBMS is used to describe systems in which a single instance of a DBMS hosts a number of applications from independent users (tenants) [14]. Multi-tenant databases align naturally with the Software as a Service (SaaS) and cloud computing architectures, promising the benefits of denser consolidation and centralized management. The independence of tenants' applications justifies the practical importance of a completely disjoint partitioning model in which fragments may be moved between nodes more easily. We used the same approach to partitioning in this thesis. A detailed discussion of the techniques of the dynamic migration of disjoint partitions appears in [31]. It is worth noting that building multi-tenant systems atop of state-of-the-art database implementations introduces some challenges, particularly, in terms of isolation, security, and resource scheduling in the presence of non-uniform resource usage by different tenants [13], [15].

The issue of data migration efficiency was studied within the multi-tenant database context [30]. In the paper, several classification schemes are proposed for migration scenarios. One of them describes the degree of service interruption, according to which, the proposed migration forms include *live, synchronous,* and *asynchronous*. *Live migration* allows the system to continue to service transactions with no unavailability window. With s*ynchronous migration* a system performs most work in parallel with current activities, incurring only minimal interruption during the final switch. Finally, *asynchronous migration* stands for the approach which blocks active transactions that may interfere with the migration process and implies the longest service disruption. With live migration dismissed as practically impossible to implement, synchronous migration is proposed as the preferred approach. The characteristics of various migration types are summarized in Table 11.

Form of Migration	**Downtime**	**Interruption of Service**	**External Coordination**	**Operation Overhead**	**Migration Overhead**
Live	None	Very Minimal	Minimal	Low/Moderate	Minimal
Synchronous	Minimal	Minimal	Moderate	Minimal	Moderate
Asynchronous	Moderate	Moderate	High	None	High

Table 11. Summary of the forms of migration and the associated costs [30].

Multi-tenant models are also classified [30] by the performance and service disruption for clients; we summarize this classification in Table 12. Reading this paper raises the question whether the attributed costs and the severity of service disruption are determined by the fundamental approach to elastic service organization or they are merely the result of using particular, commonly used implementations. Addressing this question requires understanding the reasons for various performance behaviours of scalable systems, which was a part of motivation for our work.

Two practical implementations of multi-tenant database migration were proposed recently [32][33]. Those implementation addresses the very same problem as we did in this thesis, elastic scale-out of a multi-tenant database. Zephyr [32] is a DBMS-level implementation for *live* migration of tenants in the shared-nothing architecture. The support of active transactions is an advantage over our work. Zephyr's migration algorithm combines bulk asynchronous transfer of a tenant and synchronous on-demand transfers of the pages which were accessed in the destination system. Another architectural difference is Zephyr's use of intermediary query routers to maintain the locations of tenants and send transactions accordingly. Such routers may need to be scaled out along with the database backends. Despite the claims that Zephyr implements live migration, there are limitations on type of operations that transactions are allowed during migration. For example, upper levels of indexes cannot be modified and pages which have already been transferred cannot be modified at the source node. In both those cases transactions will have to abort.

Another related approach to live tenant migration is implemented in a technique, called Albatross, but for *shared-disk* architecture [33]. This approach uses the iterative copying

Model	**Type of isolation**	**Runtime overhead (redundancy)**	**Service interuption during migration**	**Cost of migration**
Shared hardware	Virtual machine	Moderate	No/minimal	Low
Shared instance	Physical DBMS copy	Minimal	Minimal	Minimal
Shared database	Logical DBMS schema	No	Moderate	High
Shared table	Rows	No	Efficient migration is challenging due to heavy coupling between fragments	

Table 12. Milti-tenancy models from [30] and their implications on migration

algorithm. In this algorithm, a buffer pool snapshot, which may be not consistent, is sent first. Then, the changes to the buffer pool since the time the snapshot was taken, including the pages read from disk as well as modified by transactions, are sent. This step may be repeated several times until the amount of sent data in each iteration stops decreasing. Finally, the rest of the modifications are sent to the destination and the state of query routers is updated. This is done synchronously, imposing a small unavailability window.

Curino at al [28] address the combination of three problems: automatic partitioning, live migration, and dynamic resource allocation. The live migration part closely corresponds to our research, however, only requirements and suggestions for implementation are proposed. The eventual goal is to allow frequent migration of partitions with minimal cost while allowing the usual workload to execute. The steps to achieve this goal included: partitioning into smaller fragments; copying a snapshot of a partition during the active load and sending the change logs afterwards; distributing the read-only workload between replicas; and warming-up stand-by replicas. An on-demand approach is also considered as a way to minimize the cost of migration. In that case, the new node fetches data from the old one while it is trying to execute the load, remembering the data it receives. We consider this approach to be similar to the one studied in the thesis.

Chapter 7
Conclusions and Future Work

In this thesis we studied the problem of scale-out of elastic transactional database services. Our approach was limited by some constraints, particularly, the requirement of perfect data partitioning and disabling transaction processing for affected partitions. During our study we investigated the use of a hypervisor-based, on-demand block copying mechanism built using Snowflock. As an alternative, we developed a DBMS-level mechanism that can take advantage of the partition physical locations and that uses bulk data transfers and other optimizations. We built this mechanism into a major open-source database server, MySQL.

To show the relative efficiency of those two main scale-out mechanisms, we performed a series of experiments, comparing their performance characteristics, as well as the impact of scale-out on client applications. In addition to the two main methods we included a simple, application-based scale-out mechanism as a baseline. The application-level mechanism is the simplest and is portable between database systems. However, the application-based method was inferior in all our experiments, demonstrating the fact that the efficiency of a scale-out mechanism is important for an elastic system. We suppose the application-level mechanism can be improved, however, it might lose its simplicity and portability in this case.

The experiments were conducted in CPU-bound and disk-bound settings. We found that the presence of disk access latency is the key factor affecting the balance between the on-demand transfers and bulk transfers. In the CPU-bound experiments, the Snowflock-based method showed the best results in terms of throughput and the DBMS-based method was less efficient. In terms of client-experienced service disruption, those two methods were comparable, however, their impact was different. The Snowflock-based method caused short downtime for all the migrating clients while the DBMS-based one affected one client at a time but taking longer. This partition-wise downtime pattern may be preferable for some applications, for example, ones that prioritize work made by different clients. For such applications, the controller may choose to migrate partitions of lower-priority clients. In the disk-bound setting the Snowflock-based method showed greater throughput loss as well as service disruption, compared to the DBMS-level one. The main contributing factors of Snowflock's poorer performance in the disk-bound case are its need to maintain consistent snapshots of disk images while accommodating concurrent write activity and its dependence on synchronous high-latency transfers of individual small disk blocks over the network. The former factor was responsible for most of the performance impact in the original Snowflock implementation.

Despite the good performance of the Snowflock-based mechanism in some scenarios, we believe the biggest issue with using a hypervisor-based mechanism is not the performance characteristics, which are reasonable and can be further improved, but its architectural limitations stemming from the lack of information about the partitioning. As a result, only a very restricted scale-out scenario is possible, in which an existing server is split. It is not possible to balance the load incrementally among running servers by migrating partitions using a hypervisor-based mechanism. It is also not possible to implement scale-in. In addition, Snowflock cannot allocate resources for snapshot maintenance in a partition-wise way, instead it consumes unnecessary amount of memory and disk space for the whole server snapshot, which is never reclaimed.

Considering the relatively good results of the on-demand data approach of the Snowlock-based mechanism in the CPU-bound setting and its deficiency in the presence of disk and network latencies, as well as its architectural shortcomings noted above, we can imagine a future scale-out mechanism which leverages *both* on-demand block transfers *and* partition awareness. We believe that combination is essential for efficient scale-out. Additionally, bulk transfers can be utilized for low priority transfers of data that are not accessed by the on-demand transfers.

Another possible future direction would be elimination of the requirement to block client activity on affected partitions during partition migration. We believe it would be a very important practical advantage. At the same time, there are no principal difficulties to develop such live migration mechanism as the amount of data modifications during the migration is usually small compared to the size of the database.

Bibliography

[1] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. IEEE Computer, vol. 40, 2007.

[2] Green Procurement Initiative. Nine lessons in greening IT. http://www.greeninggreatertoronto.ca/pdf/GGT-Green-Exchange-ITSummary.pdf.

[3] RightScale. More Servers, Bigger Servers, Longer Servers, and 10x of That. http://blog.rightscale.com/2010/08/04/more-bigger-longerservers-10x/.

[4] Amazon Web Services. http://aws.amazon.com/.

[5] Amazon EC2 Auto Scaling. http://aws.amazon.com/autoscaling/.

[6] Amazon Relational Database Service. http://aws.amazon.com/rds/.

[7] Amazon Relational Database Service. User Guide. http://docs.amazonwebservices.com/AmazonRDS/latest/UserGuide/.

[8] Melinda Varian. VM and the VM Community: Past, Present, and Future. Princeton University. SHARE 89, 1997.

[9] Intel® Virtualization Technology: Hardware support for efficient processor virtualization. Intel® Technology Journal. Volume 10, Issue 03, 2006.

[10] Paul T. Barham, Boris Dragovic, Keir Fraser, Steven Hand, Timothy L. Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In Proc. ACM Symp. on Operating Systems Principles (SOSP), 2003.

[11] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2 (NSDI'05), Vol. 2. USENIX Association, Berkeley, CA, USA, 2005.

[12] H. Andres Lagar-Cavilla, Joseph Whitney, Adin Scannell, Philip Patchin , Stephen M. Rumble, Eyal de Lara, Michael Brudno, M. Satyanarayanan. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. 3rd European Conference on Computer Systems (Eurosys), Nuremberg, Germany, 2009

[13] B. Reinwald. Database support for multi-tenant applications. In IEEE Workshop on Information and Software as Services, 2010.

[14] D. Jacobs and S. Aulbach. Ruminations on multi-tenant databases. In Datenbanksysteme in Büro, Technik und Wissenschaft (BTW), pages 514–521, 2007.

[15] Stefan Aulbach, Torsten Grust, Dean Jacobs, Alfons Kemper, and Jan Rittinger. 2008. Multi-tenant databases for software as a service: schema-mapping techniques. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD '08). ACM, New York, NY, USA, 1195-1206.

[16] Cassandra: A highly scalable, eventually consistent, distributed, structured key-value store, 2010. http://incubator.apache.org/cassandra/.

[17] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. ACM Trans. Comput. Syst. 26, 2, Article 4 (June 2008), 2008.

[18] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (SOSP '07). ACM, New York, NY, USA, 2007.

[19] HBase: Bigtable-like structured storage for Hadoop, HDFS. http://hadoop.apache.org/hbase/.

[20] S. Ceri , M. Negri , G. Pelagatti, Horizontal data partitioning in database design, Proceedings of the ACM SIGMOD international conference on Management of data, 1982.

[21] David DeWitt and Jim Gray. 1992. Parallel database systems: the future of high performance database systems. Commun. ACM 35, 6, 1992.

[22] C. K. Baru, G. Fecteau, A. Goyal, H. Hsiao, A. Jhingran, S. Padmanabhan, G. P. Copeland, and W. G. Wilson. 1995. DB2 parallel edition. IBM Syst. J. 34, 2, 1995.

[23] Oracle Corporation. Oracle9i Real Application Clusters Concepts Release 1 (9.0.1), Part Number A89867-01.

[24] Understanding Federated Database Servers. MSDN Library. http://msdn.microsoft.com/en-us/library/ms187467.aspx.

[25] MySQL 5.1 Reference manual. Partitioning. http://dev.mysql.com/doc/refman/5.1/en/partitioning.html.

[26] Peter Zaitsev. How many partitions can you have? http://www.mysqlperformanceblog.com/2009/12/05/how-many-partitions-can-you-have/.

[27] Baron Schwartz, Vadim Tkachenko. Percona Server with XtraDB for Software-as-a-Service Application Databases. A Percona White Paper, 2010.

[28] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: a workload-driven approach to database replication and partitioning. Proc. VLDB Endow. 3, 1-2, 2010.

[29] Mikael Ronström. Design and Modelling of a Parallel Data Server for Telecom Applications. PhD thesis. Linköping University, Sweden. 1998.

[30] Aaron Elmore, Sudipto Das, Divyakant Agrawal, Amr El Abbadi. Who’s Driving this Cloud? Towards Efficient Migration for Elastic and Autonomic Multitenant Databases. Technical Report 2010-05, CS, UCSB, 2010. http://www.cs.ucsb.edu/research/tech_reports/.

[31] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, Amr El Abbadi. Live Database Migration for Elasticity in a Multitenant Database for Cloud Platforms. Technical Report 2010-09, CS, UCSB, 2010. http://www.cs.ucsb.edu/research/tech_reports/.

[32] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In Proceedings of the 2011 international conference on Management of data (SIGMOD '11). ACM, New York, NY, USA, 2011.

[33] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. Proc. VLDB Endow. 4, 8, 2011.

[34] Jon Olav Hauglid, Norvald H. Ryeng, and Kjetil Nørvåg. DYFRAM: dynamic fragmentation and replica management in distributed database systems. Distrib. Parallel Databases 28, 2-3, 2010.

[35] Transaction Processing Performance Council. TPC BENCHMARK™ C. Standard Specification. Revision 5.11, 2010. http://www.tpc.org/tpcc.

[36] OSDL Database Test 2 (DBT-2). http://osdldbt.sourceforge.net/.

[37] CERAS IBM Blade Centre at UW. http://www.cs.uwaterloo.ca/cscf/research/cerasblade/cerasblade.html.

[38] IBM BladeCenter LS21. http://publib.boulder.ibm.com/infocenter/bladectr/documentation/index.jsp?topic=/com.ibm.bladecenter.ls21.doc/bls_ls21_product_page. html.

[39] Seagate Savvio Specifications. Seagate Publication Number DS1563-4. 2006.