Online Scaling in a Highly Available Database

Svein Erik Bratsberg and Rune Humborstad
Clustra AS
N-7485 Trondheim, Norway
E-mail: svein.erik.bratsberg@clustra.com

Abstract

An increasing number of database applications demands high availability combined with online scalability and soft real-time transaction response. This means that scaling must be done online and non-blocking. This paper extends the primary/hot standby approach to high availability with online scaling operations. The challenges are to do this without degrading the response time and throughput of transactions and to support high availability throughout the scaling period. We measure the impact of online scaling on response time and throughput using different scheduling schemes. We also show some of the recovery problems that appear in this setting.

1 Introduction

An increasing number of applications demands high availability, often combined with online scalability and soft real-time response. The growing fields of electronic commerce, world-wide businesses and telecommunications all require databases that are highly available, they must be available for all kinds of operations anytime. This means that all maintenance must be done online.

Schema evolution is a necessity in databases due to changing requirements, design faults and the evolutionary nature of various application domains. In a distributed database there are two main categories of schema evolution. *Model evolution* is when the table design changes after the database has been populated. This includes removal or addition of key or non-key fields, change in domain of fields, splitting and unions of tables, etc.

The second category, *declustering evolution*, is due to changes in data volume, processing power or access patterns. In this paper we focus on the second category of schema evolution. However, many of the concepts are the same for these two types of evolution, but the implementation may differ. Model evolution expressed through general relational algebra operations requires general algebra operations on the log, where in the declustering evolution log records are transformed one-to-one.

To scale up the system, new nodes are added to the system, and tables are redistributed to the new nodes while the system is running. This is done by fuzzy copy of the data in parallel with log shipping and redo. Scaling down is done in the same way as scaling up, and spare capacity on nodes may be reused for other purposes. The scaling is done as a transaction so that both node failures and undo of scaling is handled by the system.

The redistribution of data may also be done by changing the distribution key, i.e. the part of the primary key which is used to decide the distribution. Normally, this is the whole primary key, but to cluster related tuples a subset of the primary key may be used.

This paper is organized as follows. We present related work in Section 2. Section 3 presents the architecture and context for the work. The method for scaling of data is presented in Section 4. In Section 5 we measure the impact from online scaling on response time and throughput for concurrent user transactions. Section 6 concludes the paper.

2 Related Work

Online scaling and schema evolution are special cases of database reorganization. Sockut and Iyer [SI00] have made a comprehensive survey and they categor-

**Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001**

ize online database reorganization into the following classes:

- restoring physical arrangement without changing the database definition
- changing the physical definition
- changing the logical definition

Our work mainly falls into the first definition since we move data around in the system only changing the declustering information in the distribution dictionary. However, our work may also be used in the second category since the physical storage is not visible in the method. One constraint to the method presented here is that tuples must be inserted in key order in the copy process.

The method to be presented is similar to the one we use in online repair [BGH+96]. The main differences are that the copy is limited to mirroring in repair, while online scaling involves running nodes with existing data and traffic. Thus, node crash recovery at repair is done by restarting repair. Recovery at scaling must be done as a normal transactional rollback.

Zou and Salzberg present a method for online reorganization of B+-trees [ZS96]. This method combines an in-place reorganization of leaf-level data, with a copying reorganization of non-leaf data. It is intended for reorganizations of very large and sparsely populated trees. The same authors have also done work on updating hard references from index tables to base tables during on-line reorganization [ZS98]. This is done by piggy-backing updates to index tables with user transactions.

Lakhamaraju et al. present an online reorganization in object databases [LRSS00]. They focus on high concurrency during move of objects and update of references from parent objects. At one point in time they lock at most two distinct objects.

Common to the two last approaches is they are not intended to solve problems for fast growing volumes of data and transactions. Lee et al. shows an approach for self-tuning placement of data in a shared-nothing system [LKO+00], where the data is partitioned into ranges. It is a two-tier index approach, where the top tier is small and is cached in main-memory and replicated to all nodes. Data at the leaf tier is stored in B+-trees. Data is migrated in bulks between nodes in the leaf tier. The top tier is updated lazily, and during the move requests may be forwarded from the old to the new location of the data. Since data is moved by bulk loading, it is unclear how recovery and undo of the reorganization is handled.

IBM has done work in online data copy that is similar to our work [SI96]. They permit both reads and writes while the data is reorganized into a shadow copy. This is done by applying the log to the shadow copy in multiple rounds. Writes are prohibited a short period while the last changes are applied to the shadow copy. The main problem in this approach is to maintain the mapping between record identifiers in the two versions. Our approach is designed to decluster data onto many nodes and to move data around as the volume grows. Hence, we do not have the problem of mapping record identifiers since both data and log records are identified by primary key.

3 An Architecture for High Availability

3.1 Shared-nothing architecture

The architecture assumes a shared nothing approach, where the nodes of the system are grouped into sites. A site is a collection of nodes which share environment characteristics, like power supply, maintenance personnel, location, etc. The architecture assumes high bandwidth, low latency networks. It may utilize duplicate networks for fault tolerance purposes. A node is a standard workstation equipped with memory and local disks. There are no shared disks.

3.2 Data declustering

Data is both fragmented and replicated. The fragmentation lets the system be scalable, the replication is done for high availability purposes.

Fragmentation may be done according to hashing or ranges of keys. This is transparent to operations except for the creation of tables, where the number of fragments and fragmentation method are given together with a node group. There are no concept of table spaces, only groups of nodes which store tables. Typically, small tables are stored on few nodes, while large tables are stored on all active nodes. A table may have multiple fragments at each node.

Replication is done for high availability. Analysis has shown that two replicas is ideal [Tor95], which means that one system consists of two sites. More replicas complicate the system and increases the probability for software failures. Multiple replicas are often used in systems with asynchronous replication and looser requirements with respect to transaction consistency.

We use mirrored declustering, where each node at one site has a mirror node at the other site. Typically, each node stores two *fragment replicas* of a table. The mirror nodes divide the primary responsibility between each other, so that one is primary for one fragment and hot standby for the other fragment. Figure 1 illustrates this. In the left part of the figure, node 0 has two fragment replicas, which are mirrored at node 1. Node 0, 2 and 4 belong to the same site, while node 1, 3 and 5 belong to the other site.

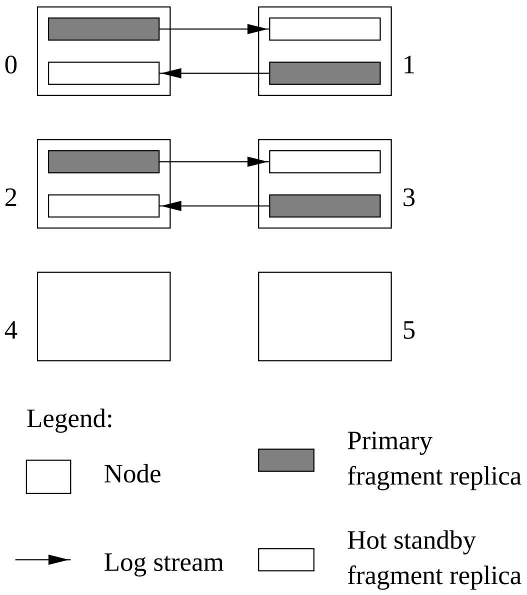

Figure 1: Declustering of data.

3.3 Location and replication independence

Location transparency for data and log records is important. When the volume of transactions or data is changing, we must be able to move data around without taking the system down. To be able to do this, log records must be *logical.* I.e., the reference to data from the log record must be by primary key. The second important concept to use is tuple state identifiers, i.e., the status identifier of the log is attached to tuples instead of blocks. This concept may be seen as an alternative to compensation-based logging to support fine-granularity locking, but is very powerful for log-based replication because it allows for tuple-granularity replication [Hva96]. The combination of these two concepts allows the system to move tuples and their associated log around in the system while the system is running.

The logical logging is implemented by a two-level logging scheme, where logical operations are logged in the tuple log and are replicated between mirror nodes. Node-internal transactions are physical "consequences" of logical operations, like block splitting and creation of files. The node-internal log is written to disk and is not replicated. These two logs are recovered in sequence [BHT97].

3.4 Fault tolerance by recovery and online repair

When a node fails, its mirror node takes over the responsibility for the data residing on the failed node. The failed node recovers and gets updated by receiving new log from its mirror node. This is called *catchup*, which terminates when the catchup log shipping has caught up with the log production.

If a node has been down too long, or it is not able to recover, online repair is used. This includes copying data and log from the node which did the takeover to either a *spare node* or to the node which failed. This copy is done online and non-blocking.

Online repair is important for multiple purposes: as an escalated recovery mechanism, for use by spare nodes, for upgrade of hardware and software, and as a general cleanup mechanism.

3.5 Main software components and their interrelationship

The main software components involved with online scaling are the transaction controller, the kernel, the log channel, and the dictionary cache. The transaction controller coordinates the execution of transactions through a two-phase commit protocol. It consults the dictionary cache to locate the fragment replicas of a given tuple. Due to its frequency of use, dictionary information is cached in main-memory at each node.

For simple operations based on primary keys the transaction controller forwards the user requests directly to the kernels for execution. When a kernel participates in a transaction, it is called a *slave.* Different tuple operations may be forwarded to different kernels and executed in parallel. More advanced queries may be sent to all kernels which involve the SQL executor to do algebra operations. Tuples are stored at the leaf level in B^{link}-trees.

Updates to data are replicated through the log channel, which eagerly reads log records and ships them to the hot standby kernels. In this way the hot standby replicas are kept updated. The transaction controller lets the transactions be executed at the primary fragment replicas and the log records be stored at the hot standby replicas before the slaves are declared to be ready. This is the *neighbor write-ahead logging strategy*, the log must be stored at a hot standby node before the effected data item is allowed written to disk at the primary side. In this way replication is cheap, it is a simple redo of the log that is already there for undo purposes.

Hot standby operations may be executed (redone) after the transaction is decided to commit. Before starting the second phase of the commit, the transaction controller consults its hot standby transaction controller residing at the other site to know whether to commit or not. This is done to register the outcome of the transaction in case the primary transaction controller dies.

Figure 2 illustrates the threads involved in execution of a simple, one-tuple update transaction. The transaction controller at node 4 receives a request to execute a single-tuple update operation. The TrolThread looks up in the dictionary cache to find the primary fragment replica for this tuple. It forwards an

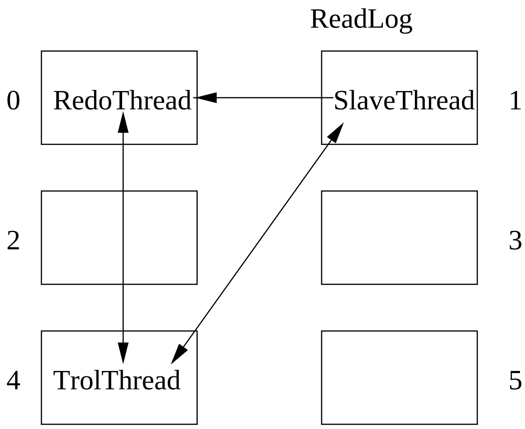

Figure 2: Threads involved in a simple transaction.

update operation to a SlaveThread at node 1, which logs the operation and executes the update. It replies to TrolThread as soon as it is ready with the execution. ReadLog at node 1 ships the log record for redo execution at node 0. When the RedoThread has received the log record, it replies ready to the TrolThread. The redo of the update will happen after the reply.

Due to mirrored declustering and the delayed redo execution at the hot standbys, primary and hot standby kernels should be equal with respect to processing capacity and storage volume.

4 Method for Online Scaling

4.1 The first phase

The transaction controller receives a command to scale a table onto a given node group from an SQL compiler, a management program, or from an api program.

The transaction controller performs the scaling in two phases. The first phase includes a number of steps. It grabs an exclusive lock for the dictionary. This is necessary to serialize the changes to the dictionary cache replicated to all nodes in the system. In the instruction a target node group is given. The standard declustering used by the transaction controller is to have two fragments per node, one primary fragment replica and one hot standby. The command may override this by specifying the number of fragments and replicas. The node group is read from the dictionary tables in the database.

The *target fragment replicas* are created at the nodes in the node group. At every target fragment replica, both primary and hot standby, there will be one *pseudo replica* for each primary *source fragment replica.* A pseudo replica is an entrance for log records to be redone at a target replica. A pseudo replica is *bound* to a target fragment replica, meaning that it is a pseudonym for the target fragment replica. The pseudo replicas are needed to distinguish different log streams into the same target replica, since each primary fragment replica assigns log sequence numbers individually. During the scaling log records are generated at the different source fragment replicas, and they are entered into the targets through a pseudo replica.

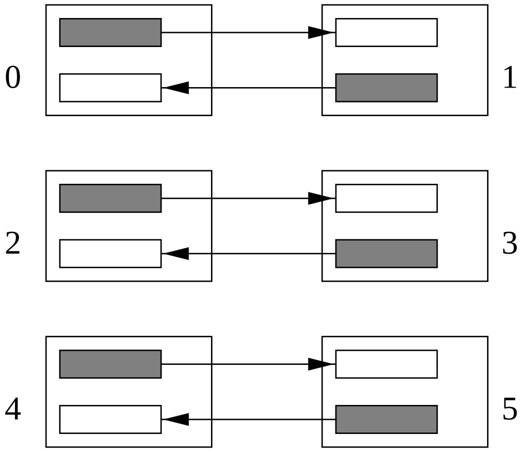

Figure 3: Declustering after scaling.

Figure 3 illustrates the declustering after the scaling. Now, the table is declustered onto all six nodes. Note that the two extra nodes may be acquired and installed when the scaling is necessary.

4.2 Data and log replication

The data and log shipping is started by sending instructions to the primary fragments replicas of the table. All log records of a node reside in a shared log buffer, before they eventually are flushed to disk. Log shipping starts at the oldest alive log record, so that transactions may be rolled back in the new copies after *takeover*, when the scaling transaction commits. The log shipping is done by a separate log shipping thread which reads the log buffer and does a partitioning lookup according to the new declustering. The log shipping from one node to another is done in two separate channels: the realtime channel and the slowlog channel. This means that the scaling may be run on low priority to prevent slowdown of parallel realtime transactions. Two mirror nodes have a bidirectional realtime log channel, while during scaling there will be a slowlog channel from all nodes storing primary fragment replicas of the table, to all node storing fragment replicas in the new declustering.

When the log shipping has started, data copying may start. This enforces the existence of old enough log records at the targets for redo purposes. The copying is done online and non-blocking, allowing parallel updates and inserts to happen. The fragment replicas are read sequentially in primary key order. Only low-level latches are needed during copy of the tuples within a block. Thus, updates may happen while data is read. This is why it is called *fuzzy copy*, it somehow resembles *fuzzy checkpointing.*

One reader puts tuples into multiple data packages according to the partitioning lookup. Each tuple will exist in two data packages, one for the primary and one for hot standby.

A data package includes the following fields:

Pseudo ReplId the identifier of the pseudo replica.

RedoLSN the lowest non-executed log sequence number for this fragment replica. This number is registered when starting to fill the data package.

Highkey the highest key read when sending the package. This is not necessarily the highest key of a tuple in the package, as the tuple with the highest key read may have gone to another pseudo replica.

When tuples in a package are inserted at a pseudo replica, a special redo thread is started to execute log records with higher LSNs than the redo-LSN for records with keys lower than highkey. In order to avoid disturbing realtime redo processing, this is done by a separate redo thread for each pseudo replica. Realtime redo happens eagerly at the front of the log. Thus, resources, like locks, buffers and transaction table slots may be released eagerly.

The tuple status identifier (log sequence number) within each tuple is transferred directly in fuzzy copy. In this way, the redo thread knows whether redo is necessary or not. Normal hot standby redo will always be done, while in fuzzy copy the redo test is necessary since operations may have been performed at the source replica.

The different pseudo replicas are merged at a target replica, i.e., the inserter waits for input on all connected pseudo replicas before starting. This is done to have sequential inserts, which is favorable with respect to fill degree and disk utilization. The reader sends to all pseudo replicas at the same time, even if some of the data packages are not filled. It is important to let the inserter have tuples to merge, and to let the redo thread advance while the log records are in the buffer.

The slaves execute fuzzy copy for a number of fragments in parallel by a parastart/parawait construction. Thus, they wait for completion of a number of parallel fuzzy inserter threads. Fuzzy copy for one target is ready when all tuples have arrived, the redo thread has caught up with the realtime redo thread, and handed over the redo responsibility. When the slaves are through with the fuzzy copy, they reply to the transaction controller.

Figure 4 illustrates the threads involved in the production of one target fragment replica. There is one FuzzyReader at each source primary fragment replica, and one FuzzyInserter at each target fragment replica. Thus, in this example there will be 4 FuzzyReaders and 12 FuzzyInserters. Only 4 of the 48 data streams are shown in the figure. In addition to the fuzzy copy

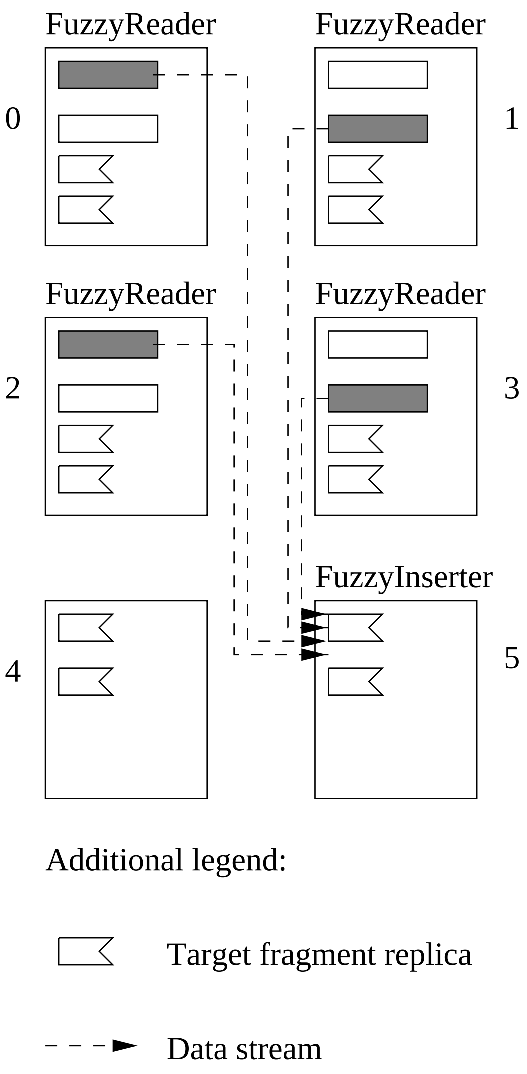

Figure 4: Threads involved in fuzzy copy.

threads, as previously explained, there are threads for log shipping and redo which are present during the scaling. For each node having source fragment replicas, there is one RefragLog thread which ship log records to pseudo replicas. For each pseudo replica, there is one RecoverWindow thread which handles redo. This thread is activated by the receipt of a data package, and is terminated when it has caught up with the corresponding RedoThread after receipt of the last data package.

4.3 The last phase

The second phase is executed when prepare-to-commit in the scaling transaction is encountered at the transaction controller.

The pseudo replicas are unbound from the target replicas. They will not be needed anymore since the log production from now on will happen at the new primary fragment replicas. The old fragments are deleted, and a takeover is issued. Takeover is sent to the old fragment replicas which become locked. When the takeover log records are received at the new fragment replicas, the slaves will be ready and the scaling transaction may commit. After commit, the new fragment replicas are used as primary. Thus, there is a short period at takeover where the fragment replicas are unavailable. This period is usually less than a millisecond.

For takeover to be fast, we use tuple-level locks instead of coarse-granularity locks at the target replicas. When takeover happens, there may be hot standby tuple locks present at the new primary fragment replicas. Hot standby tuple locks are shipped together with log records, thus, becoming effective when log records arrive in pseudo replicas. These locks are released at commit or abort of user transactions, just like ordinary tuple locks.

4.4 Two version dictionary

Parts of the dictionary tables are in frequent use by the transaction controllers. Since we use a peer-to-peer architecture, where transaction controllers may run on all nodes, the distribution dictionary is cached at all nodes. The distribution dictionary describes how tables are fragmented and replicated. Hence, to know where to send tuple operations, the transaction controller does a lookup in the dictionary cache. This means that all transaction controllers can access the distribution dictionary in main memory in parallel.

Transactions updating the distribution dictionary need to be grab the dictionary lock, which blocks the distribution dictionary for other updates. An update to the distribution dictionary is replicated to the dictionary caches at all nodes. During an update of the distribution dictionary, the dictionary cache will exist in two versions. One version is the current version of the dictionary, the other is the one being updated. Only the dictionary change transaction sees the new version during the update. When the dictionary change transactions commits, the new version becomes the current one. However, old transactions which started prior to the commit of the dictionary change transaction, will still use the old dictionary. If an old transaction access the table which just was scaled and committed, they will hit a delete lock at the old fragment replicas and they may abort. If a new dictionary change transaction tries to start when there still are transactions using the oldest dictionary version, the old transactions will be aborted by the transaction controller when accessing the dictionary cache.

4.5 Involving new nodes

When executing transactions the transaction controller informs the involved slaves about the outcome of the two-phase commit. Nodes not in use in a table in the current version of the dictionary will not be informed. During node crash recovery, the recovering node is informed about the outcome of transactions by use of the log channel, i.e. the recovering node receives commit/abort log records for already committed or aborted transactions during catchup. New nodes introduced through the scaling will not be involved in ongoing parallel user transaction, and will not be involved through the log channel. Since commit is logged once for each transaction, using the log channel would require all nodes to ship commit/abort log records to all other nodes. For short transactions this would be a considerable overhead in the system.

To solve this problem, the dictionary cache is informed about the introduction of new nodes as a part of the scaling transaction. The transaction controllers will during the scaling involve new nodes during the second phase of their commit, and the new nodes will receive commit and abort decisions. Note that they will not take part in the first phase of the commit. At commit or abort of the scaling, this “involve new nodes” information is removed from the dictionary caches. This happens when the last transaction using the old version of the dictionary is committed or aborted.

4.6 Distributed resource control

There are a number of fixed-sized resources which are configured at startup of the system, like lock table slots, transaction table slots, fragment replica descriptors, etc. In a distributed system it is not easy to tell in advance that there will be enough resources to execute a given transaction. To distribute local resource information among all nodes is not a scalable approach. Therefore, we exploit the fact that update transactions acquire approximately an equal amount of resources at mirror nodes.

When a transaction is executed at a primary kernel, it is checked whether there are enough resources to execute it. Since hot standby resources may be acquired after the transaction is ready, we reserve some resources for hot standby execution.

If the system is overloaded it is not very wise to abort transactions at the kernel, because rollback creates extra work. A better solution is to reject new transactions at the transaction controller. This may be done based on observed timeouts of recent transactions, on information acquired by the local scheduler or on specific alarms given by the system.

4.7 Recovery issues

Fuzzy copy operations are not logged, but the scaling transaction logs its creation and bind operations. At abort of the scaling the new fragment replicas are undone by rolling back the bind and creation operations. This includes removing the half-populated B-trees.

At abort of user transactions, undo of log records referring to pseudo replicas will after takeover be mapped to the target fragment replicas. The garbage collection of pseudo replica descriptors ensures that they live long enough to cater for this mapping.

To have little impact on the response time of parallel user transactions, log receipt for pseudo replicas is not included in the ready-to-commit criteria. This lets the log shipping needed for scaling be done lazily or at bursts when the system has little to do. This means that log records for pseudo replicas may still arrive when the transaction is committing.

Slaves involved only through scaling, will not participate in the first phase of the commit. Thus, if the scaling transaction aborts, these "pure pseudo touch" transactions will be removed, they do not commit or abort.

When the scaling transaction has committed, parallel user transaction may have log records referring to (pseudo) fragment replicas which are removed. The replica log descriptors are garbage collected according to use in transactions.

Since fuzzy copy does not use logging, we either have to force fuzzily copied to disk as a part of the first phase of commit, or to detect that disk recovery is not possible due to missing data, and therefore escalate to repair. During a checkpoint the status of a fragment replica is logged. If there is any fragment replicas "under repair" in the penultimate or ultimate checkpoint, recovery is not allowed, and the node restarts with repair.

5 Measurements

A central requirement is to let online scaling have little impact on ongoing transactions, with respect to response time and throughput. Since data is not locked during fuzzy copy, the impact is from resource contention on memory, CPU, network, etc. In these measurements we try to capture the effect of scaling on the response times and throughput in a main-memory setting. All data in these measurements reside in main-memory during the copy, both the new and the old copy of the table.

Our main measurement is the *response time increase factor*, i.e., the average response time during scaling divided by the average before scaling. We do the same for throughput – the *throughput decrease factor*. We have also measured the increase factor for the maximum and minimum response time where the maximum and minimum is sampled every second. These measures are the average of the samples during scaling divided by the average before the scaling.

5.1 Parameterization

As shown in Figure 2 a user transaction is executed at the slaves by three threads. Of these three, only SlaveThread and ReadLog have direct effect on the response time, since the hot standby redo execution is only required for the second phase of the commit to terminate.

The runs are done with three different priorities and execution schemes for the seven relevant threads. The scheduler uses three different priorities: *high*, *poll* and *low*. Threads at *high* priority are executed until there is no more to execute. Messages are *poll*ed if there are no active threads at *high* priority. Threads at *low* priority are run when there are no active threads at *high* priority and no messages to *poll*.

Each thread is non-preemptive, thus, they must relinquish the control themselves. The seven thread types in question execute a number of tuples or log records before giving up the control. The test is run with three different numbers of operations before preempting themselves. We have summarized these two parameters in this table:

Thread type	*eager*		*medium*		*lazy*	
SlaveThread	high	20	high	20	high	20
ReadLog	high	20	high	5	high	1
RedoThread	high	10	low	4	low	1
FuzzyReader	high	53	low	25	low	1
FuzzyInserter	high	20	low	5	low	1
RefragLog	high	20	low	5	low	1
RecoverWindow	high	20	low	4	low	1

The idea with these three priority schemes is that *eager* tries to give much priority to fuzzy copy, *lazy* gives priority to user transactions and *medium* is in between these two.

5.2 Numbers

We have used simple clients which either read 1 tuple, update 1 or 4 tuples during the scaling. Each client sends transactions back-to-back, possibly with a wait time. We have run clients with 0 or 50 milliseconds wait time. There is one client connected to each node in the system (six in all).

The table in question consists of 900000 tuples, a total of 115 MB at each site. The clients do random updates or reads to the table, but with separate key ranges, so that conflicts are avoided in these measurements. Each node is single-CPU at 1.1 GHz running Linux, connected through a 100 Mbit switched network. Most results are relative, so that we do not emphasize individual results like response time and throughput. The software used is Clustra Database 4.0 with Clustra C++ API.

The time (seconds) to scale from four to six fragments (and from four to six nodes) is given in the following table:

client	wait time	eager	medium	lazy
no		10.6	23.6	27.4
1-read	0	19.7	58.7	68.5
1-upd	0	53.1	58.2	31.8
4-upd	0	76.4	57.5	34.5
1-read	50	10.6	20.9	24.5
1-upd	50	13.9	23.2	24.6
4-upd	50	17.1	25.0	31.9

Eager scheduling lets parallel user transactions have little impact on the time to scale, except at high update loads. For high update loads, lazy scheduling has least impact on the time to scale. This is explained by the decrease in throughput which hits lazy harder than the others.

When there are no load we observe the scaling time more than doubles at medium or lazy scheduling. This is explained by the fuzzy copy algorithm which is very sensitive to workload at all nodes. Data is merged at each FuzzyInserter, thus insert does not happen before all readers have sent their data packages. Each FuzzyReader does not continue before all inserters have replied. The rationale behind this is to insert sequentially and thus fill blocks. The consequence is that the system at medium and lazy scheduling does a lot of waiting, and there is spare capacity for processing parallel user transactions.

The increase factor for the average response time is given in the following table:

client	wait time	eager	medium	lazy
1-read	0	5.26	2.51	1.98
1-upd	0	4.35	1.52	1.32
4-upd	0	3.39	1.08	0.85
1-read	50	8.02	3.43	3.12
1-upd	50	5.46	2.63	2.47
4-upd	50	4.57	2.26	1.91

Medium scheduling has an increase factor less than 3.5 for all loads, while lazy scheduling has a factor less than 2.5 for updates and less than 3.2 for reads. However, if we measure the increase as a difference instead of as a factor, reads do better than updates. The factor below 1 is due to low throughput, which is caused by the distributed resource control.

The increase factor for the minimum response time is given in the following table:

client	wait time	eager	medium	lazy
1-read	0	1.57	1.48	1.48
1-upd	0	1.56	1.33	1.28
4-upd	0	1.76	1.07	0.89
1-read	50	2.42	2.38	2.29
1-upd	50	1.76	1.76	1.57
4-upd	50	1.73	1.51	1.45

The increase factor for the maximum response time is given in the following table:

client	wait time	eager	medium	lazy
1-read	0	2.54	1.64	1.52
1-upd	0	1.72	1.19	1.04
4-upd	0	1.51	0.94	0.84
1-read	50	11.94	5.96	6.14
1-upd	50	4.00	2.95	3.05
4-upd	50	3.00	2.66	2.20

The decrease factor for the throughput is given in the following table:

client	wait time	eager	medium	lazy
1-read	0	0.40	0.59	0.59
1-upd	0	0.25	0.37	0.16
4-upd	0	0.34	0.36	0.17
1-read	50	1.00	0.99	1.00
1-upd	50	0.95	1.00	1.00
4-upd	50	0.96	1.00	0.98

When transactions are run back-to-back with zero wait time, the throughput is highly affected. At worst the system is able to cope with only 16% of the throughput compared with prior to the scaling. At low loads the scaling hardly is noticeable on the throughput. The conclusion is that online scaling highly affects the throughput if the database is already loaded. Do the scaling before this happens.

For eager scheduling the decrease in throughput is explained by the characteristics of the scheduler. The scheduler polls for messages only when it has no high priority jobs to do. The throughput is reduced because the transaction controller polls for messages less frequently, so that it acquires new work less frequently as well.

For the other scheduling schemes, when the throughput is seriously damaged, the response time is not. This is explained by the scheduler-based resource control which hits high update loads. This lets transactions within the system have priority over new transactions arriving when the scheduler is busy. When the transaction controller rejects a client request due to resource control, the load is somewhat controlled by the retry interval of the client. In our measurement we used 50 milliseconds as the retry interval.

Figure 5 shows plots of the average response time for user transactions from one transaction controller during the scaling. The plots show this for eager, medium and lazy scheduling for one-tuple reads with no wait time. The raised parts of the plots are when the scaling is done, starting at 40 seconds.

Figure 6 shows the same for 1 tuple update transactions. In this situation the checkpointing may be seen on the average response time by regular bumps.

6 Conclusions

Online scaling degrades the response time and throughput of transactions during the reorganization. We have measured this impact on simple user transactions for different scheduling schemes. Lazy scheduling gives the best overall performance with respect to response time. For throughput the impact is seen only at high loads.

It is important that the scaling goes through and does not abort due to too few resources. Distributed resource control is important to let the system handle both user transactions and the scaling. A highly available system must be designed to have spare processing capacity, and online scaling is probably not performed at peak load periods, and should be done before the system is saturated with respect to processing power.

Through our measurements we have seen that the copying of data is very load sensitive. To improve this we could limit the declustering used in the scaling, so that we do not get all-to-all dependencies. E.g., by splitting existing fragments into two new fragments, the number of dependent nodes for each reader is limited to two. We expect this to become more important with an increased number of nodes.

An idea to reduce the impact is to reduce the fault tolerance level during the scaling by discarding one of the old replicas while producing a new modified replica. Alternatively, one could produce a single new copy and after the takeover start producing a new, second copy. However, this reduces the fault tolerance level during a vulnerable period. Lazy scheduling of fuzzy copy also reduces the impact, but retains the fault tolerance level.

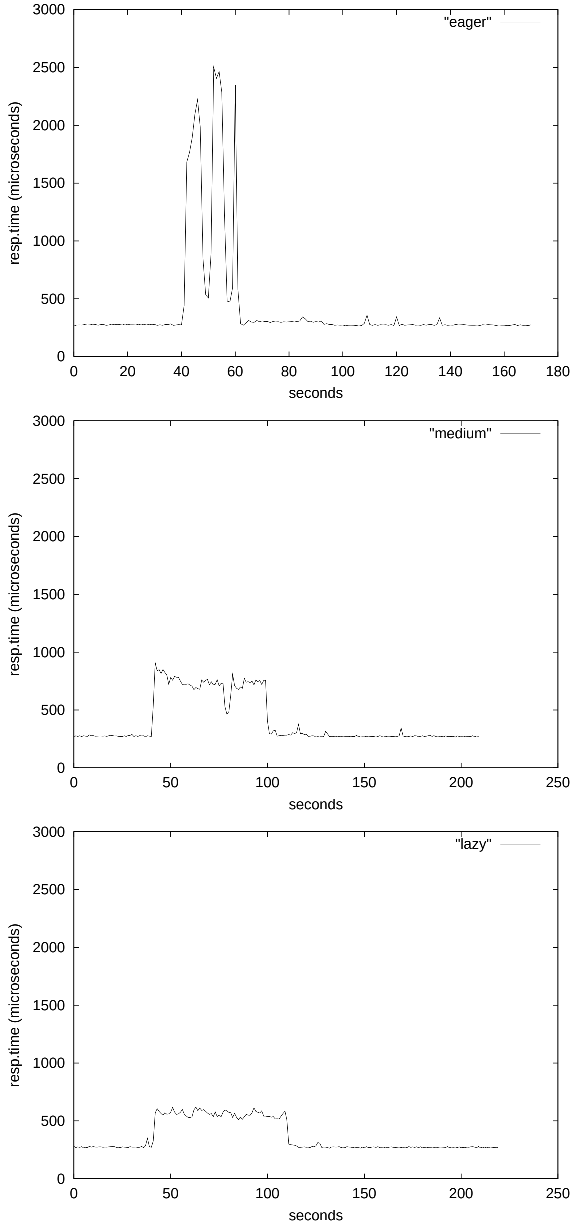

Figure 5: Eager, medium and lazy 1read, 0 wait time.

References

[BGH+96] Svein Erik Bratsberg, Øystein Grøvlen, Svein-Olaf Hvasshovd, Bjørn P. Munch, and Øystein Torbjørnsen. Providing a highly available database by replication and online self-repair. *International Journal of Engineering Intelligent Systems for Electrical Engineering and Communications, Special issue on Data-*

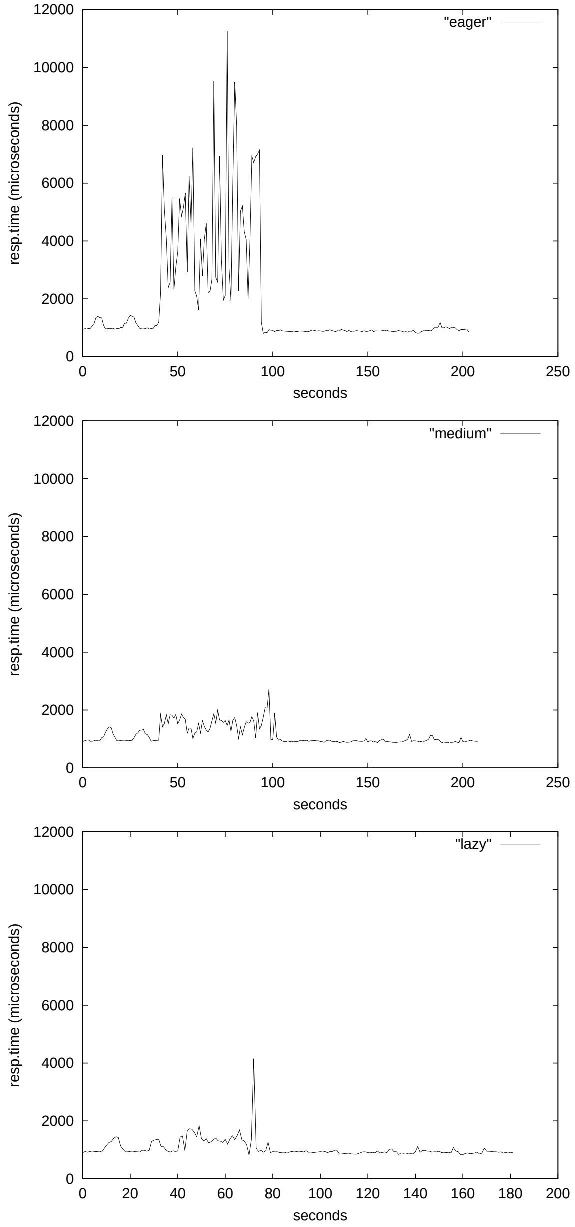

Figure 6: Eager, medium and lazy 1upd, 0 wait time.

bases and Telecommunications, 4(3):131–139, September 1996.

[BHT97] Svein Erik Bratsberg, Svein-Olaf Hvasshovd, and Øystein Torbjørnsen. Location and replication independent recovery in a highly available database. In *Advances in Databases, 15th British National Conference on Databases*, pages 23–37. Springer-Verlag LNCS 1271, July 1997.

[Hva96] Svein-Olaf Hvasshovd. *Recovery in Parallel Database Systems*. Verlag Vieweg, Wiesbaden, Germany, 1996.

[LKO+00] Mong Li Lee, Masaru Kitsuregawa, Beng Chin Ooi, Kian-Lee Tan, and Anirban Mondal. Towards self-tuning data placement in parallel database systems. In *Proceedings of ACM/SIGMOD (Management of Data)*, pages 225–236, May 2000.

[LRSS00] Mohana H. Lakhamraju, Rajeev Rastogi, S. Seshadri, and S. Sudarshan. On-line reorganization in object databases. In *Proceedings of ACM/SIGMOD (Management of Data)*, pages 58–69, May 2000.

[SI96] Gary H. Sockut and Balakrishna R. Iyer. A survey of online reorganization in IBM products and research. *IEEE Data Engineering Bulletin*, 19(2):4–11, 1996.

[SI00] Gary H. Sockut and Balakrishna R. Iyer. Online reorganization of databases, June 2000. Available from authors: IBM Silicon Valley Laboratory, 555 Bailey Ave, San Jose, CA 95141, USA.

[Tor95] Øystein Torbjørnsen. *Multi-Site Declustering Strategies for Very High Database Service Availability*. PhD thesis, The Norwegian Institute of Technology, University of Trondheim, January 1995. 186 p., ISBN 82-7119-759-2.

[ZS96] Chendong Zou and Betty Salzberg. On-line reorganization of sparsely-populated b+-trees. In *Proceedings of ACM/SIGMOD (Management of Data)*, pages 115–124, June 1996.

[ZS98] Chendong Zou and Betty Salzberg. Safely and efficiently updating references during on-line reorganization. In *Proceedings of the 24th International Conference on Very Large Databases, New York City, New York (VLDB '98)*, pages 512–522, September 1998.