CARL VON OSSIETZKY universität OLDENBURG

Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik
Systemsoftware und verteilte Systeme

Diplomarbeit

Optimale Replikation

vorgelegt von

Christian Hülsen
Matrikelnummer: 9182860

Gutachter:

Prof. Dr.-Ing. Oliver Theel
Dipl.-Inform. Eike Möhlmann

14. November 2012

Zusammenfassung

Replikationsstrategien werden in vielen Anwendungsfällen eingesetzt. Die Zahl der verschiedenen Strategien ist mindestens ebenso groß. Dabei unterscheiden sich die Strategien teilweise sehr stark und sind für unterschiedliche Anwendungsfälle optimiert. Diese Arbeit beschäftigt sich damit, für jeden Anwendungsfall die beste Instanz einer Replikationsstrategie aus einer Liste vorhandener Replikationsstrategien zu ermitteln.

Da es für Replikationsstrategien keine einfach Möglichkeit des Vergleichs gibt, kommt in dieser Arbeit die Brute-Force-Methode zum Einsatz. Das bedeutet, dass für jede sinnvolle Instanz der Replikationsstrategien aus der Liste die Kosten und Verfügbarkeiten für den Anwendungsfall berechnet werden. Der Anwendungsfall gibt dabei unter anderem die Anzahl der Replikate und die Intaktwahrscheinlichkeit der Rechner an. Auch wenn dieses Vorgehen wenig Effizient erscheint, ist es die momentan beste Möglichkeit, Replikationsstrategien für wechselnde Anwendungsfälle wirksam zu bewerten.

Zu diesem Zweck wurde eine Software geschrieben. Diese Software enthält die Berechnungsfunktionen der Replikationsstrategien und nimmt als Eingabe die Daten des Anwendungsfalls entgegen. Anschließend wird die beste, bekannte Instanz einer Replikationsstrategie ermittelt. Zudem werden die Ergebnisse in einer Datenbank gespeichert um doppelte Berechnungen zu vermeiden.

Die Software bedient verschiedene Einsatzgebiete. Sie soll eigenständig genutzt werden können oder Ein- und Ausgaben über eine Webseite verarbeiten. Zudem soll auch die Möglichkeit bestehen, die Software in anderen Programmen einzubinden. Damit bietet sich die Möglichkeit auf Basis der Software eine dynamische Replikationsstrategie zu entwerfen, die für jede mögliche Situation die beste statische Replikationsstrategie anwendet.

Inhalt

1 Einleitung

Replikationsstrategien werden eingesetzt, um die Erreichbarkeit von Daten zu erhöhen. Diese Daten können zum Beispiel eine Datei oder einen Anwendung sein. Die Erreichbarkeit von nicht replizierten Daten kann durch Rechner- oder Leitungsausfälle, Netzwerkpartitionierung oder ein erhöhtes Aufkommen von Abfragen sinken. Replikationsstrategien wirken dem entgegen, indem die Replikate auf verschiedene Rechner verteilt werden. Dadurch steigt allerdings auch der Aufwand der Verwaltung und das Aktualisieren der Replikate. Einzelne Replikationsstrategien unterscheiden sich in der Art, wie Replikate gelesen und geschrieben werden, teilweise drastisch. Einige Replikationsstrategien sind beispielsweise nicht sicher gegen Netzwerkpartitionierung, andere verursachen unverhältnismäßig hohe Kosten durch Rechnerzugriffe.

Diese Arbeit beschäftigt sich mit dem Problem, für einen beliebigen, vorgegebenen Anwendungsfall die optimale Instanz einer Replikationsstrategie aus einer Menge vorgegebener Replikationsstrategien zu finden. Dazu müssen Strategien aus einer Liste mit dem vom Anwendungsfall gegebenen Voraussetzungen, wie Art und Größe des Netzwerks und der Intaktwahrscheinlichkeiten der Rechner, einzeln getestet werden. Diese Tests liefern für jede sinnvolle Instanz der bekannten Replikationsstrategien die Kosten sowie die Schreib- und Leseverfügbarkeit. Diese Werte werden dann miteinander verglichen, so dass die Replikationsstrategien bestimmt werden, die den vom Anwendungsfall gewünschten Werten am ehesten entspricht.

Nach dieser Einleitung werden in Kapitel 2 Replikationsstrategien genauer beschrieben und die für diese Arbeit umgesetzten Replikationsstrategien erklärt. Kapitel 3 geht auf die Anforderungen der Software ein, die im Zuge dieser Arbeit erstellt wird. Im folgenden Kapitel 4 wird der Entwurf der Software und der Datenbank erklärt. Kapitel 5 beschreibt die Umsetzung des Entwurfs. Das Kapitel 6 beschreibt anhand eines Beispiels, wie die Software zu Bedienen ist und welche Performanz sie bietet. Kapitel 7 beinhaltet ein abschließendes Fazit mit einem Ausblick auf weitere Möglichkeiten der Nutzung der Software. Das Handbuch in Kapitel 8 erklärt, was nötig ist, um die Software einzusetzen.

2 Replikationsstrategien

Dieses Kapitel erklärt, was Replikationsstrategien sind und wozu sie eingesetzt werden. Danach werden bestimmte Replikationsstrategien genauer betrachtet, für die im Zuge dieser Arbeit die Methoden zur Berechnung der Kosten und Verfügbarkeiten erstellt werden. Sie bilden den Grundstock der Software, der später beliebig erweitert werden kann.

2.1 Allgemeines

Die Replikation von Daten, also beispielsweise die Speicherung einer Datei auf mehreren Rechnern, wird unter anderem eingesetzt, um die Ausfallsicherheit zu erhöhen oder die Last der Anfragen zu verteilen [The93b] [The93a]. Die größte Herausforderung ist die Sicherstellung der Korrektheit der Strategie. Das bedeutet, dass beim Lesen immer die aktuelle Version des Replikates gelesen werden muss. Dafür müssen sich die Mengen der gelesenen und geschriebenen Replikate überschneiden. Die Replikate werden mit einer Versionsnummer versehen, um das aktuelle Replikat in der Lesemenge zu erkennen. Um dieses zu gewährleisten, gibt es unterschiedliche Replikationsstrategien, die sich hauptsächlich darin unterscheiden, wie ein Replikat geschrieben und gelesen wird [FKT91]. Auch sind nicht alle Strategien gegen Partitionierung gefeit.

Jeder Rechner, auf dem ein Replikat gespeichert wird, hat eine Intaktwahrscheinlichkeit, also eine Wahrscheinlichkeit, mit der der Rechner korrekt arbeitet. Dabei wird angenommen, dass Rechnerausfälle nicht voneinander abhängig sind und Kommunikationsverbindungen nicht ausfallen. Die Rechner sollen untereinander vollständig vernetzt sein. Diese Annahmen werden in der Realität selten erreicht, sie ermöglichen uns aber, die Verfügbarkeiten von Lese- und Schreiboperationen einer Strategie zu errechnen.

Weiter können für jede Replikationsstrategie die Kosten für Lese- und Schreiboperationen ermittelt werden. Diese Kosten geben an, wieviele Replikate geschrieben beziehungsweise gelesen werden müssen, um sicherzustellen, dass beim Lesen eines Replikates immer die aktuelle Version gelesen werden kann. Die Gewichtung dieser Kosten hängt vom Einsatzgebiet der Daten ab. Werden die Daten hauptsächlich gelesen und selten geändert, so sollten die Lesekosten möglichst gering sein, die Schreibkosten können jedoch hoch sein. Das angestrebte Verhältnis von Lese- und Schreibkosten ergibt sich also aus dem Anwendungsfall. Damit soll erreicht werden, dass die durchschnittlich verursachten Kosten für den Betrieb möglichst gering sind.

2.1.1 Statische Replikationsstrategien

Alle in dieser Arbeit betrachteten Replikationsstrategien sind sogenannte statische Replikationsstrategien. Das bedeutet, sie ändern ihre Vorgehensweise während des Betriebs nicht mehr. Das hat den Vorteil, dass sich die Kosten und Verfügbarkeiten im Voraus berechnen lassen und sich nicht ändern.

Jede laufende Instanz einer Replikationsstrategie ist auf die Anzahl der Replikate festgelegt, da die Anzahl das Vorgehen bei Schreib- und Leseoperationen bestimmt. Eine nachträgliche Änderung ist nicht möglich, ohne die Strategie neu zu konfigurieren.

2.1.2 Dynamische Replikationsstrategien

Dynamische Replikationsstrategien basieren auf statischen Replikationsstrategien, die je nach Bedarf eingesetzt und getauscht werden können [Sto12]. Ändert sich ein Parameter, etwa durch das Hinzukommen oder das dauerhafte Ausfallen eines oder mehrerer Replikate, so kann bei dynamischen Replikationsstrategien einfach die zugrunde liegende statische Replikationsstrategie ausgetauscht werden, sollte es für die neue Situation eine bessere statische Replikationsstrategie geben. Eine dynamische Replikationsstrategie passt sich somit immer der aktuellen Situation an und minimiert die Kosten und maximiert die Verfügbarkeiten.

2.1.3 Vergleich von Replikationsstrategien

Um Replikationsstrategien zu Vergleichen, betrachten wir die Verfügbarkeiten und die Kosten. Die Verfügbarkeit gibt die Wahrscheinlichkeit an, dass eine Lese- oder Schreiboperation ausgeführt werden kann. Die Kosten beziffern die Zugriffe auf Replikate, die eine Lese- oder Schreiboperation verursacht.

2.2 Spezifikation des Szenarios

Ein Szenario beschreibt die Rahmenbedingungen eines Anwendungsfalls. Die Daten des Szenarios werden vom Nutzer eingegeben und ändern sich nicht mehr.

Es wird zwischen Daten unterschieden, die für die Berechnungen wichtig sind und Daten, die für die Auswertung wichtig sind.

Für die Berechnung werden die Anzahl der Replikate und die Intaktwahrscheinlichkeit der Rechner benötigt. Diese beiden Werte liefern alle wichtigen Informationen über das (Teil-)Netzwerk, in dem die Replikationsstrategie eingesetzt werden soll. Mit ihnen lassen sich die Kosten für einen Lesezugriff und einen Schreibzugriff sowie die Verfügbarkeiten der Lese- und Schreiboperationen errechnen.

Anzahl der Replikate	N
Intaktwahrscheinlichkeit	p

Durch multiplizieren der erwarteten Zugriffen pro Stunde mit den Kosten für einen Lese- oder Schreibzugriff können die Replikationsstrategien bezüglich der verursachten Kosten pro Stunde verglichen werden. Diese Daten sind für die Berechnungen uninteressant, allerdings besteht die Möglichkeit, dass für ein Szenario mit gleicher Anzahl von Replikaten und gleicher Intaktwahrscheinlichkeit der Rechner verschiedene Replikationsstrategien sinnvoll sein können, wenn sich die Anzahl und Verteilung der Zugriffe deutlich ändert.

Erwartete Lesezugriffe pro Stunde	QR
Erwartete Schreibzugriffe pro Stunde	QW

Um die so ermittelte Werte der Replikationsstrategien auszuwerten, sind diese Maximal- und Minimalwerte erforderlich. Alle Instanzen von Replikationsstrategien, die diese Werte nicht erfüllen,

werden aussortiert. Die Reihenfolge gibt dabei an, in welcher Reihenfolge der Maximal- und Minimalwerte die verbliebenen Instanzen sortiert werden sollen.

Maximale Kosten pro Stunde	$maxC$
Minimale Leseverfügbarkeit	$minAW$
Minimale Schreibverfügbarkeit	$minAR$
Reihenfolge	

2.3 Betrachtete Replikationsstrategien

Für diese Arbeit werden drei Strategien genauer betrachtet. Die Formeln sind, soweit nicht anders angegeben, aus „Entwurf und Bewertung von Replikationsverfahren“ [Koc94] entnommen.

2.3.1 Read-One-Write-All-Strategie

Wie der Name schon sagt genügt, es bei der Read-One-Write-All-Strategie [BG84], ein einzelnes Replikat zu lesen. Da bei einer Änderung alle Replikate geschrieben werden müssen, ist sichergestellt, dass das gelesene Replikat aktuell ist. Dadurch sind die Lesekosten so gering wie überhaupt möglich:

$$cr = 1$$

Im Gegenzug sind die Schreibkosten maximal hoch:

$$cw = N$$

Die Leseverfügbarkeit berechnet sich aus der Intaktwahrscheinlichkeit der Replikate wie folgt:

$$ar(p) = 1 - (1 - p)^N$$

Für eine Schreiboperation müssen alle Replikate erreichbar sein, ansonsten schlägt sie fehl. Dieser Umstand spiegelt sich auch in der Schreibverfügbarkeit wieder:

$$aw(p) = p^N$$

Daraus ergibt sich, dass die Leseverfügbarkeit mit der Anzahl der Replikate steigt, während die Schreibverfügbarkeit sinkt. Das macht die Strategie höchstens für Szenarien interessant, bei denen fast ausschließlich gelesen wird. Bei Szenarien mit vielen Replikaten ist diese Strategie aber ungeeignet.

2.3.2 Weighted-Voting-Strategie

Die Weighted-Voting-Strategie [Gif79] basiert auf Lesequoren (LQ) und Schreibquoren (SQ). Dazu wird jedem Replikat eine Anzahl an Stimmen gegeben. Die Menge an Replikaten, die eine Lese- oder Schreiboperation anspricht, muss mindestens eine vorgegebene Anzahl an Stimmen haben. Für die Quoren gelten folgende Regeln, die die Korrektheit sicherstellen, wobei V die Summe der Stimmen aller Replikate ist:

$$LQ + SQ > V$$
$$2 * SQ > V$$

Diese Bedingungen stellen sicher, dass sich Lese- und Schreibquoren sowie zwei Schreibquoren in mindestens einem Replikat überschneiden. Der Zweck der Stimmen ist es, Rechnern mit einer höheren Intaktwahrscheinlichkeit oder einer besseren Anbindung eine höhere Anzahl an Stimmen zu geben, damit diese eher angesprochen werden. Für die Berechnung der Kosten und Verfügbarkeiten wird angenommen, dass jedes Replikat die gleiche Anzahl an Stimmen, nämlich genau eine, hat. Dieses deckt sich mit der Grundannahme, dass alle Rechner die gleiche Intaktwahrscheinlichkeit haben und vollständig vernetzt sind. Damit ergeben sich die Kosten aus den Quorengrößen:

$$cr = LQ$$
$$cw = SQ$$

Die Leseverfügbarkeit errechnet sich wie folgt:

$$ar(p) = \sum_{k=LQ}^{N} \left(\begin{array}{c} N \\ k \end{array} \right) p^k (1-p)^{N-k}$$

Die Schreibverfügbarkeit ergibt sich aus:

$$ar(p) = \sum_{k=SQ}^{N} \left(\begin{array}{c} N \\ k \end{array} \right) p^k (1-p)^{N-k}$$

Die Schreibverfügbarkeit steigt mit der Annäherung des Schreibquorums an das Lesequorum. Damit steigen aber auch die Kosten für einen Lesezugriff.

2.3.3 Grid-Strategie

Die Grid-Strategie [CAA92] ordnet die Replikate in einem Gitter an, in dem jedes Replikat mit allen Replikaten der Nachbarspalten verbunden ist.

Eine Leseoperation sperrt in jeder Spalte ein Replikat. Eine Schreiboperation sperrt eine komplette Spalte und ein Replikat aus jeder anderen Spalte. Bei s Replikaten in einer Spalte und r Replikaten in einer Zeile ergeben sich die Kosten für die Leseoperation:

$$cr = r$$

Für die Schreiboperation ergeben sich Kosten von:

$$cw = r + s - 1$$

Für ein vollständig gefülltes Gitter berechnet sich die Leseverfügbarkeit aus der Wahrscheinlichkeit, dass es keine Spalte gibt, in der alle Replikate ausgefallen sind:

$$ar(p) = [1 - (1-p)^s]^r$$

Für die Schreibverfügbarkeit muss zusätzlich die Wahrscheinlichkeit einer komplett verfügbaren Spalte betrachtet werden. So ergibt sich:

$$aw(p) = [1 - (1-p)^s]^r - [1 - p^s - (1-p)^s]^r$$

Die besten Ergebnisse liefern quadratische Gitter. Darum werden auch, wenn möglich, nahezu quadratische Gitter betrachtet. Ist ein quadratisches Gitter nicht möglich, wie beispielsweise bei 12 Replikaten, so werden beide Ausrichtungen des Gitters berechnet. Bei 12 Replikaten also 4x3 und 3x4 Replikate.

Da es aber selten möglich ist, ein Gitter vollständig zu füllen, müssen die Formeln entsprechend angepasst werden. Es wird angenommen, dass das Gitter immer von oben nach unten und von links nach rechts gefüllt wird. Das bedeutet, dass freie Knoten in der untersten Zeile von rechts beginnen.

Für die Lesekosten brauchen die freien Knoten nicht beachtet zu werden. Die Schreibkosten reduzieren sich um einen, wenn die komplett gesperrte Spalte im Bereich der freigelassenen Knoten liegt. Da aber nicht vorhergesagt werden kann, welche Spalte komplett gesperrt wird, werden hier die mittleren Kosten berechnet. Die Anzahl der vollständigen Spalten sei j.

$$cw = r + s - 2 + (j/N)$$

Für die Verfügbarkeiten sind die freien Knoten ebenfalls wichtig. Bei der Leseverfügbarkeit ändert sich bei den Spalten mit fehlendem Knoten die Wahrscheinlichkeit, dass alle Replikate dieser Spalten nicht erreichbar sind. In der originalen Formel wird dieser Fall mit $[1 - (1-p)^s]$ für jede Spalte berechnet. Diese Werte werden dann miteinander multipliziert.

$$ar(p) = [1 - (1-p)^{s1}] * [1 - (1-p)^{s2}] * ... * [1 - (1-p)^{sr}])$$

Da die originale Formel auf vollständigen Gitter aufbaut, ergibt sich damit $[1 - (1-p)^s]^r$. Ist das Gitter nicht vollständig, wird es in zwei neue, vollständige Gitter geteilt. Abbildung 2.1 verdeutlicht dieses.

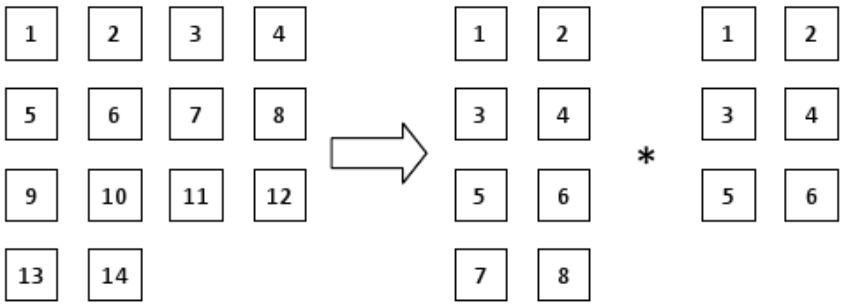


Abbildung 2.1: Aufteilung eines unvollständigen Gitters

Mit rl und sl für das linke Gitter und rr und sr für das rechte Gitter ergibt sich somit:

$$ar(p) = ([1 - (1-p)^{sl}]^{rl}) * ([1 - (1-p)^{sr}]^{rr})$$

Die Schreibverfügbarkeit berechnet sich aus der Wahrscheinlichkeit, dass in jeder Spalte mindestens ein Replikat erreichbar ist, minus der Wahrscheinlichkeit, dass mindestens eine Spalte komplett erreichbar ist. Der erste Term deckt sich mit der Leseverfügbarkeit. Der zweite Term kann auch ausgedrückt werden als: Die Wahrscheinlichkeit, dass es keine Spalte gibt, die komplett erreichbar ist. Da im ersten Term schon alle Spalten mit einem vorhandenen Replikat abgedeckt wurden, dürfen diese allerdings nicht doppelt betrachtet werden. Daraus ergibt sich dann die Wahrscheinlichkeit, dass es keine Spalte gibt, in der nicht alle und nicht ein Replikat erreichbar sind. Oder als Formel für eine Spalte ausgedrückt $[1 - p^s - (1 - p)^s]$. Diese Wahrscheinlichkeit muss von dem ersten Term abgezogen werden.

Abgeleitet von der originalen Formel für vollständige Gitter berechnet sich die Schreibverfügbarkeit wie folgt:

$$aw(p) = ([1 - (1 - p)^{sl}]^{rl}) * ([1 - (1 - p)^{sr}]^{rr}) - ([1 - p^{sl} - (1 - p)^{sl}]^{rl} * [1 - p^{sr} - (1 - p)^{sr}]^{rr})$$

Da für das Schreiben immer eine komplette Spalte gesperrt werden muss, nimmt die Schreibverfügbarkeit mit der Größe des Gitters stark ab. Die Kosten dieser Strategie sind allerdings sowohl für die Lese- wie auch für die Schreiboperation sehr gering.

3 Anforderungen

3.1 Funktionale Anforderungen

Die Software besitzt eine Vielzahl von funktionalen Anforderungen, auf die hier genauer eingegangen werden soll.

3.1.1 Eingabe der Szenariodaten

Die Daten eines Szenarios werden über eine Internetseite vom Nutzer eingegeben. Hierbei müssen alle notwendigen Daten abgefragt und eingetragen werden, die für die Berechnungen der Kosten und Verfügbarkeiten sowie die Sortierung und Gewichtung der Instanzen der Replikationsstrategien wichtig sind. Diese sind im Einzelnen:

- `N` - Die Anzahl der Replikate.
- `p` - Die Intaktwahrscheinlichkeit der Rechner, wobei jeder Rechner die gleiche Intaktwahrscheinlichkeit hat.
- `QR` - Die erwartete Anzahl der Lesezugriffe pro Stunde.
- `QW` - Die erwartete Anzahl der Schreibzugriffe pro Stunde.
- `maxC` - Die gewünschten maximalen Kosten.
- `minAR` - Die gewünschte minimale Leseverfügbarkeit.
- `minAW` - Die gewünschte minimale Schreibverfügbarkeit.
- Die Gewichtung von Kosten und Verfügbarkeiten, falls keine Strategie gefunden wird, die genau den gewünschten Werten entspricht.

Diese Daten werden als Szenario bezeichnet.

3.1.2 Replikationsstrategien

Für jede Replikationsstrategie werden Funktionen bereitgestellt, mit denen die Kosten und die Verfügbarkeit von Schreibe- und Lesezugriffen errechnet werden können. Sind mit der vorgegebenen Anzahl an Replikaten mehrere Instanzen einer Replikationsstrategie möglich, beispielsweise verschiedene Größen des Gitters einer Grid-Strategie, so werden alle diese Möglichkeiten berechnet.

Ein nachträgliches Einführen neuer Replikationsstrategien soll durch die Implementierung eines Interface einfach möglich sein.

Bei schon vorhandenen Szenarien wird für jede Replikationsstrategie geprüft, ob Ergebnisse für dieses Szenario vorhanden sind. Dadurch wird eine erneute Berechnung der Kosten und Verfügbarkeiten vermieden und die Laufzeit deutlich verringert.

3.1.3 Datenbank

In der Datenbank werden die schon berechneten Szenarien sowie die Ergebnisse der Berechnungen pro Replikationsstrategie gespeichert. Dadurch sollen Mehrfachberechnungen der selben Grundwerte vermieden werden.

Von einem Szenario werden lediglich die Anzahl der Replikate und die Intaktwahrscheinlichkeit der Rechner gespeichert, da diese Werte die rechenintensivsten sind. So können zwei Szenarien mit gleicher Anzahl der Replikate und gleicher Intaktwahrscheinlichkeit der Rechner, unterschiedliche viele Zugriff haben, ohne dass die Kosten und Verfügbarkeiten erneut berechnet werden müssen.

3.2 Nichtfunktionale Anforderungen

Folgende nichtfunktionale Anforderungen werden an die Software gestellt.

3.2.1 Bedienbarkeit

Die Eingabe erfolgt über eine Internetseite. Die Seiten für Eingabe und Ausgabe sollen verständlich und intuitiv bedienbar sein.

3.2.2 Zuverlässigkeit

Das System wird während der Berechnung stabil laufen.

3.2.3 Leistung

Das System muss mit einer großen Anzahl an Replikationsstrategien arbeiten können.

3.2.4 Implementierung

Der Entwickler schreibt den Quelltext nach den Richtlinien der C/C++ Coding Conventions. Kommentare werden in englischer Sprache erstellt. Auftretende Ausnahmebedingungen werden abgefangen und gesondert bearbeitet, so dass dem Benutzer keine undurchsichtigen Fehlermeldungen oder Programmabstürze angezeigt werden.

3.2.5 Erweiterbarkeit/Modularisierung

Die Software wird modular entwickelt. Das heißt, dass jedes Teilsystem ein eigenes Modul bildet, welches einfach austauschbar ist. Eine Interface-Klasse ermöglicht eine einfache Einführung neuer Replikationsstrategien.

3.2.6 Schnittstellen

Durch den modularen Aufbau des Systems werden verschiedene Schnittstellen zur leichteren Erweiterbarkeit angeboten.

3.2.7 Unterstützung

Die Struktur des Systems wird modular gehalten, um die Möglichkeit der Erweiterung durch andere Projekte zu gewährleisten.

3.2.8 Rechtliches

Der Quelltext unterliegt dem Urheberrecht und ist Eigentum des Entwicklers und der Universität Oldenburg.

4 Architekturentwurf

Als erstes soll der Ablauf der Software anhand von drei Sequenzdiagrammen deutlich gemacht werden. Danach wird genauer auf die Klassen eingegangen.

4.1 Konzept

Der Ablauf einer Anfrage wird durch das Sequenzdiagramm in Abbildung 4.1 ersichtlich.

Die Daten eines Szenarios werden durch die Klasse `Input` empfangen und eine Instanz von `Szenario` erstellt, die diese Daten speichert. Die Instanz von `Szenario` wird an die Klasse `Manager` übergeben. Hier geschieht die Abfrage, ob ein Szenario mit identischen Daten bereits in der Datenbank existiert. Danach erfolgt die Berechnung der Kosten und Verfügbarkeiten, und die Auswertung und Aufbereitung der Ergebnisse. Die Ergebnisse werden dann an die Klasse `Output` übergeben. Diese Klasse generiert die Ausgaben, die an das PHP-Script zurückgegeben und angezeigt werden.

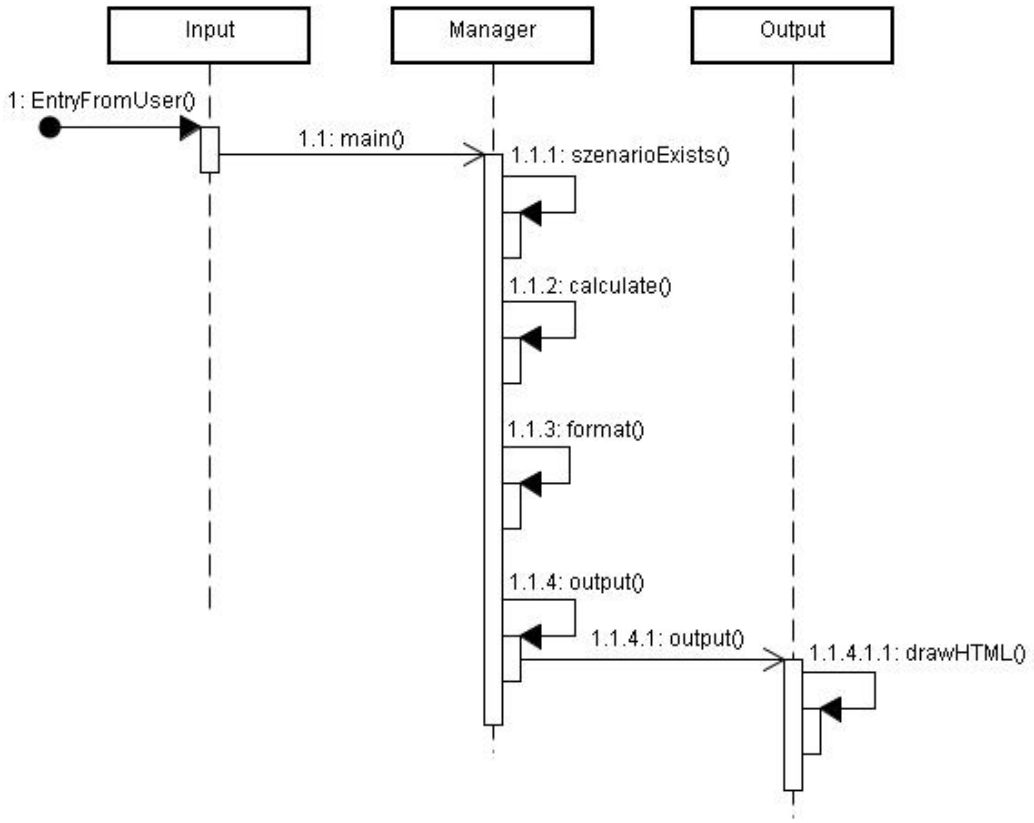


Abbildung 4.1: Ablauf der Software

In den folgenden Sequenzdiagrammen wird genauer auf den Ablauf der Berechnungen von Kosten und Verfügbarkeiten eingegangen. Die Eingabe und Ausgabe wird nicht genauer betrachtet.

Das Vorgehen bei einem neuen Szenario ist in dem Sequenzdiagramm in Abbildung 4.2 zu sehen. Zuerst wird überprüft, ob es in der Datenbank einen Eintrag mit identischen Werten für die Anzahl

der Replikate und die intaktwahrscheinlichkeit der Rechner wie im neuen Szenario gibt. Ist dieses nicht der Fall, wird das neue Szenario in der Datenbank gespeichert und die ID zurückgegeben.

Jetzt werden für jede Replikationsstrategie die Kosten und Verfügbarkeiten berechnet und in der Datenbank gespeichert. Durch die Klasse `ListOfReplikationStrategies` werden alle vorhandenen Replikationsstrategien angesprochen. Dabei können für eine Strategie mehrere Ergebnisse möglich sein.

Wenn alle Replikationsstrategien ihre Ergebnisse berechnet haben, dann erstellt die Klasse `ListOfReplikationStrategies` ein SQL-Statement, mit dem die Ergebnisse von der Datenbank sortiert werden. Die Ergebnistabelle wird in einem zweidimensionalen Vector gespeichert. Dieses geschieht, weil jede Strategie unterschiedliche sekundäre Parameter besitzt: Eine Gitterstruktur beispielsweise hat zusätzlich die Anzahl an Reihen und Spalten gespeichert. Diese sekundären Parameter werden jetzt durch die Implementierung der Replikationsstrategie in einen sinnvollen, lesbaren Kontext für die Ausgabe umgewandelt.

Bei dem Sequenzdiagramm in Abbildung 4.3 existiert das Szenario bereits in der Datenbank. Die Datenbankabfrage gibt darauf die ID des Szenarios zurück. Nun wird für jede Replikationsstrategie geprüft, ob Ergebnisse zu dieser ID in der Datenbank gespeichert sind. Wenn dieses nicht der Fall ist, dann werden die Berechnungen analog zu der Situation eines neuen Szenarios durchgeführt. Die Formatierung der Daten erfolgt ebenfalls analog zu der Situation eines neuen Szenarios.

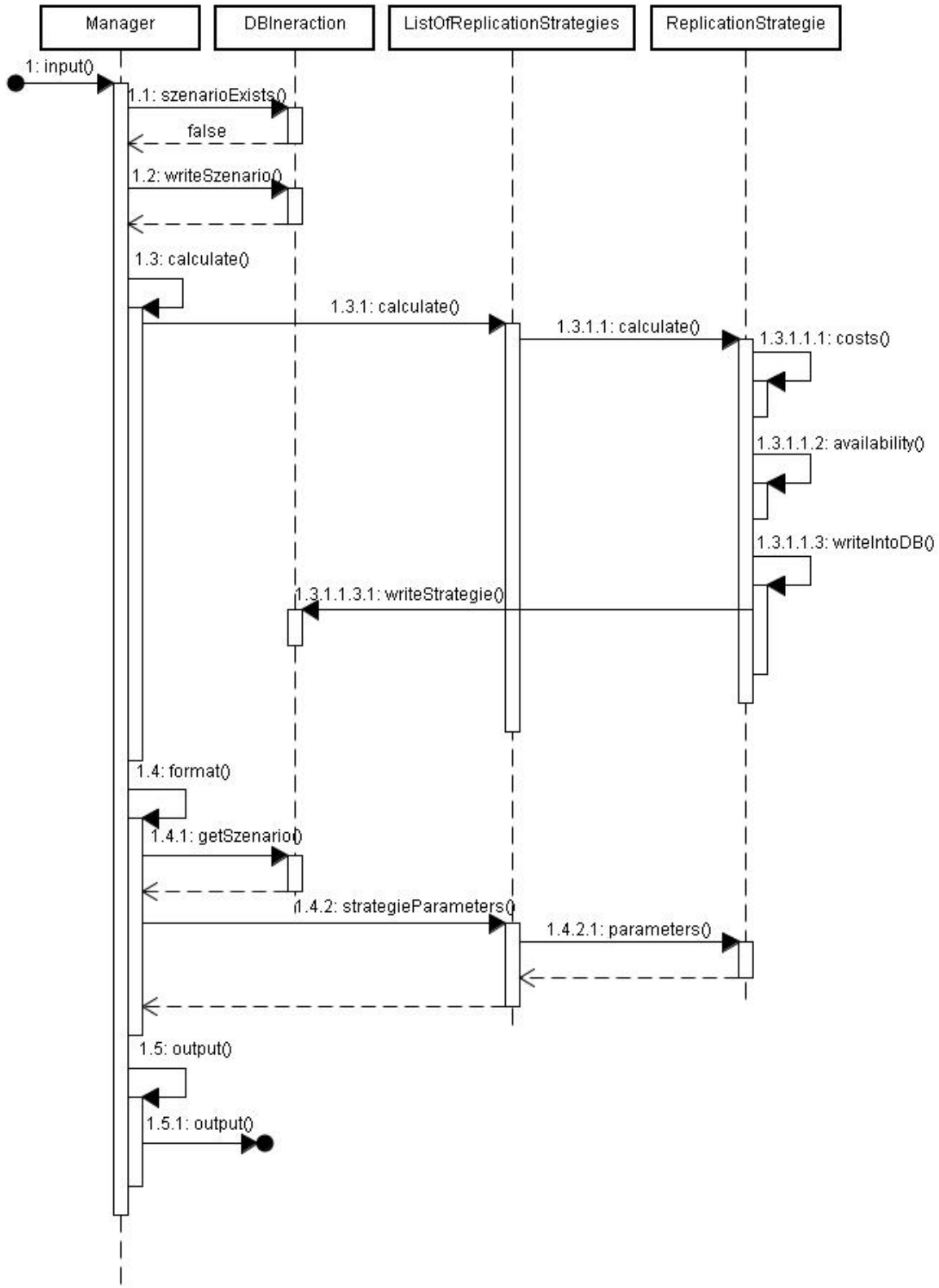


Abbildung 4.2: Sequenzdiagramm für ein neues Szenario

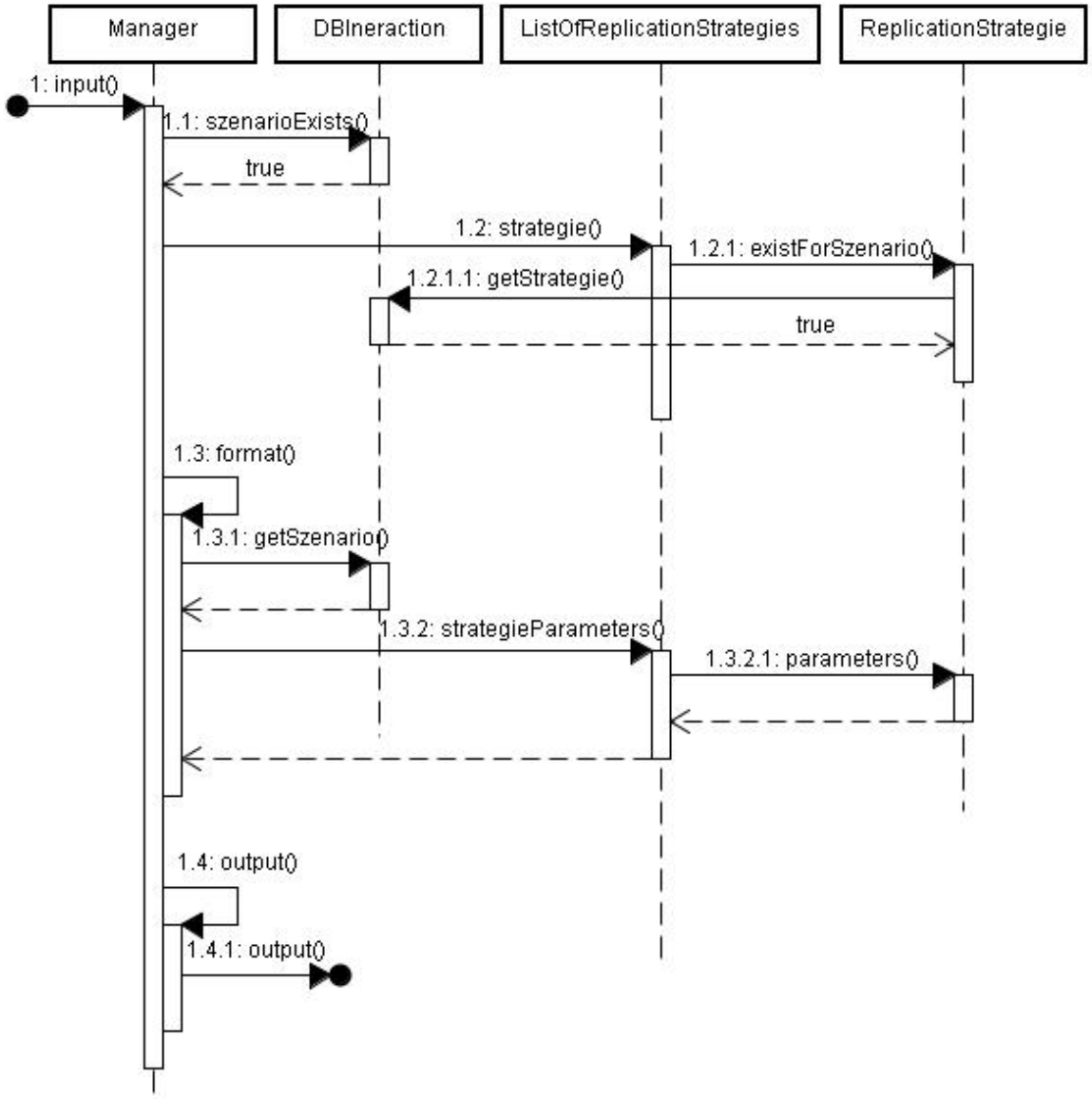


Abbildung 4.3: Sequenzdiagramm für ein bekanntes Szenario

4.2 Klassendiagramme

Die Software zur Ermittlung der optimalen Replikationsstrategie ist in drei Pakete aufgeteilt. Diese Pakete sind in Abbildung 4.4 zu sehen, auf die im Folgenden genauer eingegangen wird.

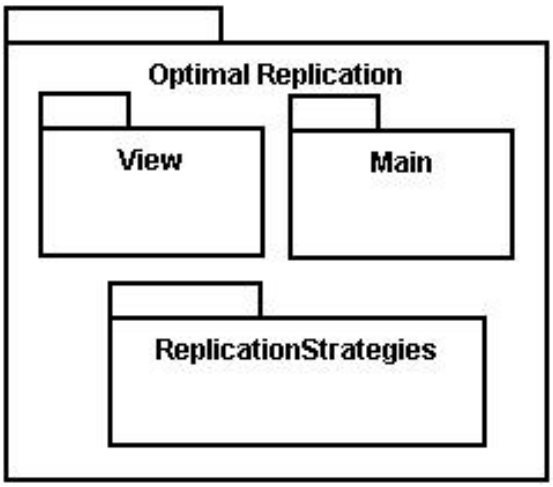
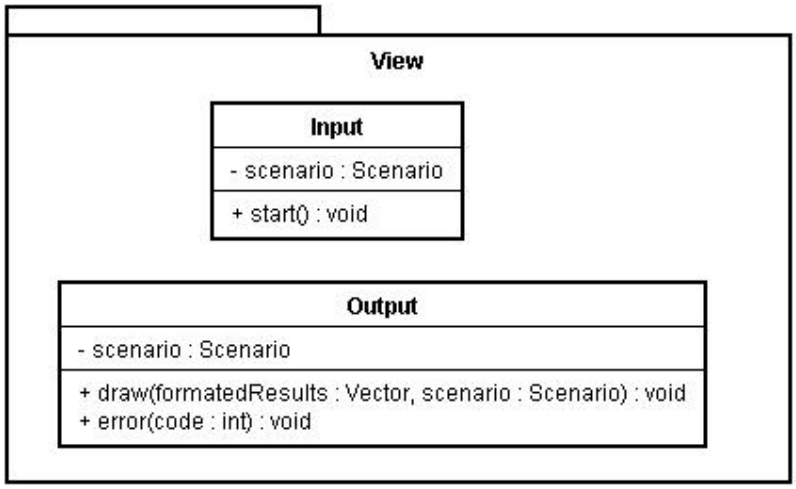


Abbildung 4.4: Pakete der Software

4.2.1 Das Paket View

Das Paket View beinhaltet die Klassen zur Eingabe der Szenariodaten und zur Ausgabe der Ergebnisse.



Input

Erstellt ein Szenario und startet die Berechnung.

scenario:Scenario
Ein Objekt von Typ `Szenario`.

start():scenario
Diese Methode ruft die Main-Methode von `Manager` auf, um die Berechnung zu starten.

Output

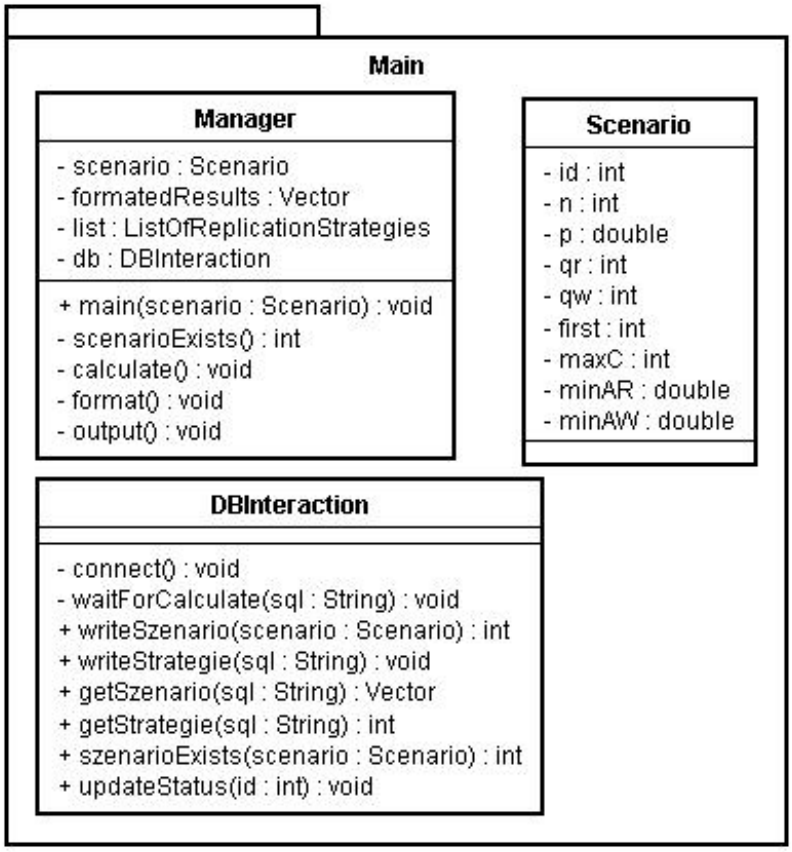
Diese Klasse bereitet die Ausgabe der Ergebnisse auf und gibt für Fehler wie falsche Eingaben, das Szenario wird bereits berechnet oder wenn keine passenden Ergebnisse gefunden werden konnten, eine Meldung aus.

draw(formatedResults:Vector<String>)
Erstellt die Ausgabe der Ergebnisse. Dazu wird ein lesbarer Text aus ganzen Sätzen mit den eingegebenen Werten ausgegeben und die Ergebnisse aus `formatedResults` für eine optisch ansprechende Anzeige formatiert.

error(code:int)
Entsprechend dem Wert von `code` wird ein passender Fehlertext ausgegeben.

4.2.2 Das Paket Main

In diesem Paket befinden sich die Klassen, die die Berechnungen koordinieren, die Klasse zur Speicherung der Szenariodaten und die Schnittstelle zur Datenbank.



Manager

Diese Klasse steuert den gesamten Ablauf der Berechnung. Sie ist unabhängig von der Klasse `input`, so dass auch Berechnungen von anderen Klassen aus gestartet werden können. Damit besteht die Möglichkeit, das Projekt auch in andere Projekte einzubinden.

scenario:Scenario
Ein Objekt von Typ `Szenario`.

formatedResult:Vector<Sring>
In diesem Vector werden die formatierten Ergebnisse gespeichert. Dabei wird jedes Ergebnis in einem eigenen String gespeichert.

list:ListOfReplikcationStrategies
Die Instanz von `ListOfReplikcationStrategies`.

db:DBInterface
Die Instanz von `DBInterface`.

main(scenario:Scenario)
Die Klasse steuert den kompletten Ablauf von der Berechnung bis zur Ausgabe.

szenExist()
Prüft, ob ein Szenario mit identischen Startwerten bereits in der Datenbank gespeichert ist.

calculate()
Startet die Berechnungen.

format()
Fragt die sortierte Ergebnisliste aus der Datenbank ab und bereitet sie zur Ausgabe auf.

output()
Stößt die Ausgabe der Daten an.

DBInteraction

Diese Klasse dient der Kommunikation mit der Datenbank. Alle Datenbankaufrufe werden von diese Klasse verarbeitet. Dadurch ist es nicht nötig, die Verbindungsdaten der Datenbank in mehreren Klassen anzugeben. Zudem muss bei einem Wechsel der Datenbank nur diese Klasse angepasst, beziehungsweise ausgetauscht werden.

connect()
Baut eine Verbindung mit der Datenbank auf.

waitForCalculate(SQL:String)
Wenn ein Szenario bereits in der Datenbank eingetragen ist, aber noch nicht fertig berechnet wurde, wartet diese Methode bis zu 60 Minuten auf das Ende der Berechnungen und fragt dabei jede Minute erneut in der Datenbank an, ob die Berechnung inzwischen abgeschlossen ist.

writeScenario(scenario:Scenario):int
Speichert ein neues Szenario in der Datenbank und gibt dessen ID zurück.

writeStrategie(SQL:Sting)
Speichert ein neues Ergebnis in der Datenbank. Da jede Strategie ihre eigene Table mit unterschiedlichen Attributen hat, liefert jede Strategie ein entsprechendes SQL-Statement.

getSzenario(SQL:String):Vector<Vector>double»
Führt ein SQL-Statement aus und speichert das Ergebnis Zeilenweise in einem zweidimensionalen Vector. Das Sortieren nach den Gewichtungen des Szenarios erfolgt hier durch die Datenbank.

getStrategie(SQL:String):int
Führt ein SQL-Statement mit einer Count-Anweisug aus, mit dem die Anzahl der Ergebnisse für eine Szenario-ID gezählt werden, und liefert das Ergebnis zurück. Da die Ergebnisse jeder Strategie in einer eigenen Tabelle gespeichert werden, muss jede Strategie ein für sie passendes Statement liefern.

scenarioExists(scenario:Scenario):int
Prüft, ob ein Szenario mit diesen Werten bereits in der Datenbank existiert. Wenn ein existierendes Szenraio gefunden wurde, wird die ID des Szenarios zurückgegeben. Ansonsten wird `0` zurückgegeben.

updateStatus(id:int)
Ändert den Status eines Szenarios nach dessen kompletter Berechnung.

Scenario

Dieser Datentyp enthält alle Daten eine Szenarios. Damit müssen diese Daten nicht einzelnd weitergereicht werden sondern es genügt, eine Instanz von `Scenario` zu übergeben. Diese Instanz enthält die Daten aus den Nutzereingaben.

id:int
Die ID dieser Strategie in der Datenbank.

n:int
Die Anzahl der Replikate.

p:double
Die Intaktwahrscheinlichkeit der Rechner.

qr:int
Die durchschnittliche Anzahl der Lesevorgänge pro Stunde.

qw:int
Die durchschnittliche Anzahl der Schreibvorgänge pro Stunde.

first:int
Die Reihenfolge der Gewichtung von Kosten, Lese- und Schreibverfügbarkeiten.

maxC:int
Die gewünschten maximalen Kosten.

minAR:double
Die gewünschte minimale Leseverfügbarkeit.

minAW:double
Die gewünschte minimale Schreibverfügbarkeit.

4.2.3 Das Paket Replikationsstrategien

Dieses Paket beinhaltet die Verwaltung der Replikationsstrategien sowie die Replikationsstrategien selber.

ListOfReplikcationStrategies

Diese Klasse dient zur Verwaltung der vorhandenen Replikationsstrategien. Neue Replikationsstrategien müssen hier eingetragen werden.

replicationStrategies : Vector<ReplicationStrategie>
Ein Array, in dem die Instanzen der einzelnen Replikationsstrategien gespeichert sind.

formatedResult:Vector<String>
In diesem Vector werden die formatierten Ergebnisse gespeichert. Dabei wird jedes Ergebnis in einem eigenen String gespeichert.

calculate(scenario:Scenario)
Stößt für jede Strategie aus `replicationStrategies` die Berechnungsmethode an.

strategy(scenario:Scenario)
Stößt für jede Strategie aus `replicationStrategies` eine Überprüfung an, ob Ergebnisse dieser Strategie zu diesem Szenario existieren.

strategyParameters(unformatedResults:Vector<Vector>double»):Vector<String>
Stößt je nach `STID` jeder Zeile des Vectors die Formatierungsfunktion der enstprechenden Replikationsstrategie an und übergibt dieser den Inhalt einer Zeile.

resultString(scenario:Scenario):String
Diese Methode erstellt das SQL-Statement, welches zur Abfrage und Sortierung der Daten eines Szenarios benötigt wird. Dabei wird auf die Besonderheiten jeder Strategie eingegangen.

ReplicationStrategies

Dieses ist ein Interface, welches die Attribute und Methoden für jede Replikationsstrategie vorgibt. Durch dieses Interface soll das Hinzufügen neuer Strategien vereinfacht werden.

id:int
Die ID ist wichtig, um in der Ergebnistabelle zwischen den Strategien unterscheiden zu können.

scenario:Scenario
Ein Objekt von Typ `Szenario`.

cr:double
Die Kosten für einen Lesezugriff.

cw:double
Die Kosten für einen Schreibzugriff.

ar:double
Die Verfügbarkeit eines Lesezugriffes.

aw:double
Die Verfügbarkeit eines Schreibzugriffes.

calculate(scenario:Scenario)
Die Hauptmethode, die die Berechnungen koordiniert.

costs()
Berechnet die Kosten dieser Strategie für das Szenario.

availability()
Berechnet die Verfügbarkeiten dieser Strategie für das Szenario.

writeIntoDB()
Lässt die Ergebnisse der Berechnung in die Datenbank schreiben.

existForSzenario(scenario:Scenario)
Prüft, ob für diese Strategie schon Ergebnisse für das Szenario vorhanden sind. Wenn nicht, werden diese berechnet.

parameters(column:Vector<double>):String
Formatiert den String aus dem Ergebisvector in einen verständlichen Text. Dieser Text enthält je nach Strategie alle Informationen, die wichtig sind.

4.3 Datenbankentwurf

Die Datenbank dient zur Speicherung der Szenarien und der Ergebnisse der Berechnungen der einzelnen Replikationsstrategien für den Fall, dass ein Szenario mit identischen Daten erneut abgefragt wird. In diesem Fall werden die Berechnungen nicht erneut ausgeführt, sondern die Ergebnisse aus der Datenbank gelesen. Zudem übernimmt die Datenbank das Sortieren der Ergebnisse für die Ausgabe.

Das Schema für die Szenarien sieht wie folgt aus:

ID	N	p

- ID - Primary Key
- N - Anzahl der Replikate
- p - Intaktwahrscheinlichkeit der Rechner
- completed - Bei neuen Szenarien wird dieser Wert erst nach erfolgter Berechnung gesetzt. Das verhindert bei zufälliger gleichzeitiger Eingabe eines gleichen Szenarios Seiteneffekte.

Für jedes Szenario müssen lediglich die Anzahl der Replikate sowie die Intaktwahrscheinlichkeit der Rechner gespeichert werden, da nur diese für die Berechnungen wichtig sind. Alle weiteren Daten des Szenarios dienen lediglich der Auswertung und Sortierung der Ergebnisse.

Jede Replikationsstrategie besitzt ihr eigenes Schema. Da jede Replikationsstrategie unterschiedliche Parameter besitzt, wäre bei einem Schema für alle Replikationsstrategien ein großer Teil der Attribute leer. Außerdem müsste unter Umständen bei jeder neuen Replikationsstrategie das Schema angepasst werden.

Für die in Kapitel 2.3 genannten Replikationsstrategien sehen die Schemata wie folgt aus:

Read-One-Write-All-Strategie

ID	SID	CR	CW	AR	AW	STID

Da diese Strategie keine zusätzlichen Parameter hat, sind dieses auch die Grundwerte, die jede Strategietabelle enthalten muss.

- ID - Primary Key.
- SID - ID des zugehörigen Szenarios
- CR - Kosten einer Leseoperation
- CW - Kosten einer Schreiboperation
- AR - Verfügbarkeit einer Leseoperation
- AW - Verfügbarkeit einer Schreiboperation
- STID - Die ID einer Strategie

Weighted-Voting-Strategie

ID	SID	CR	CW	AR	AW	LQ	SQ	STID

Zusätzlich sind in dieser Tabelle diese Werte enthalten:

- LQ - Größe des Lesequorums
- SQ - Größe des Schreibquorums

Grid-Strategie

ID	SID	CR	CW	AR	AW	R	S	STID

Zusätzlich sind in dieser Tabelle diese Werte enthalten:

- R - Anzahl der Reihen des Gitters
- C - Anzahl der Spalten des Gitters

Wenn die Anzahl der Replikate kein vollständig gefülltes Gitter ermöglicht, wird das Gitter immer von oben nach unten und von links nach rechts befüllt. Die freien Knoten befinden sich also in der untersten Zeile, von rechts beginnend. Durch diese Vorgabe ist es Ausreichend zu der Anzahl der Replikate die Größe des Gitters zu kennen.

5 Umsetzung des Entwurfs

In diesem Kapitel wird aufgezeigt, wie der Entwurf umgesetzt wurde.

5.1 Grundlagen

5.1.1 C++

Als IDE kam Netbeans zum Einsatz. Das Betriebssystem war Ubuntu 12.04.1.

Nach Rücksprache mit dem Betreuer wurde sich für diese Arbeit für die Programmiersprache C++ entschieden. Die Wahl fiel auf C++, da das Hauptaugenmerk der Software auf der Berechnung von Formeln liegt. Hier bietet C++ die bessere Performanz.

5.1.2 MySQL

Für die Speicherung der Daten wurde das Datenbankmanagementsystem MySQL genutzt. Den Ausschlag gaben hier die weite Verbreitung, die Einstufung als Open-Source-Software sowie das einfache Ansprechen aus dem C++-Code heraus.

Für Letzteres bietet MySQL sogenannte Connectoren für verschiedene Programmiersprachen an. Jede Anfrage erfolgt nach dem gleichen Prinzip. Es wird eine Verbindung zur Datenbank aufgebaut, das Statement abgesetzt, gegebenenfalls die Ergebnistabelle abgefragt und die Verbindung wieder geschlossen.

Das Aufbauen der Verbindung erfolgt über einen sogenannten `driver`.

```
1 sql::mysql::MySQL_Driver *driver;
2 sql::Connection *con;
3
4 driver = sql::mysql::get_driver_instance(); \\
5 con = driver->connect("tcp://127.0.0.1:3306", "user", "password");
```

Statements werden über die Methode `Statement` ausgeführt. Für Statements, die kein Ergebnis liefern, wird `execute` verwendet.

```
1 sql::Statement *stmt;
2
3 stmt = con->createStatement();
4 stmt->execute("USE␣" EXAMPLE_DB);
5 stmt->execute("DROP␣TABLE␣IF␣EXISTS␣test");
6 stmt->execute("CREATE␣TABLE␣test(id␣INT,␣label␣CHAR(1))");
7 stmt->execute("INSERT␣INTO␣test(id,␣label)␣VALUES␣(1,␣'a')");
```

Um ein Ergebnis zu speichern, wird ein `ResultSet` benötigt. Das Statement wird hierbei mit `executeQuery` abgesetzt.

```
sql::ResultSet  *res;

res = stmt->executeQuery("SELECT id, label FROM test ORDER BY id ASC");
```

Auf das `ResultSet` kann immer Zeilenweise auf zwei Arten zugegriffen werden.
Mit `res->getInt(1)` wird das erste Feld der aktuellen Zeile als Integer zurückgegeben.
`res->getString(„label")` gibt den Inhalt der Spalte `label` als String zurück.

Nach jedem Aufruf werden die Verbindung, das Statement und das ResultSet wieder gelöscht.

```
delete res;
delete stmt;
delete con;
```

5.1.3 HTML und PHP

Die Software soll über eine Webseite angesprochen werden können. Dafür wurde ein intuitiv zu bedienendes Design gewählt, in dem die Daten des Szenarios in separaten Eingabefeldern eingetragen werden.

Nach Betätigung eines Buttons wird die Seite neu geladen. Die Eingaben werden dabei durch eine PHP-Methode eingelesen und über die PHP-Funktion `exec` an eine kompilierte, ausführbare Version der Software übergeben. Die PHP-Funktion wartet auf die Rückgaben der Software und zeigt diese beim neuen Erstellen der Webseite an.

5.2 Umsetzung der vorhandenen Strategien

In diesem Kapitel wird die Umsetzung der drei vorhandenen Replikationsstrategien genau erklärt.

5.2.1 Read-one-Write-All-Strategie

Die Berechnung der Kosten ist trivial, da die Lesekosten immer 1 und die Schreibkosten immer N sind.

```
    virtual void costs() {
        //Readcosts
        cr = 1;
        //Writecosts
        cw = scenario.getN();
    }
```

Auch die Berechnung der Verfügbarkeiten benötigt nur simple Formeln:

```
1     virtual void availability() {
2         //Readavalability
3         ar = 1.0 - pow((1.0 - scenario.getP()), scenario.getN());
4
5         //Writeavalability
6         aw = pow(scenario.getP(), scenario.getN());
7     }
```

5.2.2 Weighted-Voting-Strategie

Da es bei dieser Strategie mehrere mögliche Kombinationen der Quoren gibt, müssen diese natürlich alle berechnet werden. Dazu wird das Lesequorum mit 2 und das Schreibquorum mit $N-1$ begonnen und gleichmäßig inkrementiert beziehungsweise dekrementiert solange das Lesequorum größer als $N/2$ ist. Natürlich kann die Weighted-Voting-Strategie auch mit einem Lesequorum von 1 und einem Schreibquorum von N beginnen. Dieser Fall ist aber Deckungsgleich mit der Read-One-Write-All-Strategie und wird darum nicht betrachtet.

```
1     virtual void calculate(Scenario & new_scenario) {
2         scenario = new_scenario;
3         n = scenario.getN();
4         qr = 2;
5         for (qw = n - 1; qw > n / 2; --qw) {
6             costs();
7             availability();
8             writeIntoDB();
9             ++qr;
10        }
11    }
```

Die Kosten ergeben sich aus den Größen der Quoren:

```
1     virtual void costs() {
2         //Readcosts
3         cr = qr;
4         //Writecosts
5         cw = qw;
6     }
```

Da die Verfügbarkeiten über eine Summe berechnet wird, wird hier eine Schleife eingesetzt, welche die Summenformel wiedergibt:

```
1     virtual void availability() {
2         //Readavalability
3         ar = 0;
4         for (int k = qr; k <= n; ++k) {
5             ar += aSum(k);
6         }
7
8         //Writeavalability
9         aw = 0;
10        for (int k = qw; k <= n; ++k) {
11            aw += aSum(k);
12        }
13    }
```

Da beide Formeln identisch sind, wurde die Berechnung in eine eigene Methode ausgelagert:

```
double aSum(int k) {
    long double answer = 1;
    answer = (nOverK(n, k)) * (pow(scenario.getP(), k)) * (pow((1 - scenario.
        getP()), (n - k)));
    return (double) answer;
}
```

Die Berechnungen von $\begin{pmatrix} N \\ k \end{pmatrix}$ und der hierfür nötigen Fakultät wurden in eine eigene Methode verschoben, um hier mit einem größeren Datentypen arbeiten zu können:

```
long double nOverK(int n, int k) {
    long double answer = 1;
    answer = factorial(n) / (factorial(n - k) * factorial(k));
    return answer;
}

long double factorial(int i) {
    long double answer = 1;
    for (; i > 0; i--) {
        answer *= i;
    }
    return answer;
}
```

5.2.3 Grid-Strategie

Das Gitter dieser Replikationsstrategie sollte möglichst quadratisch sein. Wenn das nicht möglich ist, dann wird das Gitter so gleichmäßig wie möglich aufgebaut. r und s unterscheiden sich in diesem Fall um genau 1. Um die Größe des Gitters zu bestimmen, wird die Wurzel aus N gezogen. Das Ergebnis wird mit $0,9$ beziehungsweise $0,5$ addiert und anschließend beim Umwandeln des Datentypen auf die volle Zahl gekürzt. Durch das Addieren wird sichergestellt, dass bei nicht quadratischer und vollständiger Belegung das Netz groß genug wird.

Wenn r und s unterschiedlich sind, das Gitter also nicht quadratisch ist, dann wird zusätzlich das Gitter mit vertauschten r und s berechnet.

```
virtual void calculate(Scenario & new_scenario) {
    scenario = new_scenario;
    n = scenario.getN();
    double d = sqrt(n);
    r = static_cast<int> (d + 0.9);
    s = static_cast<int> (d + 0.5);
    costs();
    availability();
    writeIntoDB();
    if (s != r) {
        int i = s;
        s = r;
        r = i;
        costs();
        availability();
```

```
16              writeIntoDB();
17          }
18      }
```

Die Kosten ergeben sich aus der Größe des Gitters. Dabei muss bei nicht vollständig gefüllten Gittern für die Schreibkosten ein Mittelwert gefunden werden. j ist dann die Anzahl der vollständig belegten Spalten.

```
1       virtual void costs() {
2           //Readcosts
3           cr = r;
4           //Writecosts
5           int i = (r * s) % n;
6           //if complete
7           if (i == 0) {
8               cw = r + s - 1;
9           }
10          else {
11              int j = n - (r * (s - 1));
12              cw = r + s - 2 + (j / n);
13          }
14      }
```

Wie in Kapitel 2 beschrieben, wurden die Formeln für nicht vollständig belegte Gitter entsprechend angepasst. Dazu wird erst kontrolliert, ob das Gitter vollständig belegt ist um dann je nach Fall mit dem vollständigen oder den beiden aufgespaltenen Gittern zu rechnen.

```
1       virtual void availability() {
2           //Readavailability
3           ar = 0;
4           int i = (r * s) % n;
5           //if complete
6           if (i == 0) {
7               ar = arIntern(r, s);
8           } else {
9               int j = n - (r * (s - 1));
10              ar = arIntern(j, s) * arIntern(r - j, s - 1);
11          }
12
13          //Writeavailability
14          aw = 0;
15          //if complete
16          if (i == 0) {
17              aw = arIntern(r, s) - awIntern(r, s);
18          } else {
19              int j = n - (r * (s - 1));
20              aw = (arIntern(j, s) * arIntern(r - j, s - 1)) - (awIntern(j, s) *
                    awIntern(r - j, s - 1));
21          }
22      }
```

Da die Berechnung der Leseverfügbarkeit mehrfach, unter anderem auch für die Schreibverfügbarkeit angewendet wird, wurde sie in eine eigene Methode ausgelagert. Die Berechnung des zweiten Teils der Schreibverfügbarkeit ebenso.

```
    double arIntern(int r, int s) {
        double answer;
        answer = pow(1 - pow(1 - scenario.getP(), s), r);
        return answer;
    }
    double awIntern(int r, int s) {
        double answer;
        answer = pow(1 - pow(scenario.getP(), s) - pow(1 - scenario.getP(), s), r
            );
        return answer;
    }
```

5.3 Einfügen neuer Replikationsstrategien

Wenn eine neue Replikationsstrategie in die Software eingefügt werden soll, dann muss eine Klasse für die Replikationsstrategie geschrieben werden, die von der Klasse `ReplicationStrategie` abgeleitet wird.

In dieser Klasse müssen diese Methoden ausprogrammiert werden:

```
protected:
    //Berechnung der Lese- und Schreibkosten.
    virtual void costs() = 0;
    //Berechnung der Lese- und Schreibverfügbarkeiten.
    virtual void availability() = 0;
    //Speichern der Werte in die Datenbank.
    virtual void writeIntoDB() = 0;

public:
    //Durchführung der Berechnungen, dabei möglicherweise mehrere Kombinationen
        beachten.
    virtual void calculate(Scenario & new_scenario) = 0;
    //Überprüfung, ob es für dieses Szenario Ergebnisse in der Datenbank gibt.
    virtual void existsForScenario(Scenario & this_scenario) = 0;
    //Formatierung einer Ergebnisspalte in einen lesbaren Satz.
    virtual std::string parameters(std::vector<double> column) = 0;
```

Gibt es für diese Replikationsstrategie mehrere Möglichkeiten für das Schreiben und Lesen, so müssen alle sinnvollen Möglichkeiten von dieser Klasse berechnet werden.

Zusätzlich muss in der Klasse `ListofReplicationStrategies.cpp` die neue Klasse im Konstruktor eingefügt werden, um sie in den Vektor einzufügen. Der Replikationsstrategie ist dabei eine fortlaufende ID zu geben.

Außerdem ist das zu erstellende SQL-Statement in der Methode `std::string resultstring(Scenario & this_scenario)` der Klasse `ListofReplicationStrategies.h` um die neue Replikationsstrategie zu erweitern.

Dieses Statement wird zentral in der Klasse `ListofReplicationStrategies.h` erstellt. Wenn die Erstellung des Statements auf die Klassen der Replikationsstrategien verteilt wäre, dann wäre das Fehlerpotential deutlich größer. Wird zum Beispiel die Anzahl der freien Spalten für die strategiespezifischen Werte erhöht, so müssen alle Abfragen der einzelnen Tabellen der Replikationsstrategien angepasst werden.

5.4 SQL-Statements

Während der Berechnung werden verschiedene SQL-Statements erstellt, auf die hier genauer eingegangen werden soll.

5.4.1 Überprüfen und Speichern des Szenarios

Das erste Statement überprüft, ob es in der Datenbank bereits ein Szenario mit diesen Daten gibt:

```
SELECT *
FROM `OptimalReplication`.`Scenarios`
WHERE N=70 AND p=0.86
```

Wenn das nicht der Fall ist, dann wird das Szenario in der Datenbank gespeichert und die Berechnung gestartet. Existiert das Szenario bereits in der Datenbank, wird mit Kapitel 5.4.2 weiter gemacht, ansonsten mit der Berechnung in Kapitel 5.4.3.

```
INSERT INTO `OptimalReplication`.`Scenarios`
(`N`,`p`,`completet`)
VALUES(70,0.86,0)
```

5.4.2 Bekanntes Szenario

Wird ein Szenario mit gleichen Werten von `N` und `p` noch einmal angefragt, wird bei der ersten Abfrage aus Kapitel 5.4.1 die ID zurückgegeben. Die Berechnungen entfallen dann, jedoch muss überprüft werden, ob für jede im System bekannte Strategie Ergebnisse zu diesem Szenario vorliegen. Dazu wird von jeder Strategie ein Statement gesendet, mit dem die Anzahl der Zeilen mit der ID des Szenarios gezählt werden:

```
SELECT COUNT(*)
FROM `OptimalReplication`.`rowa`
WHERE SID=64
```

Existieren keine Datensätze zu diesem Szenario, werden sie berechnet. Dieses Vorgehen ist wichtig, da Replikationsstrategien nach dem erstmaligen Berechnen eines Szenarios dazu kommen können. Danach wird mit der Abfrage der Ergebnisse in Kapitel 5.4.5 weiter gemacht.

5.4.3 Speicherung der Berechnungen

Da jede Strategie eine eigene Tabelle mit spezifischen Attributen hat, unterscheiden sich auch die Statements zum Eintragen der Ergebnisse.

Für die Read-One-Write-All-Strategie sieht das Statement so aus:

```
INSERT INTO `OptimalReplication`.`rowa`
(`SID`,`CR`,`CW`,`AR`,`AW`,`STID`)
VALUES(64,1,70,1,2.59951e-05,1)
```

Die extrem niedrige Schreibverfügbarkeit folgert sich daraus, dass alle 70 Replikate erreicht werden müssen, damit das Schreiben möglich ist.

Bei der Weighted-Voting-Strategie gibt es mehrere mögliche Größen für die Schreib- und Lesequoren. Jede sinnvolle, mögliche Kombination wird einzeln betrachtet und in die Datenbank geschrieben:

```
1 INSERT INTO `OptimalReplication`.`WV`
2 (`SID`,`CR`,`CW`,`AR`,`AW`,`QR`,`QW`,`STID`)
3 Values (64,2,69,1,0.000322218,2,69,2)
```

```
1 INSERT INTO `OptimalReplication`.`WV`
2 (`SID`,`CR`,`CW`,`AR`,`AW`,`QR`,`QW`,`STID`)
3 Values (64,35,36,1,1,35,36,2)
```

Hier sieht man schön, wie die Schreibverfügbarkeit stark von der Größe der Quoren abhängt. Wenn 69 Replikate geschrieben werden müssen, dann ist die Schreibverfügbarkeit nahe der Schreibverfügbarkeit der Read-One-Write-All-Strategie. Im zweiten Beispiel sind die Quoren fast gleich groß. Die Verfügbarkeiten haben sich auch angeglichen.

Als letztes folgt noch die Grid-Strategie:

```
1 INSERT INTO `OptimalReplication`.`Grid`
2 (`SID`,`CR`,`CW`,`AR`,`AW`,`R`,`S`,`STID`)
3 Values (64,9,16,0.999997,0.527092,9,8,3)
```

Wie hier zu sehen ist, können 70 Replikate nicht optimal in ein quadratisches Gitter eingetragen werden. Darum wird noch eine zweite Möglichkeit mit getauschten Gittergrößen berechnet und in die Datenbank eingetragen:

```
1 INSERT INTO `OptimalReplication`.`Grid`
2 (`SID`,`CR`,`CW`,`AR`,`AW`,`R`,`S`,`STID`)
3 Values (64,9,16,1,0.423512,8,9,3)
```

Durch das Drehen hat sich die Leseverfügbarkeit leicht verbessert, die Schreibverfügbarkeit jedoch deutlich verschlechtert.

Auffällig sind die Verfügbarkeiten von 1. Diese Verfügbarkeiten sind natürlich gerundet von 0.9 Periode.

5.4.4 Markieren des Szenarios

Nachdem die Berechnungen abgeschlossen sind, wird der Status des Szenarios von 0 auf 1 geändert um anzuzeigen, dass das Szenario vollständig mit allen Replikationsstrategien im System berechnet wurde.

```
1 UPDATE `OptimalReplication`.`Scenarios`
2 SET completet=1
3 WHERE idScenarios=64
```

5.4.5 Abfragen der Ergebnisse

Jetzt wird die Ergebnistabelle erstellt. Dazu müssen die Ergebnisse der verschiedenen Replikationsstrategien in einer Tabelle vereint werden. Dieses geschieht mit folgendem Statement:

```
1  SELECT STID, (CR *1500) + (CW *100) AS C, AR, AW,
2  NULL AS eins, NULL AS zwei, NULL AS drei, NULL AS vier
3  FROM `OptimalReplication`.`rowa`
4  WHERE SID = 64
5  AND ((CR *1500) + (CW *100)) < 650000
6  AND AR > 0.95
7  AND AW > 0.95
8  union
9  SELECT STID, (CR *1500) + (CW *100) AS C, AR, AW,
10 QR AS eins, QW AS zwei, NULL AS drei, NULL AS vier
11 FROM `OptimalReplication`.`WV`
12 WHERE SID = 64
13 AND ((CR *1500) + (CW *100)) < 650000
14 AND AR > 0.95
15 AND AW > 0.95
16 union
17 SELECT STID, (CR *1500) + (CW *100) AS C, AR, AW,
18 R AS eins, S AS zwei, NULL AS drei, NULL AS vier
19 FROM `OptimalReplication`.`Grid`
20 WHERE SID = 64
21 AND ((CR *1500) + (CW *100)) < 650000
22 AND AR > 0.95
23 AND AW > 0.95
24 ORDER BY AR DESC, AW DESC, C ASC
```

Mit dem Statement werden gleich die Gesamtkosten pro Stunde berechnet, sowie alle Strategien ausgefiltert, die nicht den Ansprüchen des Szenarios bezüglich der Kosten und Verfügbarkeiten entsprechen. Außerdem erfolgt eine Sortierung gemäß der Vorgabe, in diesem Fall erst absteigend nach der Leseverfügbarkeit, dann absteigend nach der Schreibverfügbarkeit und anschließend aufsteigend nach den Gesamtkosten. Für die strategiespezifischen Parameter sind vier Spalten vorgesehen. Werden weniger benötigt, müssen die freien Spalten mit `NULL` gefüllt werden.

Hiernach liegen die Ergebnisse vor und müssen nur noch für die Ausgabe aufbereitet werden.

6 Ergebnisse

Für eine leichte Bedienung kann die Software über eine HTML-Seite mittels PHP angesprochen werden. Eingabe und Ausgabe werden auf der gleichen Seite angezeigt. In Abbildung 6.1 ist die Seite zu sehen, wie sie nach dem Aufrufen erscheint.

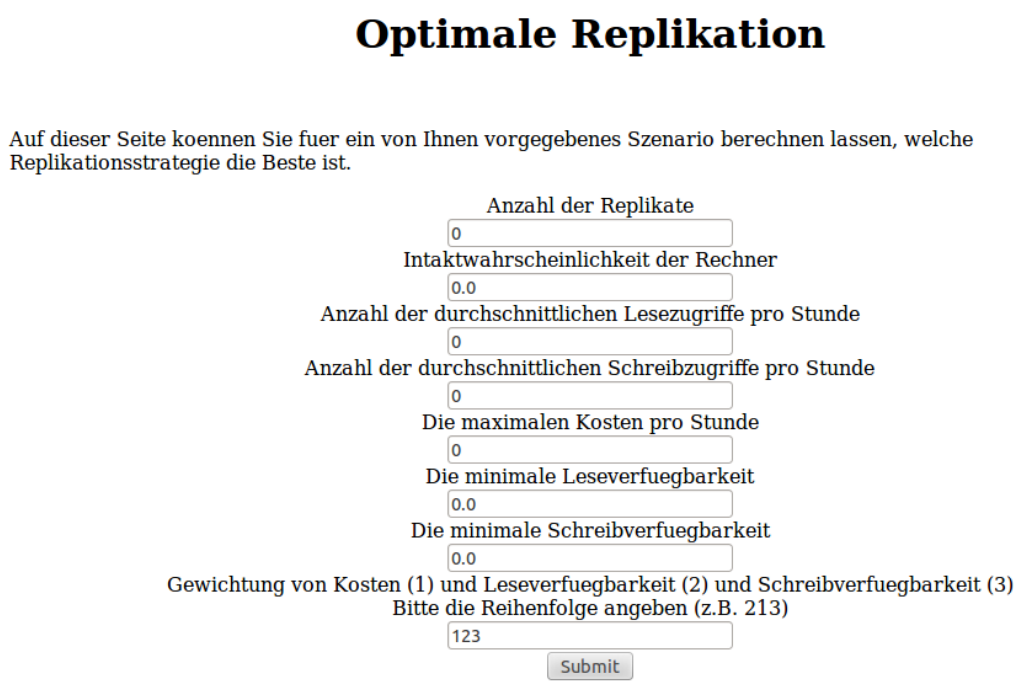


Abbildung 6.1: HTML-Seite zur Bedienung der Software

6.1 Beispiel

Der genaue Vorgang soll anhand eines Beispiels verdeutlicht werden.

6.1.1 Eingabe

Als Beispiel wird folgendes Szenario verwendet:

Anzahl der Replikate	N	70
Intaktwahrscheinlichkeit	p	0.86
Erwartete Lesezugriffe pro Stunde	QR	1500
Erwartete Schreibzugriffe pro Stunde	QW	100
Maximale Kosten pro Stunde	maxC	680.000
Minimale Leseverfügbarkeit	minAW	0.95
Minimale Schreibverfügbarkeit	minAR	0.95

Reihenfolge 231

In Abbildung 6.2 ist zu sehen, wie die Szenariodaten in die Eingabemaske eingetragen werden.

6.1.2 Ausgabe

Die Ausgabe der Ergebnisse erfolgt als eine sortierte Liste. Die Ausgabe des Beispielszenarios ist in Abbildung 6.3 zu sehen.

6.2 Performanz

Das Testsystem verfügte über einen Intel Core Duo Prozessor mit 1,83 GHz und 2038 MB Arbeitsspeicher.

6.2.1 Maximale Anzahl Replikate

Die maximale Anzahl der Replikate liegt im Moment bei 1500. Bei mehr Replikaten kommt es zu einem Werteüberlauf bei der Berechnung der Fakultät für die Verfügbarkeiten der Weighted-Voting-Strategie trotz des größtmöglichen Standartdatentypen von C++.

6.2.2 Berechnungsdauer

Die Dauer der Berechnung hängt stark von der Anzahl der Replikate ab. Auch hierbei ist die Weighted-Voting-Strategie dafür verantwortlich. Bei der Read-One-Write-All-Strategie gibt es nur eine Berechnung und bei der Grid-Strategie maximal zwei. Die Tabelle zeigt, wie lange die Software auf dem Testsystem benötigt. Die benötigte Zeit für ein neues Szenario steigt dabei nahezu linear an.

Anzahl der Replikate	Neues Szenario	Bekanntes Szenario
100	1,99s	etwa 300ms
500	10,66s	etwa 300ms
1000	25,06s	etwa 300ms
1500	42,50s	etwa 300ms

Bei bekannten Szenarien variiert die Zeit nur leicht. Der große Unterschied der Zeiten macht deutlich, wie erfolgreich das Speichern der Daten in der Datenbank ist.

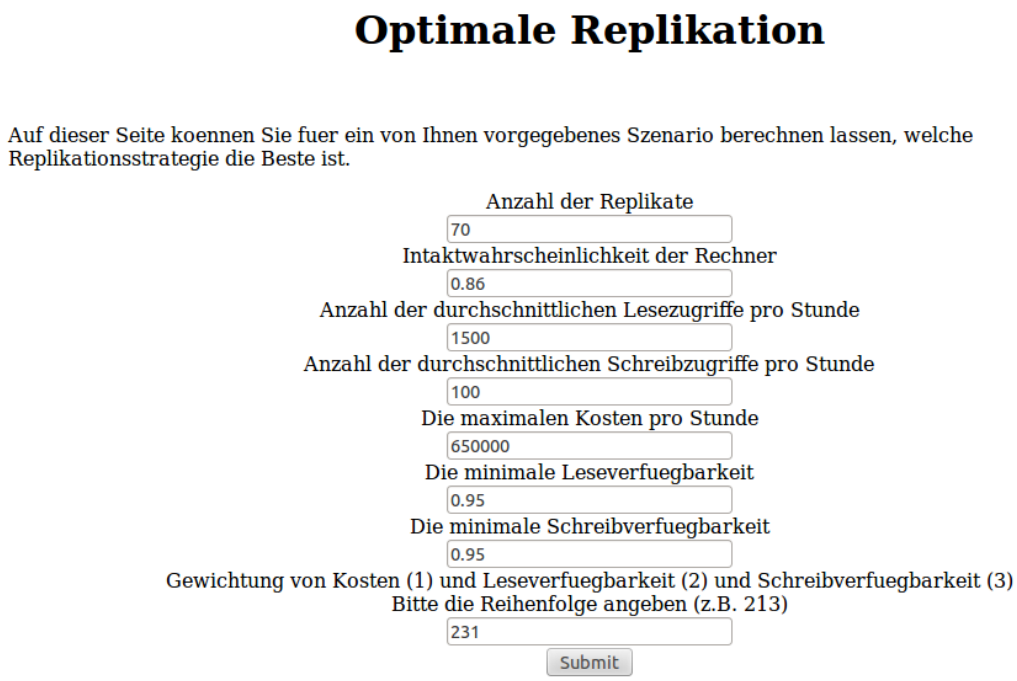


Abbildung 6.2: Eingabe des Beispielszenarios

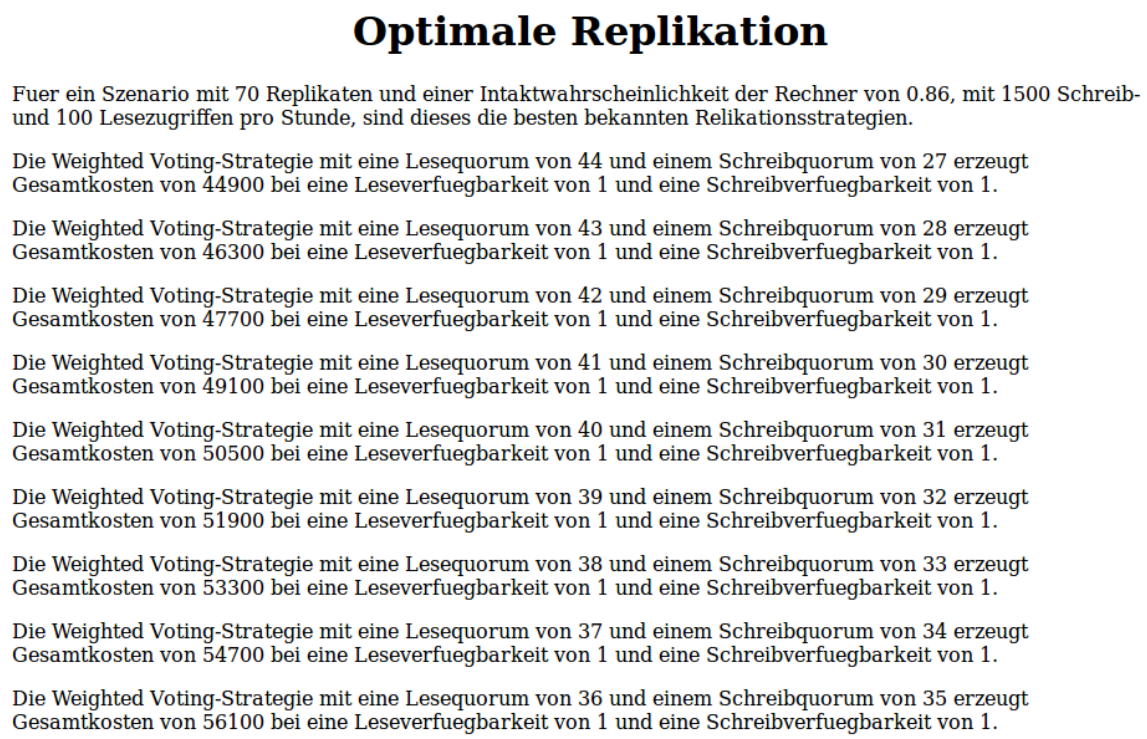


Abbildung 6.3: Ergebnis des Beispielszenarios

7 Fazit

7.1 Replikationsstrategien

Für diese Arbeit lag das Hauptaugenmerk nicht auf der Vorgehensweise der Replikationsstrategien, sondern auf der Berechnung der Kosten und Verfügbarkeiten. Hierfür lieferte mir „Entwurf und Bewertung von Replikationsverfahren“ [Koc94] alle nötigen Formeln. Allerdings sind diese Formeln immer auf die optimale Anzahl an Replikaten ausgerichtet, so dass bei einigen Replikationsstrategien Anpassungen nötig sind.

7.2 C++

Der Umstieg von Java zu C++ fiel erstaunlich leicht. Die ersten Erfolge mit funktionierenden Methoden und Klassen ließen nicht lange auf sich warten. Nichts desto trotz bin ich mir sicher, dass die Software schneller sein könnte. Gerade bei der Parameterübergabe bietet C++ viele Möglichkeiten mit Kopien und Referenzen zu arbeiten, welche ich sicher nicht vollständig ausgenutzt habe.

7.2.1 C++ und MySQL

Große Probleme traten erst bei der Anbindung an die MySQL Datenbank auf. Für diesen Zweck stellt Oracle einen Connector zur Verfügung mit einer ausführlichen Dokumentation [MyS]. Auf den ersten Blick ist dieser Connector wirklich simpel und sollte das Absenden von Statements und Abholen von Ergebnistabellen einfach gestalten. Leider war es mir nicht möglich, nach dieser Dokumentation mit den Beispielklassen eine funktionierende Verbindung aufzubauen. Nach ausführlicher Recherche habe ich herausgefunden, dass der für Windows angebotene Connector C++ mit VC++ kompiliert wurde und somit nicht mit dem vom mir verwendeten Cygwin-Kompiler kompatibel sein soll.

Ich habe mehrere Anleitungen befolgt, den Connector mit Cygwin neu zu kompilieren. Schließlich bin ich von Cygwin zu MinGW gewechselt, um Anleitungen zu testen, die auf diesem Kompiler aufbauten. Anschließend habe ich versucht, die Verbindung über eine andere API, MySQL++ [You], aufzubauen. Auch dieses scheiterte. Interessant dabei ist, das ich bei gleicher Vorgehensweise und gleichen Voraussetzungen auf verschiedenen Systemen verschiedene Fehler produzieren konnte.

Daraufhin bin ich von Windows zu Linux gewechselt. Auch hier funktionierte der Connector C++ nicht auf Anhieb und gab beim Kompilieren einen Fehler aus. Nach weiterer Internetrecherche habe ich schlussendlich herausgefunden, dass in sämtlichen Beispielen von Oracle ein Fehler ist. Nach beheben dieses Fehlers funktionierte die Anbindung der Datenbank allerdings problemlos und so einfach wie zu Anfang gedacht.

Ich habe dann testweise unter Windows die gleichen Änderungen an dem Beispielcode vorgenommen. Dieses führte aber nicht zum Erfolg.

7.2.2 Email durch C++ versenden

Am Anfang gab es Überlegungen, dass der Nutzer eine Email-Adresse hinterlegen kann, an die das Ergebnis der Berechnungen gesendet wird für den Fall, dass die Berechnungen sehr lange dauern. Das Versenden von Emails direkt durch C++ ist allerdings nur unter Windows möglich, da hier auf Windowsinterne Konstrukte zurück gegriffen werden kann. Unter Linux ist es nicht möglich, mit C++ Emails zu versenden. Darum habe ich zu Gunsten der Universalität auf dieses Feature verzichtet.

7.3 Ausblick

Die drei integrierten Replikationsstrategien stellen natürlich nur einen kleinen Teil der Möglichkeiten dar. Mit mehr Replikationsstrategien würde das Ergebnis deutlich Aussagekräftiger werden. Das Implementieren neuer Replikationsstrategien kann sehr kompliziert werden, vor allem, wenn die Idealbedingungen, mit denen die Replikationsstrategie arbeiten will, nicht immer gegeben sein können. Als Beispiel sei hier das Tree-Quorum-Protokoll genannt, das möglichst einen vollständigen Baum haben will. Für nicht vollständige Bäume müssen hier Lösungen zur Berechnung der Kosten und Verfügbarkeiten gefunden werden.

Die Arbeit könnte mit weiterem Ausbau nicht nur zur Berechnung von per Hand eingegebenen Szenarien dienen, sondern auch dynamischen Replikationsstrategien dazu dienen, für das momentane Szenario die beste statische Replikationsstrategie zu finden.

8 Handbuch

Dieses Handbuch erklärt, wie die Entwicklungsumgebung für die Software eingerichtet werden muss.

8.1 Netbeans

Als IDE kam Netbeans zum Einsatz. Das Betriebssystem war Ubuntu 12.04.1. Die aktuelle Version von Netbeans ist über das Ubuntu Software-Center zu installieren. Für die C++-Unterstützung muss das entsprechende Plugin installiert werden [1].

Wählen Sie dafür unter „Tools“ in der Menüleiste den Punkt „Plugins“. Navigieren Sie zu dem Reiter „Available Plugins“. Aktivieren Sie die Checkbox vor dem Plugin „C/C++“ und klicken Sie auf den Button „Install“. Folgen Sie den Anweisungen des Installationsassistenten und starten Sie Netbeans nach erfolgter Installation neu.

Fügen Sie das Projekt über das Menü „File“ und „Open Project“ hinzu. Sie finden das Projekt auf dem Datenträger im Ordner „Source“.

8.2 MySQL-Connector

Für den MySQL-Connector muss das Paket „libmysqlcppconn-dev“ installiert werden [2].

Damit Netbeans auch mit dem MySQL-Connector arbeiten kann, müssen die Pfade zu den Dateien eingetragen werden [3].

Öffnen Sie dafür die Projekteigenschaften unter „File“, „Project Properties“, vergewissern Sie sich, dass die Auswahl unter „Configuration“ auf „Release“ steht und navigieren Sie in dem Baum auf der linken Seite zu „Build“, „C++ Compiler“. Im rechten Feld wählen Sie unter „General“ den Button „...“ hinter „Include Directories“.In dem sich öffnenden Fenster fügen Sie über den Button „Add“ den Pfad zu den MySQL Connector/C++ header files hinzu. Dieser sollte „/usr/local/include“ sein. Schließen Sie die Fenster mit einem Klick auf „Select“ und „Ok“.

Jetzt muss die Library hinzugefügt werden. Ich habe die Dynamische Library gewählt. Navigieren Sie dafür im linken Baum zu „Linker“. Auf der rechten Seite klicken Sie auf den Button „...“ hinter „Additional Library Directories“. Es öffnet sich ein Fenster, in dem Sie über den Button „Add“ den Pfad zu der Datei „libmysqlcppconn.a“ hinzufügen. Die Datei sollte unter „/usr/local/lib“ oder „/usr/lib“ zu finden sein. Schließen Sie die Fenster mit einem Klick auf „Select“ und „Ok“.

Schließen Sie die Projekteigenschaften mit „Ok“. Der MySQL-Connector sollte jetzt erfolgreich installiert sein.

[1] Eine Anleitung zur Installation des Plugins ist zu finden unter: http://netbeans.org/community/releases/60/cpp-setup-instructions.html

[2] Das Paket ist hier zu finden: http://packages.ubuntu.com/lucid/libmysqlcppconn-dev

[3] Eine ausführliche Anleitung gib es unter: http://dev.mysql.com/doc/refman/5.1/en/connector-cpp-apps-linux-netbeans.html

Wenn Sie eines der vom MySQL-Connector mitgelieferten Beispiele probieren, achten Sie darauf, dass in jedem Beispiel ein Fehler enthalten ist [4].

Für eine funktionierende Verbindung ändern Sie die Zeile:

```
driver = get_driver_instance();
```

in

```
driver = sql::mysql::get_driver_instance();
```

, die Zeile

```
sql::Driver *driver;
```

ändern Sie in

```
sql::mysql::MySQL_Driver *driver;
```

und fügen folgende Zeilen hinzu:

```
include "mysql_driver.h"
include "mysql_connection.h"

using namespace sql::mysql;
```

8.3 MySQL

Die MySQL-Datenbank sollte auf jedem Ubuntu-System installiert sein [5]. Zum Erstellen des Schemas und der Tabellen für die Szenarien und die drei Replikationsstrategien führen Sie das SQL-Statement aus, welches Sie auf dem Datenträger im Ordner „SQL“ finden.

Erstellen Sie einen Datenbanknutzer mit vollem Zugriff für dieses Schema.

8.3.1 Zugangsdaten

Um eine Verbindung zur Datenbank zu erstellen, muss die Software folgende Daten kennen:

- Adresse der Datenbank
- Nutzername
- Passwort des Nutzers

Diese Daten sind in der Klasse `DBInteraction` in der Methode `connect` hinterlegt. Tragen Sie die Daten in diese Zeile ein:

```
con = driver->connect("tcp://127.0.0.1:3306", "user", "password");
```

[4] Die Änderungen sind aus den Antworten dieser Seite entnommen: http://stackoverflow.com/questions/3872388/mysql-c-connector-undefined-reference-to-get-driver-instance

[5] Ist die Datenbank nicht installiert, ist hier eine Anleitung zum Installieren zu finden: http://wiki.ubuntuusers.de/MySQL

8.3.2 Anzahl der Spalten der Ergebnistabelle

Für die spezifischen Werte der Replikationsstrategien sind im Moment vier Spalten in der Ergebnistabelle vorgesehen. Sollte diese Anzahl im späteren Gebrauch nicht mehr ausreichen, so kann sie einfach erhöht werden. Neben der Anpassung des für die Abfrage zuständigen Statements aus Kapitel 5.4.5 muss in der Klasse `DBInteraction` in der Methode `getScenario` der Wert von `numColumns` angepasst werden.

8.4 Webseite

Kompilieren Sie das Projekt in Netbeans mit der Konfiguration „Release“. Kopieren Sie dann die Dateien „Source/OptimalReplication/dist/Release/GNU-Linux-x86/optimalreplication“ und „WWW/OptimalReplication.php“ in den öffentlichen Ordner des Webservers, im Allgemeinen ist das der Ordner „public_HTML“.

Glossar

Nachfolgend sind noch einmal wesentliche Begriffe dieser Arbeit zusammengefasst und erläutert. Eine ausführliche Erklärung findet sich jeweils in den einführenden Abschnitten sowie der jeweils darin angegebenen Literatur. Das im Folgenden im Rahmen der Erläuterung verwendete Symbol $\sim$ bezieht sich jeweils auf den im Einzelnen vorgestellten Begriff, das Symbol $\uparrow$ verweist auf einen ebenfalls innerhalb dieses Glossars erklärten Begriff.

Brute-Force-Methode Die $\sim$ ist eine Möglichkeit zum Lösen von Problemen, bei der alle möglichen Fälle betrachtet werden. Sie wird eingesetzt, wenn kein effizienter Algorithmus existerit, der das Problem effizienter lösen kann.

C++ Die Programmiersprache, in der die Software geschrieben wurde.

Instanz Die $\sim$ einer $\uparrow$Replikationsstrategie hat eine bestimmte Konfiguration.
Von einer $\uparrow$Replikationsstrategie kann es mehrere $\sim$ geben, deren Konfigurationen unterschiedlich sind.

Intaktwahrscheinlichkeit Die Wahrscheinlichkeit, mit der der Rechner, auf dem das $\uparrow$Replikat gespeichert ist, funktioniert.

HTML $\sim$ steht für Hypertext Markup Language und ist eine Sprache zur Darstellung von Internetseiten.

Kosten Jeder Zugriff auf ein $\uparrow$Replikat verursacht $\sim$ von 1. Dieses gilt sowohl für einen $\uparrow$Lese- wie auch für einen $\uparrow$Schreibzugriff.

Leseoperation Eine $\sim$ besteht aus einem oder mehreren $\uparrow$Lesezugriffen auf verschiedene Replikate.

Lesezugriff Ein $\sim$ beschreibt das Lesen eines Replikates. Für einige $\uparrow$Replikationsstrategien können mehrere $\sim$ nötig sein, um eine $\uparrow$Leseoperation durchzuführen.

MySQL Ein weit verbreitetes Open-Source-Datenbankmanagementsystem.

PHP $\sim$ ist eine Scriptsprache die Anwendung bei dynamischen Internetseiten oder Internetanwendungen findet.

Replikat Ein $\sim$ ist die Kopie einer Datei, einer Funktion oder sonstigen Daten, das auf einem Rechner gespeichert ist und dessen Zugriffe durch eine $\uparrow$Replikationsstrategie geregelt werden.

Replikationsstrategie Eine $\sim$ beschreibt das Vorgehen, mit dem $\uparrow$Lese- und $\uparrow$Schreiboperationen die $\uparrow$Replikate lesen und schreiben. Vorgabe ist, dass beim Lesen immer ein aktuelles $\uparrow$Replikat gelesen wird. $\uparrow$Lese- und $\uparrow$Schreiboperationen müssen dementsprechend aufeinander abgestimmt sein.

Schreiboperation Analog zu $\uparrow$Leseoperation für das Schreiben.

Schreibzugriff Analog zu $\uparrow$Lesezugriff für das Schreiben eines Replikates

Szenario Ein $\sim$ beschreibt die Umgebung, in der die $\uparrow$Replikationsstrategie eingesetzt werden soll. Die wichtigsten Parameter sind die Anzahl der $\uparrow$Replikate und die $\uparrow$Intakwahrscheinlichkeit der Rechner.

Verfügbarkeit Die $\sim$ gibt an, mit welcher Wahrscheinlichkeit zwischen 0 und 1 ein ↑Lese- oder ↑Schreibzugriff ausgeführt werden kann. Je höher der Wert ist, desto eher gelingt der Zugriff. Die $\sim$ hängt von der Anzahl der ↑Replikate und der ↑Intaktwahrscheinlichkeit ab.

Abkürzungen

N	Anzahl der Replikate
p	Intaktwahrscheinlichkeit
cr	Kosten für einen Lesezugriff
cw	Kosten für einen Schreibzugriff
ar	Verfügbarkeit einer Leseoperation
aw	Verfügbarkeit einer Schreiboperation
LQ	Lesequorum
SQ	Schreibquorum
QR	Erwartete Lesezugriffe pro Stunde
QW	Erwartete Schreibzugriffe pro Stunde
maxC	Maximale Kosten pro Stunde
minAW	Minimale Leseverfügbarkeit
minAR	Minimale Schreibverfügbarkeit
ROWA	Read-One-Write-All-Strategie
WV	Weighted-Voting-Strategie

Abbildungen

Literatur

[BG84] BERNSTEIN, Philip A. ; GOODMAN, Nathan: An algorithm for concurrency control and recovery in replicated distributed databases. In: *ACM Trans. Database Syst.* 9 (1984), Dezember, Nr. 4, 596–615. http://dx.doi.org/10.1145/1994.2207. – DOI 10.1145/1994.2207. – ISSN 0362–5915

[CAA92] CHEUNG, S. Y. ; AMMAR, M. H. ; AHAMAD, M.: The Grid Protocol: A High Performance Scheme for Maintaining Replicated Data. In: *IEEE Trans. on Knowl. and Data Eng.* 4 (1992), Dezember, Nr. 6, 582–592. http://dx.doi.org/10.1109/69.180609. – DOI 10.1109/69.180609. – ISSN 1041–4347

[FKT91] FREISLEBEN, Bernd ; KOCH, Hans-Henning ; THEEL, Oliver: Designing Multi-Level Quorum Schemes for Highly Replicated Data. In: *Proc. of the 1991 Pacific Rim International Symposium on Fault Tolerant Systems, Kyoto, Japan*, 1991, S. 154–159

[Gif79] GIFFORD, David K.: Weighted voting for replicated data. In: *Proceedings of the seventh ACM symposium on Operating systems principles*. New York, NY, USA : ACM, 1979 (SOSP '79). – ISBN 0–89791–009–5, 150–162

[Koc94] KOCH, Hans-Henning: *Entwurf und Bewertung von Replikationsverfahren*, Dept. of Computer Science, University of Darmstadt, Germany, Diss., 1994

[MyS] MYSQL 5.1 REFERENCE MANUAL: *MySQL Connector/C++*. http://dev.mysql.com/doc/refman/5.1/en/connector-cpp.html. – zuletzt besucht: 13.10.2012

[Sto12] STORM, Christian: *Specification and Analytical Evaluation of Heterogeneous Dynamic Quorum-Based Data Replication Schemes*, Department of Computer Science, Carl von Ossietzky University of Oldenburg, Germany, Diss., 2012

[The93a] THEEL, Oliver: General Structured Voting: A Flexible Framework for Modelling Cooperations. In: *In Proc. of the 13th International Conference on Distributed Computing Systems*, IEEE, 1993, S. 227–236

[The93b] THEEL, Oliver E.: A General Framework for Modelling Data Replication Schemes. In: *Proceedings of the International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems*. San Diego, CA, USA : Society for Computer Simulation International, 1993 (MASCOTS '93). – ISBN 1–56555–018–8, 247–250

[You] YOUNG, Warren: *MySQL++*. http://tangentsoft.net/mysql++/. – zuletzt besucht: 13.10.2012

Index

Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg festgelegt sind, befolgt habe.

Oldenburg, den 14. November 2012

Christian Hülsen