RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

Aaron Taylor

Untersuchung verschiedener Datenlager für hochskalierende Monitoringsysteme und Entwicklung einer generischen Schnittstelle für multiple Datenlager

Diplomarbeit

HD-KIP-04-06

KIRCHHOFF-INSTITUT FÜR PHYSIK

Fakultät für Physik und Astronomie

Ruprecht-Karls-Universität Heidelberg

Diplomarbeit
im Studiengang Physik

vorgelegt von

Aaron Taylor
aus Heppenheim

April 2004

Untersuchung verschiedener Datenlager für hochskalierende Monitoringsysteme und Entwicklung einer generischen Schnittstelle für multiple Datenlager

Aaron Taylor

Die Diplomarbeit wurde ausgeführt am

Kirchhoff-Institut für Physik

unter der Betreuung von

Herrn Prof. Dr. Volker Lindenstruth

Untersuchung verschiedener Datenlager für hochskalierende Monitoringsysteme und Entwicklung einer generischen Schnittstelle für multiple Datenlager

Durch das umfassende Monitoring hochgradig verteilter Systeme entstehen große Datenmengen. Diese müssen gespeichert und unterschiedlichen Anwendungen zur Verfügung gestellt werden. In dieser Arbeit werden verschiedene Datenlagerungssysteme auf ihre Eignung zur Verwaltung der anfallenden Daten untersucht. Dazu werden neben Round-Robin Datenbanken und Dateisystem basierten Datenlagern auch native XML- und relationale Datenbank-Management-Systeme vorgestellt und getestet. Dabei zeigt sich, dass kein Datenlager alle, z. T. sehr unterschiedlichen Anforderungen gleichzeitig zufriedenstellend erfüllt.
Dieses Ergebnis motiviert die Entwicklung einer generischen Schnittstelle, welche die gleichzeitige Verwendung unterschiedlicher Datenlager ermöglicht. Eine entsprechende Software wird vorgestellt und ihre Leistungsfähigkeit getestet.

Analysis of different data storage systems adequate for high-scaling monitoring systems and development of a generic interface for multiple data storage systems

Extensive monitoring of highly distributed systems produces large amounts of data. This data must be stored und made available to different applications. In this thesis four different data storage systems are being examined for their suitability to manage the resulting data. Thereto file system based data storage, round-robin and native XML databases as well as relational database management systems are analysed and tested.
It becomes apparent that no data storage system can satisfy the very different requirements at the same time. Therefore a generic interface for different systems is being devolved. An appropriate software is presented and its performance tested.

Inhaltsverzeichnis

1 Einleitung

1.1 LHC

Die moderne Hochenergie- und Elementarteilchen-Physik versucht, den Aufbau und die Struktur von Materie besser zu verstehen. Dazu werden in Beschleunigern Teilchen zur Kollision gebracht. Da sich das Auflösungsvermögen mit steigender Teilchenenergie erhöht, werden immer leistungsfähigere Beschleuniger benötigt. Am Europäischen Zentrum für Elementarteilchenforschung (CERN)[1] [1] bei Genf (Schweiz) wird nach einer langjährigen Planungs- und Vorbereitungsphase zur Zeit mit **L***arge* **H***adron* **C***ollider* (LHC) [2] im Tunnel des bisherigen Beschleunigers **L***arge* **E***lectron* **P***ositron Collider* (LEP) [3] ein neuer Teilchenbeschleuniger installiert. Aktuellen Planungen zufolge wird LHC im Jahr 2007 den Regelbetrieb aufnehmen. Für die kommenden zwei Jahrzehnte wird der derzeit noch im Bau befindliche Beschleuniger einer der wichtigsten Messinstrumente moderner Hochenergiephysik sein.

LHC kann in zwei verschiedenen Modi betrieben werden. In Proton-Proton-Kollisionen mit einer geplanten Schwerpunktsenergie von bis zu 14 TeV werden die elementaren Bausteine der Hadronen – Quarks und Gluonen – untersucht. Dafür beschleunigen starke, hochfrequente elektrische Wechselfelder die Protonen in gegenläufiger Richtung, welche daraufhin im Inneren der geplanten Experimente zur Kollision gebracht werden. LHC soll im Proton-Proton-Betrieb Bedingungen für Reaktionen erzeugen, wie sie auch etwa 10^{-13} bis 10^{-14} s nach dem Urknall herrschten.

Alternativ kann LHC mit schweren Ionen betrieben werden. Dabei wird mit 1.150 TeV pro Kollision 30-mal mehr Energie als am Schwerionenbeschleuniger **R***elativistic* **H***eavy* **I***on* **C***ollider* (RHIC) [4] in Brookhaven (USA) freigesetzt, wo die derzeit höchsten Energien in Schwerionenkollisionen erreicht werden. Ziel der Forschung ist dabei, Zustände vor und während des so genannten QCD-Phasenübergangs[2], der etwa 10^{-6} s nach dem Urknall

[1] **C**onseil **E**uropéen pour la **R**echerche **N**ucléaire

[2] Ein von der (Gitter-) Quantenchromodynamik (QCD) vorhergesagter Phasenübergang zwischen Quark-Gluon- und Hadron / Resonanz-Materie

stattfand, zu untersuchen, bei dem sich die Hadronen aus den ursprünglich vorhandenen freien Quarks und Gluonen bildeten.

An der Entwicklung und dem Bau des Beschleunigers aber auch der vier Experimente arbeiten in weltweiten Kooperationen weit über 5 000 Wissenschaftler aus 46 Nationen. Damit ist LHC das größte gemeinschaftliche Projekt der physikalischen Grundlagenforschung.

1.1.1 Die LHC-Experimente

Am LHC werden vier große Experimente aufgebaut: **A T**orroidal **L**HC **A**ppartu**S** (ATLAS) [5], **C**ompact **M**uon **S**olenoid (CMS) [6], **A L**arge **I**on **C**ollider **E**xperiment (ALICE) [7] und LHCb [8]. Mit ATLAS und CMS sollen in erster Linie Proton-Proton-Kollisionen untersuchen werden. Man erwartet, z. B. die Frage nach dem Higgs-Boson endgültig klären zu können. Das durch das Standardmodell vorhergesagte, bis jetzt allerdings experimentell nicht eindeutig nachgewiesene Higgs-Boson ist das Teilchen, das durch Interaktion mit anderen massiven Teilchen (z. B. Elektronen, Gluonen, Quarks) diesen ihre Masse verleiht. Das Standardmodell erwartet dabei eine Masse für das Higgs-Boson zwischen 100 $\mathrm{GeV/c^2}$ und 1 $\mathrm{TeV/c^2}$ [9]. Da LHC diesen Energiebereich vollständig abdeckt, ist sichergestellt, dass das Higgs-Boson auf jeden Fall gefunden werden kann, sofern es dem Standardmodell genügt. Weiterhin hofft man, neue Phänomene zu entdecken, die von Theorien jenseits des Standard-Modells vorausgesagt werden [10]. Zu diesen gehört z. B. die Theorie der Supersymmetrie (SUSY) [11], welche u. a. so genannte „Superpartner“ für jedes fundamentale Teilchen vorhersagt (z. B.: Partner des Myons wäre dann das Smyon). Bis heute wurde keine direkte empirische Evidenz für die Existenz dieser letzten, theoretisch noch möglichen Symmetrie[3] gefunden.

ALICE ist zur Untersuchung von Schwerionen-Kollisionen entwickelt worden. Die bei solchen Schwerionenreaktionen auftretenden Wechselwirkungen der Materie finden bei extremen Energiedichten (100-mal größer als in normaler Materie in Atomkernen) statt. Unter normalen Bedingungen trägt die Confinement-Hypothese der Tatsache Rechnung, dass Quarks und Gluonen in der Natur nicht einzeln beobachtet werden können, sondern hadronisieren, d. h. zu Hadronen binden. Bei den durch die Kollision erreichten Energiedichten wird allerdings ein neuer Materiezustand, das Quark-Gluon-Plasma, erreicht. Darunter versteht man die Koexistenz von Quarks und Gluonen ohne Bildung von Hadronen. Mit dem ALICE-Detektor soll nun das Quark-Gluon-Plasma genauer studiert werden, von dessen Verständnis

[3] Alle anderen theoretisch möglich erscheinenden Symmetrien wurden entdeckt.

man hofft, Kernfragen der Quantenchromodynamik beantworten zu können, sowie ein tieferes Verständnis des Confinements zu erhalten.

Mit LHCb wird schließlich ein Experiment entwickelt, das u. a. die Untersuchungen der CP-Verletzung im System der B-Mesonen maßgeblich beeinflussen soll. Die CP-Verletzung wird durch die Eigenschaften der Cabibbo-Kobayashi-Maskawa-Matrix (CKM) im Standardmodell erklärt. Die Parameter der CKM-Matrix sind dabei allerdings noch experimentell zu bestimmen. Eine entscheidende Schwierigkeit bisheriger Experimente mit B-Mesonen ergibt sich aus der physikalischen Tatsache, dass die zur Messung der CP-Verletzung notwendigen Zerfälle relativ selten sind. Um den statistischen Einfluss zu reduzieren, sind wiederum sehr hohe Ereigniszahlen erforderlich, wie sie bei LHC erzeugt werden (10^{12} Paare von B-Mesonen und Anti-B-Mesonen pro Jahr). LHCb wird somit eine Präzisionsmessung der Parameter der CKM-Matrix ermöglichen.

1.1.2 Die Datenflut

LHC ist somit eines der größten wissenschaftlichen Messinstrumente weltweit und ermöglicht eine komplexe Grundlagenforschung in der Teilchenphysik. Dadurch allerdings, dass die physikalisch relevanten Kollisionen (d. h. Kollisionen mit noch nicht gut verstandenen Reaktionen) mehr als 100 mal seltener sind, als die so genannten Standardereignisse bekannten physikalischen Ursprungs, muss man mit einem hohen Untergrund kämpfen. Um den statistischen Einfluss zu reduzieren und aussagekräftige Messwerte zu erhalten, wird eine hohe Anzahl an Kollisionen erzeugt. Dabei entstehen wiederum sehr viele Teilchen (über 3000 Teilchen in 25 ns), die in den Detektoren erfasst und verarbeitet werden müssen. Pro Sekunde hat man es daher mit gut 10^{11} Teilchen oder aber mit Datenraten in der Größenordnung vieler TB/s zu tun [10]. Triggersysteme direkt hinter den Detektoren reduzieren diese Datenrate deutlich (um mehrere Größenordnungen), dennoch stellt die Aufzeichnung und Analyse der verbleibenden Daten neue Herausforderungen an Rechenleistung und Datenspeicherung.

1.2 Konventionelle Ansätze zur Lösung rechenintensiver Probleme

Um rechenintensive Probleme zu lösen, werden z. Z. in der Regel monolithische Supercomputer[4] oder Cluster verwendet. Als Supercomputer bezeichnet man oft Rechner, die aus vielen parallel arbeitenden Prozessoren (engl.: *Central Processing Unit* (CPU)) bestehen, so dass Rechengeschwindigkeiten erreicht werden, die mit einem Prozessor allein mit heutiger Designtechnik nicht zu realisieren wären. Supercomputer sind für das *High Performance Computing* (HPC) Paradigma prädestiniert. Unter HPC versteht man, dass jeder einzelne Prozess eine möglichst kurze Laufzeit benötigt und dazu auf mehrere Prozessoren verteilt wird. Durch die Aufteilung eines Prozesses in mehrere Teilprobleme wird dabei u. U. Kommunikation zwischen den einzelnen CPUs nötig, da diese z. B. auf einen gemeinsamen Datenbestand zurückgreifen und von einander abhängige Aufgaben sequentiell ausführen müssen. Durch die geringe Latenz der sehr schnellen Netzwerke zwischen den Prozessoren eines Supercomputers ist solche Kommunikation effizient möglich.

Im Gegensatz zu den Supercomputern stehen so genannte Cluster. Darunter versteht man einen räumlich konzentrierten, meist homogenen Verband von zunächst unabhängigen Computern (Knoten genannt). Bei solchen Knoten handelt es sich oft um handelsübliche Personal Computer (PC). Diese haben in den letzten 15 Jahren einen enormen Zuwachs an Leistung bei gleichzeitiger Senkung der Kosten pro Einheit erfahren. Im Vergleich dazu haben sich Supercomputer nicht so schnell entwickelt. Neben ihrer günstigen Anschaffung zeichnen sich PCs durch ihre weite Verbreitung und der damit verbundenen guten Verfügbarkeit von Ersatzteilen aus. Diese Faktoren reduzieren die Gesamtkosten[5] für einen Cluster, auf welche besonders im industriellen und wirtschaftlichen Umfeld geachtet wird.

Cluster können mit unterschiedlicher Zielsetzung implementiert werden. Das in diesem Zusammenhang interessante Cluster Computing in so genannten Beowulf-Clustern [12] bedient sich dabei vieler Rechner, hauptsächlich „Hardware von der Stange“ (engl: Commodity Off The Shelf (COTS)), zur Lösung rechenintensiver Aufgaben. Die Knoten in Beowulf-Clustern werden dabei allerdings nicht als Arbeitsplatzrechner genutzt, sondern stehen nur für die Ausführung von Prozessen innerhalb des Clusters zur Verfügung. Dieser auf hohe Perfor-

[4] Eng verbunden mit dem Begriff Supercomputer ist die nach ihrem Gründer Seymour Cray benannte Firma Cray (mittlerweile von Silicon Graphics übernommen), welche die ersten Supercomputer in den 70er Jahren des 20. Jahrhunderts herstellte.

[5] Auch als *Total Cost of Ownership* (TCO) bekannt. Darunter versteht man die Summe der Kosten für Anschaffung, Installation, Wartung und Betrieb.

mance optimierte Cluster-Typ wurde erstmals 1994 gebaut, um für die NASA[6] Simulationen mit großen Datenmengen durchzuführen.

Ein weiteres entscheidendes Kriterium für die Entstehung und Verbreitung von Clustern ist der schnelle Fortschritt in den Netzwerk-Technologien in den letzten Jahren. Allein durch ihren Aufbau sind Cluster in erster Linie für *High Throughput Computing* (HTC) geeignet. Bei HTC liegt der Schwerpunkt im Gegensatz zu HPC auf einer möglichst hohen Rechenleistung über längere Zeiträume, d. h., dass nicht die Laufzeit sondern der Durchsatz der Prozesse entscheidend ist. Da Cluster aus unabhängigen Knoten bestehen, kann auf jedem Knoten ein eigener Prozess abgearbeitet werden. Erst die höhere Performance von Netzwerken ermöglicht es, Cluster nach dem HPC-Paradigma zu betreiben [14]. Weiterhin gestatten schnelle Netzwerke eine höhere Auslastung der Ressourcen (Rechenleistung, Speicherplatz) einzelner Knoten, indem man nicht benötigte Kapazitäten anderen Knoten im Cluster zur Verfügung stellt.

Ob ein monolithischer Supercomputer im Einzelfall einem Cluster vorgezogen werden sollte, hängt somit u. a. von der Parallelisierbarkeit des verwendeten Lösungsalgorithmuses ab. Bei der Verwendung eigenständiger Einheiten, wie im Fall eines Clusters, sollte das zu bearbeitende Problem in möglichst viele von einander unabhängige Teilaufgaben zerlegt werden. Dies ist bei sequentiellen Algorithmen jedoch nicht immer möglich. Die Parallelisierbarkeit eines Algorithmuses hängt stark von dessen Struktur ab, so dass für eine effizientere Ausführung oft auch die Entwicklung neuer „parallelerer“ Algorithmen erforderlich ist [15].

Die Daten und die Lösungsalgorithmen der LHC Experimente werden eine vergleichsweise hohe Parallelisierbarkeit aufweisen. Allerdings werden selbst nach dem Einsatz von Triggersystemen sehr hohe Datenraten erwartet. Die Leistungsfähigkeit herkömmlicher Supercomputer und Cluster reicht bei weitem nicht aus, um die entstehenden Daten aufzunehmen oder zu verarbeiten. Hier soll Grid-Computing einen Ausweg bieten.

1.3 Was ist Grid-Computing?

Unter einem Grid versteht man einen räumlich verteilten Rechnerverbund vieler unterschiedlicher Einheiten (Cluster, einzelne Knoten) verschiedener Organisationen, welche über Weitverkehrsnetze (z. B. das Internet) zusammen genutzt werden, um rechenintensive Applika-

[6] National Aeronautics and Space Administration [13]

tionen auszuführen. Im Unterschied zu Clustern sind Grids daher in der Regel nicht homogen und werden typischerweise auch nicht von einer zentralen Stelle aus verwaltet [16].

Das Fernziel der Entwicklung von Grids ist die Schaffung einer zusätzlichen, allgemein zugänglichen Ressource ähnlich der Ressource elektrische Energie. Bei der elektrischen Energie ist der Durchbruch hin zur allgemein verfügbaren Ressource erst durch die Schaffung von Energieversorgungsnetzen[7] gelungen. Bis dahin besaß jeder, der Elektrizität nutzen wollte, seinen eigenen Generator. Überschüssige Energie ging (wirtschaftlich gesehen) verloren und bei erhöhtem Energiebedarf führte kein Weg an der Anschaffung zusätzlicher Generatoren vorbei.

Heute kann jeder, egal ob Privathaushalt oder Industrie, die gewünschte elektrische Leistung einfach „aus der Steckdose“ beziehen, ohne selbst über Generatorleistung zu verfügen. Genauso soll es später einmal möglich sein, Rechenleistung unabhängig von den eigenen Rechenkapazitäten zu verwenden. Dabei spielt es – genau wie bei der Nutzung von Strom – keine Rolle, wo die benötigte Rechenleistung herkommt, noch wie die Verteilung der Rechenleistung tatsächlich vonstatten geht. Ausreichend für die Nutzung eines solchen Grids sollte, analog zur Steckdose, ein passendes Interface sein. Die aktuelle Forschung versucht somit, durch die Schaffung von Computer-Grids ähnliche Fortschritte bei der Verfügbarkeit von Rechenleistung zu erzielen, wie durch die Elektrifizierung.

1.4 Probleme verteilter Architekturen

Grids aber auch Cluster haben einen erheblichen strukturellen Nachteil gegenüber Supercomputern. Verteilt man rechenintensive Applikationen auf viele unabhängige Einheiten, so muss unweigerlich auch ein Konzept zur Fehlerbehandlung vorhanden sein. Jeder vom Anwender in das System eingebrachte Prozess wird, falls möglich, zur schnelleren Berechnung in viele kleinere Prozesse („Jobs“) aufgeteilt und an die einzelnen Einheiten verteilt werden. Auf jeder Einheit können dann völlig unterschiedliche Probleme auftreten. So können Prozesse aufgrund eines Programmfehlers außer Kontrolle geraten oder abbrechen. Ergebnisse sind dann vielleicht unvollständig oder fehlerhaft. Weiterhin können die einzelnen unabhängigen Einheiten selbst oder Teile davon ausfallen. Eine übliche Methode zur Quantifizierung der Ausfallsicherheit einzelner Bauteile ist die so genannte *MeanTime Between Failure*[8]

[7] engl.: power grid
[8] engl.: mittlere Zeit zwischen zwei Ausfällen

(MTBF). So haben gute Lüfter z. B. eine MTBF von 70 000 Stunden. Bei 7 000 verwendeten Lüftern fällt somit im Durchschnitt alle 10 Stunden ein Lüfter aus! Nicht zuletzt muss man die Auslastung jeder Einheit kennen, um eine optimierte Verteilung der Last auf die einzelnen Einheiten[9] vornehmen und damit die Effizienz des gesamten Systems steigern zu können.

Monolithische Supercomputer werden mit besonderem Blick auf Ausfallsicherheit optimiert, dennoch können natürlich auch Probleme wie z. B. Programmfehler und Hardwareausfälle auftreten. Hochleistungsrechner sind jedoch (zumindest theoretisch) meist relativ einfach zu überwachen, da das Gesamtsystem von einem Hersteller konzipiert wird. Dieser stimmt alle Hardwarekomponenten aufeinander ab und passt die Systemsoftware (vor allem das Betriebssystem) speziell an diese Hardware an. Entsprechend existieren dann in der Regel proprietäre Überwachungsfunktionen.

Für ein Cluster oder Grid mit sehr vielen, z. T. völlig unterschiedlichen Einheiten ist eine solche Überwachung a priori nicht vorgesehen. Handelsübliche PCs, wie sie in Clustern oft Verwendung finden, werden zudem nicht primär auf Ausfallsicherheit optimiert. Daher ist zu erwarten, dass ständig irgendwelche Einheiten nicht wie erwünscht funktionieren und dass man von deren Zustand ohne eine Überwachung keine oder nur unzureichende Kenntnis erlangt. Da man Fehler in der Regel nur bei Kenntnis der Ursache beseitigen kann, wäre es nicht überschaubar, ein solch verteiltes System manuell, d. h. ohne automatisierte Überwachungsfunktionen, zu kontrollieren. Der Betreiber eines Clusters oder Grids muss daher ein geeignetes Werkzeug – ein Monitoringsystem – bereitstellen.

Monitoringsysteme haben somit die primäre Aufgabe, Daten über den Zustand einer jeden Einheit, und damit über das gesamte System, zu sammeln. Dabei ist es sinnvoll, viele Zustandsinformationen über die betreffende Einheit zur Verfügung zu stellen, da u. U. die Kombination verschiedener, augenscheinlich nicht korrelierter Daten das frühzeitige Erkennen von Problemen ermöglicht, noch bevor sie außer Kontrolle geraten und die Stabilität bzw. die Verwendbarkeit beeinträchtigen.

Als weitere Gründe für die Erfassung von Zustandsparametern verteilter Systeme sind, neben der Gewährleistung der Verwendbarkeit, die Verfügbarkeit von Statistiken und nicht zuletzt von Daten zum Zwecke der Abrechnung zu nennen. Da Grids in der Regel organisationsübergreifend gebildet werden, sind auch entsprechende Gebühren für die Bereitstellung von Rechenleistung an andere zu erwarten.

[9] load-balancing

1.4.1 Datenvolumina

Mit der Ansammlung vieler Daten durch Monitoringsysteme in einem verteilten System ist die Problematik der Datenspeicherung und -vorhaltung verbunden. Bei einer rudimentären Überwachung eines Knotens entsteht im Durchschnitt eine Datenrate von deutlich unter 1 kB/s[10], welches bei intensiverem und umfassenderem Monitoring zwangsläufig steigt. Sammelt man alle Daten zentral an einem Rechner, so wird dessen Datenannahmegeschwindigkeit und der Netzwerkverkehr hin zu diesem Rechner bei großen Verbünden zu einem Flaschenhals. Die für Cluster und Grids unbedingt notwendige Skalierbarkeit ist also nicht gegeben. Zudem reagiert das gesamte System kritisch auf den Ausfall eines solchen zentralen Rechners (engl.: single point of failure). Folglich müssen die aus dem Monitoring gewonnenen Daten verteilt werden.

1.4.2 Fehlertoleranz

Registriert das Monitoringsystem einen Fehler, könnten die Administratoren darauf reagieren. Dies ist bei einem Netzwerk, bestehend aus mehreren tausend Einheiten, sehr aufwändig und daher oft inpraktikabel, zumal Grids in der Regel (geographisch) verteilte Systeme darstellen. Viele übliche (Software-) Fehler können mit einer flexiblen, automatisierten Fehlerbehandlung (Fault Tolerance (FT) [17]) abgefangen werden. Diese überwacht festgelegte, vom Monitoringsystem erfasste Zustandsparameter und kann Diagnosen erstellen sowie evtl. geeignete Reparaturmaßnahmen beim Auftreten von Standardproblemen vornehmen. Nur Fehler, die die FT nicht beheben kann, erfordern dann menschliche Intervention.

1.5 Gliederung dieser Arbeit

Wie man sieht, werden viele verschiedene Personen (Administratoren, Benutzer, Eigentümer, ...) und unterschiedliche Anwendungen (Fehlertoleranz, Statistik, ...) auf die Daten eines Monitoringsystems zurückgreifen. Deshalb muss insbesondere das Format, in dem Daten zwischen den einzelnen Instanzen ausgetauscht werden sollen, einigen Bedingungen (z. B. Plattformunabhängigkeit) genügen. In Kapitel 2 wird daher eine Sprache vorgestellt, welche gut geeignet ist, um die im Rahmen eines Cluster- und Grid-Monitorings erhobenen Daten darzustellen.

[10] Eine aufgeschlüsselte Abschätzung wird in Tabelle 4.1 vorgenommen.

Ziel dieser Arbeit ist die Untersuchung der Eignung verschiedener Systeme zur Datenlagerung im Rahmen umfassenden Monitorings hochgradig verteilter Systeme wie Cluster und Grids. Dazu wird in Kapitel 3 zuerst eine Einführung in Monitoringsysteme gegeben und die Funktionsweise einiger ausgewählter Systeme erläutert. Prinzipiell dafür in Frage kommende und verwendete Datenlager werden in Kapitel 4 vorgestellt und im darauf folgenden Kapitel auf ihre Eigenschaften hin untersucht. Diese Ergebnisse motivieren die Entwicklung einer generischen Schnittstelle für unterschiedliche Datenlager, welche in Kapitel 6 vorgestellt wird. Die wesentlichen im Rahmen der Arbeit gewonnenen Erkenntnisse werden im letzten Kapitel zusammengefasst und es wird ein kurzer Ausblick gegeben.

2 XML

2.1 Einleitung

In diesem Kapitel wird ein kurzer Überblick über die Auszeichnungssprache *e***X***tensible* **M***arkup* **L***anguage* (XML) vermittelt. Dabei wird nur auf einige wesentliche Eigenschaften eingegangen[1]. Ziel dieser Einführung ist es zu zeigen, dass die Verwendung von XML für Monitoringsysteme viele Vorteile bietet. Daher werden lediglich Fähigkeiten, die im Rahmen eines Cluster- und Grid-Monitorings wichtig erscheinen, näher beleuchtet.

XML ist eine Metasprache für das Definieren beliebiger Dokumenttypen. Sie entstand als vereinfachte Form der *Standard Generalized Markup Language*[2] (SGML), einem Standard zur Beschreibung von Dokumenten, welcher 1986 von der *International Standardization Organization* (ISO) verabschiedet wurde[3]. Eine der wesentlichen Eigenschaften von XML ist, dass damit formulierte Inhalte stets für den Menschen lesbare Textdokumente bleiben (engl.: human-readable).

XML bietet die Möglichkeit, beliebige strukturierte Informationen mit einem kleinen Satz einfacher Regeln einheitlich darzustellen. Die Vorteile bei Verwendung von XML zur Darstellung liegen hauptsächlich in der großen Flexibilität und der weiten Verbreitung. Durch letztere existieren im Umfeld der XML-Spezifikation viele weitere etablierte und nützliche Standards (z. B. DOM, SAX, SOAP, Hyperlinks, ...). Jedes XML-basierte Datenformat kann somit von den Vorteilen einer Reihe nützlicher Zusatz-Standards profitieren, so dass Kompatibilität und Möglichkeiten für den Datenaustausch deutlich einfacher zu realisieren sind, als bei proprietären Lösungen.

[1] Für eine umfassende Darstellung wird auf [18] verwiesen.
[2] HTML wurde in den ersten Versionen durch SGML definiert.
[3] ISO 8879: 1986

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<messwerte>
  <messnr station="hd-mensa-inf" nummer="25322">
    <messzeit type="datum"> 15.04.2004 </messzeit>
    <messzeit type="uhrzeit"> 12:21:22 </messzeit>
    <temperatur messgeraet="ACTR5" einheit="Grad Celsius">
      25
    </temperatur>
    <sonne> ja </sonne>
    <bewoelkung> wolkenfrei </bewoelkung>
  </messnr>
</messwerte>
```

Abbildung 2.1: Exemplarische Darstellungen eines Datensatzes einer Wetterstation in XML

2.2 Prinzipieller Aufbau

Im Prinzip wird die abzulegende Information zuerst in ihre kleinsten (atomaren) Teilinformationen zerlegt. Diese werden nun klassifiziert und durch so genannte Tags[4] ausgezeichnet. Die einzelnen Teilinformationen bezeichnet man als Elemente, denen zusätzlich Attribute mit Wertzuweisungen zugeordnet werden können, welche sie näher definieren. Die Elemente werden nun in ihrer logischen (informationstragenden) Reihenfolge hierarchisch so verschachtelt, dass sie auf eine Baumstruktur abbildbar sind. In der Informatik beschreibt der Begriff eines Baums eine Datenstruktur, welche an den Aufbau eines biologischen Baums angelehnt wurde, der sich ausgehend vom Stamm immer weiter verzweigt. Ein Baum der Informatik besteht dabei aus Knoten, die untereinander hierarchisch verbunden sind, also selbst Unterelemente (so genannte Kindknoten) besitzen können, wobei die Knoten, in denen der Baum endet, als Blätter bezeichnet werden. Unter der Wurzel (engl.: root) versteht man den Ursprung aller Knoten und Blätter[5]. Dem Umstand, dass in XML Elemente, Attribute und Struktur des Baumes ansonsten völlig frei gewählt werden können, trägt das „extensible“ im Namen Rechnung.

Zum besseren Verständnis wird folgendes Beispiel betrachtet: Die Wetterstation „Heidelberg Zentralmensa“ möchte die Messwerte ihrer 25322. Messung in XML abspeichern. Angenommen, die Temperatur betrage 25 °C (gemessen mit dem Thermometer ACTR5), die Sonne scheine und der Himmel sei wolkenfrei. Der Zeitpunkt der Messung sei 15.04.2004 um 12:21:22 Uhr. Abbildung 2.1 zeigt eine mögliche Darstellung dieses Datensatzes.

[4] engl.: Etikett
[5] Graphentheoretisch betrachtet ist ein Baum ein zusammenhängender, gerichteter nicht-zyklischer Graph.

Jedes XML-Dokument sollte mit einer XML-Deklaration beginnen. In dieser werden üblicherweise die XML-Version und der verwendete Zeichensatz (Kodierung) als Attribute angegeben. Durch die Unterscheidung verschiedener XML-Versionen ist die Abwärtskompatibilität gewährleistet. Die Angabe des verwendeten Zeichensatzes stellt die richtige Interpretation jedes Zeichens sicher, da je nach Zeichensatz verschiedene Zeichen und unterschiedliche Kodierungen für das gleiche Zeichen verwendet werden können. Somit können XML-Dokumente u. a. auch in Unicode[6] und in anderen ISO-Zeichensätzen geschrieben werden[7].

2.3 Dokumenttypen

Im vorhergehenden Abschnitt wurde anhand eines Beispiels der Aufbau eines XML-Dokuments vorgestellt. Allerdings will man oft die Verwendung auf einige ausgezeichnete der sonst frei wählbaren Elemente einschränken. Für Dokumente, die viele strukturelle Gemeinsamkeiten aufweisen, können dazu so genannte Dokumenttypen definiert werden, welche sich durch ein eigenes Vokabular und eine eigene Grammatik auszeichnen. Eine Festlegung auf einen Dokumenttyp verpflichtet einerseits den Verfasser eines Dokuments, lediglich die vorgegebenen Verschachtelungen und erlaubten Elemente zu verwenden, und erleichtert andererseits dem Lesenden durch die vorgegebene Struktur die Interpretation. Man kennt für XML zwei Möglichkeiten, um Dokumenttypen zu definieren:

DTD In einer *Document Type Definition* (DTD) werden die einzelnen in einem XML-Dokument erlaubten Elemente, ihre möglichen Attribute sowie ihre Reihenfolge festgelegt. Diese so genannten Vereinbarungen müssen vollständig sein, d. h. ein dieser DTD genügendes XML-Dokument darf ausschließlich in der DTD definierte Elemente enthalten sowie nur Verschachtelungen verwenden, die die DTD explizit vorsieht. Unter einer DTD versteht man somit die Gesamtheit der Richtlinien, die für das Organisieren von Daten innerhalb eines Dokuments gelten (Abbildung 2.2).

DTDs weisen allerdings auch einige Unzulänglichkeiten auf. So kann man z. B. keine Aussagen und Einschränkungen über den Inhalt bzw. den Datentyp eines Elements

[6] Unicode ist in der Standardform ein 16-Bit breiter Zeichencode, mit dem allerdings durch Erweiterungsmechanismen mehr als eine Millionen Zeichen dargestellt werden können. Unicode strebt die plattformunabhängige, möglichst vollständige Erfassung und Darstellung aller bekannten Zeichen aus gegenwärtigen und vergangenen Schriftkulturen an (vgl. [19]).

[7] In Abbildung 2.1 wurde zum Beispiel der Latin1-Zeichensatz, der für westeuropäische Sprachen optimiert wurde und u. a. die Verwendung von Umlauten erlaubt, verwendet.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!ELEMENT messwerte (messnr+)>

<!ELEMENT messnr (messzeit+, temperatur, sonne?, bewoelkung?)>
<!ATTLIST messnr station CDATA #REQUIRED
                  nummer CDATA #REQUIRED>

<!ELEMENT messzeit (#PCDATA)>
<!ATTLIST messzeit type (datum|uhrzeit)>

<!ELEMENT temperatur (#PCDATA)>
<!ATTLIST temperatur messgeraet CDATA #REQUIRED
                        einheit CDATA #IMPLIED)>

<!ELEMENT sonne (#PCDATA)>
<!ELEMENT bewoelkung (#PCDATA)>
```

Abbildung 2.2: Beispiel einer DTD für die in Abbildung 2.1 beschriebene XML-Datei. Durch `ELEMENT` wird ein Element mit den zugelassenen bzw. erforderlichen untergeordneten Elementen und / oder der Datentyp (`#PCDATA`) festgelegt. `ATTLIST` definiert für ein bekanntes Element die zugeordneten Attribute und legt u. a. fest, ob diese obligatorisch oder optional sind.

vornehmen. Auch Kommentare sind nicht vorgesehen. Weiterhin sind die DTDs selbst nicht in XML verfasst, wodurch die meisten der mit XML verbundenen Vorteile verloren gehen. Da XML als einer ihrer großen Stärken jedoch in der Lage ist, jeden strukturierten Inhalt darstellen zu können, sollte es auch möglich sein, einen Weg zur Definition eines XML-Dokumenttyps zu finden, welcher die anderen bereits aufgeführten Nachteile von DTDs vermeidet. Aus dieser Motivation heraus entstanden XML-Schemata.

XML-Schemata Ein XML Schema ermöglicht Dokumenten-Definitionen, welche über die Funktionalität von DTD hinausgehen. Zu den wichtigsten Eigenschaften zählt, dass XML-Schemata selbst in XML beschrieben werden[8]. Weiterhin können mit Hilfe eines XML-Schemas durch Festlegen von Datentypen genaue Angaben und Einschränkungen zum Inhalt der Elemente gemacht werden. Dabei stehen über 30 verschiedene vordefinierte Datentypen zur Verfügung. Daneben können vom Entwickler eines XML-Schemas jederzeit neue Datentypen vereinbart werden. Abbildung 2.3 zeigt ein Beispiel für ein einfaches XML-Schema.

[8] Der Vollständigkeit halber wird erwähnt, dass allerdings DTDs existieren, welche die Elemente der Schemadokumente beschreiben. Allerdings ist das für die Praxis unbedeutend, da man in der Regel auf diese DTDs nicht explizit zurückgreift.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
..<xsd:complexType name="messwerte" type="xsd:string">
.. . <xsd:sequence>
.. . . <xsd:element name="messnr" type="xsd:string"
.. . .                 minOccurs="0" maxOccurs="unbounded">
.. . . . .<xsd:complexType>
.. . . . . . <xsd:sequence>
.. . . . . . . <xsd:element name="messzeit" type="xsd:string">
.. . . . . . . . ..<xsd:complexType>
.. . . . . . . . .. . <xsd:attribute name="type" type="xsd:string"/>
.. . . . . . . . ..</xsd:complexType>
.. . . . . . . </xsd:element>
.. . . . . . . <xsd:element name="temperatur" type="xsd:float">
.. . . . . . . . ..<xsd:complexType>
.. . . . . . . . .. . <xsd:attribute name="messgeraet" type="xsd:string"/>
.. . . . . . . . .. . <xsd:attribute name="einheit" type="xsd:string"/>
.. . . . . . . . ..</xsd:complexType>
.. . . . . . . </xsd:element>
.. . . . . . . <xsd:element name="sonne" type=""/>
.. . . . . . . <xsd:element name="bewoelkung" type=""/>
.. . . . . . </xsd:sequence>
.. . . . . . <xsd:attribute name="station" type="xsd:string"/>
.. . . . . . <xsd:attribute name="nummer" type="xsd:decimal"/>
.. . . . . .</xsd:complexType>
.. . . </xsd:element>
.. . </xsd:sequence>
..</xsd:complexType>
</xsd:schema>
```

Abbildung 2.3: Beispiel eines XML-Schemas für die in Abbildung 2.1 beschriebene XML-Datei. Zur besseren Übersicht wurden die verschiedenen Stufen der Verschachtelung farblich hervorgehoben. Alle Elemente derselben Farbe sind logisch in der gleichen Hierarchie-Ebene. (Die in dem Text abgebildeten Hilfspunkte dienen zur besseren Visualisierung der unterschiedlichen Ebenen)

2.4 Wohlgeformte und gültige XML-Dokumente

Trotz der großen Flexibilität von XML müssen dennoch einige Regeln eingehalten werden, damit ein Dokument als ein XML-Dokument bezeichnet werden kann. So muss die Syntax der XML-Spezifikation des W3C [18] entsprechen. Insbesondere gibt es einige markante Eigenschaften, die (z. B. im Vergleich zu HTML) beachtet werden müssen:

- Es gibt genau ein abgeschlossenes Element auf der obersten Hierarchie-Ebene (Wurzelelement), dass alle anderen Elemente (Knoten / Blätter) enthält. Die XML-Deklaration ist dabei nicht als Element zu verstehen und steht somit außerhalb des Wurzelelements.

- Elemente in einem XML-Dokument müssen stets so verschachtelt werden, dass das gesamte Dokument als Baum darstellbar ist. Das bedeutet, dass Elemente stets in der umgekehrten Reihenfolge ihres Auftretens geschlossen werden müssen. So ist die Verschachtelung von
  `<sonne> 1 <helligkeit> 100 </sonne> </helligkeit>`
  illegal. Korrekt muss es in diesem Falle z. B. heißen:
  `<sonne> 1 <helligkeit> 100 </helligkeit> </sonne>`
- Jedes Element muss geschlossen werden. Diese Forderung hängt stark mit der zuvor gestellten zusammen, da sonst die Baumstruktur nicht immer eindeutig ist. Leere Elemente können allerdings durchaus auftauchen (`<sonne></sonne>` und können sogar durch einen abschließenden "/" vor der schließenden Klammer abgekürzt werden (`<sonne />`)
- XML unterscheidet zwischen Groß- und Kleinschreibung (engl.: case-sensitive).
- Attribute müssen immer einen Wert haben (Beispiel: `<temperatur Celsius>` ist falsch, korrekt muss es z. B. `<temperatur einheit="Celsius">` heißen). Werte müssen zudem mit Anführungszeichen umschlossen sein.

Erfüllt ein Dokument alle syntaktischen Kriterien gemäß der XML-Spezifikation des W3C, so handelt es sich um ein **wohlgeformtes** Dokument. Ist einem Dokument eine DTD oder ein Schema zugeordnet (existiert also eine Eingrenzung des Vokabulars und / oder der Syntax), so kann man prüfen, ob das Dokument **gültig** (engl.: valid) im Sinne der DTD bzw. des Schemas ist. Daraus ergibt sich sofort, dass Wohlgeformtheit eine notwendige, jedoch nicht hinreichende Bedingung für Gültigkeit ist.

XML bietet noch viele weitere Möglichkeiten[9], auf die hier allerdings nicht weiter eingegangen werden soll, da sie im Folgenden keine Verwendung finden.

2.5 XML-Parser

Will man in einer Anwendung – im Falle dieser Arbeit in einem generischen Interface für Datenlager – XML-Dateien verarbeiten, so greift man in der Regel auf einen XML-Parser zurück. Darunter versteht man ein Programm, das eine XML-Datei einliest und Inhalt und Struktur über definierte Schnittstellen zugänglich macht. Ein solcher XML-Parser sollte die

[9] Bsp.: Verwendung von Namensräumen ähnlich dem Namensraumkonzept unter *C++* [20] und *Java* [21]

XML-Datei mindestens auf Wohlgeformtheit überprüfen, im Falle der Einschränkung auf Dokumenttypen allerdings auch die Gültigkeit an Hand einer DTD oder eines Schemas validieren. Durch die weite Verbreitung von XML hat sich eine Vielzahl von XML-Parsern für fast jede Programmiersprache entwickelt. So stehen allein für die Programmiersprache *C++* u. a. folgende, frei verfügbare, gut gewartete und betreute Parser zur Verfügung: *eXpat* [22], *libxml2* [23] aus dem *GNOME*-Toolkit [24], *Xerces-C++* [25], *XML Parser v2* von Oracle [26], *TinyXML* [27] und viele weitere. Die meisten auf dem Markt verfügbaren Parser benutzen eines von zwei üblichen Verfahren, um den Zugriff auf den Inhalt eines Dokuments zu ermöglichen. Diese werden im Folgenden vorgestellt.

2.5.1 Document Object Model

Das erste Verfahren – das Document Object Model (DOM) – liest das komplette XML-Dokument ein und erstellt daraus im Speicher eine Baumstruktur, in welcher alle Elemente, Attribute und Inhalte als Knoten und Blätter des DOM-Baums dargestellt werden (vgl. Abbildung 2.4). Die große Stärke aber auch gleichzeitig die große Schwäche dieser Repräsentation ist, dass das gesamte Dokument im Speicher vorliegt. Dies ermöglicht einerseits den wahlfreien Zugriff auf einzelne Knoten des DOM-Baums (gleichbedeutend mit dem wahlfreien Zugriff auf verschiedene Teile des XML-Dokuments), andererseits bedeutet dies auch, dass Dokumente erst komplett in den Speicher eingelesen und dort für die Dauer der Verarbeitung gehalten werden müssen. Für große Dokumente kann der hohe Speicherverbrauch u. U. Probleme bereiten. Der DOM-Standard ist wie XML auch eine Spezifikation des W3C [28].

2.5.2 Simple API for XML

Das zweite Verfahren – Simple API for XML (SAX) – arbeitet ereignisorientiert. Ein SAX-basierter Parser liest ein Dokument sequentiell ein und meldet so genannte „Ereignisse“ an das Programm, das den Parser verwendet. Unter einem Ereignis versteht man den Beginn oder das Ende eines Dokuments oder Elements, aber auch Attribute, Werte, Kommentare usw. Das den SAX-Parser verwendende Programm kann für es relevante Ereignisse weiterverarbeiten, andere hingegen kann es ignorieren. Der Vorteil dieser Methode im Vergleich zu DOM ist, dass immer nur ein kleiner Ausschnitt des Dokuments im Speicher gehalten werden muss. Deswegen allerdings kann auch nicht wie mit DOM wahlfrei auf einzelne Teile des Dokuments zugegriffen werden. SAX basiert nicht auf einem W3C-Entwurf, sondern

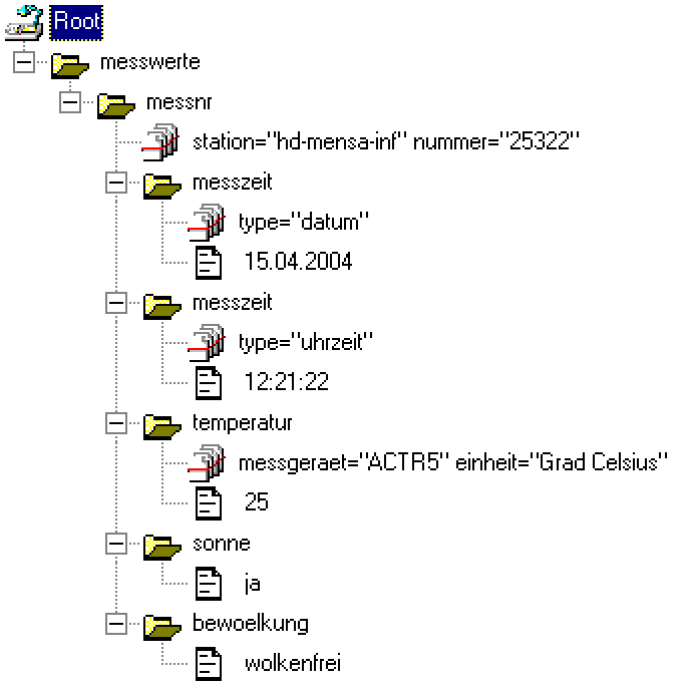

Abbildung 2.4: Baum-Darstellung des in Abbildung 2.1 definierten XML-Dokuments

entstand aus der Anwendung heraus von Entwicklern von XML-Parsern und -Anwendungen. Dennoch ist es ein de facto Standard. Die aktuelle Version ist SAX 2.0.1 (vgl. [29]).

2.6 Nachteile des XML-Formats

XML-Dokumente haben – neben all ihren Vorteilen – allerdings auch einen großen Nachteil. Der Informationsgehalt ist unabhängig vom eigentlichen Inhalt recht klein, die Entropie[10] also relativ groß. Da XML vom Menschen direkt lesbar ist, werden (bei Verwendung von ASCII) nur lesbare Zeichen aus dem 256 Zeichen umfassenden ASCII 8 Zeichensatz verwendet. Die effektive Anzahl der verwendbaren Zeichen liegt mit etwa 110 somit bei unter 50 Prozent. Weiterhin enthalten menschliche Sprachelemente, wie sie auch in XML vorkommen, eine hohe Redundanz. Mit Hilfe geeigneter Kompressionsalgorithmen (z. B. LZ77

[10] Shannon entwickelte 1948 in seiner Arbeit „Mathematische Theorie der Nachrichtenübertragung“ [30] den Begriff der informationstheoretischen Entropie $H(N)$ als ein Maß für Redundanz einer Nachricht I: $H(I) = -\sum_i p_i \log_2 p_i$, wobei p_i die Wahrscheinlichkeit ist, mit der das i-te Symbol in I auftritt.

(*gzip* [31]), Burrows-Wheeler Algorithmus (*bzip2* [32])) lassen sich XML-Dateien allerdings auch oft um bis zu 90 % oder mehr komprimieren.

2.7 Zusammenfassung

Einige, im Zusammenhang dieser Arbeit besonders wichtige Eigenschaften von XML werden im Folgenden kurz zusammengefasst.

- XML ist eine von Menschen und Maschinen gut lesbare und, bei geeigneter Wahl der Element- und Attributnamen, selbstdokumentierende Metasprache.
- XML ist Rechnerarchitektur und Betriebssystem unabhängig. Die freie Wahl des Zeichensatzes ermöglicht zudem u. a. auch die Nutzung von Unicode und somit die Verwendung von fast allen verfügbaren Zeichen in fast allen Sprachen.
- XML-Dokumente können aufgrund des kleinen Regelsatzes leicht auf ihre Wohlgeformtheit geprüft werden.
- XML sieht die optionale Spezifizierung von Dokumenttypen durch Festlegung eines Vokabulars und einer Grammatik mit Hilfe von DTD- oder XML-Schema-Dateien vor. XML-Dokumente können somit mit Hilfe eines validierenden Parsers auch auf ihre Gültigkeit geprüft werden.
- Es existiert eine große Zahl von Werkzeugen, wie z. B. XML-Parser, für fast alle Programmiersprachen.
- Teilweise beschädigte XML-Dokumente bleiben im nicht beschädigten Teil noch immer lesbar. Abhängig vom Grad der Schädigung kann man durch die hohe Redundanz menschlicher Sprache sogar beschädigte Teile relativ leicht rekonstruieren (Beispiel: `<soxxe> ja </sonne>` könnte von einem korrigierenden Parser unter der Annahme, dass das Dokument ansonsten gültig ist, leicht zu `<sonne> ja </sonne>` verbessert werden, sofern nur `<sonne>` ein zulässiges Element ist.

Aufgrund dieser Eigenschaften ist XML gut geeignet, um die im Rahmen eines Cluster- und Grid-Monitorings erhobenen Daten darzustellen, da flexibles Monitoring, wie es im nächsten Kapitel motiviert wird, auf eine flexible Darstellung der Messwerte angewiesen ist. Nicht zuletzt deswegen verwenden einige der in Kapitel 3 vorgestellten Monitoringsysteme XML als Datenaustauschformat.

3 Monitoringsysteme

3.1 Einleitung

Cluster aber auch Grids werden verwendet, um die vorhandenen Ressourcen (Rechenleistung, Speicherkapazität) einer Menge von Rechnern einem Benutzer koordiniert zur Verfügung zu stellen. Dabei soll der Benutzer keine genaue Kenntnis über das verteilte System benötigen, um es nutzen zu können. Vielmehr soll das gesamte System nach außen hin als eine Einheit erscheinen. Diese Kapselung der dahinter liegenden Struktur erreicht man durch die Schaffung von Schnittstellen, die es dem Anwender erlauben, eigene Prozesse einzubringen und die Ergebnisse abzurufen.

Dennoch arbeiten hinter der Schnittstelle im Endeffekt einzelne Rechner, deren Zustand über einen dynamischen Satz dynamischer Parameter beschrieben werden kann. Zu der Aufgabe eines Monitoringsystems gehört nun die Überwachung verschiedener, frei definierbarer Zustandsparameter. Der Begriff des Zustandsparameters ist dabei bewusst sehr allgemein gehalten. Im Folgenden wird eine sehr begrenzte Auswahl typischer Zustandsparameter aufgezählt:

- Auslastung der Prozessoren
- verfügbarer Arbeitsspeicher
- verfügbarer Festplattenspeicher
- Eingabe / Ausgabe (E/A) -Verkehr (z. B. Netzwerk)
- Anzahl laufender Prozesse
- CPU-Temperatur
- Umdrehungsgeschwindigkeit eines Lüfters
- Laufgeräusch einer Festplatte
- Neueinträge des Syslog-Dienstes[1]

[1] System-Protokoll

Allen oben aufgeführten Zustandsparametern ist gemein, dass man sie jeweils lokal auf jedem Rechner (die entsprechenden Rechte und evtl. notwendige physikalische Sensoren vorausgesetzt) messen kann. Die Software, die diese Messungen vornimmt, wird im Folgenden als Sensor bezeichnet. Sensoren übergeben ihre Messdaten an das Monitoringsystem, welches die gemessenen Daten anderen Programmen bzw. dem Anwender zur Verfügung stellt.

Erweitert man die Definition eines Sensors, so kann im Prinzip jede Software, die ein Ergebnis als Funktion ihrer Eingangsparameter zurückliefert, ein Sensor sein. Insbesondere stellt damit Software, die Messwerte anderer Sensoren als Eingangsparameter für ihr Ergebnis verwendet, einen Sensor dar. Ein solcher Sensor kann somit abstraktere bzw. konsolidierte Informationen über ein System zurückliefern (z. B.: gesamter freier Speicher im Grid, Gesamtrechenzeit eines Prozesses, ...).

3.2 Motivation

Im Folgenden werden die Notwendigkeit und die Aufgaben eines Monitoringsystems dargelegt. Ein umfassendes Monitoring verschiedener Zustandsparameter kann helfen, Fehler vorherzusagen und sie somit evtl. zu vermeiden bzw. abzumildern. Fällt zum Beispiel auf einem Knoten ein CPU-Lüfter aus, so könnte man den gesamten Rechner noch rechtzeitig vor der Überhitzung der CPU herunterfahren. Sollte es doch zu einem Fehler kommen, dienen geeignete Sensorwerte der Erkennung der Ursache, die Voraussetzung für ihre Behebung ist. Im eben genannten Beispiel muss der Techniker lediglich den Lüfter austauschen und nicht erst eine aufwändige Fehlersuche starten. Weiterhin kann man die gewonnenen Daten zur Optimierung der Auslastung (load-balancing) und zur Kontrolle der Verfügbarkeit einzelner Ressourcen (Beispiele: aktive Dienste[2], vorhandene Massenspeicher, aktive Netzwerkanbindungen) einsetzen. Ziel ist es also, mit einem geeigneten Satz an Parametern einen Überblick und, wenn nötig, detaillierte Informationen über den Status des Systems zu gewinnen. Monitoringsysteme sollten daher die mittels möglichst frei definierbarer Sensoren ermittelten Messwerte aufnehmen und diese verwalten.

Damit ein Monitoringsystem für die Überwachung von Grids eingesetzt werden kann, muss dieses einige allgemeinen Anforderungen erfüllen. Neben der unbedingt notwendigen Skalierbarkeit und Fehlertoleranz gegen den Ausfall einzelner Komponenten muss eine leichte

[2] Bezeichnung für Programme, die typischerweise beim Hochfahren eines Rechners gestartet werden und in der Regel im Hintergrund arbeiten, um eine gewisse Funktionalität bereitzustellen. Dienste werden allerdings nicht automatisch mit dem Betriebssystem-Kern gestartet.

Handhabung gewährleistet bleiben. Wissen über die genaue Struktur des verteilten Systems sollte nur wenn nötig vorausgesetzt werden. Ganz zu vermeiden ist es natürlich nicht, geben Sensoren in der Regel doch Auskunft über einen ganz bestimmten Parameter eines ganz bestimmten Knotens.

Zuletzt gehört zu einem sinnvollen Monitoring auch, dass die gesammelten Daten anderen Applikationen tatsächlich zur Verfügung stehen. Ein so genanntes Datenlagerungssystem, im Folgenden auch Datenlager genannt, stellt Schnittstellen zum Ablegen für das Monitoringsystem auf der einen Seite und zum Abrufen durch die abfragenden Komponenten auf der anderen Seite zur Verfügung. Ein solches Datenlager kann auf unterschiedliche Weisen realisiert werden. Kapitel 4 gibt einen Überblick über gebräuchliche Datenlager.

3.3 Auf dem Markt frei und kommerziell verfügbare Systeme

Auf dem Markt existieren mehrere Monitoring-Programme mit unterschiedlichen Schwerpunkten und Architekturen. Exemplarisch werden einige davon kurz vorgestellt und ihr wesentlicher Leistungsumfang dargelegt. Für einen weiteren Überblick, vor allem in Bezug auf die Skalierbarkeit verschiedener Monitoringsysteme, wird auf [33] verwiesen.

3.3.1 Ganglia

Das *Ganglia Cluster Toolkit* [34] ist ein skalierbares und verteiltes Cluster- und Grid-Monitoringsystem. Es basiert auf einem hierarchischen Design:

- Monitor Demon *gmond*: läuft auf jedem Knoten und sammelt Daten.
- Meta Demon *metad*: läuft auf einem oder wenigen ausgezeichneten Knoten und fast die gesammelten Daten **aller** *gmond* zusammen, welche, je nach Position in der Hierarchie, zum Teil direkt und zum Teil durch andere *metad* zur Verfügung gestellt werden.

Dieses Verfahren erhöht die Ausfallsicherheit, da jeder *metad* dadurch einen Überblick über den gesamten Cluster besitzt. Realisiert wird es durch Verwendung eines eigenen Protokolls zur Kommunikation zwischen mehreren verschiedenen Knoten, welches auf Multicasts[3] basiert. Dies hat allerdings den Nachteil einer hohen Netzlast.

[3] Multicast ermöglicht die Kommunikation zwischen einem einzelnen Sender und mehreren Empfängern in einem Netzwerk.

Ganglia nutzt XML zur Darstellung der Sensordaten, und ist somit flexibel und einfach selbst anzupassen. Allerdings werden die gesammelten Daten durch den Meta Demon zur Speicherung und späteren visuellen Darstellung *RRDtool* [35] übergeben, welches seinerseits Daten in einer Round-Robin-Datenbank[4] ablegt. Da *RRDtool* lediglich zahlenbehaftete Werte verwalten kann, wird das Spektrum zulässiger Sensorinformationen stark eingeschränkt.

Ganglia steht unter der *BSD License* [36] für mehrere Betriebssysteme (Linux, Solaris, MacOS X sowie weitere Unix-Varianten) und Prozessor-Architekturen (i386, IA-64, Sparc, Alpha, S/390 und weitere) zur Verfügung.

3.3.2 xmlsysd

xmlsysd [37] ist ein Monitoringsystem für Linux-Cluster, das ebenfalls XML-basierte Messwerte zur Verfügung stellt. Dazu muss der *xmlsysd* Dienst auf jedem Knoten eines Clusters installiert werden. Dieser Dienst kann daraufhin gezielt vorher festgelegte Pseudo-Dateien aus `/proc`[5] auslesen und dadurch verschiedene Parameter des Systems (z. B. CPU-Load, Speicherverbrauch, Netzwerkauslastung, laufende Prozesse, etc.) überwachen.

Die Messergebnisse werden in XML verpackt und können mit Hilfe des Terminalprogramms *wulfstat* über eine Netzwerkverbindung zu dem *xmlsysd*-Dienst ortsunabhängig abgerufen werden. Um einen Überblick über den gesamten Cluster zu erhalten, muss somit eine gezielte Verbindung mit jedem einzelnen Knoten hergestellt werden. Doch selbst dann stehen lediglich die aktuellen Daten zur Verfügung, da eine Speicherung alter Daten nicht vorgesehen ist.

Der Entwickler stellt allerdings auch einen anderen Klienten namens *wulflogger* zur Verfügung. *wulflogger* steht ständig in Kontakt mit den zu überwachenden Knoten und leitet deren Messwerte, mit einem Zeitstempel versehen, auf die Standardausgabe um. Diese Daten könnten nun einem Datenlager zugeführt werden, jedoch obliegt dies der Implementation des Benutzers.

xmlsysd steht unter der freien, nicht-kommerziellen *Modified Open Publication License* (OPL) [38] für das Betriebssystem Linux unter allen von Linux unterstützten Prozessor-Architekturen zur Verfügung.

[4] Datenlagerungssystem, welches sich durch die NICHT- Dauerhaftigkeit der Daten auszeichnet. In Kapitel 4.3.1 wird dies ausführlich besprochen.

[5] In dem Verzeichnis `/proc` stehen unter Linux viele dynamische und feste Parameter, die das System und seinen aktuellen Zustand beschreiben, in Pseudo-Dateien zur Verfügung.

3.3.3 Supermon

Das *Supermon*-Paket [39] wird am Los Alamos National Laboratory entwickelt. Als eine Sammlung von Werkzeugen für das Cluster-Monitoring wurde es auf eine hohe Abtastrate der Sensoren hin optimiert und sammelt bzw. organisiert Sensormesswerte auch in großen Clustern. Dabei stellt es lediglich aktuelle Messwerte zur Verfügung, da auch hier, wie bereits bei *xmlsysd*, eine Speicherung alter Daten nicht vorgesehen ist.

Supermon besteht aus drei Programmteilen:

- Ein Kernel-Modul[6] sammelt alle zu beobachtenden Sensordaten. Dabei wird stets ein Mindestsatz an fest definierten Sensoren (integrierte Sensoren) mit abgefragt.
- Der Daten-Server *mon* läuft auf jedem Knoten und stellt die vom Kernel-Modul gesammelten Daten zur Verfügung
- Der Daten-Konzentrator *supermon* läuft auf einigen ausgezeichneten Knoten und fast die gesammelten Daten einiger *mon* Daten-Server zusammen.

Supermon verwendet nicht XML zum Datenaustausch zwischen den einzelnen Komponenten, sondern mit den so genannten „s-expressions“ ein auf der Programiersprache *Lisp* [40] basierendes Datenaustauschformat. In der *Lisp*-Syntax sind Daten aus symbolischen Ausdrücken aufgebaut, die entweder Atome, d. h. nicht weiter zerlegbare Einheiten, oder aber Listen sein können. Dadurch allerdings wird auf die allgemeine Verständlichkeit und Portabilität, wie sie XML bietet, zu Gunsten der effizienteren Darstellung verzichtet. Dies wiederum erschwert die Einbringung eigener Sensoren, die über die zu diesem Zweck zur Verfügung gestellte *monhole*-Schnittstelle integriert werden können. Zudem ist nach Angeben der Entwickler die Datenübergabe von eigenen Sensoren an die *monhole*-Schnittstelle bei weitem nicht so effizient, wie die Datenerfassung der integrierten Sensoren.

Supermon steht unter der *GNU General Public License* (GPL) [36] für die Betriebssysteme Linux und MacOS X für alle unterstützen Prozessor-Architekturen zur Verfügung.

[6] Ein Kernel-Modul ist ein spezieller Programmcode, der in den Kernel, dem Hauptbestandteil eines Betriebssystems, geladen und wieder entfernt werden kann. In der Regel werden Gerätetreiber als Kernel-Module implementiert. Der Einsatz eines Kernel-Moduls für Monitoring ist indes ungewöhnlich und im Prinzip unnötig.

3.3.4 HCMS

Das *HCMS* ist ein Monitoringsystem für Linux-Cluster, welches aufgrund eines hierarchischen Prinzips gut skaliert und somit auch mit vielen Rechnern benutzt werden kann. *HCMS* wurde im Rahmen einer Diplomarbeit am Lehrstuhl für Technische Informatik an der Universität Heidelberg entwickelt [33].

Die Software muss auf jedem Knoten installiert werden. In dieser Phase wird dabei keine direkte Unterscheidung zwischen den einzelnen Hierarchie-Ebenen vorgenommen. Auf jedem Knoten lassen sich über eine einfache Schnittstelle leicht beliebig viele Sensoren zur Überwachung frei definierbarer Parameter integrieren. Die Sensordaten werden von einem *Escalator* genannten Programm in die nächste Hierarchie-Ebene übermittelt. Jeder Knoten kann dabei feststellen, welcher Ebene er angehört und welche Knoten der nächst höheren Ebene zugeordnet sind. Da dieses System selbstkonfigurierend ist, kann der Ausfall beteiligter Rechner ohne Datenverlust abgefangen werden.

Die Kommunikation zwischen Sensoren und Monitoringsystem, aber auch zwischen den einzelnen Hierarchie-Ebenen erfolgt in XML. Das *HCMS* fasst dabei einzelne XML-Sensorwerte im Zuge der so genannten Konsolidierung zu größeren XML-Objekten zusammen. Als Datenlager zur persistenten Speicherung verwendet das *HCMS* eine relationale Datenbank[7], die durch das Monitoringsystem gespeist und durch den Anwender abgefragt werden kann.

3.4 Zusammenfassung

In diesem Kapitel wurden die Vorteile, wenn nicht sogar die Notwendigkeit von Monitoringsystemen aufgezeigt und ihre Aufgaben definiert. Zusätzlich wurden exemplarisch einige auf dem Markt verfügbare Monitoringsysteme kurz vorgestellt und ihr Leistungsumfang grob umrissen. Dabei zeigte sich, dass unterschiedliche Strategien der Datenhaltung existieren. Umfassendes Monitoring allerdings ist auf persistente Datenhaltung angewiesen, welches von den vorgestellten Systemen lediglich durch das *HCMS* anboten wird.

Der Sinn umfassenden Monitorings ist es, den Administratoren und Nutzern eines verteilten Systems die Möglichkeit zu geben, Prozesse, Ereignisse und die Systemumgebung zu

[7] In Kapitel 4.3.3 werden relationale Datenbanken ausführlich vorgestellt.

überwachen. Durch dieses erlangte Verständnis über Monitoringsysteme sind einige Anforderungen an die anzuschließenden Datenlagerungssysteme abzusehen, welche im folgenden Kapitel genauer untersucht werden.

4 Datenlagerungssysteme

4.1 Einleitung

In Kapitel 3 wurden einige auf dem Markt verfügbare Monitoringsysteme für Cluster und Grids vorgestellt. In diesen wird allerdings hauptsächlich das aus klassischen Clustern bekannte Monitoring betrieben. Dabei werden einige vorher festgelegte Werte erfasst – meist einige einfache, numerische Daten wie CPU Auslastung, Speicherverbrauch und E/A-Verkehr bzw. Netzwerklast. Diese Daten dienen meist statistischen Zwecken und einer rudimentären Überwachung des gesamten Systems. Wenn in einem Knoten ein gravierender Fehler auftritt, erkennt der Administrator eines Clusters den betreffenden Knoten meist dadurch, dass dieser in mindestens einem der oben aufgeführten Messwerte deutlich von den anderen Knoten abweicht. Der Administrator muss dann eine Fehlersuche durchführen, wobei er, wenn überhaupt, nur auf eine geringe Historie einiger weniger Messdaten zurückgreifen kann. Die erfassten Daten sind dann in einem Datenlager archiviert und stehen für Abfragen zur Verfügung. Ein solches Datenlager kann prinzipiell unterschiedlich realisiert werden. In diesem Kapitel werden einige unterschiedliche Datenlager-Plattformen vorgestellt. Zuvor jedoch müssen die Anforderungen an Datenspeicherung und -haltung im Rahmen umfassenden Grid-Monitorings formuliert werden.

4.2 Anforderung an ein Datenlagerungssystem im Umfeld eines umfassenden Grid-Monitorings

Umfassendes Monitoring ist eine unverzichtbare Voraussetzung für das Betreiben großer Grids. Man möchte jedem autorisierten Nutzer die Möglichkeit einräumen, unabhängig von der Kenntnis der genauen Struktur des hochgradig verteilten Systems, beliebige eigene Prozesse einzubringen und diese zu überwachen. Um eine hohe Unabhängigkeit und Flexibilität der zu beobachtenden Parameter vom eigentlichen Monitoringsystem zu gewährleisten, bietet es sich an, die Überwachung und Messwertgewinnung durch eigenständige Sensoren

vornehmen zu lassen. Zur Überwachung eigener Prozesse kann der Nutzer dann selbst Sensoren entwickeln und einbringen, die genau die Daten und Messwerte auslesen, die für die eigenen Prozesse relevant sind.

4.2.1 Inhomogenität der Messwerte in Typ und Frequenz

Eine entscheidende Herausforderung, der sich ein umfassendes Monitoringsystem stellen muss, wird im Folgenden besonders verdeutlicht: Sensorergebnisse können vollkommen verschiedener Natur und Struktur sein.

Sicher wird es bei Grid-Monitoringsystemen eine Anzahl einfacher Sensoren geben, die mit einer festen Frequenz einen numerischen Messwert oder eine Zeichenkette und den dazugehörigen Zeitstempel generieren. Diese kann man als „flache Sensoren“ bezeichnen, da sie einen zweidimensionalen Wertebereich besitzen – zu jedem Zeitpunkt t_i wird genau ein Messpunkt y_i gemessen. Daneben können sicher auch Sensoren existieren, die aperiodisch Messwerte zurückliefern. Als anschauliches Beispiel könnte man sich einen Syslog-Sensor für einen Nutzerprozess vorstellen: jeden Eintrag in das Syslog, den dieser Prozess auslöst, registriert der Sensor und reicht ihn weiter. Auch Sensoren dieser Art kann man als flache Sensoren bezeichnen, ergeben sie doch auch einen zweidimensionalen Wertebereich. Zusätzlich sind auch Sensoren vorstellbar, die komplexere Messdaten erzeugen. Zwei einfache Beispiele sollen dies verdeutlichen:

Beispiel 1 Ein Sensor soll die gesamte verbrauchte Rechenzeit eines Nutzerprozesses ermitteln. Auf den ersten Blick entspricht dies dem Anforderungsprofil eines numerischen flachen Sensors. Erzeugt dieser Nutzerprozess nun aber je nach Situation verschieden viele Kindprozesse, deren verbrauchte Rechenzeit man ebenfalls getrennt erfassen möchte, erweist sich ein numerischer flacher Sensor als ungeeignet, da eine a priori unbekannte Anzahl an Messdaten-Tupeln (ID des Kindprozesses, verbrauchte Rechenzeit, Zeitstempel) vorliegen. Ein flacher Stringsensor kann zwar die Tupel in Form einer langen Zeichenkette aufnehmen, allerdings muss zur Verwendung dieser Daten später genau bekannt sein, dass dies verschiedene Tupel sind und nicht etwa eine durch einen „echten“ Stringsensor erzeugte Zeichenkette. Viel geeigneter wäre es, wenn der Sensor seine hierarchischen Informationen tatsächlich auch hierarchisch übergeben könnte. Analog zu flachen Sensoren führen wir hier daher den Begriff des hierarchischen oder auch „multidimensionalen Sensors“ ein. Dies ist ein Sensor, dessen Wertebereich im all-

gemeinen Falle beliebig viele Dimensionen tief sein kann. Das Datenlager, in dem der Sensor seine Daten ablegt, sollte also idealerweise beliebige Hierarchien unterstützen.

Beispiel 2 Ebenso ist vorstellbar, dass ein Sensor binäre Daten übertragen soll. Ein Sensor könnte z. B. die Laufgeräusche einer Festplatte überwachen. Stellt dieser nun fest, dass sich das Geräusch über einen bestimmten Toleranzbereich hinaus verändert, nimmt er eine „Hörprobe“ dieses Geräusches auf. Eventuell macht der Sensor anhand diverser ihm bekannter Profile schon eine Annahme über den wahrscheinlichen Grund der Veränderung und leitet die Audiodatei, mit einem Zeitstempel und dem vermuteten Problem versehen, weiter. Es ist ersichtlich, dass dieses Szenario andere Anforderungen an das Datenlager stellt als hierarchische Messwerte.

Sensorergebnisse können somit vollkommen verschiedener Struktur bezüglich Datentypen, Tiefe und Frequenz sein. Die genannten Beispiele verdeutlichen die Tatsache, dass man bei der Entwicklung eines solchen Systems nicht spezifizieren kann, welche Sensoren existieren und was genau sie messen werden.

4.2.2 XML als Sensorformat

In Kapitel 2 wurden viele Vorteile von XML für die Ablage von strukturierten Objekte aufgezeigt. Da die Sensoren eines Monitoringsystems wie gezeigt z. T. sehr unterschiedliche Daten produzieren, liegt es nahe, dass Sensoren ihre Messdaten stets in XML „formulieren“. Dadurch wird ein großes Spektrum an Sensorwerten ermöglicht, wie einige der in Kapitel 3 vorgestellten Monitoringsysteme bereits zeigen.

Durch die XML-Formatierung wird jedoch das effektive Volumen des eigentlichen Messwerts erhöht. Jeder Messwert, sowie alle zusätzlichen Informationen, werden durch XML-Tags ausgezeichnet, Bei kurzen Messwerten hat dies, abhängig von den verwendeten Tags, eine signifikante Erhöhung des Volumens eines Datensatzes zur Folge. Wenn im Folgenden daher von Datenvolumina gesprochen wird, so sind damit ausschließlich die Volumina der „Rohdaten“, d. h. der reinen Messwerte gemeint.

Durch die Wahl von XML für den Datenaustausch werden weitere Anforderungen an das Datenlagerungssystem motiviert: es sollte auf einfache Art möglich sein, die in XML strukturierten Messdaten verarbeiten zu können. Multidimensionale sowie binäre Messwerte sollten

gleichermaßen ohne Informationsverlust abgelegt werden können. Auch sollte die Größe eines Datensatzes keinen wesentlichen Einschränkungen unterliegen, da sowohl kurze als auch im Verhältnis viel umfangreichere Messwerte gespeichert werden müssen.

Insbesondere stellt sich bei allen Datenlagern die Frage, wie die jeweiligen Datenlager mit Messwerten ihnen unbekannter Sensoren zurechtkommen.

4.2.3 Erwartetes Datenvolumen

Ein umfassendes Grid-Monitoring produziert ein Vielfaches an Datenmenge im Vergleich zum klassischen. Um dies zu quantifizieren wird eine Abschätzung des Datenvolumens vorgenommen.

Man kann zwischen drei Arten von Sensoren unterscheiden: Messwerte, die im klassischen Monitoring ermittelt werden, sollen im Folgenden als „klassisch“ bezeichnet werden. Ihre Anzahl $N_{klassisch}$, durchschnittliche Größe $g_{klassisch}$ und ihre typischen Frequenzen $\nu_{klassisch}$ sind gut bekannt, die entstehende Datenrate $R_{klassisch}$ lässt sich daher leicht abschätzen. Sensorwerte klassischen Monitorings sind meist zahlenbehaftet und ändern sich in der Regel schnell.

Weiterhin gibt es Messwerte, die im Rahmen eines erweiterten Monitorings gemessen werden. Ihre Anzahl $N_{erweitert}$ ist größer als die der klassischen Sensoren, so wie die durchschnittliche Größe $g_{erweitert}$ eines Messwerts aufgrund der verschiedenen zugelassenen Datentypen (wie z. B. Zeichenketten) ebenfalls ansteigt. Allerdings erwartet man, dass solche Sensoren im Schnitt mit einer etwas geringeren Frequenz $\nu_{erweitert}$ Daten produzieren, da die meisten sich schnell ändernden Sensorwerte bereits im klassischen Monitoring erfasst werden.

Es verbleiben also noch alle anderen Sensoren, die im umfassenden Grid-Monitoring auftauchen können. Ihre Anzahl $N_{umfassend}$ ist viel größer als die der beiden anderen Sensorarten, da es im Prinzip für fast jeden denkbaren Sensor auch ein geeignetes Anwendungsszenario gibt. Für alle Sensoren gilt, dass sie im Idealfall durch ihre Messungen keinen signifikanten Einfluss auf den Rechner haben sollten. Deswegen wird in der Regel die durchschnittliche Messfrequenz $\nu_{umfassend}$ der Sensoren dieses Typs relativ gering sein, da eine große Anzahl von ständig aktiven Sensoren das zu überwachende System ausbremsen würde. Die Breite des Spektrums an möglichen Ergebnissen, die solche Sensoren produzieren können (vom einfachen Zahlenwert über Dokumente und Log-Files bis hin zu nicht-textualen Objekten) lässt auf eine nochmals deutlich größere durchschnittliche Messwertgröße $g_{umfassend}$ schließen.

Sensortyp	Anzahl N der Sensoren	durchschn. Größe g eines Messwerts (Byte)	durchschn. Frequenz ν $[min^{-1}]$	durchschn. Datenrate R [kB/sek]
klassisch	10	25	120	0,49
erweitert	20	75	60	1,46
umfassend	70	225	1	0,26
gesamt	**100**	**55**	**25**	**2,21**

Tabelle 4.1: Auf Erfahrungswerten basierende Annahmen über die zu erwartende Größenordnung des Datenaufkommens je Knoten. Die Summe errechnet sich wie folgt: $N_{gesamt} = \sum_i N_i \qquad \nu_{gesamt} = \frac{\sum_i N_i \cdot \nu_i}{N_{gesamt}}$ und $g_{gesamt} = \frac{\sum_i R_i}{N_{gesamt} \cdot \nu_{gesamt}}$

In Tabelle 4.1 werden erwartete Größenordnungen pro Knoten für die eingeführten Parameter angegeben und die daraus resultierende Gesamtdatenrate errechnet.

Wie ersichtlich, können dabei je nach Anzahl der Sensoren, durchschnittlicher Frequenz und durchschnittlicher Größe eines Messwertes leicht einige kByte pro Sekunde und Knoten anfallen. Geht man davon aus, dass einem Grid insgesamt ca. 50-100 tausend Knoten angehören, muss das gesamte Datenlagerungssystem Datenraten in der Größenordnung von zweihundert Megabyte pro Sekunde aufnehmen können. Des weiteren muss es, wie oben erläutert, in Typ und Struktur unterschiedliche Sensorwerte verwalten, idealerweise ohne vorherige Kenntnis des genauen Aufbaus.

Geht man von etwas über 200 Megabyte unkomprimierter Rohdaten pro Sekunde aus, so fallen im Verlauf eines Tages fast 18 Terabyte an Daten an, was auf Festplatten abgelegt Kosten von ca. 9 000 EUR pro Tag verursacht[1]. Notwendigerweise sind bei diesen Datenmengen daher strenge Konzepte zur Datenkompression, -Löschung und -Archivierung zu implementieren. Ohne Kombination dieser Verfahren würden selbst an sich billige Tertiärspeicher[2] auf sehr absehbare Zeit zu teuer.

4.2.4 Verteilte Systeme

Einer der wesentlichen Gründe für die Sammlung so vieler Daten ist es, den Administratoren und Nutzern von Clustern und Grids die Möglichkeit zu geben, Prozesse, Ereignisse und die Systemumgebung zu überwachen. Zu diesem Zweck stellt man Anfragen an das Datenlagerungssystem, die dieses mit möglichst geringer Latenz beantworten soll. Im klassischen

[1] Die günstigsten heute verfügbaren Festplatten kosten ca. 0,50 EUR pro GB.
[2] Primärspeicher: Arbeitsspeicher, RAM; Sekundärspeicher: Festplatten; Tertiärspeicher: Magnetbänder

Monitoring werden die anfallenden Daten meist nicht verteilt, sondern das Datenlager ist, sofern überhaupt eins existiert, üblicherweise auf einem einzelnen Rechner eingerichtet. Neben dem bereits erwähnten Problem, dass ein solches Design einen „single point of failure“ mit sich bringt, den zu vermeiden ein weiteres wichtiges konzeptionelles Ziel sein muss, wird aktuelle Hardware mit den oben benannten Anforderungen überfordert. Geht man daher davon aus, dass das Datenlagerungssystem aus einem Verbund an sich autarker Einheiten besteht, so muss ein Verfahren existieren, das eine lesende Anfragen auf die verschiedenen Einheiten verteilt und die einzelnen Antworten gebündelt zurückliefert. Einige relationale **D**aten**B**ank-**M**anagement-**S**ysteme (DBMS)[3] unterstützen dies nativ, für alle anderen muss ein solches Verfahren bei Bedarf selbst implementiert werden.

4.3 Unterschiedliche Datenlager-Plattformen

In diesem Abschnitt werden verschiedene vorhandene und teilweise von klassischen Systemen verwendete Datenlager-Plattformen vorgestellt und es wird auf ihre Stärken und Schwächen eingegangen.

Bei allen Datenlagerungssysteme ist jedoch zu erwarten, dass sie nicht beliebig große Datenmengen verwalten können. Aufgrund der hohen erwarteten Datenvolumina werden immer mehr Sekundärspeicher (Festplatten) benötigt; auch bei effektiver (Speicher-) Organisation wachsen die Zugriffszeiten zu den Daten an; gewisse Schwellenwerte des jeweiligen Datenlagers (wie z. B. maximale Anzahl an Dateien, die Maximalgröße von Tabellenbereichen) werden erreicht und drohen, einen Stillstand zu verursachen bzw. erfordern massives Eingreifen durch den Administrator. Es gibt zwei Möglichkeiten, auf diese Situation zu reagieren: Zum einen kann man durch Datenlöschung das Anwachsen der Datenmengen zum Stillstand bringen oder wenigstens bremsen, oder es erfolgt eine Datenarchivierung aus dem Datenlager heraus auf andere geeignete Speichermedien. In folgenden werden beide Verfahren etwas näher erläutert.

Datenlöschung Eine Datenlöschung sollte nur vorgenommen werden, wenn die entsprechenden Daten sicher nicht mehr gebraucht werden. Im Umfeld eines Monitorings gehört demnach z. B. die CPU-Last eines bestimmten Knotens zu einem ganz bestimmten Zeitpunkt relativ weit in der Vergangenheit (z. B. vor 196 Tagen 42 Minuten und 14 Sekunden) wahrscheinlich zu den unwichtigen Informationen. Hingegen ist es

[3] Eine genaue Definition folgt in Kapitel 4.3.3.

sicher potentiell interessant zu wissen, wie sich die CPU-Ausnutzung/Tag eines Knotens über das letzte Jahr hinweg entwickelt hat. Die aufgezeigte Diskrepanz zwischen gespeicherten Daten und nützlicher Information ermöglicht eine starke Datenreduktion durch Zusammenfassung. Dies setzt selbstverständlich voraus, dass man zum Zeitpunkt der Löschung weiß, welche Informationen tatsächlich benötigt werden.

Archivierung Archivierung bedeutet, dass man solche Messdaten, die aktuell nicht mehr oder nur noch extrem selten gebraucht werden, auf günstige (Tertiärspeicher-) Medien auslagert; diese werden dann „in den Schrank“ gestellt, bis die auf ihnen abgelegten Daten eventuell wieder einmal nachgefragt werden.

In der Praxis eines umfassenden Monitoringsystems wird sicher eine Kombination beider Methoden sinnvoll sein. Deshalb wird bei der nachfolgenden Besprechung ein besonderes Augenmerk auf die möglichen Techniken zur Datenreduktion gelegt.

4.3.1 Round-Robin

Beim klassischen Cluster-Monitoring werden häufig Round-Robin-Datenbanken eingesetzt. In Kapitel 3.3.1 z. B. wurde das *Ganglia*-Monitoringsystem vorgestellt, welches seine Daten in einem solchen Datenlager ablegt.

Round-Robin-Datenbanken zeichnen sich dadurch aus, dass ihnen zur Datenablage nur ein begrenzter Speicher zur Verfügung steht. Die eingehenden Daten werden hintereinander in den Speicher geschrieben. Ist dieser gefüllt, so überschreiben neu eintreffende Daten die ältesten. Gepeicherte Daten stehen somit immer nur eine endliche Zeit zur Verfügung

Offensichtlich erfüllen Round-Robin-Datenbanken keine der oben definierten Anforderungen für den Einsatz im Umfeld umfassenden Grid-Monitorings, da diese konstruktionsbedingt nur sehr einfache Datentypen (Zahlen, sehr kurze Zeichenketten) halten können. Bereits unbegrenzt lange Zeichenketten (von komplexeren Datentypen ganz abgesehen) überfordern das Konzept, da nur ein begrenzter Speicher zur Datenverwaltung zur Verfügung steht.

Weiterhin scheidet die Verwendung von Round-Robin-Datenbanken für Grids aus, da große Datenmengen nicht über einen längeren Zeitraum gehalten werden können. Man kann das Überschreiben lediglich durch das Anlegen großer Puffer hinauszögern, was allerdings erheblichen zusätzlichen Speicherbedarf verursacht, das Problem aber nicht löst.

4.3.2 Plain-File-Datenlager

4.3.2.1 Grundlagen

Handelsübliche Festplatten werden üblicherweise in verschiedene Partitionen unterteilt. Um jedoch Dateien auf einer Partition ablegen zu können, benötigt diese ein Dateisystem, welches die Organisation der Dateien regelt. Dateien haben in einem Dateisystem in der Regel mindestens einen Dateinamen, sowie gewisse Attribute, die weitere Metainformationen über die Datei enthalten (z. B. Besitzer, Zugriffsrechte, Datum der Erstellung, ...). Ein Dateisystem sieht sich dabei mit folgenden Anforderungen konfrontiert:

- Es muss eine große Menge an Informationen speichern können.
- Die Informationen müssen dauerhaft zur Verfügung stehen (Persistenz).
- Der Zugriff auf die gespeicherten Informationen muss effizient möglich sein.

Fast alle Dateisysteme sind hierarchisch organisiert, wobei mehrere Dateinamen in eine Gruppe zusammengefasst werden, die Verzeichnis genannt wird. Unter Unix-artigen Betriebssystemen wie z. B. Linux oder BSD werden die einzelnen Partitionen normalerweise als Verzeichnisse in den Linux-Dateibaum[4] eingehängt (engl.: to mount). Der Zugriff auf andere Partition oder sogar einen anderen Datenträger erfolgt somit durch den Zugriff auf das dem Medium zugeordnete Verzeichnis.

Eine Möglichkeit der Implementation eines Datenlagers ist nun, die einzelnen Sensorwerte als Datei direkt in ein Dateisystem zu schreiben. Ein solches Plain-File-Datenlager kann unter jedem aktuellen Betriebssystem realisiert werden und stellt eine intuitive Möglichkeit dar, Daten zu schreiben. Dabei ermöglicht die Verwendung von geeigneten Dateinamen und Verzeichnisebenen die Strukturierung des Datenbestands, wobei die erwarteten Abfragekriterien soweit wie möglich in die Struktur der Verzeichnisebenen einfließen sollten.

Allerdings hat dieses sehr einfach zu realisierende Datenlager große Nachteile. Die Organisation, Verwaltung und Sicherung der Datenbestände erfolgt nicht durch eine einheitliche Software (z. B. das Dateisystem), welches die komplette Kontrolle besitzt. Vielmehr müssen diese Aufgaben mit anderen Hilfswerkzeugen wahrgenommen werden. Als weitere Nachteile muss man aufführen, dass Dateisysteme keine Rollback- oder Recovery-Dienste

[4] / ist die Wurzel (engl.: root) des Dateisystems und wird daher auch als Wurzelverzeichnis bezeichnet. Es enthält alle anderen Verzeichnisse.

anbieten. Rollback-Dienste ermöglichen die Wiederherstellung des letzten als permanent definierten Zustands eines Datenlagers und garantieren somit z. B. nach einem Systemabsturz einen definierten Anfangszustand. Recovery-Dienste erfüllen die Forderung, dass Modifikationen am Datenbestand permanent sind, auch wenn diese direkt vor einem Systemabsturz durchgeführt wurden, und somit eventuelle Änderungen noch nicht physikalisch auf dem Speichermedium sondern nur in einem Zwischenspeicher vorgenommen wurden. Weiterhin erstellen Dateisysteme in der Regel keine Indexstrukturen für den Dateiinhalt. Dies erhöht den Aufwand für Zugriffe auf Messwerte, wenn ihr genaues Auftreten nicht bekannt ist – ein offensichtlich häufiges Szenario eines Monitoringsystems. Die Suche nach bestimmten Kriterien muss dann im Falle von Textdateien (XML!) durch eine Volltextsuche innerhalb der in Frage kommenden Dateien erfolgen.

4.3.2.2 Verarbeitung in XML strukturierter Messdaten

In Plain-Systemen können die Messdaten bekannter, einfach strukturierter Sensoren mit Hilfe eines Parsers leicht extrahiert werden. Zur Speicherung der Daten muss nun aus den Informationen zu dem Messwert (Sensor, Knoten, Messzeit, etc.) der dazu passende Pfad erzeugt werden. Dieser Vorgang wird als „Mapping“ (engl.: Zuordnung) bezeichnet. Dabei ist zu beachten, dass die hierarchische Struktur der Verzeichnisse eine eindeutige Abbildung erfordert. Man muss also im Vorhinein genau wissen, nach welchen Kriterien die Daten in das Plain-System einzuordnen sind. Dann können sowohl multidimensionale als auch binäre Sensormesswerte sinnvoll abgebildet werden.

Unbekannte komplexe Inhalte müssen im Plain-File-System als eine einzelne XML-Datei abgelegt werden, da a priori nicht bekannt ist, nach welchen Kriterien die verschiedenen Strukturelemente der neuen Daten zu unterscheiden sind. Durch die Vorgabe obligatorischer Felder eines Datensatzes (z. B. der Bezeichnung des Knotens, Name des Sensors, Messzeit, etc.) kann allerdings ein minimales Mapping sichergestellt werden.

Plain-File-Systeme können prinzipiell zwar sowohl große als auch kleine Datensätze aufnehmen, jedoch ist mit jeder Datei ein gewisser Overhead sowohl im Speicherverbrauch als auch bei der Erstellung bzw. beim Zugriff verbunden, der von der Größe des Datensatzes im Wesentlichen unabhängig ist. Für die Erstellung und den lesenden Zugriff auf viele sehr kleine Dateien und Datensätze sind Plain-File-Systeme daher eher ungeeignet.

4.3.2.3 Techniken zur Datenreduktion

Bei geschickter Wahl der Einhängepunkte von Partitionen in die Verzeichnisstruktur erweist sich bei geeigneter Organisation insbesondere das Archivieren und Löschen als unkompliziert. Zum einen existieren viele bewährte Hilfswerkzeuge zur Archivierung und Kompression von Dateien und Verzeichnissen (*tar*, *gzip*, *bzip2*, ...). Andererseits kann, wenn eine ganze Partition zu löschen ist, diese einfach aus dem Verzeichnisbaum ausgehängt und formatiert werden, was in der Regel nur wenige Sekunden in Anspruch nimmt. Danach kann die gleiche Partition – an einer geeigneten Stelle in den Verzeichnisbaum eingehängt – wieder neue Daten aufnehmen. Sollte jedoch z. B. vor der Löschung der Mittelwert eines Messwerts über eine große Anzahl an Dateien ermittelt werden, so sind sehr viele Lesezugriffe nötig, was die benötigte Zeit für diese Operation stark ansteigen lässt.

4.3.2.4 Zusammenfassung

Plain-File-Datenlager können unter allen aktuellen Betriebssystemen leicht realisiert werden. Der Vorteil der an sich intuitiven Nutzung relativiert sich allerdings durch das Fehlen integrierter Dienste zur Verwaltung und Organisation. Die Schaffung geeigneter Strukturen bietet zwei Vorteile: einerseits ermöglicht sie eine effiziente uneingeschränkte Datenlöschung, andererseits lassen sich bekannte Messwerte strukturiert speichern und gezielt auslesen. Unbekannte Messwerte hingegen sind nur durch Vorgabe obligatorischer Felder innerhalb eines Datensatzes eingeschränkt abbildbar.

Plain-File-Systeme eignen sich gut zur Ablage großer Datensätze, die später gar nicht oder nur selten modifiziert werden müssen, weil Anfragen auf diese Daten relativ aufwändig sind, da ohne Indizierung ganze Dateien durchsucht werden. Insbesondere sind Anfragen, die das Durchsuchen mehrerer Datensätze erfordern, schnell ineffizient.

4.3.3 Relationale Datenbank-Management-Systeme

4.3.3.1 Grundlagen

Ein relationales Datenbank-Management-System (DBMS) ist eine Software, die die einheitliche Beschreibung und sichere Bearbeitung einer Datenbank ermöglicht. Dabei garantiert das DBMS u. a. die Einhaltung von Konsistenzregeln zur Gewährleistung der Korrektheit

der Daten, sowie Datensicherheit nach Programm- und Systemabstürzen und funktionierenden Mehrbenutzerbetrieb. In der Regel wird auch eine fein abstufbare Steuerung von Zugriffsrechten angeboten [41].

Alle relationalen Datenbanken können in drei voneinander unabhängige Ebenen eingeteilt werden:

Externe Ebene: Dies ist die einzige Ebene, zu der Anwender- bzw. Anwenderprogramme Zugang haben. Ihnen wird eine so genannte Sicht (engl: View) auf den Datenbestand eingeräumt. Jede Anwendung kann dabei eine eigene Sicht auf die Daten erhalten.

Konzeptuelle oder logische Ebene: Diese Ebene garantiert die logische Datenunabhängigkeit, indem logische Strukturen bereitgestellt werden, die die Daten enthalten. In einem relationalen Umfeld handelt es sich dabei um Tabellen, die nach dem Entity-Relationship-Modell entworfen werden [41]. Dabei wird ein Ausschnitt der realen Welt – die so genannte Miniwelt – mit Hilfe von Entity-Typen, Datentypen und Beziehungen[5] vereinfacht dargestellt (modelliert). Entity-Typen beschreiben mit den erlaubten Datentypen eigenständige Gegenstandsmengen, mehrere Entity-Typen werden über Beziehungen miteinander verbunden. Abbildung 4.1 und Tabelle 4.2 veranschaulichen dies an einem sehr einfachen Beispiel.
Änderungen an der logischen Struktur sind für Anwendungen und Abfragen unsichtbar, solange die entsprechenden Sichten auf den Datenbestand der veränderten logischen Struktur angepasst werden.

Interne oder physikalische Ebene: Diese Ebene garantiert die physische Datenunabhängigkeit. Hier werden die konkrete physikalische Speicherung und Datenverwaltung geregelt. Änderungen der Speicherstrukturen sind für die logische und externe Ebene nicht sichtbar.

Weitere wichtige Eigenschaften von DBMS werden im Folgenden kurz aufgeführt und erläutert:

- Datenschutz: Das DBMS kontrolliert den Zugriff auf Daten und deren Sichtbarkeit.
- Datensicherheit: Das DBMS garantiert, dass die Datenbank immer in einem konsistenten Zustand ist. Dies wird durch Einführen von Transaktionen erreicht. Jede Modifikation des Datenbestands findet im Rahmen einer Transaktion statt. Erst nach dem Beenden einer Transaktion werden die modifizierten Daten in der Datenbank

[5] engl.: relationship = Beziehung, Verwandschaft

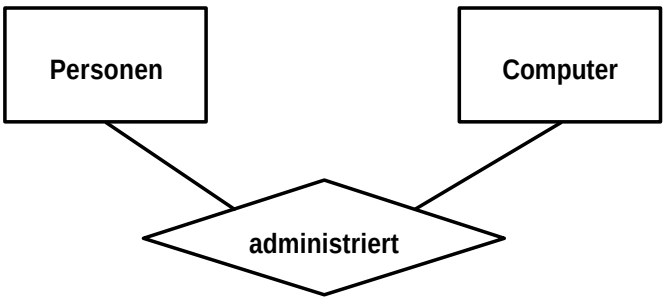

Abbildung 4.1: Sehr einfaches Beispiel für den konzeptionellen Entwurf im Entity-Relationship-Modell. *„Computer“* und *„Personen“* sind Gegenstandsmengen (Entity-Typen), die Relation *„administriert“* hingegen stellt eine Beziehung zwischen einem Computer und dessen Administrator her.

Personen

Pers. Nr.	Name
007	Bond
0815	Schmidt
...	...

Computer

Inventarnr.	OS
KIP/308A	FreeBSD
KIP/308B	SuSE 9.0
...	...

administriert

Inventarnr.	Pers. Nr.
KIP/308A	0815
KIP/308B	007
...	...

Tabelle 4.2: Relationale Darstellung des in Abbildung 4.1 vorgestellten, sehr einfachen Entwurfs im Entity-Relationship-Modell. Die Relation *„administriert“* nimmt eine Sonderstellung ein, da sie die Beziehung zwischen einem Computer und dessen Administrator modelliert.

als gültig markiert. Dieser Vorgang wird als „Commit“[6] bezeichnet. Man kann jedoch eine Transaktion auch vollständig zurücksetzen („Rollback“), also alle Modifikationen verwerfen.

- Konsistenzprüfung / Integritätskontrolle: Das DBMS garantiert, dass ihm bekannte Konsistenzregeln (z. B. „Jeder Rechner hat genau einen Administrator“) eingehalten werden.
- Kontrolle gleichzeitiger Zugriffe: Das DBMS stellt sicher, dass mehrere parallel aktive Transaktionen isoliert voneinander und konsistent ablaufen.

[6] engl.: festlegen, binden, übergeben

- Archivierungsdienste: Viele DBMS bieten Funktionen zum Sichern und Archivieren der Datenbestände an. Dieser Punkt wird später noch näher beleuchtet.

Relationale Datenbanken haben somit sehr viele Vorteile gegenüber einem Plain-File-Datenlager. Weiterhin existiert eine einheitliche Schnittstelle – die **S***tructured* **Q***uery* **L***anguage* (SQL) – mit deren Hilfe man u. a. Daten, die in relationalen Datenbanken gespeichert sind, auslesen und manipulieren kann. Auch ermöglicht SQL ein einfaches Einfügen neuer Daten. Neben dem Datensatz ist lediglich die Angabe eines abstrakten Ziels (Tabelle) erforderlich, d. h. es wird keine Kenntnis darüber vorausgesetzt, wo und wie die Daten physikalisch abgelegt werden. Auch Abfragen (engl.: queries) an die Datenbank, gestalten sich einfach und dennoch flexibel. So kann man sich z. B. leicht mit einem Befehl alle Werte eines Sensors eines bestimmten Knotens innerhalb eines Zeitfensters ausgeben lassen, ohne sich mit der Organisation und der Verwaltung der Daten zu beschäftigen . Weiterhin bieten heutige Datenbanken ausgefeilte Transaktions- und Backupmechanismen an. Man kann daher die Gefahr von erheblichem Datenverlust bei geeigneter Konfiguration quasi vernachlässigen.

4.3.3.2 Verarbeitung in XML strukturierter Messdaten

Wie bei Plain-File-Systemen auch, können bei bekanntem (XML-) Schema Messdaten mit Hilfe eines Parsers leicht extrahiert werden, um diese dann problemlos in einer relationalen Datenbank zu speichern. Abbildung 4.2 zeigt exemplarisch, wie die Struktur und der Inhalt eines kumulierten XML-Dokuments (enthält viele einzelne Messwerte unterschiedlicher Sensoren) auf bereits vorhandene Strukturen einer relationalen Datenbank abgebildet wird.

Anders verhält es sich bei unbekanntem Schema der Sensordaten. In relationalen Datenbanken ergibt sich ein ähnliches Problem wie bereits bei den Plain-File-Systemen: Mapping eines unbekannten XML-Dokumentes ist nur bei Vorgabe obligatorischer Felder im Datensatz möglich. Es wird ansonsten immer mindestens ein Schema benötigt, um die nötigen Tabellen anzulegen und die Konvertierung durchzuführen. Schemalose Daten können bestenfalls als so genanntes *Binary Large Object* (BLOB), also nur als strukturlose Einheit gespeichert werden.

Relationale Systeme unterliegen nur sehr schwachen Einschränkungen bezüglich der Größe eines Datensatzes oder des Datentyps eines Messwertes. Einzelne binäre Objekte können je nach Implementation des DBMS sogar einige Gigabyte groß werden – Datengrößen, die im

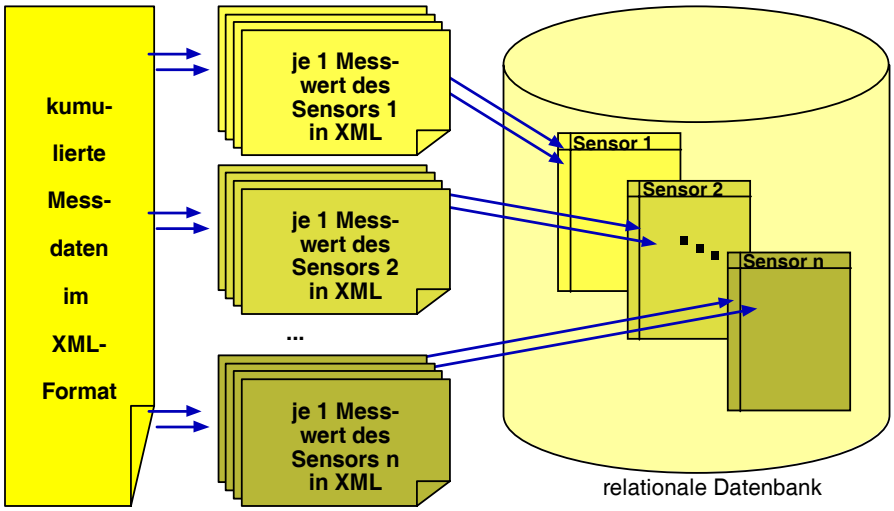

Abbildung 4.2: Beispiel für XML-Mapping in eine relationale Datenbank. Die Struktur und der Inhalt eines XML-Dokument wird auf bereits vorhandene andere Strukturen abgebildet wird.

Umfeld von Monitoring nicht anfallen werden, da selbst sehr große einzelne Messwerte nicht die Größenordnung einiger Megabyte überschreiten werden.

4.3.3.3 Techniken zur Datenreduktion

Aktuelle relationale DBMS sind in der Lage, Datenmengen im Bereich mehrerer Gigabyte problemlos zu organisieren. Probleme können jedoch dann auftreten, wenn die Datenmengen in den Bereich vieler Hundert Gigabyte oder in den Terabyte-Bereich anwachsen ([42]). Grid-Monitoring wird diese Datenmengen in relativ kurzer Zeit erreichen (vgl. Abschnitt 4.2.3), so dass sowohl Datenlöschungs- als auch Archivierungsstrategien vorhanden sein sollten.

Datenlöschung Relationale DBMS trennen zwischen logischem und physikalischem Datenbestand. Somit sollte eine gezielte Datenlöschung in relationalen Systemen durch die zur Verfügung gestellten Schnittstellen und nicht physikalisch durch Löschung einer Datei oder Formatierung einer Partition erfolgen. Durch entsprechende SQL-Befehle ist es möglich, eine gewünschte Tabelle oder Teile davon zu löschen. Eine Datenreduktion ist somit auch leicht zu realisieren (Bsp.: man ermittelt über eine entsprechende Datenbankabfrage einen Mittelwert, speichert diesen ab und löscht daraufhin die ursprünglichen Daten). Diese Form der Datenlöschung muss von außerhalb des Datenlagers angestoßen werden.

Archivierung Zur Archivierung existieren zwei unterschiedliche Ansätze, die im Folgenden näher erläutert werden:

datenbanksystembasierte Archivierung Bei der datenbanksystem**basierten** Archivierung werden die zu archivierenden Daten mittels einer Anwendungsschicht aus der Datenbank herausgeholt und in ein separates Archiv verschoben (und somit aus der Datenbank gelöscht) [43]. Das Lesen archivierter Daten und ein mögliches Zurückladen in das DBMS muss allerdings von außerhalb des DBMS angestoßen werden und zeigt die Schwerfälligkeit von datenbanksystembasierter Archivierung, deren entscheidender Nachteil darin liegt, dass sich die archivierten Daten nicht unter Kontrolle des DBMS befindet. Dieses Verfahren ist in der Praxis weit verbreitet, wobei es von den verschiedenen DBMS unterschiedlich gut unterstützt wird. Bekannte Anwendungssysteme, wie das *System R/3* von SAP [44], verfolgen ein Archivierungssystem, das auf diesem einfachen Vorgehen basiert. Auch Oracle[7] bietet „von Haus aus" die Möglichkeit an, Tabellenbereiche „offline" zu setzen und auf Tertiärspeicher auszulagern, bis eine spätere Nutzung eine Wiedereinlagerung erforderlich macht.

datenbanksystemintegrierte Archivierung Das theoretische Gegenstück zur datenbanksystembasierten Archivierung ist die datenbanksystem**integrierte** Archivierung. Man versteht darunter, dass das DBMS selbst Archivierungsdienste anbietet. Der große Vorteil dieser Methode liegt darin, dass die Datenbank das Archiv kennt und auch kontrolliert. Dadurch eröffnen sich Möglichkeiten, die wegen der sonst losen Anbindung zwischen Datenbank und Archiv nicht ohne weiteres realisierbar wären. So könnte z. B. eine Datenbank anhand der Historie automatisch erkennen, ob Daten oft abgefragt werden oder nicht. Daraufhin würde sie selten benötigte Daten eigenständig aus- und nur bei Bedarf wieder einlagern. In den heutigen (relationalen) DBMS wird eine solche Funktionalität leider noch nicht angeboten [42].

Relationale DBMS bieten somit zur Zeit nur die Möglichkeit der datenbanksystembasierten Archivierung.

4.3.3.4 Zusammenfassung

Das Konzept relationaler DBMS ist bereits sehr ausgereift und es existieren viele bewährte Implementationen. Das Modifizieren der Datenbestände wird durch die mächtige, standardisierte Schnittstelle SQL ermöglicht, welche auch komplexe Anfragen unterstützt. Eine

[7] Oracle [26] ist der größte kommerzielle Anbieter für relationale Datenbanken.

Reihe zusätzlicher Dienste vereinfacht die Behandlung von Ausnahmesituationen. Im Rahmen eines Datenlagers für Monitoring eignen sich relationale Systeme gut zur Speicherung bekannter Messdaten. Unbekannte Messwerte hingegen können nur umständlich und ineffizient verarbeitet werden. Weiterhin unterstützen relationale Datenbanken eine selektive Datenlöschung sowie die datenbanksystembasierte Archivierung zur Datenreduktion. Zur Zeit wird noch von keiner relationalen Implementation eine datenbanksystemintegrierte Archivierung angeboten.

4.3.4 Native XML-Datenbanken

Den bisher vorgestellten Datenlagerungssystemen ist gemeinsam, dass sie Messwerte fest strukturierter Datenquellen bei vorheriger Kenntnis der Struktur durch Schaffung geeigneter Infrastruktur (Verzeichnisse bzw. Tabellen) relativ einfach aufzunehmen vermögen. Komplexe unbekannte Datenstrukturen und Abhängigkeiten können so allerdings nicht (ohne weiteres) sinnvoll abgebildet werden. Dies ist bei nativen XML-Datenbanken hingegen möglich.

4.3.4.1 Grundlagen

Durch die Verwendung von XML zur Darstellung von Messdaten gewinnt man die Möglichkeit, komplexe Daten außerhalb einer vorher festgelegten Struktur zu formulieren; die bisher vorgestellten Datenlager konnten diesen Vorteil allerdings nur begrenzt nutzen. Sie sind stets auf Mapping angewiesen, um die Daten für ihre Ablageorganisation strukturieren zu können. Benutzt man hingegen eine Datenbank, die XML nativ versteht, muss man im Prinzip kein Wissen über die ankommenden Daten besitzen, bis auf die Tatsache, dass sie XML-konform sind. Der Vorteil nativer XML-Datenbanken ist, dass sie die Dokument-Reihenfolge, Kommentare und die logische Struktur komplexer Datensätze oder ganzer Dokumente erhalten. Das ist unter Plain-File-Datenlagern sowie relationalen DBMS meist nur durch redundante Speicherung des gesamten XML Dokumentes möglich, deren Nachteile bereits diskutiert wurden.

Weiterhin bietet eine XML-Datenbank die von den relationalen Systemen bekannten Dienste wie Datenschutz, Datensicherheit, Konsistenzprüfung und Kontrolle gleichzeitiger Zugriffe sowie Archivierungsdienste. Insbesondere ermöglicht sie die in Kapitel 4.3.3 vorgestellte Datenbank typische Trennung zwischen logischer und physikalischer Datenhaltung.

Die Größe der Datensätze ist ähnlich zu den relationalen Systemen kaum Einschränkungen unterworfen. Allerdings unterscheidet insbesondere die Möglichkeit der Verwendung beliebiger komplexer Datentypen die nativen XML-Datenbanken von allen anderen Systemen und stellt einen ihrer größten Vorteile dar.

XML-Datenbanken sind allerdings nicht für den Einsatz in allen Gebieten geeignet, so dass sie relationale Systeme auf absehbare Zeit nicht ersetzen können. Eine Datenverwaltung in Tabellen ist – gleiche und gleichmäßige Struktur der Daten vorausgesetzt – aufgrund des geringeren Verwaltungsaufwands in der Regel effizienter. XML-Datenbanken spielen vielmehr bei komplexen, unregelmäßigen oder unbekannten Datenstrukturen ihren Vorteil aus. Allerdings müssen sie selbst dort, wo sie relationalen Datenbanken Konkurenz machen können, erst den Reifegrad etablierter relationaler DBMS erreichen, da das Gebiet der nativen XML-Datenbanken noch sehr neu ist. Dies liegt nicht zuletzt daran, dass XML erst vor etwas über sechs Jahren (Februar 1998) als Standard verabschiedet wurde. Seitdem ist die Verbreitung von XML rasant fortgeschritten und mit ihr die Nachfrage nach nativen XML-Datenbanken. Im Gegensatz zu relationalen DBMS existieren aufgrund der kurzen Zeit allerdings noch keine allgemein akzeptierten Standards, so dass Entwickler nativer XML-Datenbanken ihre eigenen Standards entwickeln, wodurch künftige Kompatibilitätsprobleme wahrscheinlich sind.

Anfragesprachen Damit eine XML-Datenbank auch tatsächlich den Anforderungen an Datenbanken gerecht wird, muss eine geeignete Anfragesprache existieren. Da SQL auf relationale Systeme zugeschnitten ist und somit Tabellen und Spalten voraussetzt, bietet es offensichtlich keine Möglichkeit, um geeignete Anfragen auf XML-Dokumente zu formulieren. Um flexible Anfragen stellen zu können, muss eine Anfragesprache mindestens folgende Eigenschaften erfüllen:

- Eine Anfrage muss sich auf mehrere Fragmente eines XML-Dokuments bis hin zu einzelnen Elementen beziehen lassen
- Eine Anfrage muss sich auf beliebig viele XML-Dokumente formulieren lassen
- Daten verschiedener Quellen müssen zusammengeführt werden können
- Man muss Operationen auf gefundenen Informationen durchführen können
- Informationen sollten sich nach vorgegebenen Kriterien bewerten / sortieren lassen

Je mehr voneinander abhängige und unabhängige Operationen eine Anfragesprache erlaubt, desto höher ist ihre Ausdrucksstärke.

In der Zwischenzeit wurden vom W3C die bisherigen XML-Anfragesprachen wie XLS und XPATH in der Sprache XQuery zusammengefasst. Durch eine (trotz vieler Unterschiede) durch SQL vertraute Syntax und Struktur wird zur Zeit versucht, XQuery als eine mächtige und leicht zu erlernende Anfragesprache für den Zugriff auf XML-Daten zu etablieren. Dennoch existieren einige offensichtliche Unterschiede zwischen XQuery und SQL. XQuery bietet keine Möglichkeiten zum Modifizieren der Daten innerhalb der Datenbank. Für solche Aufgaben müssen zur Zeit noch die proprietären Lösungen der Hersteller oder eigene Algorithmen, welche auf die meist vorhandene DOM-Schnittstelle aufsetzen, verwendet werden.

4.3.4.2 Techniken zur Datenreduktion

Da XML-Datenbanken an das Prinzip relationaler Datenbanken angelehnt sind, stehen ihnen die gleichen theoretischen Techniken zur Datenreduktion zur Verfügung wie den relationalen Systemen. Prinzipiell steht somit einer datenbanksystembasierten Archivierung und Datenlöschung mittels einer Anwendungsschicht nichts entgegen, jedoch kann eine solche Lösung nicht wie mit SQL einheitlich für alle nativen Datenbanken verwendet werden, da XQuery eine Modifikation und somit die Löschung von Daten nicht unterstützt. Eine selektive Löschung innerhalb des Datenbestands ist einheitlich nur durch prozedurale Nutzung der DOM-Schnittstelle möglich, wodurch jedoch bei weitem nicht die Flexibilität und Effizienz von SQL-Modifikationen auf den Datenbestand erreicht wird. Da die Hersteller nativer XML-Datenbanken sich zur Zeit sehr auf die Entwicklung der grundlegenden funktionalen Eigenschaften konzentrieren, wurden bis jetzt, wenn überhaupt, nur sehr rudimentäre Archivierungsdienste realisiert.

4.3.4.3 Zusammenfassung

Native XML-Datenbanken können beliebig strukturierte, sogar schemalose XML-Dateien in ihrer ursprünglichen Form speichern. Dabei stehen dem Anwender im Prinzip alle Datenbank typischen Vorteile zur Verfügung. Aufgrund des frühen Entwicklungsstadiums dieser Datenbank-Familie jedoch wird, insbesondere im Bereich der Archivierung, noch nicht die Breite der von relationalen Systemen bekannten Funktionalität angeboten.

Die Standardisierung einer einheitlichen Anfragesprache ermöglicht den Hersteller unabhängigen lesenden Zugriff auf den Datenbestand, allerdings existiert kein Standard zu dessen Modifikation. Dies wiederum erschwert die Entwicklung implementationsunabhängiger Techniken zur Datenreduktion.

4.4 Zusammenfassung

In diesem Kapitel wurde ein Überblick über verschiedene Familien von Datenlagerungssystemen vermittelt. Dabei wurde ihre prinzipielle Funktionsweise vorgestellt sowie auf ihre Eignung als Datenlager für Monitoringsystem eingegangen. Es wurde herausgearbeitet, dass Round-Robin-Datenbanken sich nicht für dieses Einsatzgebiet eignen. Bei den anderen Datenlagern wurden stets Vor- und Nachteile festgestellt, so dass sich kein abschließendes Urteil bilden lässt. Deswegen werden im nächsten Kapitel verschiedene Tests an ausgewählten Vertretern jeder Familie durchgeführt.

5 Tests verschiedener Datenlager

5.1 Einleitung

In diesem Abschnitt wird bis auf die Round-Robin-Datenbanken je ein ausgewähltes Mitglied der in Kapitel 4 vorgestellten Familien von Datenlagerungssystemen auf seine Eignung für Monitoringsysteme getestet. Die ausgewählten drei Familien weisen grundlegende Eigenschaften auf, die sie für die nähere Betrachtung qualifizieren:

- Daten können theoretisch beliebig lange aufbewahrt werden.
- Es existiert Software auf dem Markt, mit der sich das Datenlager realisieren lässt.
- Es existieren geeignete Schnittstellen, um aus einer Programmiersprache heraus das Datenlager zu manipulieren.

Ob sich jedoch alle Familien gleichermaßen für ein umfassendes Monitoring wie in Kapitel 3 definiert eignen, soll dieser Test zeigen. Dabei wird auf die prinzipiellen Eigenschaften mehr Wert gelegt als auf eine spezielle Implementation.

5.2 Testumgebung

Für alle Messungen wurden stets nur Knoten aus dem „hauseigenen“ Cluster des Lehrstuhl für Technische Informatik [45] (TI-Cluster) der Universität Heidelberg verwendet. Hier eine kurze Auflistung wichtiger Kenndaten:

- **Betriebssystem:** SuSE Linux 9.0
- **Kernel:** 2.4.23
- **Prozessor:** 2 x Intel Pentium III (Coppermine) mit je 800 MHz
- **Hauptspeicher:** 512 MB PC-133 SD-RAM
- **Netzwerk:** 100 Mbit Fast Ethernet

- **Festplatte mit 2 MB Cache**
- **relationale Datenbank:** MaxDB V7.5.0.8
- **XML-Datenbank:** Natix Prerelease

5.3 Plain-File-Datenlager

Das Plain-File-Datenlager wird für die Performance-Messungen mit ANSI-C-Programmen [46] implementiert. Dabei erstellen diese jeweils *N* Dateien in einer komplexen Verzeichnisstruktur. Anschließend werden einige ausgewählte Verzeichnisse nach einer vorgegebenen Zeichenkette durchsucht. Zum Durchsuchen wurden zwei Verfahren getestet: Zum einen verwenden *Shell*[1]-Skripte das unter Unix bekannte Tool *grep*, zum anderen öffnet ein selbst geschriebenes C-Programm jede benannte Datei und untersucht die ausgelesene Zeichenkette mit Hilfe der *strstr()*-Funktion der Bibliothek *string.h*.

Die *Shell*-Skripte haben den Vorteil, dass sie sehr leicht zu implementieren sind. Von großem Nachteil sind die eingeschränkte Wartungsfähigkeit und Flexibilität sowie die begrenzte Portabilität, da die Skripte von der verwendeten Shell – in diesem Fall der *Bash*-Shell – abhängig sind. Auch C-Programme können abhängig von der verwendeten Plattform sein. Der ANSI-C-Standard definiert jedoch genau, welche Eigenschaften und Funktionen in welchem Umfang plattformübergreifend zur Verfügung stehen. Bei geeigneter Programmierung kann daher ein ANSI-C-konformes Programm auf allen Systemen, für die ein entsprechender Compiler zur Verfügung steht, genutzt werden.

5.3.1 Verwendete Dateisysteme

Die Tests wurden mit zwei verschiedenen Dateisystemen, *ReiserFS* und *Ext3* durchgeführt. Dabei handelt es sich ausschließlich um so genannte Journaling Dateisysteme. Deren wesentlicher Vorteil liegt darin, dass sie ein Protokoll (Journal) ihrer Manipulation auf dem Datenbestand führen. Bei den meisten (älteren) Dateisystemen werden bei Schreibzugriffen die Daten zur Geschwindigkeitssteigerung zuerst in den Cache[2] und erst später in einer vom Cache definierten Weise auf den Datenträger geschrieben. Ein ungeplanter Neustart birgt somit das Risiko einer Inkonsistenz im Dateisystem. So kann es z. B. vorkommen, dass

[1] Kommandozeilen-Interpreter

[2] Schneller Zwischenspeicher (Puffer), der häufig benutzte Daten aufnimmt und bei Bedarf wieder zur Verfügung stellt. Der tatsächliche Nutzen hängt stark von der Lokalität der Daten bzw. Programme ab.

noch nicht „endgültig“ gespeicherte Daten verloren gehen. Stürzt der Rechner während der Manipulation des Root-Verzeichnisbaums ab, kann sogar das ganze Dateisystem zerstört werden. Bei dem einem unkontrollierten Abbruch folgenden Neustart werden daher alle Dateien überprüft, was abhängig von der Anzahl der Dateien Stunden bis Tage dauern kann. Für ein Datenlager ist dies ein eindeutiges Ausschlusskriterium, da dieses im Falle eines ungeplanten Neustarts schnell wieder einsatzbereit sein muss.

Bei einem Journaling Dateisystem werden, ähnlich wie bei Datenbanken, Transaktionen eingeführt, die protokolliert werden. Sobald eine „Transaktion“ beginnt, wird dies im Journal registriert. Anschließend werden die Daten geschrieben, dann das erfolgreiche Ende der Transaktion im Journal festgehalten (Logging). Der wesentliche Vorteil eines Journaling Dateisystems liegt nun darin, dass nach einem Systemabsturz oder -ausfall dank des Protokolls auf zeitaufwändige Überprüfung aller Dateien und Strukturen verzichtet werden kann. Es muss nur der Bruchteil der Dateien überprüft werden, bei dem kein Eintrag über die erfolgreiche Beendigung der Transaktion erfolgt ist.

ReiserFS unterschiedet sich von *Ext3* vor allem in der Organisationstruktur der so genannten Filetable. Dabei handelt es sich um ein Index-Register, welches darüber Auskunft gibt, wo genau eine Datei physikalisch auf dem Datenträger aufzufinden ist. *Ext3* verwendet – wie sein direkter Vorgänger *Ext2* auch – eine ungeordnete Liste zur Organisation. Der Aufwand für das Hinzufügen neuer Dateien sowie der Suchaufwand in der Filetable beim gezielten Zugriff auf eine Datei ist daher von der Ordnung $O(n)$, wobei n die Anzahl der abgelegten Dateien im Dateisystem ist.

ReiserFS dagegen verwendet so genannte B-Trees (Balanced Trees). Dabei handelt es sich um ausgeglichene Bäume, wie sie bei Datenbanken schon seit vielen Jahren eingesetzt werden. Die Blätter eines balancierten Baumes enthalten den Verweis auf die Daten, die Knoten des Baums enthalten Verweise auf die nächste Baumebene oder schließlich die Blätter. Ein Baum ist dabei dann balanciert, wenn sich die Anzahl der zu durchlaufenden Knoten auf dem Weg von der Wurzel zu jedem beliebigen Blatt nicht mehr als um eins unterschiedet. Man bezeichnet die maximale Anzahl der zu durchlaufenden Knoten auch als die Höhe h des Baumes.

Dadurch, dass beim Hinzufügen der Baum evtl. neu balanciert werden muss (spätestens beim Erzeugen einer neuen Ebene), erhöht sich der Verwaltungsaufwand beim Schreiben neuer Dateien. Das Suchen hingegen ist recht unaufwändig. Für den Zugriff auf eine beliebige Datei muss man lediglich maximal h Knoten durchlaufen. Der Suchaufwand (beim gezielten Zugriff

Sensoren s	Knoten k	Tage d	Stunden h	Messwerte/Std. n	Messwertgröße g [Bytes]
100	10	1	4	1500	55

Tabelle 5.1: Die Wahl der Parameter für die durchgeführten Tests zum Schreibverhalten eines Plain-Files-Systems mit simulierten Messdaten. Selbst bei einem eingeschränkten Testlauf auf nur 4 Stunden pro Tag entstehen bereits $6 \cdot 10^6$ Datensätze, jeder wird in einer eigenen Datei abgelegt.

auf eine Datei) wächst somit nur logarithmisch mit der Anzahl der abgelegten Dateien im Dateisystem an. Für weitergehende Performancetests verschiedener Dateisystemen, darunter auch *Ext3* und *ReiserFS*, sowie für einen Überblick über deren Vorzüge und Nachteile wird auf [47] verwiesen.

5.3.2 Vorgehensweise der Schreibtests

Die entwickelte Testsoftware erstellt in einem ersten Schritt die komplette Verzeichnisstruktur. Für jeden Sensor s wird dabei ein Unterverzeichnis angelegt. In jedem „Sensor"-Verzeichnis werden nun Verzeichnisse für jeden simulierten Knoten k erstellt, der Daten liefern wird. Somit existieren bereits $s \cdot k$ Verzeichnisse. Weiterhin werden in jedem „Knoten"-Verzeichnis so viele Unterverzeichnisse angelegt, wie Messtage d simuliert werden sollen. Jeder Tag enthält dabei nochmals h Unterverzeichnisse – eines für jede Stunde des Messtages. Somit sind $s \cdot k \cdot d \cdot h$ Verzeichnisse auf der untersten Ebene vorhanden.

Im zweiten Schritt werden nun in diesen Unterverzeichnissen die simulierten Messdaten als einzelne Dateien abgelegt. Alle Parameter (Anzahl der Sensoren, Knoten, Tage, Stunden, Messdaten pro Stunde n, durchschnittliche Größe g eines Messwertes, sowie das Ursprungsverzeichnis für den Test) können dabei innerhalb sinnvoller Grenzen frei gewählt werden.

Um die Tests realistisch zu gestalten, wurden die Parameter so festgesetzt, dass sie den Abschätzungen aus Tabelle 4.1 nahe kommen und dennoch die Tests in absehbarer Zeit beendet werden können. Tabelle 5.1 zeigt die gewählten Parameter. Insgesamt werden somit $4 \cdot 10^3$ Verzeichnisse mit je $1,5 \cdot 10^3$ Dateien, also $N = 6 \cdot 10^6$ Dateien mit einem gesamten Datenvolumen von ca. 315 Megabyte angelegt.

Um den Einfluss der Reihenfolge der Erstellung beim Schreiben sowie beim Lesen zu untersuchen, wurden einmal die Verzeichnisse nacheinander in ihrer logischen Reihenfolge gefüllt (zuerst alle n Messdaten für [Sensor 1, Knoten 1, Tag 1, Stunde 0], danach für die Stunden

1, 2, 3, usw.). Dies entspricht allerdings nicht der Reihenfolge, in der die Daten in der Realität geschrieben werden. Dort erwartet man eine Reihenfolge, in der eine zeitliche Nähe der Messungen sich in der Reihenfolge der Ablage widerspiegeln wird. Deswegen wurden in einem zweiten Test die Dateien nicht logisch sondern „chronologisch“, d. h. in der Reihenfolge ihrer Messung, erstellt.

5.3.3 Messergebnisse des Schreibtests

Um eine Datei zu schreiben müssen in der Regel zwei Dinge passieren: Das Dateisystem muss erstens den Inhalt der Datei auf einem freien Speicherplatz auf der Festplatte speichern und zweitens einen Eintrag in der Filetable vornehmen. Diese enthält, neben der genauen Position der Datei, weitere Metainformationen wie z. B. Dateiname und -Größe. Die genaue Organisation einer solchen Filetable ist dabei wie oben besprochen dateisystemspezifisch.

Festplatten sind so genannte Block-Devices. Das bedeutet, dass sie nur blockweise Daten lesen und schreiben können. Die typische Blockgröße einer Festplatte beträgt 512 Bytes. Werden Daten am Stück, also Block hinter Block, geschrieben, so sind leicht Transferraten von vielen MB/s zu erreichen, da der Lesekopf nur wenig bewegt werden muss. Das häufige Hin- und Herbewegen des Lesekopfs ist hingegen sehr langsam – marktübliche Festplatten haben eine mittlere Zugriffszeit von ca. 10 ms bei wahlfreiem Zugriff auf das Medium. Dies entspricht einer Frequenz von ca. 100 Hz, mit der man willkürlich auf die Platte zugreifen kann. Da es sich bei den Tests allerdings nicht um willkürliche Zugriffe handelt und außerdem Festplatten zwecks Zugriffsbeschleunigung einen eigenen Cache mitbringen, wird die tatsächliche Zugriffsfrequenz voraussichtlich von diesem Wert abweichen.

Zwischen dem C-Programm, das die Daten schreibt, und dem entsprechenden Device liegen allerdings noch weitere Caches. Neben dem bereits erwähnten (2 bis 8 MB großen) Festplattencache stellt das Dateisystem zusätzlich dynamische Page- sowie Buffer-Caches zur Verfügung. Das Zusammenspiel vieler Caches ist eine sehr komplexes System, dessen Interpretation nicht trivial ist.

5.3.3.1 Logische Reihenfolge

Die in Tabelle 5.2 gezeigten Messergebnisse der Schreibtests in logischer Reihenfolge ergeben, dass $6 \cdot 10^6$ Dateien in etwas über einer halben Stunde geschrieben werden können. *ReiserFS* ist dabei rund 20 % schneller als *Ext3* und schreibt die Dateien mit einer Frequenz

	ReiserFS log.	*Ext3* log.
benötigte Zeit t [s]	1890	2323
Schreibfrequenz [Hz]	3175	2583

Tabelle 5.2: Ergebnisse der Messung für die Erzeugung von $6 \cdot 10^6$ Dateien in **logischer Reihenfolge** unter *ReiserFS* und *Ext3*.

	ReiserFS chron.	*Ext3* chron.
benötigte Zeit t [s]	53 814	322 200
Schreibfrequenz [Hz]	111	19

Tabelle 5.3: Ergebnisse der Messung für die Erzeugung von $6 \cdot 10^6$ Dateien in **chronologischer Reihenfolge** unter *ReiserFS* und *Ext3*.

von ca. 3,2 kHz (*Ext3*: 2,6 kHz). Unter beiden Dateisystemen ist zu beobachten, dass die mittlere Zugriffszeiten je Datei über den Verlauf des Tests relativ gleichmäßig um einen Mittelwert oszilliert (vgl. Abbildung 5.1). Die gemessenen Schwankungen liegen jedoch nur im Prozentbereich.

Der tatsächlich belegte Speicherplatz zur Ablage der $6 \cdot 10^6$ Dateien hängt ebenfalls vom verwendeten Dateisystem ab. *ReiserFS* belegt deutlich weniger Speicherplatz als *Ext3* – 1,2 GB im Vergleich zu 6,0 GB. Dennoch liegt der Speicherbedarf bei *ReiserFS* ein vielfaches über dem tatsächlichen Datenvolumen von ca. 315 MB. Die 4 kB (*ReiserFS*) bzw. 1 kB (*Ext3*) großen Blöcke des Dateisystems sind zum Teil die Ursache. *Ext3* kann pro Block max. eine Datei ablegen, so dass selbst bei sehr kleinen Dateien immer mindestens 1 kB Speicherplatz belegt wird. Dies erklärt die gemessenen 6 GB für $6 \cdot 10^6$ Dateien.

ReiserFS hingegen geht effizienter mit kleinen Dateien um. Es allokiert den benötigten Platz in exakter Größe und nicht in Blöcken zu je 4 kB. Der dennoch deutlich über dem tatsächlichen Datenvolumen liegende Speicherverbrauch könnte sich durch die Speicherung der ebenfalls unbedingt notwendigen, von der eigentlichen Dateigröße relativ unabhängigen Metainformationen erklären.

Die Überlegungen bezüglich des benötigten Plattenplatzes gelten unabhängig davon, in welcher Reihenfolge die Daten auf die Platte geschrieben werden. Die Messungen ergaben daher auch, dass der Speicherplatzverbrauch der folgenden chronologischen Tests genauso hoch ist, wie der ihrer logischen Pendants.

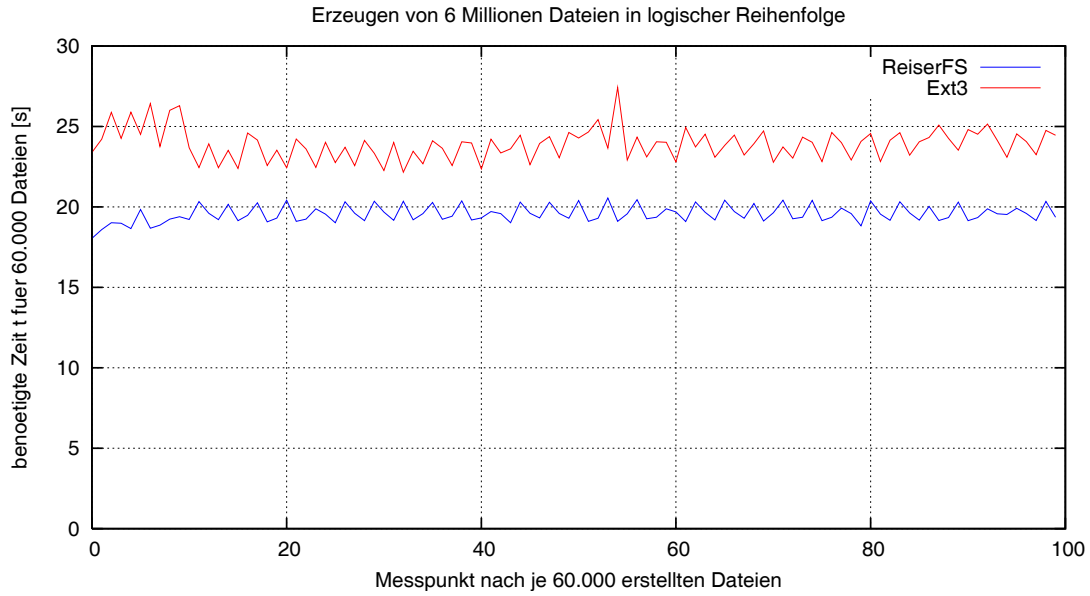

Abbildung 5.1: Verlauf der benötigten Zeit für die Erstellung von je $6 \cdot 10^4$ Dateien in **logischer Reihenfolge** bei insgesamt $6 \cdot 10^6$ erzeugten Dateien

5.3.3.2 Chronologische Reihenfolge

Das Ergebnis des chronologischen Tests ist ernüchternd (Tabelle 5.3). Um die geforderten $6 \cdot 10^6$ Dateien auf *ReiserFS* zu Schreiben, benötigt das Testsystem über fünfzehn Stunden und ist somit dreißig mal langsamer als der entsprechende logische Test (Abbildung 5.2). Nochmals deutlich schlechter als bei *ReiserFS* sind die Ergebnisse unter *Ext3*. Die Erstellung in chronologischer Reihenfolge dauert fast 70 Stunden und ist somit für den realen Einsatz völlig ungeeignet. Die benötigte Zeit zum Erstellen einer Datei steigt bei *ReiserFS* deutlich an, während bei *Ext3* starke Schwankungen von bis zu 1000 % gemessen werden (vgl. Abbildung 5.3).

Das Verhalten von *ReiserFS* könnte man qualitativ wie folgt deuten: Die Zugriffszeiten steigen an, da der B-Tree für jede weitere Datei größer wird. Er muss immer wieder neu gelesen und ausgeglichen werden. Weiterhin werden einige Metainformationen eines Verzeichnisses wie das Datum der letzten Änderung ebenfalls verändert, sobald in dem betreffenden Verzeichnis eine Datei angelegt wird. Bei der logischen Erstellung wurden ja viele Dateien im gleichen Verzeichnis direkt nacheinander eingefügt. Der B-Tree-Ausschnitt für das Verzeich-

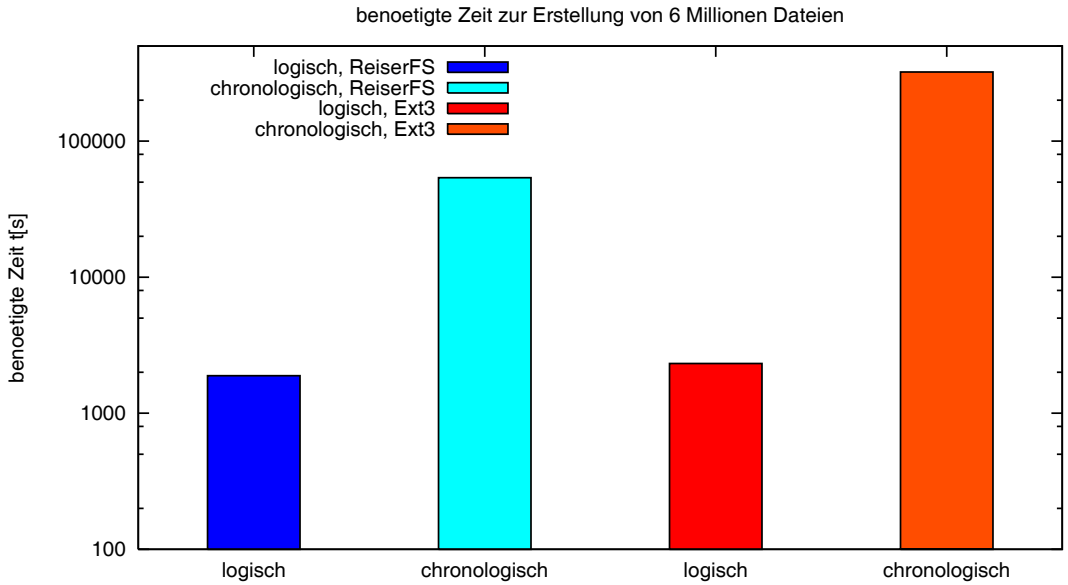

Abbildung 5.2: Logarithmische Darstellung der benötigten Zeit zum Erstellen von insgesamt $6 \cdot 10^6$ Dateien auf den Dateisystemen *ReiserFS* und *Ext3*, sowohl in logischer als auch in chronologischer Reihenfolge.

nis wird daher bereits nach dem ersten Zugriff im Buffer-Cache zur Verfügung stehen. Erst der Zugriff auf das nächste Verzeichnis kann eventuell einen „Cache-Miss“[3] erzeugen, der jedoch nicht ins Gewicht fällt, da der Cache ab jetzt wieder den aktuellen B-Tree-Ausschnitt enthält. Chronologisch hingegen werden völlig unterschiedliche Verzeichnisse angesprungen, bei denen jedes Mal der Baum von relativ nah bei der Wurzel durchlaufen werden muss. Diese Informationen finden sich nicht im Cache[4], so dass tatsächlich die Platte „befragt“ wird. Dadurch dauert jeder Zugriff entsprechend lange.

Diese Deutung wird weiterhin dadurch gestützt, dass *ReiserFS* die Dateien in diesem Test mit einer Frequenz von ca. 100 Hz auf die Platte schreibt, und sich somit etwa im Rahmen der Zugriffszeiten bewegt, die man von einer Platte bei wahlfreiem Zugriff ohne Caches erwarten würde. Da abwechselnd eine Datei geschrieben und die Filetable aktualisiert werden müssen, springt der Kopf wahrscheinlich hin und her.

[3] engl: Fehlschlag: Der gesuchte Eintrag ist nicht im Cache vorhanden.

[4] Nach dem Zugriff stehen sie durchaus im Cache, nur werden die Werte durch die nächsten Zugriffe wieder überschrieben, da andere Daten benötigt und diese dann im Cache abgelegt werden. Dieses Verhalten bezeichnet man auch als „Cache-Trashing“.

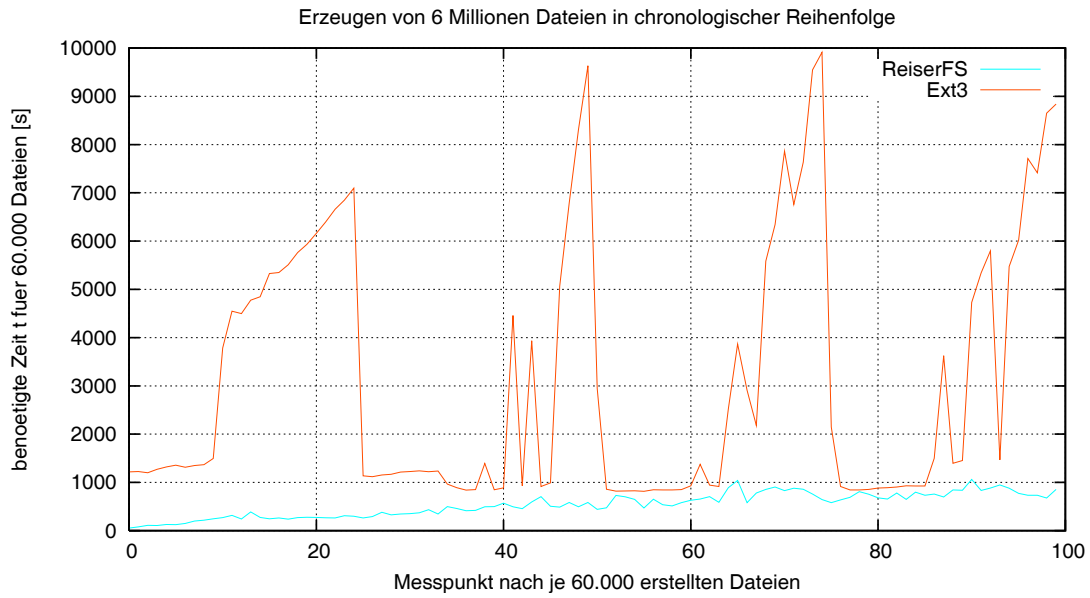

Abbildung 5.3: Verlauf der benötigten Zeit für die Erstellung von je $6 \cdot 10^4$ Dateien in **chronologischer Reihenfolge** bei insgesamt $6 \cdot 10^6$ erzeugten Dateien

Das Verhalten von *Ext3* ist schwerer zu deuten. Zur genaueren Untersuchung wurden die Zugriffszeiten mit einer höheren Auflösung gemessen (Abbildung 5.4). Statt die benötigte Zeit nach 60 000 Dateien zu ermitteln, wurde diese nach bereits 1 000 Dateien festgestellt. Die in Abbildung 5.3 bereits beobachtbaren Peaks in der Zugriffszeit werden hier relativ deutlich aufgelöst. Man erkennt, dass die Peaks periodisch auftreten und selbst stark schwanken, dabei aber linear in der Amplitude anwachsen.

Um dieses Verhalten zu verstehen, wurde noch ein weiterer Test durchgeführt, der im nächsten Abschnitt beschrieben ist.

5.3.4 Messung mit DWARW

Um die Zusammenhänge der extrem unterschiedlichen Zugriffszeiten zwischen logischer und chronologischer Erstellung sowie bei Verwendung unterschiedlicher Dateisysteme zu erkennen, wurden weitere Untersuchungen mit Hilfe des Kernel-Moduls *DWARW* (**D**evice **W**rite **A**nd **R**ead **W**atcher) [48] durchgeführt. Wie der Name bereits vermuten lässt, zählt

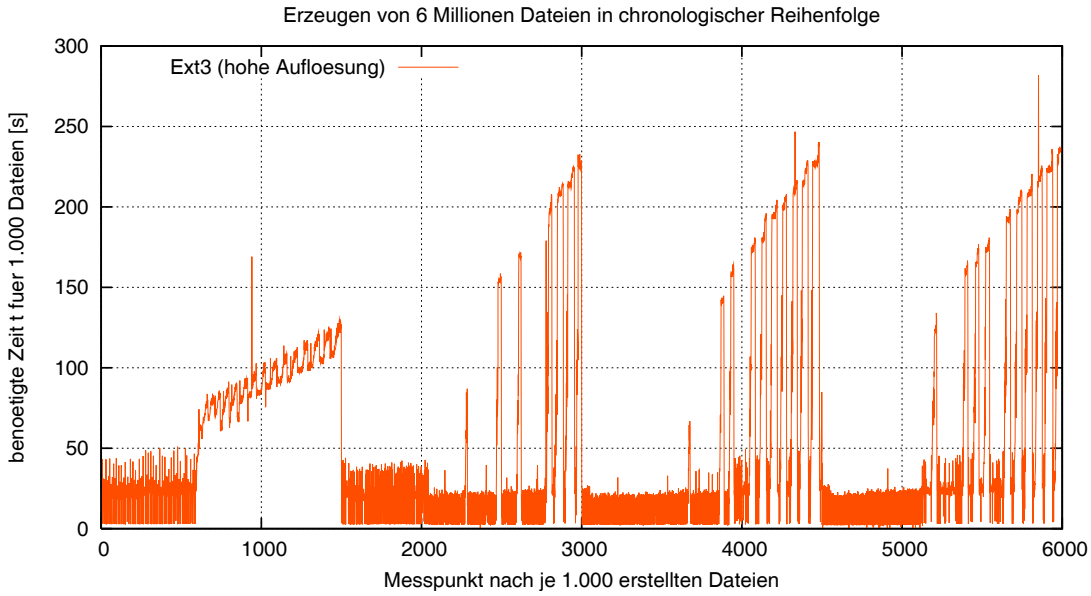

Abbildung 5.4: Verlauf der benötigten Zeit für die Erstellung von je $1\,000$ Dateien in **chronologischer Reihenfolge** bei insgesamt $6 \cdot 10^6$ erzeugten Dateien. Es handelt sich dabei um die gleiche Messung wie in Abbildung 5.3, nur mit einer höheren Auflösung durch Messpunkte nach je 1 000 statt 60 000 Dateien

DWARW alle anfallenden Lese- und Schreibzugriffe eines Devices[5]. Dazu wird das Modul an das zu beobachtende Device gebunden und stellt sich für das Betriebssystem selbst als ein neues Device zur Verfügung. Zugriffe auf dieses virtuelle *DWARW*-Device werden registriert (gezählt) und unverändert auf das eigentliche Ziel-Device umgeleitet. Es wird somit eine Softwareebene zwischen dem ursprünglichen Device und dem Betriebssystem (und damit unterhalb der Caches) eingezogen (Abbildung 5.5). Obwohl die eingefügte Lage sehr „dünn“ ist, könnte dies zu Lasten der Geschwindigkeit gehen, was aber in diesem Fall keine Rolle spielt, da mit diesen Tests nicht die Performance untersucht werden soll. Vielmehr soll gemessen werden, ob die Anzahl an Lese-/Schreibzugriffen auf das Device sich bei den verschiedenen Spielarten (*ReiserFS* / *Ext3* / logisch / chronologisch) unterscheidet.

Naiv müsste man davon ausgehen, dass die Gesamtzahl der Zugriffe (zumindest beim gleichen Dateisystem) gleich ist, da exakt gleich viele Dateien und Verzeichnisse angelegt wer-

[5] engl: Gerät; Als ein Device bezeichnet man unter Unix spezielle Dateien, die Zugriff auf ein physikalisches oder virtuelles E/A-Gerät (z. B. Festplatte, Tastatur, ...) ermöglichen.

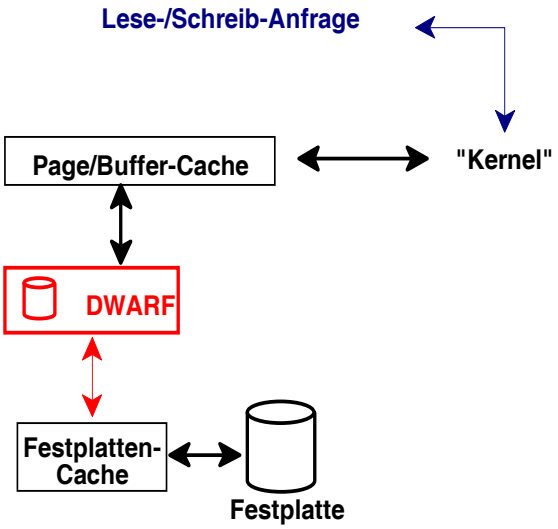

Abbildung 5.5: Schematische Darstellung der eingefügten Softwarelage *DWARW* (rot)

	ReiserFS		*Ext3*	
	logisch	chronologisch	logisch	chronologisch
Schreibzugriffe w	561 304	54 656 775	7 880 364	62 579 099
Lesezugriffe r	773	2 405 690	784 185	52 874 996
$\frac{w}{r}$	726	23	10	1,2
benöt. Zeit t [s]	1 912	55 842	2 385	353 763
$\frac{w+r}{t}$ [Hz]	294	1 022	3 633	326

Tabelle 5.4: Ergebnisse der Messung mit *DWARW* für die Erzeugung von $6 \cdot 10^6$ Dateien in logischer und in chronologischer Reihenfolge unter *ReiserFS* und *Ext3*.

den. Dies trifft allerdings nur zu, wenn Caches unberücksichtigt bleiben. Sollte ein Unterschied in der Anzahl der Zugriffe feststellbar sein, kann man eine fundierte Aussage über die beteiligten Caches treffen, da *DWARW* sich genau zwischen die Caches des Dateisystems und der Festplatte setzt.

5.3.5 Ergebnisse mit DWARW

Tabelle 5.4 und Abbildung 5.6 zeigen das Ergebnis der Messungen. Bei der Erstellung von $6 \cdot 10^6$ Dateien in logischer Reihenfolge wurden unter *ReiserFS* in etwa $5,6 \cdot 10^5$ Schreibzugriffe auf die Platte registriert. Pro Schreibzugriff werden somit ungefähr zehn Dateien à 55 Byte geschrieben, wobei die Metainformationen zu jeder Datei noch nicht berücksichtigt wurden.

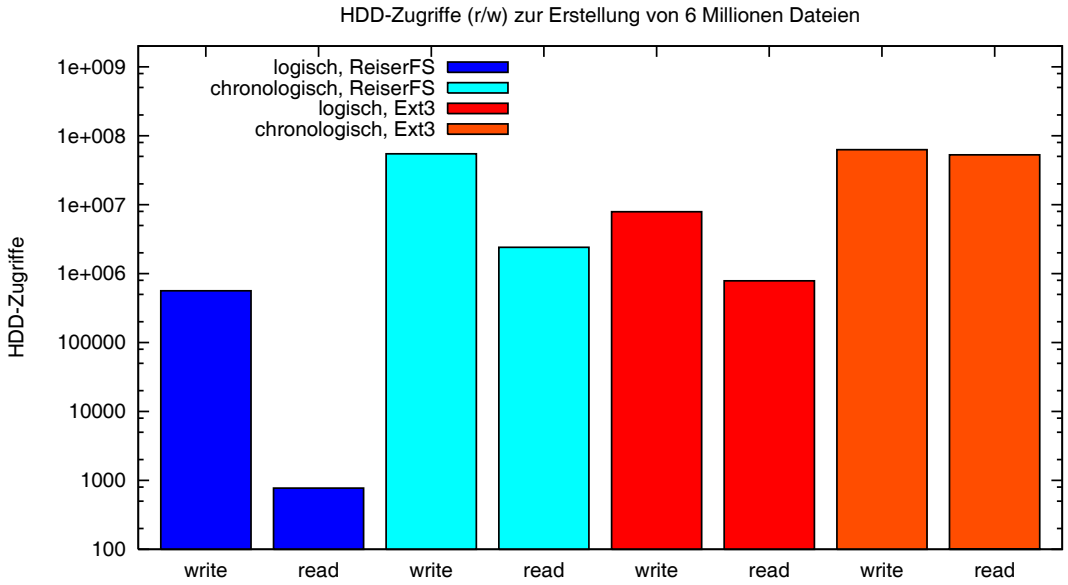

Abbildung 5.6: Logarithmische Darstellung der Anzahl der benötigten Festplattenzugriffe zum Erstellen von $6 \cdot 10^6$ Dateien auf den Dateisystemen *ReiserFS* und *Ext3*, sowohl in logischer als auch in chronologischer Reihenfolge.

Ext3 benötigt gut 14 mal so viele Schreibzugriffe wie *ReiserFS*, so dass nunmehr nicht einmal eine Datei pro Schreibzugriff auf die Platte geschrieben wird. Damit ist gezeigt, dass das Schreiben in logischer Reihenfolge unter *ReiserFS* in erster Linie in die vom Betriebssystem zur Verfügung gestellten Caches stattfindet.

Im Folgenden werden einige Vermutungen anhand der zur Verfügung stehenden Daten angestellt, die im Rahmen dieser Arbeit nicht durch weitere Messungen verifiziert werden konnten. Daher wird ausdrücklich darauf hingewiesen, dass es sich nicht um analytische Schlussfolgerungen handelt, zumal, wie bereits erwähnt, das Zusammenspiel mehrerer Caches zu äußerst komplexen Situationen führen kann.

Interpretation der Messergebnisse Die im Vergleich zu *ReiserFS* hohe Anzahl an Lesezugriffen bei *Ext3* könnte mit den Eigenschaften des Zugriff auf Festplatten zusammenhängen. Diese werden durch den Kernel als block-orientierte Geräte bedient, so dass Änderungen eines Blockfragments nur dadurch realisiert werden können, indem der Block zuerst gelesen, das Fragment eingefügt und der so modifizierte Block wieder geschrieben werden. Finden diese Transaktionen im Buffer-Cache statt, so werden entsprechend weniger Anfragen bis an

den Massenspeicher weitergereicht. Die unter *Ext3* fast gleiche Anzahl an Lese- und Schreibzugriffen im chronologischen Test legt allerdings den Schluss nahe, dass die Anfragen selten aus dem Buffer-Cache beantwortet werden können (das *DWARW*-Modul zählt die Zugriffe auf die Platte bzw. in den Festplatten-Cache). Diese Interpretation wird schließlich auch durch die deutlich schlechtere Laufzeit gestützt.

Ein weiterer Grund für die hohen Zugriffszeiten im Vergleich zur logischen Erstellung ist wahrscheinlich, dass, genau wie unter *ReiserFS*, auch unter *Ext3* die Metainformationen der Verzeichnisse beim Hinzufügen einer neuen Datei aktualisiert werden müssen. Werden nun abwechselnd die Daten und Metainformationen geschrieben, treten im chronologischen Test Cache-Misses auf, die lange Zugriffszeiten der Festplatte zur Folge haben.

Es bleiben noch die in Abbildung 5.4 beobachtbaren starken Schwankungen zu erklären. Die hochfrequenten, kleinen Schwankungen im „flachen“ Bereich sind wahrscheinlich aufgrund der in dieser Zeit geschriebenen Datenmenge von ca. 2 MB auf den Disk-Cache zurückzuführen. Der interessantere lineare Anstieg innerhalb der periodischen Peaks könnte bedeuten, dass hier Zugriffe auf den Buffer-Cache fehlschlagen. Der Aufbau des chronologischen Tests sieht vor, dass die Dateien stundenweise nacheinander erzeugt werden. Die genaue Analyse der Messergebnisse ergibt, dass in den vier großen Peaks die letzten Dateien der jeweiligen simulierten Stunde geschrieben werden. Bis zum Beginn der Peaks scheint der Buffer-Cache nicht vollständig gefüllt gewesen zu sein, so dass er noch Anfragen angenommen hat. Sobald der Cache allerdings voll ist, muss für die nächste Anfrage durch Überschreiben Platz geschafft werden. Einige Zugriffe können nun wieder den Buffer-Cache füllen, bis erneut gelöscht wird. Somit wäre eine Erklärung für die Peaks gefunden. Es bleibt noch der lineare Anstieg zu erklären.

Extrapoliert man den Anstieg zurück, so liegt der Schnittpunkt mit der x-Achse in der Nähe des Beginns einer Stunde. Die Einträge der Filetable werden unter Ext3 in verketteten Listen organisiert. Die Zugriffszeit auf eine solche Liste wächst linear mit der Anzahl der Elemente (in diesem Fall der Dateien). Solange die Daten allerdings im Buffer-Cache zur Verfügung standen, war dieser Effekt nicht zu beobachten. In dem Moment, in dem der Buffer-Cache allerdings das Cache-Trashing beginnt, könnte dieser Effekt zum Vorschein kommen.

Im Fall der logischen Erstellung unter *Ext3* liegt die Anzahl der Lesezugriffe um den Faktor 10 unter der Anzahl der Schreibzugriffe. Hier könnte durchaus der Buffer-Cache die „fehlenden“ Anfragen auflösen. Das mit einer Frequenz von über 3,6 kHz erfolgreich Anfragen an die Festplatte gestellt werden, die diese in sehr kurzer Zeit beantwortet, lässt darauf schließen,

dass entweder der Disk-Cache die Daten aufnimmt bzw. zur Verfügung stellt, oder aber viele Blocks hintereinander gelesen bzw. geschrieben werden.

Abschließend sei noch bemerkt, dass die durch *DWARW* zusätzlich eingezogene Softwarelage kaum Einfluss auf die Performance hatte. Die gemessenen Zugriffszeiten liegen nur geringfügig über den Referenzmessungen ohne *DWARW*.

5.3.6 Vorgehensweise der Lesetests

Mit Hilfe der Lesetests soll ein Eindruck gewonnen werden, wie aufwändig Volltext-Suchoperationen sind. Die Suche wird nur auf einen kleinen Ausschnitt des Datenbestands ausgeführt, da jedes Öffnen einer Datei einen Systemaufruf zur Folge hat, welcher Overhead produziert und somit Zeit kostet. Ziel bei der Verwendung als Datenlager muss sein, bereits durch geeignetes Eingrenzen der Parameter die Anzahl der zu öffnenden Dateien zu minimieren. Je weniger Dateien am Ende geöffnet werden, desto besser ist die Performance. Idealerweise wird nur ein Datensatz benötigt und somit nur eine Datei geöffnet, was zu sehr guten Zugriffszeiten führt.

Das Durchsuchen der Dateien erfolgt daher nach folgendem Prinzip: Im ersten Schritt muss festgelegt werden, auf welche Datenbestände sich die Suche reduzieren soll. Die Suchparameter sollten in einem Plain-File-Datenlager sinnvollerweise mit der Struktur des Verzeichnisbaums korrelieren, in diesem Fall beträgt die mögliche Spanne für die Sensoren 1 bis s, für die Knoten 1 bis k, für die Tage 1 bis d und für die Stunden 0 bis $h-1$.

Konstruieren wir eine mögliche, bereits relativ komplexe Abfrage: *„Welche Fehlermeldungen hat der Sensor 35 auf dem Knoten 3 und 6 innerhalb der ersten 2 Stunden des 1 Tages produziert?“* Geht man nun von der Annahme aus, dass eine Fehlermeldung in den Datensätzen eine eindeutige Zeichenkette zur Identifizierung enthält (z. B. `***FEHLER!***`), so kann man im zweiten Schritt eine Volltextsuche auf den eingegrenzten Datenbestand anwenden. Welchen wichtigen Unterschied die Eingrenzung macht, wird bei folgender Betrachtung sofort ersichtlich: Es müssen in diesem Test nur die folgenden vier von 4 000 Verzeichnissen durchsucht werden:

- *„sensor_35/node_3/day_1/hour_0“*
- *„sensor_35/node_3/day_1/hour_1“*
- *„sensor_35/node_6/day_1/hour_0“*

	ReiserFS		*Ext3*	
	logisch	chronologisch	logisch	chronologisch
C/C++: Zeit t [s]	0,70	2,37	1,47	2,66
grep: Zeit t [s]	1,40	3,63	2,28	2,72

Tabelle 5.5: Ergebnisse der Messungen für das Lesen von $6 \cdot 10^3$ Dateien, welche in logischer und in chronologischer Reihenfolge unter *ReiserFS* und *Ext3* erstellt wurden. Die Tests wurden einmal mit einem *C/C++* -Programm und einmal mit dem Unix-Tool *grep* durchgeführt.

- *„sensor_35/node_6/day_1/hour_1“*

Damit beschränkt sich die Suche auf $6\,000$ anstatt der ursprünglich 6 Millionen Dateien.

5.3.7 Messergebnisse der Lesetests

Die gemessenen Ergebnisse sind in Tabelle 5.5 aufgeführt. Weil nur wenige Dateien geöffnet wurden, liegen alle Messwerte im akzeptablen Bereich für die Verwendung als Datenlager, da Latenzen im Bereich einiger Sekunden durchaus zulässig sind. Der Unterschied zwischen logisch und chronologisch erstellten Daten sowie zwischen den beiden Dateisystemen *ReiserFS* und*Ext3* ist zwar wie erwartet vorhanden, doch liegen nicht wie bei der Erstellung mehrere Größenordnungen zwischen den Ergebnissen.

Die Verwendung der *C/C++* Zugriffsbibliotheken bringt ebenfalls einen geringen Geschwindigkeitsvorteil, allerdings liefern auch die Shell-Tools akzeptable Zugriffszeiten. Unabhängig von den verwendeten Tools und der Reihenfolge der Daten sollten im realen Einsatz jedoch Anfragen vermieden werden, die den Zugriff auf sehr viele Datensätze bzw. Dateien erfordern.

5.3.8 Zusammenfassung

Wie die Messungen belegen, sind bei der Realisierung eines Plain-File-Datenlagers zwei wichtige Faktoren zu berücksichtigen:

- Die Reihenfolge der zu schreibenden Daten
- Die Wahl des Dateisystems

Sollte die Verwendung eines solches System anstehen, so sollten im Vorfeld genaue Test bezüglich der zu erwartenden Reihenfolge durchgeführt werden. Deren Ergebnisse, d. h. die Reihenfolge der Daten sollten zum einen die Verzeichnisstruktur entscheidend prägen, zum anderen empfiehlt es sich, auch über eine Sortierung der Datensätze in Zwischenpuffern nachzudenken, so dass zumindest teilweise die Vorteile der vorhandenen Caches zur Geltung kommen. Allerdings erhöht man dadurch die Latenz zwischen Erfassung des Messwerts und Verfügbarkeit für andere Systeme. Zudem sollte die Verzeichnisstruktur die zu erwartenden Suchparameter ebenfalls widerspiegeln, was unter Umständen zu einem Interessenkonflikt führen kann. In so einem Fall kann vielleicht über eine zweite parallele Verzeichnisstruktur bestehend aus symbolischen Links auf die Verzeichnisse ein Kompromiss gefunden werden.

5.4 Relationale Datenbanken

Die Tests für relationale Datenbanken sind ähnlich aufgebaut wie die Plain-File-Tests. Für jeden Sensor wurde eine eigene Tabelle mit den Spalten Knoten, Messzeit und Messwert angelegt. Anschließend wird, wie bei den Plain-File-Tests auch, eine eingegrenzte Suche nach einer bestimmten Zeichenkette durchgeführt.

5.4.1 MaxDB

Als Vertreter der Kategorie der relationalen Datenbanken wurde *MaxDB* [49] gewählt. *MaxDB* ist die aktuelle Version der als *SAP-DB* bekannten, von der SAP AG zertifizierten, und für den professionellen Einsatz in Produktivsystemen empfohlenen, relationalen und SQL-basierten Open Source Datenbank [49]. *MaxDB* wird seit November 2003 unter der GNU General Public License (GPL) von der Firma MySQL [50] weiterentwickelt, welche durch ihr gleichnamiges Datenbankprodukt für kleinere Systeme bekannt geworden ist.

Im Gegensatz zu den anderen frei verfügbaren Datenbanken wie *MySQL* oder *PostgreSQL* richtet sich *MaxDB* in erster Linie an professionelle Anwender mit hohen bis sehr hohen Anforderungen an Stabilität und Leistung.

5.4.2 Interface zur Datenbank

MaxDB stellt verschiedene Schnittstellen zur Steuerung und Kommunikation der Datenbank zur Verfügung. Neben ODBC[6] unterstützt sie auch JDBC[7], *Perl* [53] und *Python* [54] durch geeignete Programmierschnittstellen. Weiterhin gehört ein *C/C++*-Precompiler zum Lieferumfang, der es erlaubt, *MaxDB*-Instanzen mit Hilfe so genannter „Embedded SQL"-Statements direkt aus *C/C++*-Programmen heraus anzusprechen [41].

ODBC ist dafür bekannt, aufgrund des großen Protokolloverheads recht langsam zu sein [55]. Außerdem existiert kein einheitlicher ODBC-Treiber für unix-artige Systeme wie Linux. Deswegen wurde für die Tests, welche in *C/C++* realisiert wurden, auf Basis des *C/C++*-Precompilers die kleine Bibliothek *libsapdb* entwickelt. Diese kapselt ähnlich zum ODBC-Treiber die Zugriffe auf die Datenbank, ist jedoch bei weitem nicht so umfangreich wie ODBC. Die Bibliothek *libsapdb* wird im Rahmen dieser Arbeit auch für das in Kapitel 6 entwickelte *Storage-Interface* benötigt.

Die Verwendung dieser Bibliothek erlaubt es, die Verbindung zu einer Datenbankinstanz zu organisieren, sowie diese mit SQL-Befehlen zu steuern. Weiterhin werden Funktionen bereitgestellt, die das Einfügen vieler ähnlicher Daten beschleunigen, indem die Datenbank-Instanz zuerst auf die Struktur der kommenden SQL-Kommandos vorbereitet wird (*WritePrepare()*), und danach lediglich immer nur die einzufügenden Daten übermittelt werden (*WriteExecute()*). Messungen zeigen, dass damit im Vergleich zu einzeln abgeschickten SQL-Kommandos Geschwindigkeitsvorteile um den Faktor 2 und mehr auftreten.

Neben den Schreiboperationen bietet die Bibliothek auch Funktionen zum Abfragen und Auslesen der Datenbankinstanz an. Zur Zeit werden allerdings nur die wichtigsten Datentypen wie z. B. Zeichenketten, Ganz- und Gleitkommazahlen unterstützt. Treten in den Ergebnissen andere Datentypen auf, wird eine entsprechende Fehlermeldung ausgegeben.

[6] ODBC ist ein von Microsoft entwickelter Datenbank-Zugriffsstandard, dessen Ziel es ist, den Zugriff auf Daten unabhängig von der verwendeten Datenbank aus jeder Applikation heraus zu ermöglichen. ODBC könnte ein Akronym für **O**pen **D**ata**B**ase **C**onnectivity sein, ist es aber laut Microsoft nicht, da sich Akronyme warenrechtlich nicht schützen lassen [51].

[7] **J**ava **D**ata**B**ase **C**onnectivity: ein von Sun Microsystems [52] entwickelter, auf ODBC basierender Zugriffsstandard für die Programiersprache *Java*.

	Commit nach c Datensätzen				
	c=1	c=1 500	c=6 000	c=60 000	c=6 000 000
benötigte Zeit [s]	4,145	2,020	1,982	1,978	2,199
Einfüllrate [Hz]	1,448	2,970	3,028	3,033	2,729

Tabelle 5.6: Ergebnisse der Messung für die Erzeugung von je $6 \cdot 10^4$ Datensätzen in 100 verschiedenen Tabellen, insgesamt also $6 \cdot 10^6$ Datensätzen, bei unterschiedlich häufigem Commit unter Verwendung der relationalen Datenbank *MaxDB*.

5.4.3 Vorgehensweise der Schreibtests

Wie bereits erwähnt, werden die Parameter aus Tabelle 5.1 auch für diesen Test verwendet. Die Organisation der Daten im relationalen Modell ist bewusst einfach gehalten. Für jeden Sensortyp wird eine eigene Tabelle mit Spalten für Knoten, Datum, Uhrzeit und Messwert erstellt und mit simulierten Werten gefüllt. Im Gegensatz zu einem Plain-File-Datenlager sind relationale Systeme transaktionsbasiert. Dies bedeutet, dass erst nach dem Beenden einer Transaktion („Commit“) die bis dahin neu übermittelten Einträge in der Datenbank als gültig markiert werden. Je kleiner eine Transaktion, desto mehr Overhead entsteht dabei, so dass man die geringste Performance beim Durchführen eines Commits nach jedem Datensatz erwartet. Um einen Eindruck von der Quantität dieses Effekts zu erhalten, wurden in den Tests daher bei sonst gleicher Konfiguration Commits nach unterschiedlich vielen Datensätzen ausgeführt. Um den Einfluss des verfügbaren Arbeitsspeichers zu untersuchen, wurden zudem Referenzmessungen mit unterschiedlich stark reduziertem zur Verfügung stehendem Arbeitsspeicher durchgeführt. Dabei ließen sich allerdings keine erkennbaren Veränderungen feststellen, so dass auf weitere Untersuchungen in dieser Richtung verzichtet wurde.

5.4.4 Messergebnisse der Schreibtests

Tabelle 5.6 zeigt die Ergebnisse der Schreibtests. Die Einfüllraten bewegen sich alle im Bereich von Kilohertz und liegen somit mehr als eine Größenordnung über den Raten, die bei chronologischem Schreiben auf Dateisystemen erreicht wurde. Wie erwartet, ist das Ausführen eines Commits nach jedem Einfügevorgang relativ zeitaufwändig, die Einfüllrate ist im Vergleich zur Strategie, nur alle 1 500 Datensätze ein Commit auszuführen, um ca. 50 % geringer. Hingegen ist kein signifikanter Zuwachs an Einfüge-Geschwindigkeit zu registrieren, wenn man die Anzahl der Commits weiter reduziert. Der Unterschied zwischen 6 000 und 60 000 Datensätzen Abstand beträgt nicht einmal 2 ‰. Man kann in Abbildung 5.7 sogar den gegenteiligen Effekt beobachten – die durchschnittlich benötigte Zeit zum Einfügen von

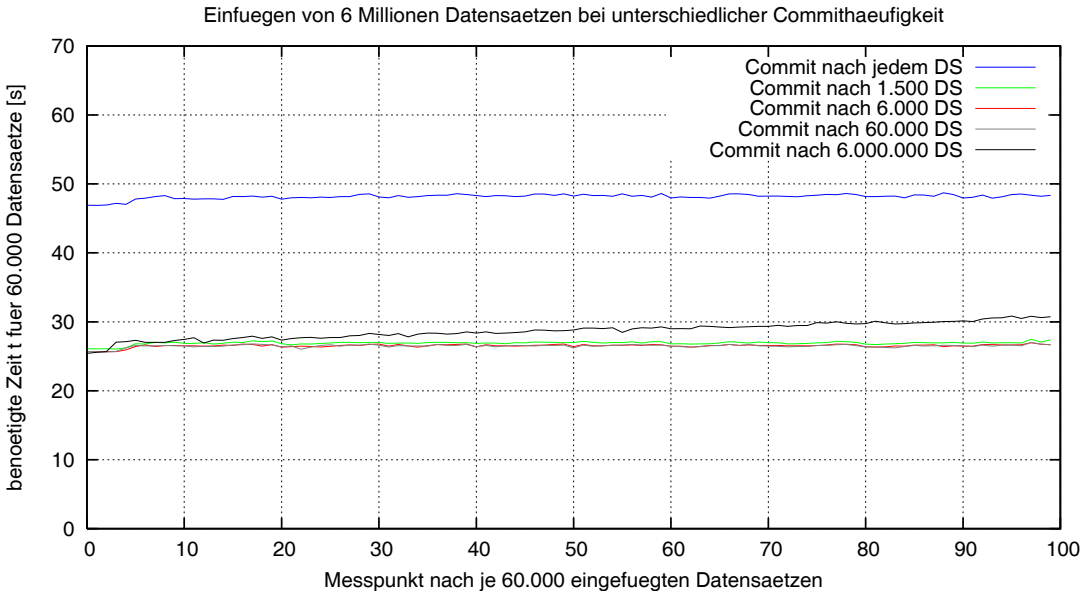

Abbildung 5.7: Verlauf der benötigten Zeit für das Einfügen von je $6 \cdot 10^4$ von insgesamt $6 \cdot 10^6$ erzeugten Datensätzen.

60 000 Dateien steigt sogar leicht an, wenn erst am Ende des gesamten Tests ein Commit ausgeführt wird. Auffallend ist dabei auch, dass die Einfüllraten relativ konstant bleiben und die Schwankungen durchweg sehr niedrig ausfallen.

5.4.5 Lesetests

Beim Lesetest für relationale DBMS wurde die gleiche Abfrage verwendet wie beim Lesetest der Plain-File-Systeme im Abschnitt 5.3.6. Allerdings wird die Abfrage - im Gegensatz zum Plain-File-System – auf den gesamten Datenbestand einer Tabelle (in diesem Fall der Tabelle *„sensor35“*) ausgeführt, da dies die einzige Granularität darstellt, auf die der Benutzer einer relationalen Datenbank Zugriff hat. Damit erweitert sich die Suche auf 60 000 anstatt der im Plain-File-System durchsuchten 6 000 Datensätze.

Im Wesentlichen zeigt das Plain-File-System bei einer großen zu durchsuchenden Anzahl an Datensätzen eine schlechte Performance, da für jede zu öffnende Datei ein Systemaufruf abgesetzt wird. Der eigentliche Suchalgorithmus innerhalb der Datei spielt eine untergeordnete Rolle. Bei einer relationalen Datenbank tritt dieses Problem nicht mehr auf, da

durchsuchte Datensätze	60 000	6 000 000
benötigte Zeit [s]	9,3	389,7

Tabelle 5.7: Benötigte Zeit in Sekunden für das Durchsuchen von $6 \cdot 10^4$ bzw. $6 \cdot 10^6$ Datensätzen

fast keine Dateien geöffnet werden. Somit hängen die Zugriffszeiten innerhalb einer Tabelle in erster Linie von der Anzahl der Datensätze, den vorhandenen Indexstrukturen und den verwendeten Suchalgorithmen ab.

Um diese Überlegungen zu verifizieren wurde die Suche einmal auf die Datensätze des Sensor 35 beschränkt, so wie oben angeführt. Danach wurde die gleiche Suche auf alle Sensorwerte ausgedehnt. Tabelle 5.7 zeigt das Ergebnis der Suchtest-Messungen. Darin ist bereits deutlich zu erkennen, dass der Suchaufwand nicht linear mit der Anzahl der Datensätze steigt. Dennoch sieht man einen deutlichen Unterschied, d. h. dass sich die Strukturierung in einzelne Tabellen für unterschiedliche Sensoren lohnt.

5.5 XML-Datenbanken

In Kapitel 4.3.4 wurde bereits erwähnt, dass sich XML-Datenbanken zur Zeit noch in einer frühen Phase der Entwicklung befinden. Zwar existieren bereits einige (kommerzielle) Systeme, bis auf *Tamino* [56], eine Entwicklung der Software AG, hat sich bis jetzt aber keines auf dem Markt behaupten können. *Tamino* hingegen ist zwar eine mächtige XML-Datenbank, deren Entwicklungsschwerpunkt allerdings auf – wie der Hersteller es nennt - „Electronic Business"-Systemen liegt und daher eher auf Abfrage- denn auf Schreiboperationen hin optimiert wurde. Deswegen wurde mit *Natix* eine andere native XML-Datenbanken verwendet.

5.5.1 Natix

Natix ist 1998 im Rahmen eines Projektes am Lehrstuhl für Praktische Informatik III der Universität Mannheim [57] in *C++* auf UNIX-Plattformen entstanden. Ziel war die Entwicklung eines DBMS zur nativen Speicherung von XML-Dokumenten. Seit dem Jahr 2000 wird *Natix* von der Firma data ex machina [58] weiterentwickelt und steht zur Zeit vor der Freigabe einer völlig überarbeiteten zweiten Version.

Natix stellt ein DBMS zur Ablage der Struktur und des Inhalts von XML-Dokumenten dar. Es bietet DOM- und SAX-Zugriff auf abgelegte Dokumente sowie Unterstützung für einige standardisierte Abfragesprachen (u. a. XQuery). Als besondere Eigenschaft wird auch ein Dateisystemtreiber zur Datenablage angeboten. Über diese Schnittstelle kann somit auf die Datenbank wie auf eine eingebundene Partition eines Massenspeichers zugegriffen werden. Dennoch verzichtet man dabei nicht auf die inhärenten Vorteile wie Indizierung des Inhalts, durch welche Suchanfragen beschleunigt werden.

Natix geht den bei relationalen DBMS bewährten Weg der Gliederung in drei voneinander unabhängige Ebenen. Die Anbindungsebene ermöglicht Anwendungen die Kommunikation mit der Datenbank. Die oben bereits genannten Schnittstellen setzten hier an. Die Datenbankdienstebene realisiert die Dienste innerhalb der Datenbank. Es werden alle wichtigen von relationalen DBMS bekannten Dienste (Transaktionsmanagment, Konsistenzprüfung, Integritätskontrolle, Mehrbenutzerbetrieb, ...) angeboten. Die Speicherungsebene als unterste Schicht bildet die logischen Daten auf einen physischen Datenträger ab und übernimmt die Verwaltung der persistenten Daten. Dazu werden Standard-Indexstrukturen wie B-Trees und invertierte Listen verwendet.

Das von den Entwicklern zur Verfügung gestellte Prerelease konnte im Rahmen dieser Arbeit sowie aufgrund des Entwicklungsstadiums der Software nur sehr oberflächlich untersucht werden. Lesezugriffe konnten deshalb nicht getestet werden. Es ist allerdings zu erwarten, dass bei geeigneter Strukturierung auch unter *Natix* akzeptable Zugriffszeiten entstehen, da *Natix* nicht wie ein Plain-File-System auf viele Dateien zugreifen muss, sondern wie relationale Systeme auch die Daten selbst verwaltet.

Schreib-/Lesezugriffe wurden unter *Natix* besonders optimiert. Dazu wird der XML-Baum in Teilbäume zerlegt und in Records abgespeichert, welche über eine Record-ID identifiziert werden. Die Records wiederum sind in Header und Daten der Knoten des Teilbaums unterteilt. Im Header wird u. a. die Zugehörigkeit zu einem bestimmten Dokument sowie zur einfachen Navigation eine Referenz auf den Vater-Teilbaum gespeichert. Die Records selbst werden in so genannten Seiten organisiert, deren Zusammensetzung am Ende die Anzahl der Zugriffe auf den Datenträger und somit den tatsächlich erreichbaren Datendurchsatz bestimmt. Eben dieser Zusammenhang wurde bei der Entwicklung von *Natix* besonders bedacht.

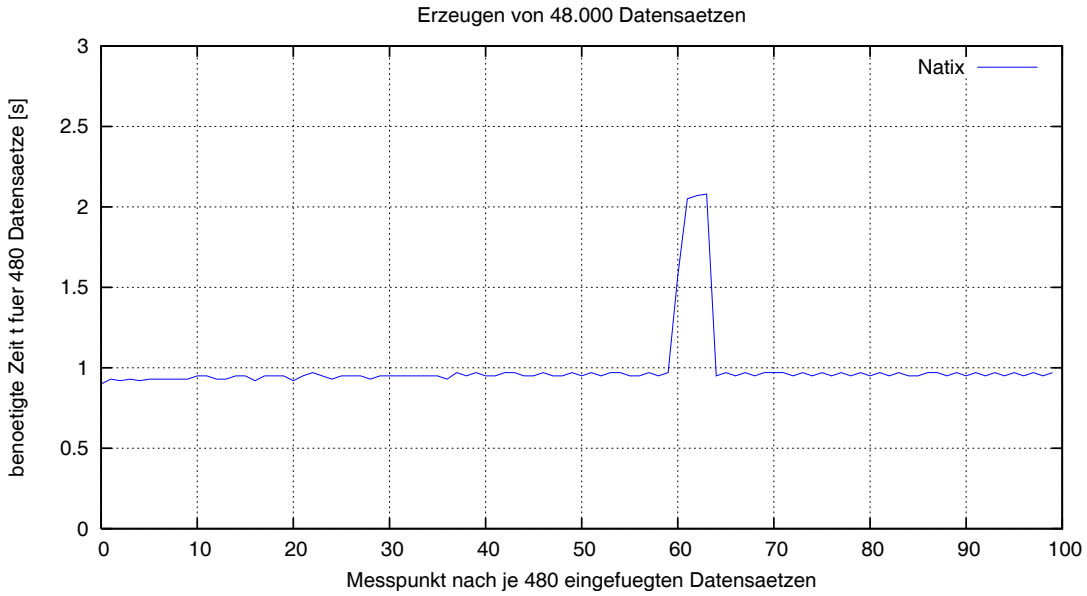

Abbildung 5.8: Verlauf der benötigten Zeit für das Einfügen von je 480 von insgesamt 48 000 erzeugten Datensätzen.

5.5.2 Schreibtests

Da die Datenbank sich noch im Entwicklungsstadium befindet, mussten die Tests angepasst werden. Die vorliegende Version von *Natix* bricht nach etwas über 50 000 eingefügten Datensätzen ab[8]. Um den Fehler zu umgehen, wird der Test daher auf 48 000 Datensätze mit je 300 Byte beschränkt. Entscheidend für die Performance ist letztendlich aber die tatsächliche Schreibfrequenz auf die Festplatte, welche mit dem Test nicht sicher erfasst werden konnte. Die Tatsache, dass die Zugriffszeiten bei den Messungen einmal kurz ansteigen (Abbildung 5.8) lässt allerdings darauf schließen, dass die Daten nicht nur in den Hauptspeicher sondern tatsächlich auf Platte geschrieben werden. Diese Vermutung wird dadurch gestützt, dass die gleichen Tests, mit stark reduziertem Hauptspeicher durchgeführt[9], fast identische Ergebnisse liefern.

[8] Der Fehler wurde den Entwicklern gemeldet.

[9] Der der Datenbank zur Verfügung stehende Hauptspeicher wurde durch andere im Hintergrund laufende Anwendungen so blockiert, dass die entsprechenden Speicherbereiche nicht ausgelagert werden konnten (*mlock()*).

XML-Datenbanken wie *Natix* stehen zur Zeit noch am Anfang ihrer Entwicklung, so dass in den nächsten Jahren erhebliche Fortschritte zu erwarten sind. Trotz der gelegentlich auftauchenden Probleme vermittelt *Natix* einen sehr vielversprechenden Eindruck und einen Ausblick darauf, welche Vorteile aus der Verwendung von XML-Datenbanken resultieren können.

5.6 Zusammenfassung

In diesem Kapitel wurde je ein Vertreter der für umfassendes Monitoring in Frage kommenden Datenlager getestet. Die Plain-File-Systeme zeigten dabei, je nach Schreibreihenfolge und verwendetem Dateisystem, die größte Streuung. Sie erwiesen sich nur unter einigen, sehr strengen Rahmenvoraussetzungen, die in der Regel nicht erfüllt werden können, als geeignet für den Einsatz als Datenlager für verteilte Systeme.

Das relationale System zeigte erwartungsgemäß die beste Schreib-Performance, ist aber durch die inhärenten Eigenschaften relationaler Datenbanken beim Umgang schemaloser Messwerte den XML-Datenbanken immer unterlegen. Diese allerdings befinden sich noch am Anfang ihrer Entwicklung und lassen für die nahe Zukunft auf deutliche Fortschritte hoffen.

Eine pauschale Aussage, welches Datenlager nun für Monitoringsysteme am besten geeignet erscheint, kann somit nicht ohne Einschränkungen getroffen werden. Im nächsten Kapitel wird daher das *Storage-Interface* vorgestellt, welches die Nutzung verschiedener Datenlager für Monitoringsysteme ermöglicht.

6 Das Storage-Interface

6.1 Einleitung

In Kapitel 4 wurden vier verschiedene Datenlager-Plattformen vorgestellt. Anhand je einer Beispiel-Implementation wurden in Kapitel 5 grundlegende Eigenschaften von drei der Plattformen untersucht. Dabei wurden auch die jedem vorgestellten Lagerungssystem inhärenten Vor- und Nachteile diskutiert.

Konventionelle Monitoringsysteme ermöglichen nur eine bestimmte Form der Datenhaltung, meist in Round-Robin- oder relationalen Datenbanken. Dabei wird allerdings übersehen, dass nicht alle Messwerte gleiche Anforderungen an ein Datenlager stellen. Umfassendes Monitoring setzt im Gegenteil mit einer Vielzahl verschiedener und verschiedenartiger Sensoren je nach Sensor unterschiedliche Eigenschaften des Datenlagers voraus. Einige Werte müssen sehr schnell aufzufinden sein, werden allerdings nur über einen bestimmten Zeitraum benötigt, z. B. die aktuelle CPU-Last. Andere Werte müssen hingegen über einen langen Zeitraum hinweg zur Verfügung stehen, so z. B. die genutzte und damit zu bezahlende Rechenzeit eines Benutzers. Einige Werte werden gezielt immer nur einzeln abgefragt, andere sind meist Ergebnisse einer Suche auf einer großen Datenmenge nach bestimmten Suchkriterien. Keines der oben benannten Datenlager kann alle Ansprüche gleichzeitig zufriedenstellend erfüllen.

6.2 Konzeption eines generischen Storage-Interface

Um diesen unterschiedlichen, zum Teil sogar widersprüchlichen Anforderungen gerecht zu werden, verteilt man die verschiedenen Sensormesswerte auf unterschiedliche Datenlager. Dies bedeutet, dass das Monitoringsystem nicht nur auf eine Datenbank zugreift, sondern auf viele verschiedene. Im Prinzip handelt es sich um ein verteiltes Datenlager, das nicht wie „üblich“ Datenbanken vom selben Typ umfasst, sondern auch unterschiedliche Plattformen benutzen kann. Messwerte des Sensors A werden somit in „seine“ Datenbank – zum Beispiel

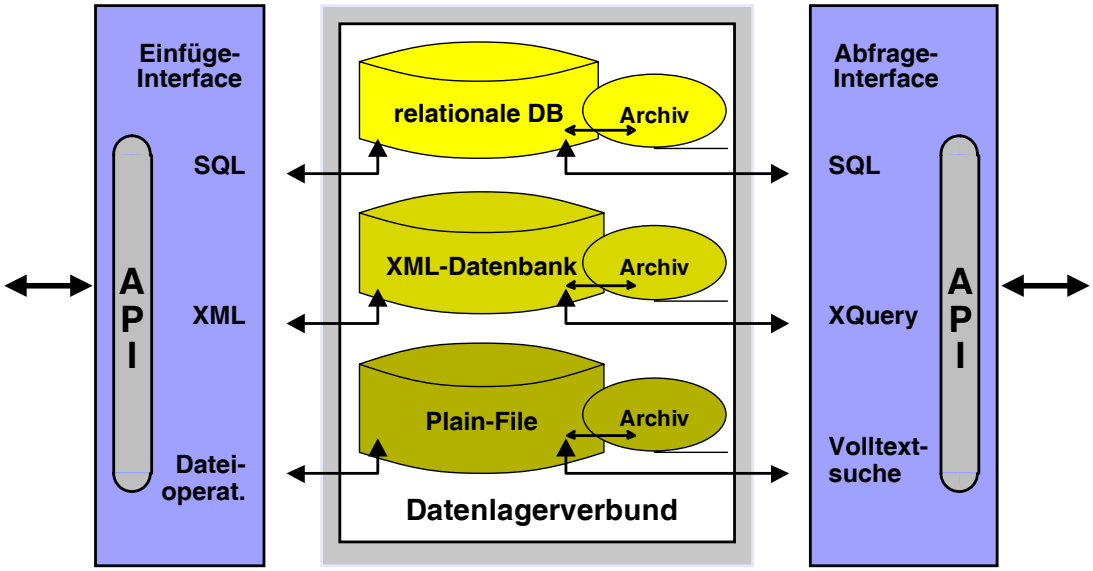

Abbildung 6.1: Prinzipieller Aufbau des *Storage-Interface*, welches eine generische Schnittstelle für unterschiedliche Datenlager zur Verfügung stellt.

eine relationale Datenbank – eingetragen, während Daten des Sensors B besser in ein Plain-File System abgelegt werden sollen. Messwerte eines neuen, unbekannten Sensors würde man hingegen wahrscheinlich in einer XML-Datenbank ablegen, da nicht bekannt ist, nach welchen Kriterien auf die Daten zugriffen werden wird (Abbildung 6.1).

Die grundlegende Idee besteht demnach darin, für jeden Sensor eine eigene „Zieldatenbank“ zu festzulegen. Allerdings entstünde für das Daten schreibende Monitoringsystem dadurch ein ziemlich großer Overhead an Verwaltung. Es müsste kontrollieren, welche Sensorinformationen wohin geschrieben werden und wie genau das jeweilige Datenlager angesprochen werden muss (auf welchem Rechner, welcher Benutzer, welche Datenbankinstanz, . . .). Im Rahmen modularer und objektorientierter Programmierung ist es daher nur konsequent, diese Fähigkeiten vollständig zu kapseln, so dass das Monitoring nur noch die Daten liefert, und eine andere Software – das *Storage-Interface* – die richtige Zuordnung der Daten und der Speicherung vornimmt. Weiterhin sollten andere Anwendungen und Dienste (z. B. Fehlertoleranz, Abfrage GUI, . . .) ohne genaue Kenntnis der exakten Lagerungsorte auf die vom Monitoringsystem gesammelten Daten zugreifen können. Das *Storage-Interface* muss also über zwei Teile verfügen: einen zur Datenannahme und einen, um Abfragen unabhängig vom darunter liegenden Datenlager zu formulieren.

6.2.1 Wahl der Programmiersprache

Bei der Entwicklung von Software muss entschieden werden, welche Programmiersprache verwendet werden soll. Das *Storage-Interface* wurde in *C++* entwickelt. Grundlage dieser Entscheidung ist eine Analyse der gestellten Anforderungen an die Programmiersprache. Dazu gehören:

- Plattformübergreifend verwendbar durch weite Verbreitung der benötigten Compiler (Grids sind typischerweise inhomogene verteile Systeme)
- Schnelligkeit
- gute Lesbarkeit des Quellcodes (Wartbarkeit)
- hoher Verbreitungsgrad (Gründe: Anbindung an vorhandene Schnittstellen wie HCMS, Datenbanken, Wart- und Erweiterbarkeit durch andere Entwickler)
- objektorientierte Struktur (Klassenkonzept, Implementationsdetails können hinter einfacher Schnittstelle versteckt werden)
- Verfügbarkeit vieler Standarddatentypen und geeigneter Methoden, die ihre Bearbeitung ermöglichen, und dennoch leicht erweitert bzw. abgeändert werden können
- Wiederverwertbarkeit

Mit der Anforderungen nach Schnelligkeit scheidet die Verwendung von Skript-Sprachen[1] wie z. B. Perl aus. Auch Java [59] erfüllt nicht alle Anforderungen, obwohl es viele Vorteile bietet. Der Java-Bytecode ist plattformunabhängig und muss nur einmal und nicht für jede Plattform einzeln kompiliert werden. Zudem ist Java eine weit verbreitete, mächtige objektorientierte Programmiersprache, mit der sich auch größere Projekte entwickeln lassen. Allerdings muss der Java-Bytecode von einer (auf jedem System zu installierenden) virtuellen Java-Maschine in die Maschinensprache des Prozessors übersetzt und interpretiert werden. Dies macht Java deutlich langsamer als z. B. *C++*.

C++ basiert auf *C* und ist die zur Zeit dominierende objektorientierte Programmiersprache am Markt. Dadurch, dass *C/C++* Compiler für sehr viele verschiedene Plattformen existieren, können *C/C++* Programme gut auf verschiedene Hard- und Softwareumgebungen portiert werden, sofern auf die Verwendung Plattform spezifischer APIs verzichtet wird.

[1] Das ausführbare Programm wird mit Hilfe eines Interpreters zur Laufzeit erzeugt.

Nicht zuletzt ist von Bedeutung, dass *C++* durch seine STL-Basisbibliotheken ein sehr mächtiges Werkzeug anbietet. Die *Standard Template Library* (STL) ist eine umfangreiche Klassenbibliothek, die effiziente und gut gepflegte Datenstrukturen und Algorithmen zur Verfügung stellt. Darüber hinaus bildet sie eine solide Grundlage, die der Programmierer um eigene Komponenten erweitern kann. Diese Flexibilität basiert auf einem als „Generisches Programmieren“ bekannten Konzept. Insgesamt zeichnet sich die STL durch eine konsistente und daher gut verständliche Architektur aus [60].

6.2.2 Erwarteter Grundaufbau eines Sensorergebnisses für das Storage-Interface

Das *Storage-Interface* verarbeitet Sensormesswerte im XML-Format. Gründe wie Plattformunabhängigkeit, Wiederverwendbarkeit, leichte Lesbarkeit, Erweiterungsfähigkeit und somit Flexibilität für die Verwendung von XML wurden in den vorangegangenen Kapitel hinreichend dargelegt. Nicht zuletzt verwenden einige Monitoringsysteme wie z. B. auch das HCMS (Kapitel 3.3.4) XML als Datenübermittlungsformat.

Die Struktur eines Sensorergebnisses sollte so einfach wie möglich gehalten werden. Um dieser Vorgabe gerecht zu werden, und um die XML-Dateien auf die verschiedenen Datenlager im Rahmen ihrer Möglichkeiten mappen zu können, sollten alle Sensorergebnisse folgenden grundlegenden Aufbau besitzen:

- Jeder XML-Knoten mit der Bezeichnung `<sensor>` umfasst einen Datensatz und wird durch sein obligatorisches `name`-Attribut genau einem Sensor zugeordnet. Die eigentlichen Daten des Datensatzes sind untergeordnete XML-Knoten. Sollten die Werte eines Sensors in einem relationalen DBMS gespeichert werden, so sind diese in einer Tabelle abgelegt, die den Namen des Sensors trägt. In einem Plain-File-System dient die verwendete Bezeichnung als Name für angelegte Unterverzeichnisse. Somit sind stets nur solche Sensorbezeichnungen zu wählen, die auch zulässige Namen einer Tabelle und der verwendeten Dateisysteme darstellen.
- Jeder Datensatz besitzt immer die XML-Knoten `<node_id>`, `<date>` und `<time>`, so dass ein Plain-File-System den Datensatz auf eine Mindest-Verzeichnisstruktur abbilden kann.
- Informationen, die in einer relationalen Datenbank sinnvoll in separaten Spalten einer Tabelle abzulegen sind (in der Regel also nicht weiter strukturierte Messwerte) sollten

```
<sensor name="cpu-load">
  <node_id>mdax</node_id>
  <date>19.03.2004</date>
  <time>16:12:53</time>
  <load_1min>0.75</load_1min>
  <load_5min>0.98</load_5min>
  <load_15min>0.92</load_15min>
  <additional>
  </additional>
</sensor>
```

(a) normaler Datensatz

```
<sensor name="cpu-load">
  <node_id>mdax</node_id>
  <date>15.04.2004</date>
  <time>12:21:22</time>
  <load_1min>3.81</load_1min>
  <load_5min>3.32</load_5min>
  <load_15min>3.26</load_15min>
  <additional>
    <suspect-program>
      <command>mem_leaker</command>
      <pid>8543</pid>
      <owner>root</owner>
      <cpu-time>511:12.81</cpu-time>
      <mem-usage>95</mem-usage>
    </suspect-program>
  </additional>
</sensor>
```

(b) Datensatz mit <additional>-Substruktur

Abbildung 6.2: Beispiel eines Datensatzes einmal ohne und einmal mit <additional>-Substruktur für den gleichen Sensor. Die Substruktur wurde vom Sensor aufgrund eines verdächtigen Verhaltens eines Programms hinzugefügt.

unabhängig von der tatsächlichen Zielplattform in getrennten XML-Knoten gespeichert werden. Dabei muss folgende Konvention eingehalten werden: Die Bezeichnung eines XML-Knotens entspricht in der relationalen Datenbank dem Namen der Spalte, so dass auch hier nur geeignete Bezeichnungen zu wählen sind.

- Darüber hinaus gehende Informationen (z. B. Dokumente) werden in einer XML-Substruktur des <additional>-Knotens abgelegt. Diese Substruktur wird in einem relationalen System entsprechend als Zeichenkette in der Spalte additional gespeichert.

Abbildung 6.2 zeigt ein Beispiel eines Datensatzes einmal ohne und einmal mit <additional>-Substruktur.

6.2.3 Konfiguration des Storage-Interface

Die Konfiguration des *Storage-Interface* sollte flexibel, erweiterbar, aber vor allem einfach erfolgen. Für jeden bekannten Sensor muss durch den Administrator, je nach dem, welche Datenlager-Plattform sinnvoll erscheint, ein existierendes Datenlager als Ziel definiert werden. Mit einem existierenden Datenlager ist dabei ein Softwaresystem gemeint, dem über

eine definierte Schnittstelle Daten zur persistenten Speicherung übergeben werden können und das Abfragen auf den von ihm verwalteten Datenbestand durchführen kann. Für unbekannte Sensoren ist eine konfigurierbare Standardeinstellung vorzusehen.

Anhand der Konfiguration bestimmt das *Storage-Interface* später, mit welchem Datenlager es bei Operationen (Schreiben / Lesen) eines bestimmten Sensors kommunizieren muss. Dazu benötigt es, neben der Kenntnis der Schnittstelle des Datenlagers, in der Regel noch weitere Daten, wie zum Beispiel, auf welchem Knoten das Lager eingerichtet ist, sowie Datenlager spezifische Informationen (Benutzername, Passwort, Datenbankinstanz, ...).

6.2.4 Datenannahme

Im Prinzip müssen die Messwerte der Sensoren möglichst in ihrer vom Sensor erzeugten Form (XML-Datensatz) entgegengenommen werden, allerdings sollte die Datenannahme idealerweise auch mehrere Sensorwerte bzw. auch kumulierte Sensorwertdateien einlesen können. Solche kumulierten Sensorwertdateien entstehen durch eine Konsolidierung von einzelnen Sensorwertdateien auf dem Weg zwischen Sensor und *Storage-Interface*. Konsolidierte Datensätze können bei der Übertragung besser komprimiert werden und gewährleisten zudem, dass nur korrekte (bzw. wohlgeformte) Datensätze an das Interface weitergereicht werden.

Die Datensätze müssen dabei nicht von unterschiedlichen Sensoren stammen, sondern können, je nach Art der Konsolidierung, auch mehrere Messwerte eines Sensors zu unterschiedlichen Messzeitpunkten enthalten. Auf der Seite des *Storage-Interface* kann es durchaus sinnvoll sein, mehrere Sensorwerte eines Sensors zu sammeln und erst nach einer geeignet erscheinenden Anzahl an Werten den Schreibprozess auf das zugeordnete Datenlager durchzuführen, da dadurch Vorteile einzelner Datenlager beim Schreiben mehrerer Datensätze ausgenutzt werden können und somit der effektive Durchsatz erhöhen werden kann (vgl. Messungen in Kapitel 5).

6.2.5 Datenabfrage

Die Datenabfrage sollte idealerweise ohne Kenntnis der verwendeten Datenlager erfolgen können. Die Schwierigkeit hierbei besteht nun darin, dass die verschiedenen Datenlager jeweils völlig unterschiedliche Anfragemechanismen (Volltextsuche, SQL, XQuery) besitzen.

SQL und XQuery sind sehr mächtige, bereits nicht eindeutig aufeinander abbildbar Abfragesprachen, so dass ein einheitliches Interface zwangsläufig nur einen kleinen Ausschnitt ihrer Möglichkeiten anbieten kann.

Aber diese systemimmanente Problematik liefert auch einen Ansatz zur Lösung. Das Interface sollte auf einfache aber wichtige Anfragen beschränkt bleiben. Es soll keine wie in Kapitel 4.3.4.1 beschriebene allgemeine Anfragesprache darstellen, sondern eine Suche auf eine entscheidend eingeschränkte Datenmenge ausführen, auf deren Ergebnis der Anfragende dann im Bedarfsfall selbst weitere (Abfrage-) Operationen ausführen kann.

6.2.6 Wahl eines geeigneten XML-Parsers

Das *Storage-Interface* erhält die in ein Datenlager zu schreibenden Sensormesswerte in Form einer XML-Datei, welche nicht auf einem Speichermedium abgelegt sein muss, sondern auch als Zeichenkette im Speicher zur Verfügung stehen kann. Bei der Wahl eines geeigneten *C++* XML-Parsers für das *Storage-Interface* muss berücksichtigt werden, dass eine solche XML-Datei durch Konsolidierung bereits viele verschiedene Sensormesswerte enthalten kann. Da dies ein gewünschter Effekt ist, der wie bereits erwähnt u. U. die effektive Datendurchsatz-Geschwindigkeit zu erhöhen vermag, sollte der Parser die maximale Größe der Messwertdatei keinen unnötigen Einschränkungen unterwerfen. Daher erscheint die Wahl eines SAX-Parsers sinnvoll, da dieser im Gegensatz zu einem DOM-Parser nur einen Ausschnitt des Dokuments im Speicher halten muss. Weiterhin ist ein wahlfreier Zugriff auf die Datei nicht nötig: das Interface muss eine Datei stückweise einlesen und einzelne Datensätze sammeln. Diese werden erst später in einem eigenen Schritt in einem Datenlager abgelegt.

6.3 Implementation

Im vorhergehenden Abschnitt wurden grundlegende Überlegungen und Entscheidungen bezüglich Leistung und Umfang des *Storage-Interface* dargelegt. In diesem Abschnitt werden einige Details der Implementation vorgestellt.

6.3.1 Wahl eines geeigneten SAX-XML-Parsers für das Storage-Interface

Das *Storage-Interface* verwendet den XML-C-Parser *libxml2*, welcher für das GNOME[2] Projekt entwickelt wurde. *libxml2* steht als freie Software unter der MIT-Lizenz [36] zur Verfügung. Weiterhin ist *libxml2* sehr portabel und kompiliert nach Angabe der Entwickler mit geringfügigen Änderungen unter vielen Systemen (Linux, Unix, Windows, CygWin, MacOS, MacOS X, RISC Os, OS/2, VMS, QNX, MVS, . . .). Für eine Software, welche in einem Grid eingesetzt werden soll, ist eine gute Portabilität von Vorteil. Dieser Parser wurde zudem gewählt, da er sich durch eine sehr hohe Geschwindigkeit auszeichnet (vgl. Abbildung 6.3) und eine einfach benutzbare SAX-Schnittstelle aufweist. Weiterhin wird *libxml2* regelmäßig von der großen Entwicklergemeinschaft um das GNOME-Projekt gewartet und optimiert, so dass auch eine zukünftige Unterstützung gewährleistet sein sollte. Zu den weiteren Vorteilen zählt die integrierte Unterstützung ZLIB / Compress komprimierter Dateien, wie sie vom *HCMS* zur besseren Ausnutzung der Bandbreite bereitgestellt werden (können). ZLIB ist die dem unter Unix bekannten Kompressionsprogramm *gzip* zugrunde liegende Bibliothek, deren Algorithmus auf dem Lempel-Ziv-Verfahren (LZ77) basiert [31].

6.3.2 Das Klassenkonzept

Das *Storage-Interface* besteht aus mehreren Klassen, die jeweils unterschiedliche Aufgaben wahrnehmen. Bei der Entwicklung wurde besonderer Wert darauf gelegt, dass bei Beibehaltung der Schnittstelle die einzelnen Klassen leicht erweitert oder durch völlig neue ersetzt werden können.

Die Klasse *cStorageBase* ist die Basisklasse des *Storage-Interface* und stellt grundlegende Methoden wie zum Beispiel für das Einlesen der Konfiguration zur Verfügung. Die beiden eigentlichen Schnittstellen für das Schreiben (*cWrite2storage*) und Lesen (*cQueryStorage*) sind von dieser Klasse abgeleitet. Daneben existieren noch weitere Klassen, die spezielle Fähigkeiten, Eigenschaften und Speicherstrukturen (z. B. Darstellung eines Datensatzes, Zwischenspeichern mehrerer Datensätze, Ergebnisse einer Abfrage) zur Verfügung stellen.

Zur genaueren Beschreibung der Funktionalität der einzelnen Klassen sowie ihrer Methoden sollte die Dokumentation zu Rate gezogen werden. Jede Klasse, jede nichttriviale Funktion und jede (globale) Variable und Konstante wurden ausreichend dokumentiert. Weiterhin existieren Kommentare für nicht-offensichtliche oder nichtportable Stellen. Mit Hilfe von

[2] GNOME ist eine Open-Source Desktop-Suite für Unix und Linux [24].

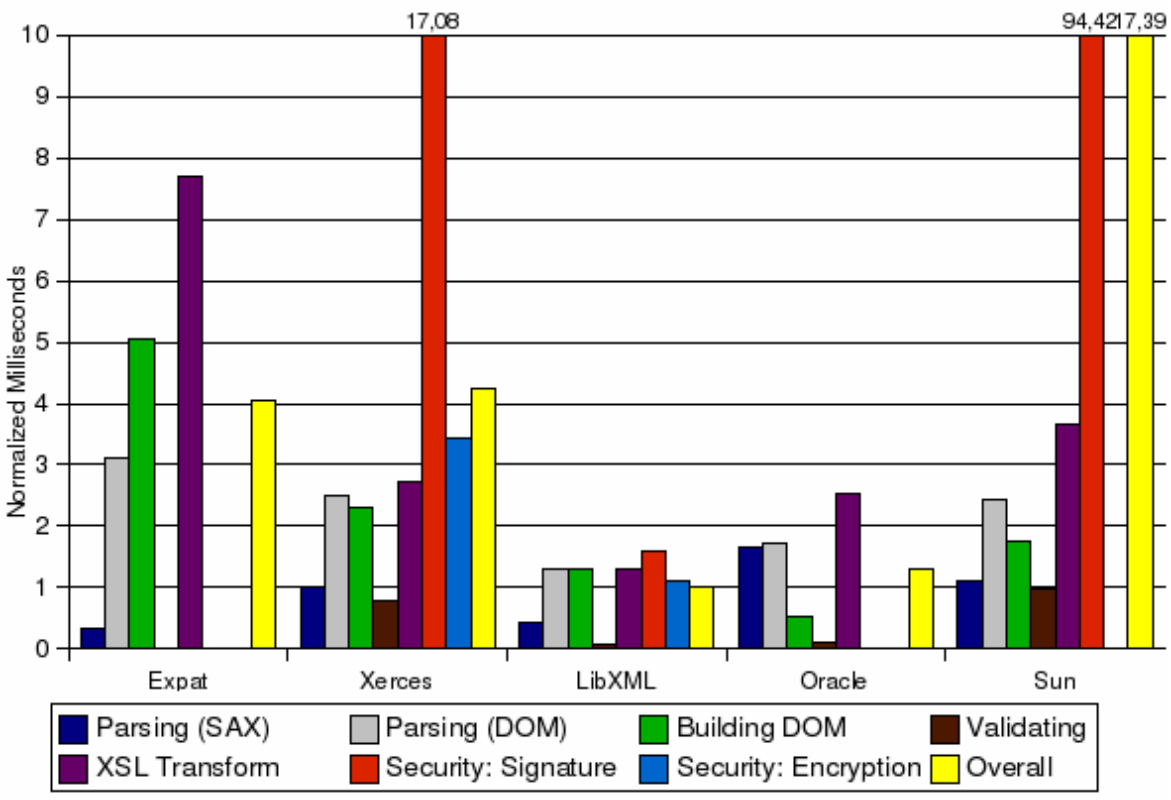

Abbildung 6.3: Testergebnis von *XML Benchmark* [61]. Dieses Open-Source-Projekt stellt einen Satz Werkzeuge zum Testen verschiedener *C/C++* XML-Parser zur Verfügung. Die Abbildung stammt aus dem unter [61] verfügbaren aktuellen Test von Feb. 2004

Doxygen[3] kann eine automatisierte Programm-Dokumentation in einer großen Anzahl von Formaten (HTML, LaTeX, PDF, RTF, PostScript, man page-Formate, ...) erstellt werden.

6.3.3 Die Konfiguration des Storage-Interface

Die Konfiguration des *Storage-Interface* wird durch XML-Dateien vorgenommen. Für diese Entscheidung können zwei wesentliche Vorteile angeführt werden:

- XML erfüllt die Anforderungen nach Flexibilität und Erweiterbarkeit

[3] Doxygen ist ein Open-Source Kommandozeilen-Dokumentationsgenerator [62].

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<config>
  <sensor name="sensor_1">
    <dbType>21</dbType>
    <dbName>DL1</dbName>
    <serverName>e101</serverName>
    <userName>user1</userName>
    <userPassword>passwd1</userPassword>
  </sensor>
  <sensor name="sensor_2">
    <dbType>21</dbType>
    <dbName>DL2</dbName>
    <serverName>e102</serverName>
              .
              .
              .
```

Abbildung 6.4: Ausschnitt aus einer Konfigurationsdatei für das *Storage-Interface*

- Für das *Storage-Interface* wird ohnehin ein XML-Parser benötigt, um Sensorwerte extrahieren zu können.

Der Aufbau der Konfigurationsdatei ist einfach gewählt. Für jeden Sensor, der anhand seines Namens identifiziert wird, werden je nach Typ des Datenlagers bestimmte Informationen erwartet. Jedem Datenlager ist eine natürliche Zahl zugeordnet[4], die Spezifizierung des Datenlagers erfolgt durch Angabe dieser Typnummer. Abbildung 6.4 zeigt einen Ausschnitt aus einer solchen Konfigurationsdatei. Die Konfiguration erfolgt durch Aufruf der Methode *readSensorDestInfo()*.

6.3.4 Datenannahme

Das eigentliche Einlesen der Sensor-Datensätze erfolgt durch den Aufruf der Methode *parseDataFile()*, welche als Argument entweder den Pfad zu einer (auch komprimierten) XML-Datei oder aber einen Zeiger auf den Speicherbereich, in der die XML-Datei zur Verfügung steht, erwartet. Die Datei wird mit dem XML-Parser durchlaufen, wobei die Datensätze in von der STL zur Verfügung gestellten Container-Klassen gesammelt werden. Durch den vorgegebenen Grundaufbau eines Sensorergebnisses wird dabei kein weiteres Wissen über den Sensor benötigt. Das Mapping eines Datensatzes ist eindeutig, da im Plain-File-System durch Sensor, Datum, Uhrzeit und Rechnername (node_id) das entsprechende Verzeichnis

[4] Die genaue Zuordnung kann man der Dokumentation der Software entnehmen.

eindeutig bestimmt ist, in relationalen Datenbanken durch das `name`-Attribut die Tabelle und durch die Bezeichnungen der jeweiligen XML-Knoten die entsprechende Spalte.

Auf diese Weise können mehrere Dateien hintereinander eingelesen werden. Der eigentliche Schreibvorgang in die jeweiligen Datenlager erfolgt durch Aufruf der Methode *write()*. Diese sortiert die gesammelten Datensätze, so dass Messwerte gleicher Sensoren in der Liste hintereinander stehen. Durch diesen Vorgang wird beim Schreiben vor allem unter Plain-File-Systemen aber auch bei relationalen Systemen ein erheblicher Geschwindigkeitsvorteil erzielt. *write()* stellt Verbindungen zu den jeweiligen Datenlagern her und übergibt diesen die Daten. Eventuell auftretende Fehler werden dabei abgefangen, wobei Daten, die nicht sicher geschrieben werden konnten, im XML-Format gesichert werden und für weitere Verwendung zur Verfügung stehen.

Im Rahmen dieser Arbeit konnte aus zeitlichen Gründen lediglich das relationale DBMS *MaxDB* unter Verwendung der in Kapitel 5.4.2 vorgestellten, selbst entwickelten Wrapper-Bibliothek als mögliches Datenlager eingebunden werden. Für XML-Datenbanken stellt das Interface jedoch nach dem Einlesen bzw. nach Ausführen der *write()*-Methode in einem Container einzelne XML-Dateien zur Verfügung, die von einer nativen XML-Datenbank (z. B. *Natix*) direkt ohne weitere Einschränkungen eingefügt werden können. Dies konnte allerdings nicht gründlich getestet werden, da *Natix* in der zur Verfügung gestellten Prerelease-Version nur eingeschränkt verwendet werden konnte.

6.3.5 Datenabfrage

Die Datenabfrage erfolgt in mehreren Schritten, in denen, durch den Aufruf verschiedener Methoden, Einschränkungen auf den Datenbestand formuliert werden. Mit Hilfe der Methode *querySensor()* muss zuerst der Sensor spezifiziert werden, auf dessen Werte zugegriffen werden soll. Jede neue Anfrage muss mit dem Aufruf dieser Methode beginnen. Im zweiten Schritt werden die Bedingungen für die Abfrage formuliert. Dies erfolgt mit Hilfe der Methode *field()*, welche als Parameter den Namen des Feldes, den dazugehörigen Vergleichswert sowie die Art der Einschränkung benötigt. Diese Methode kann mehrfach ausgeführt werden, um weitere Einschränkungen auf den Datenbestand zu formulieren. Optional besteht in einem dritten Schritt die Möglichkeit, die Auswahl der im Anfrageergebnis enthaltenen Felder mit der Methode *select()* zu bestimmen. Wird *select()* nicht aufgerufen, so steht der gesamte Datensatz, der die Bedingung erfüllt, im Ergebnis zur Verfügung. Auch diese Methode kann mehrfach aufgerufen werden, um mehrere Felder im Ergebnis zu erhalten.

Im letzten Schritt wird die eigentliche Anfrage durch Aufruf der Methode *execute()* ausgeführt. Diese übersetzt die angegebenen Bedingungen in die Sprache des entsprechenden Datenlagers, leitet sie an dieses weiter und führt sie aus.

Am Beispiel einer Abfrage, die alle Datensätze des Sensors „cpu-load“ zurückliefert, bei denen der Sensor auf dem Knoten „mdax“ etwas Außergewöhnliches registriert und deswegen den `<additional>`-Bereich genutzt hat, werden die Schritte nochmals verdeutlicht:

```
querySensor("cpu-load");
field("node_id", "mdax", "eq");
field("additional", "", "neq");
// select("*");                        /* optional */
execute();
```

Das Ergebnis der Anfrage steht in einem Objekt der *cQueryResult*-Klasse bereit. Dieses gewährt zellenweisen Zugriff auf die Ergebnismatrix (`dataset[i].field[j]`).

6.4 Tests

Um die Eigenschaften, aber auch insbesondere die Grenzen des *Storage-Interface* festzustellen, werden einige Tests durchgeführt. Eine wichtige Messgröße ist dabei der durch das Interface zu erwartende Overhead. Für Leseanfragen kann dieser mit Sicherheit vernachlässigt werden, da die Methoden lediglich eine Übersetzung der Anfragen in die Anfragesprache des verwendeten Datenlagers durchführen. Die Einfügeoperationen hingegen erfordern weitaus mehr Aufwand. Deswegen werden im Folgenden die Eigenschaften der Datenannahme ermittelt.

6.4.1 Tests der Datenannahme

Die Geschwindigkeit der Schreiboperationen auf das Datenlager wird hauptsächlich durch dessen Eigenschaften bestimmt. Diese wurden bereits in Kapitel 5 getestet. Da die Datenlager eine unterschiedliche Performance aufweisen, ist es nicht sinnvoll, die Eigenschaften der *write()*-Methode zu messen, welche im Wesentlichen nur mit dem Datenlager über dessen Schnittstelle kommuniziert. Lediglich das Sortieren der Daten vor dem Schreibvorgang, welches ebenfalls in der *write()*-Methode stattfindet, erfolgt unabhängig vom verwendeten

Datenlager. Die Sortierung im *Storage-Interface* basiert auf den von der STL zur Verfügung gestellten Methoden. Diese wiederum verwenden den 1961 von C.A.R. Hoare entwickelten Quicksort-Algorithmus [63], welcher im Durchschnitt

$$O(n \cdot \log_2 n)$$

Schritte benötigt. Quicksort ist in der Praxis schneller als andere Sortierverfahren mit ebenfalls logarithmischem Rechenzeitverhalten und gehört deswegen zu den am häufigsten benutzten Sortieralgorithmen [64]. Auf einen eigenständigen Test der Sortiermethode wurde daher verzichtet, da keine neuen Erkenntnisse erwarten wurden.

Das Einlesen der Sensor-Messwerte hingegen, welches durch den Aufruf der *parseDataFile()*-Methode durchgeführt wird, erfolgt unabhängig vom verwendeten Datenlager. Um den durch das *Storage-Interface* verursachten Overhead zu messen, werden daher verschiedene kumulierte XML-Dateien an diese Methode übergeben. Die dafür benötigte Zeit wird in Abhängigkeit von Anzahl und Größe der Datensätze ermittelt.

Die kumulierte XML-Datei wird vom *Storage-Interface* von der Festplatte gelesen. Dies hat im Wesentlichen zwei Gründe. Zum einen sollen z. T. große Datenmengen im Bereich mehrerer hundert Megabyte verarbeitet werden. Um die dazugehörige XML-Datei komplett im Speicher bereitzustellen, wird u. U. bereits mehr Speicher benötigt als Hauptspeicher zur Verfügung steht. Um genau dies zu vermeiden wird, wie in Kapitel 6.2.6 dargelegt, ein SAX-Parser verwendet. Auf der anderen Seite sollen die ausgelesenen Messwerte ihrerseits durchaus vollständig im Hauptspeicher Platz finden, da sonst durch Paging[5] die Ergebnisse signifikant verfälscht würden.

6.4.2 Ergebnisse

Die Ergebnisse der Messungen, welche auf den Knoten des TI-Clusters (Kapitel 5.2) durchgeführt wurden, sind in Abbildung 6.5 dargestellt. Man erkennt, dass der vom *Storage-Interface* verursachte Overhead bedingt durch Öffnen, Parsen und Einsortieren der Datensätze im Wesentlichen linear ansteigt. Durch Messungenauigkeiten verursacht sind im Bereich weniger Datensätze Abweichungen vom linearen Verlauf feststellbar. Beim Einfügen sehr vieler Datensätze ist, insbesondere bei den Messungen mit 180 und 360 Byte, ein nicht linearer Anstieg der benötigten Zeit zu erkennen. Die Überprüfung der Systemressourcen durch

[5] Auslagern einzelner Speicherseiten eines Prozesses aus dem Hauptspeicher auf die Festplatte

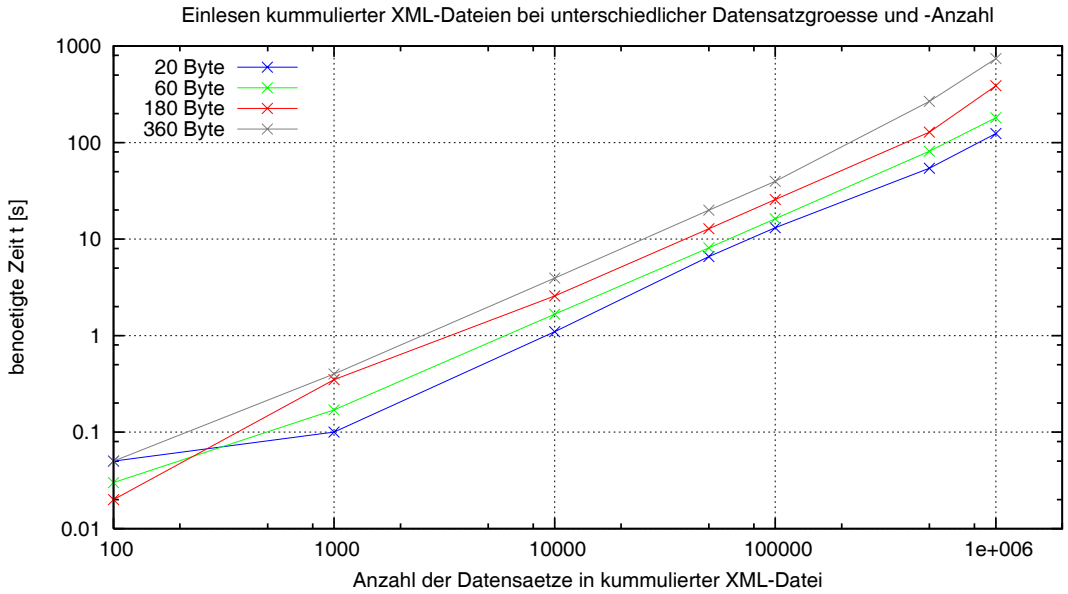

Abbildung 6.5: Ergebnisse der Messungen zur Bestimmung des vom *Storage-Interface* verursachten Overheads beim Einlesen kumulierter XML-Dateien bei unterschiedlicher Anzahl und Größe der Datensätze. Es ist zu beachten, dass die tatsächliche Dateigröße vom Produkt von Datensatzgröße und Anzahl abweicht (vgl. Kapitel 4.2.2).

Größe eines Datensatzes [Byte]	20	60	180	360
mittlere Einlesefrequenz ν [Hz]	8598	6070	3687	2514

Tabelle 6.1: Mittlere Einlesefrequenz (Datensätze pro Zeiteinheit) des *Storage-Interface* im linearen Bereich der Messung in Abbildung 6.5

top[6] legt nahe, dass dies sehr wahrscheinlich mit den vorher erwähnten Paging-Effekten zusammenhängt, da in diesem Bereich Datenmengen nur wenig unterhalb des tatsächlich verfügbaren, physikalischen Hauptspeichers verarbeitet werden.

Der ansonsten lineare Verlauf war zu erwarten, da, bis auf das Lesen der Daten von der Festplatte, alle Transaktionen im Hauptspeicher erfolgen. In Tabelle 6.1 ist die mittlere Einlesefrequenz im linearen Bereich angegeben.

[6] Das Unix-Programm *top* ermöglicht eine Kontrolle der Ressourcenausnutzung (u. a. CPU-Auslastung, realer und Swap-Speicher-Verbrauch).

Die Messungen zeigen, dass das *Storage-Interface* die Daten schneller verarbeitet als alle in Kapitel 5 vorgestellten Datenlager. Die Performance ist somit ausreichend. Insbesondere bestätigen diese Tests die Wahl des SAX-Parsers, da nur dadurch sehr große XML-Dateien verarbeitet werden können. Das *Storage-Interface* hat im Rahmen dieses Tests Dateien mit einem Volumen von knapp 500 MB problemlos verarbeitet.

6.5 Zusammenfassung

Das *Storage-Interface* stellt unter *C++* eine einfache, generische Schnittstelle für unterschiedliche Datenlager zur Verfügung. Die Konfiguration sowie das Schreiben der Daten erfolgt durch XML-Dateien. Bei eventuellen Fehlern gehen zu schreibende Daten nicht verloren, sondern stehen für alternative Einfügeversuche zur Verfügung.

Das Interface unterstützt einfache Abfragen auf den Datenbestand. Diese können leicht ausgeführt, die Ergebnisse können effizient gelesen werden.

Allerdings ist noch nicht der gesamte Leistungsumfang verfügbar, da z. B. die Anbindung an XML-Datenbanken nicht getestet werden konnte. Die gesamte Software wurde umfassend dokumentiert und realisierte Komponenten so weit wie möglich getestet.

7 Zusammenfassung

7.1 Untersuchung verschiedener Datenlager

Der erste Teil dieser Arbeit beschäftigte sich mit der Untersuchung der Eignung verschiedener Datenlager für hochskalierende Monitoringsysteme. Dazu wurden die von verschiedenen Monitoringsystemen verwendeten Datenlager vorgestellt und auf ihre Eignung hin untersucht. Auf Dateisystemen basierende Datenlager erweisen sich dabei für die meisten durch umfassendes Monitoring gewonnenen Daten als ungeeignet. Hauptsächlich stellte sich das Schreiben kleiner Datensätze als Problem heraus. Bei Verwendung des Ext3-Dateisystems wurden Schreibraten von unter 20 Hz gemessen. Nur bei geeigneter Reihenfolge der Schreibzugriffe, die in der Praxis so nicht gewährleistet werden kann, konnten gute Ergebnisse im Bereich einiger kHz erreicht werden. Bei der Durchführung weniger, ganz gezielter Anfragen hingegen konnten Plain-File-Systeme durchaus überzeugen. Die Verwendung geeigneter Verzeichnisebenen zur Strukturierung des Datenbestands ermöglichte, die Suche auf einen kleinen Datenbestand zu reduzieren und daher schnell auszuführen.

Relationale Datenbank-Management-Systeme konnten sich zur Datenablage bekannter Messwerte gut behaupten. Mit ihnen ließen sich ohne besonderen Aufwand Schreibraten im Bereich einiger kHz erreichen. Allerdings erwiesen sie sich aufgrund ihrer relationalen Konzeption für den Umgang mit schemalosen Messwerten als ungeeignet. XML-Datenbanken hingegen zeigten gerade in diesem Fall ihre Stärken. Ihre Schreibrate lag mit ca. 500 Hz im Mittelfeld zwischen Plain-File- und relationalen Systemen. Da allerdings native XML-Datenbanken noch sehr neu sind, existieren kaum wirklich ausgereifte Produkte auf dem Markt. Mit steigender Erfahrung werden sich die Ergebnisse sicher verbessern.

7.2 Entwicklung einer generischen Schnittstelle für multiple Datenlager

Die Entwicklung des *Storage-Interface* ist auf die Ergebnisse der Tests zurückzuführen. Da durch umfassendes Monitoring z. T. in Typ und Struktur sehr unterschiedliche Werte erfasst werden, erfordern diese unterschiedliche Eigenschaften bezüglich der Datenhaltung.

Um die jeweiligen Vorteile der verschiedenen Datenlager auszunutzen, wurde eine generische Schnittstelle unter *C++* entwickelt, welche durch ein einheitliches Interface multiple Datenlager zur Speicherung verwendet. Die Konfiguration sowie das Schreiben der Daten erfolgt dabei durch Übergabe von XML-Dateien. Dadurch wird eine hohe Flexibilität erreicht. Die Festlegung obligatorischer Felder innerhalb eines Datensatzes stellt zudem sicher, dass auch komplexe schemalose Sensordaten auf jedes Datenlager abgebildet werden können. Weiterhin unterstützt das Interface einfache Abfragen auf den gesamten Datenbestand. Die Abfragen können dazu vom Interface in die native Abfragesprache des darunter liegenden Datenlagers abgebildet werden. Zur Zeit ist sowohl der Lese- als auch der Schreibzugriff unter Verwendung der selbst entwickelten Wrapper-Bibliothek *libsapdb* für das relationale DBMS *MaxDB*, ehemals *SAP-DB* vollständig implementiert.

Die entwickelte Software wurde auf ihre Performance getestet. Dabei wurden z. T. große kumulierte Messwert-Dateien (mehrere hundert MB) verarbeitet. Die gemessene Einlesefrequenz lag – je nach Größe der Datensätze – zwischen 2,5 und 8,6 kHz.

7.3 Ausblick

Das *Storage-Interface* sollte in Zukunft weitere Datenlager unterstützen. Die angekündigte Version von *Natix* könnte dabei als native XML-Datenbank durchaus in Frage kommen.

Das *Storage-Interface* kann gut an das in Kapitel 3.3.4 vorgestellte *HCMS* angebunden werden, da beide Systeme zur Konfiguration und zum Datenaustausch XML verwenden. Die Kombination eines robusten, hochskalierenden Monitoringsystems und einer flexiblen Datenlagerung ermöglicht somit die effiziente Überwachung und Kontrolle großer verteilter Systeme, wie sie z. B. zur Analyse der Daten der LHC-Experimente benötigt werden.

Um eine einfache Bedienung für Administratoren zu gewährleisten, sollte auch die Entwicklung eines graphischen Benutzer-Interfaces in Betracht gezogen werden. Dieses könnte über die vom *Storage-Interface* zur Verfügung gestellte Schnittstelle Abfragen auf den gesamten Datenbestand durchführen, diese graphisch aufbereiten und die verteilte Struktur vollständig vor dem Anwender verstecken.

Literaturverzeichnis

[1] Homepage des CERN
http://public.web.cern.ch/public
(März 2004)

[2] LHC
http://lhc-new-homepage.web.cern.ch/lhc-new-homepage
(März 2004)

[3] LEP
http://public.web.cern.ch/public/about/how/experiments/experiments.html
(März 2004)

[4] Homepage des RHIC-Collider
http://www.bnl.gov/rhic
(März 2004)

[5] ATLAS-Experiment
http://pdg.lbl.gov/atlas/atlas.html
(März 2004)

[6] CMS-Experiment
http://cmsinfo.cern.ch/Welcome.html
(März 2004)

[7] ALICE-Experiment
http://alice.web.cern.ch/Alice/AliceNew
(März 2004)

[8] LHCb-Experiment
http://lhcb-public.web.cern.ch/lhcb-public/default.htm
(März 2004)

[9] Particle Data Group
Hagiwara, K. et al.:
The Review of Particle Physics
Phys. Rev. D 66, 010001; *(2002)*
http://pdg.lbl.gov/2002/gxxx.pdf
(März 2004)

[10] Deutsche Physikalische Gesellschaft (DPG) – Welt der Physik
`http://www.weltderphysik.de/themen/bausteine/teilchen/instrumente/lhc`
(März 2004)

[11] Higgs and SUSY searches at future colliders
`http://www.iisc.ernet.in/pramana/april2000/dae17.htm`
(März 2004)

[12] Beowulf Cluster
`http://www.beowulf.org/`
(März 2004)

[13] NASA
`http://www.nasa.gov`
(März 2004)

[14] HPCWire: Interview on High Throughput Computing
`http://www.cs.wisc.edu/condor/HPCwire.1`
(März 2004)

[15] Brüning, U.:
Computer Architecture II – Rechnerarchitektur II
Universität Mannheim; *(WS 03/04)*
`http://mufasa.informatik.uni-mannheim.de/lsra/lectures`
(März 2004)

[16] Foster, I.; Kesselman C.:
The Grid: Blueprint for a New Computing Infrastructure
Morgan Kaufmann; 1st edition *(1998)*

[17] Data Grid Project, Work Package 4. Fault Tolerance
`http://hep-proj-grid-fabric.web.cern.ch/hep-proj-grid-fabric`
(März 2004)

[18] XML-Spezifikation des W3C
`http://www.w3.org/XML`
(März 2004)

[19] Unicode Homepage
`http://www.unicode.org`
(März 2004)

[20] Stroustrup, B.:
Die C++-Programmiersprache
Addison-Wesley; 4. aktualisierte Auflage *(2000)*

[21] Prescod, P., Goldfarb, C.:
The XML Handbook
Prentice-Hall International; 1st edition *(1998)*

[22] Expat XML Parser
`http://expat.sourceforge.net`
(März 2004)

[23] LibXML XML Parser
`http://www.xmlsoft.org`
(März 2004)

[24] GNOME
`http://www.gnome.org`
(März 2004)

[25] Xerces XML Parser
`http://xml.apache.org`
(März 2004)

[26] Oracle
`http://oracle.com`
(März 2004)

[27] Tiny XML Parser
`http://www.grinninglizard.com/tinyxml`
(März 2004)

[28] DOM-Spezifikation des W3C
`http://www.w3.org/DOM`
(März 2004)

[29] SAX-Spezifikation
`http://www.saxproject.org`
(März 2004)

[30] Shannon, Claude E.:
A Mathematical Theory of Communication
Bell System Technical Journal, vol. 27; *(1948)*
`http://cm.bell-labs.com/cm/ms/what/shannonday/paper.html`
(März 2004)

[31] GZIP Homepage
`http://www.gzip.org`
(März 2004)

[32] BZIP2 Homepage
http://sources.redhat.com/bzip2
(März 2004)

[33] Hillesheimer, H.:
Ein skalierendes und selbstkonfigurierendes Monitoring System auf Linux-Clustern
Universität Heidelberg; Lehrstuhl für Technische Informatik *(2002)*
http://www.kip.uni-heidelberg.de/ti/publications/diploma
(März 2004)

[34] Ganglia Homepage
http://ganglia.sourceforge.net
(März 2004)

[35] RRDTool Homepage
http://www.rrdtool.com
(März 2004)

[36] Übersicht über verschiedene Softare-Linzenzen
http://opensource.org/licenses
(März 2004)

[37] xmlsysd Homepage
http://www.phy.duke.edu/~rgb/Beowulf/xmlsysd.php
(März 2004)

[38] Modified Open Publication License
http://www.phy.duke.edu/~rgb/OPL.php
(März 2004)

[39] Supermon Homepage
http://www.acl.lanl.gov/supermon
(März 2004)

[40] Die Lisp Programiersprache
http://www.apl.jhu.edu/~hall/lisp.html
(März 2004)

[41] Kemper, A. / Eickel, A.:
Datenbanksysteme
Oldenburg Verlag; 2. Auflage *(1997)*

[42] Gesellschaft für Informatik
Küspert, K. / Schaarschmidt, R.:
Archivierung in Datenbanksystemen
Friedrich-Schiller-Universität Jena; Institut für Informatik *(1998)*
http://www.gi-ev.de/informatik/lexikon/inf-lex-archivierung-db.shtml
(März 2004)

[43] Lufter, J. / Schaarschmidt, R. :
Anforderungen und Konzepte für die datenbanksystem-integrierte Archivierung mit ASQL
Friedrich-Schiller-Universität Jena; Institut für Informatik, Math/Inf/98/01 *(1998)*

[44] SAP AG
`http://www.sap.com`
(März 2004)

[45] Testcluster des Lehrstuhls für Technische Informatik
`http://www.kip.uni-heidelberg.de/ti/cluster`
(März 2004)

[46] Kernighan, B. W. / Ritchie, D. M.:
The C Programming Language
Prentice Hall; 2nd edition *(1988)* .

[47] Kleinert, J.:
Die neuen Dateisysteme
Linux Magazin; *(Ausgabe März 2004)* .

[48] DWARW
`http://www.kip.uni-heidelberg.de/ti/ClusterRAID/software/software.shtml`
(März 2004)

[49] MaxDB – MySQL AB
`http://www.mysql.com/products/maxdb`
(März 2004)

[50] MySQL – MySQL AB
`http://www.mysql.com`
(März 2004)

[51] System Software and Distributed Systems
`http://www.syssoft.uni-trier.de/systemsoftware`
(März 2004)

[52] Sun Microsystems
`http://www.sun.com`
(März 2004)

[53] Perl
`http://www.perl.com`
(März 2004)

[54] Python
`http://www.python.org`
(März 2004)

[55] PHP-Tutorial
http://www.usegroup.de/software/phptutorial
(März 2004)

[56] Tamino XML-Server – Software AG
http://www.softwareag.de
(März 2004)

[57] Universität Mannheim – Lehrstuhl für Praktische Informatik III
http://pi3.informatik.uni-mannheim.de
(März 2004)

[58] Natix – data ex machina GmbH
http://www.dataexmachina.de
(März 2004)

[59] Java
http://java.sun.com
(März 2004)

[60] Breymann, U.:
Komponenten entwerfen mit der C++ STL
Addison Wesley; 2. Auflage, korrigierter Nachdruck *(1999)*

[61] XML-Parser Bechmark
http://xmlbench.sourceforge.net/results/benchmark200402/index.html
(März 2004)

[62] DoxyGen
http://www.doxygen.org
(März 2004)

[63] Hoare, C. A. R.:
Algorithm 64: Quicksort
Communications of the ACM ; ACM Press, New York, NY, USA *(1961)*
http://doi.acm.org/10.1145/366622.366644
(März 2004)

[64] Quicksort
http://www.informatik.uni-ulm.de/ki/Edu/Vorlesungen/PI-1/WS0203
(März 2004)

Abbildungsverzeichnis

Tabellenverzeichnis

Danksagung

Zuerst möchte ich mich bei Herrn Prof. Lindenstruth für die interessante Aufgabenstellung als auch für die große Unterstützung bedanken.

Ich danke Lord Hess von Herzen für seine umfassende Betreuung, sowie für seine Unterstützung, als ich sie am nötigsten hatte.

Ein besonderer Dank geht an meine Kollegen Frank Pister, Dr. Timm Morton Steinbeck, Arne Wiebalck und Ralf Panse. Nur dank ihrer wertvollen Anregungen und vielen Hilfestellungen konnte diese Arbeit zustande kommen. Timm Morton Steinbeck und Gil Szmigiel danke ich außerdem für das sehr hilfreiche Korrekturlesen dieser Arbeit.

Den Mitglieder des Lehrstuhls danke ich für eine außergewöhnlich angenehme Atmosphäre, die mir eine sehr schöne Zeit beschert hat. Dank auch auch an alle Mitarbeiter des Kirchhoff-Instituts für die stets freundliche Zusammenarbeit.

Nicht zuletzt Dank an den bisher noch unbekannten Zweitkorrektor dieser Arbeit.

Besonderen Dank auch an meine Eltern, die immer für mich da waren.

Erklärung zur selbständigen Verfassung

Ich versichere, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, im April 2004

Aaron Taylor