Rheinisch-Westfälische Technische Hochschule Aachen

Lehrstuhl für Informatik V
Informationssysteme
Prof. Dr. Matthias Jarke

I. Physikalisches Institut
Lehr- und Forschungsgebiet Hochenergiephysik
Prof. Dr. Christoph Berger

DIPLOMARBEIT

Studie zur Anbindung von objekt-orientierten Datenbanken in der Teilchenphysik

Sven Thoennissen

25. Juni 1999

Vorwort

Diese Diplomarbeit wurde von Herrn Prof. Dr. Christoph Berger (I. Physikalisches Institut) ermöglicht und betreut. Herr Prof. Dr. Matthias Jarke (Lehrstuhl Informatik V) ist Zweitgutachter. Meine Betreuer sind Thomas Hadig (Physik I) und Thomas List (Informatik V). Dies ist die erste Diplomarbeit bei H1 im Rahmen der Informatik, wodurch bei beiden Instituten Neuland betreten wird. Beide Lehrstühle arbeiten nun enger zusammen und hoffen auf eine erfolgreiche Zukunft.

Ich möchte den H1-Mitarbeitern Claus Kleinwort, Ralf Gerhards, Sergey V. Levonian, Hans-Christian Schultz-Coulon, Christoph Grab, Martin Hennecke, Jürgen Scheins, Sascha Caron und Jan Olsson für ihre Kooperation und Unterstützung danken. Dank gilt auch Frau Eva Arderiu-Ribera (CERN, RD45) für ihre Hilfe zu Objectivity/DB, Tony Johnson (SLAC, JAS) für JAS und Support, den Mitarbeitern der Firma Micram Object Technology für den Support zu Objectivity, Objectivity Inc. für ein White-Paper, Akmal B. Chaudhri für Mithilfe bei seinen Veröffentlichungen und Marion Brandt-Röhrig für ihre technische Unterstützung. Meiner Familie möchte ich für Geist, moralische Hilfestellung und Stärke während meines Studiums danken.

Hiermit versichere ich, daß ich die Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

(Datum) (Unterschrift/Sven Thoennissen)

Inhaltsverzeichnis

Vorwort **iii**

1 Einleitung **1**
- 1.1 Das H1-Experiment . . . 1
- 1.2 Übersicht . . . 1
- 1.3 Der H1-Detektor . . . 2

2 Problemanalyse **9**
- 2.1 Zielsetzung . . . 9
- 2.2 Java Analysis Studio . . . 13

3 Datenfluß und Ereignisse **17**
- 3.1 Datennahme bei H1 . . . 17
  - 3.1.1 Ereignisse aus Trigger Level 1–5 . . . 17
  - 3.1.2 H1-Programme . . . 21
  - 3.1.3 Setup-Informationen und Slow-Control . . . 21
- 3.2 Data Warehouse . . . 22
  - 3.2.1 Einführung . . . 22
  - 3.2.2 Der H1-Datenfluß und Data-Warehousing . . . 24
  - 3.2.3 Der Aktualisierungsfluß . . . 27
- 3.3 Beispiele von physikalischen Analyse-Vorgängen . . . 29
  - 3.3.1 Eine von Grund auf neue Analyse . . . 29
  - 3.3.2 Wiederverwertung existierender Ergebnisse . . . 32
- 3.4 Möglichkeiten zur Leistungssteigerung . . . 34

4 Die Objectivity-Datenbank **39**
- 4.1 Aufgaben und Fähigkeiten . . . 39
- 4.2 Organisation der Objekte . . . 42
- 4.3 Aktueller Projektstatus . . . 47
- 4.4 Größe der Datenbank . . . 48
- 4.5 Benchmark . . . 49
- 4.6 Verbesserungen . . . 54
- 4.7 Warum Objectivity/DB ? . . . 56
- 4.8 Alternative Datenbanksysteme . . . 57

4.8.1 Relationale Datenbanken . . . 57
4.8.2 Objekt-relationale Datenbanken . . . 57
4.8.3 Objekt-orientierte Datenbanken . . . 58
4.8.4 DBMS von anderen Herstellern . . . 59

5 Die Schnittstelle zu Objectivity/DB **61**
5.1 Anforderungen an die Schnittstelle . . . 61
5.2 Funktionsweise des DIMs für Objectivity/DB . . . 62
5.2.1 Visuelle Präsentation . . . 62
5.2.2 Interne Aktivitäten . . . 65
5.2.3 Java-Klassen . . . 67
5.3 Schwierigkeiten und Lösungen . . . 71
5.3.1 Erweiterung der Ereignisdaten . . . 71
5.3.2 Optimierung des Java-Zugriffs . . . 72
5.3.3 Java-Data-Server oder RMI-Server . . . 72
5.3.4 Histogrammierung von Elementen aus dynamischen Arrays . . . 72
5.4 Andere Schnittstellen . . . 73
5.4.1 Caltech/CMS . . . 73

6 Bewertung **75**
6.1 Veränderungen durch Java Analysis Studio . . . 75
6.2 Was bringt Objectivity/DB für H1 ? . . . 76
6.3 Resonanz von Benutzern bei H1 . . . 77
6.4 Modell zur Bewertung . . . 78
6.4.1 Einführung . . . 78
6.4.2 Anwendung des Modells auf H1 . . . 80
6.5 Schlußbetrachtung . . . 83
6.6 Ausblick . . . 84

A Technical Documentation **87**
A.1 User Documentation . . . 87
A.1.1 Basic Usage in Java Analysis Studio . . . 87
A.1.2 How to use the Objy-DIM . . . 88
A.2 Administrator Documentation . . . 91
A.2.1 Installing JAS and the Objy-Interface . . . 91
A.2.2 The RMI-Server . . . 91
A.2.3 How to update or to change the TTagEvent variables . . . 92
A.2.4 JAS Updates . . . 93
A.3 Developer Documentation . . . 93
A.3.1 Package `jas.jds.module.objy` . . . 93
A.3.2 Class `jas.swingstudio.objyRunrangeTreeAdaptor` . . . 94
A.3.3 Package `h1objy` . . . 95
A.3.4 `TTagEvent` . . . 96

A.4 Future Changes 97

B Literaturhinweise **99**
B.1 Referenzen aus der Teilchenphysik 99
B.2 Andere Hinweise 101

Kapitel 1

Einleitung

1.1 Das H1-Experiment

Das Deutsche Elektronen-Synchrotron (DESY) wurde 1959 in Hamburg zur Erforschung fundamentaler Eigenschaften in der Teilchenphysik gegründet und wird aus öffentlichen Mitteln finanziert. HERA ist der Name des Speicherringes, ein unterirdischer, nicht ganz kreisrunder Tunnel mit einem Umfang von 6.3 km. Dort werden Elementarteilchen durch Vorbeschleuniger eingespeist und bei mehreren Umläufen auf sehr hohe Energien beschleunigt. Diese Teilchen stoßen schließlich im H1-Detektor entgegengesetzt zusammen[1] und erzeugen viele andere Elementarteilchen, die vom Detektor erfaßt werden. Zur Zeit wird bei HERA mit Protonen und Elektronen[2] experimentiert.

Die H1-Kollaboration besteht aus etwa 400 Wissenschaftlern, die aus 39 Instituten und 12 Ländern stammen. Das H1-Experiment ist während der Jahre 1988–1992 entstanden und begann 1992 seine Datennahme. Das Ziel sind die Erfassung der Struktur des Protons, Untersuchungen grundlegender Wechselwirkungen zwischen Elementarteilchen sowie die Suche nach Möglichkeiten, das aktuelle Standardmodell der Physik auf die Probe zu stellen.

Da die Diplomarbeit im Rahmen eines Informatikstudiums verfaßt wird, soll bei der Erklärung des H1-Detektors nicht allzusehr auf die physikalischen Details eingegangen werden. Mehr Informationen über den H1-Detektor finden sich bei geeigneten Diplom- oder Doktorarbeiten bei Herrn Prof. Dr. Berger oder im H1-Detektor-Schriftstück [Collab93].

1.2 Übersicht

Die Diplomarbeit beschäftigt sich mit der Anbindung von Analyse-Software in Java an eine Datenbank. Dort sollen physikalische Meßergebnisse gespeichert werden, die den Mitarbeitern zur Auswertung zur Verfügung gestellt werden. Zusätzlich werden Aspekte aus der Informatik in bezug auf die Datenbank betrachtet.

[1] HERA wird daher auch als "Collider" bezeichnet. Andere Beschleuniger führen "Fixed-Target"-Experimente durch, wobei sich eines der beiden Teilchen in Ruhe befindet.

[2] Protonen werden auf 920 GeV, Elektronen auf 27.5 GeV beschleunigt. 1 GeV (Giga-Elektronen-Volt) ist die Energie, die ein Elektron benötigt, um eine Potentialdifferenz von 1 Giga-Volt zu überwinden. Die Teilchen erreichen dabei etwa 99.99% der Lichtgeschwindigkeit.

Nach einer Einführung über den Detektor und das H1-Experiment wird in Kapitel 2 die Problemstellung der Diplomarbeit erläutert. Da die Art und die Größe der Daten bei der Datenbank eine wichtige Rolle spielen, wird in Kapitel 3 der Daten- und Informationsfluß von der Erfassung der Meßdaten bis hin zur Auswertung durch den Forscher dargestellt werden. Es wird dabei die Modellierungssprache UML [Fowler98] benutzt. Der Datenfluß in H1 wird auf das Data-Warehouse-Konzept abgebildet, um Lösungsansätze aus diesem Bereich zu finden. Anschließend werden zwei reale Analyse-Szenarien aus der Teilchenphysik vorgestellt, anhand deren Probleme und Lösungen diskutiert werden.

Die Objectivity-Datenbank wird in Kapitel 4 beschrieben. Die Anforderungen von H1 werden zusammen mit den Realisierungen durch Objectivity/DB erläutert. Die Organisation der Datenbank wird erklärt und die Struktur der H1-Daten wird hierauf abgebildet. Es werden eigene Benchmarks vorgestellt und diskutiert, und anschließend wird auf Probleme und Verbesserungsvorschläge aufmerksam gemacht. Beim Vergleich vom objektorientierten mit dem relationalen Datenbank-Management-Konzept wird auf deren Stärken und Schwächen eingegangen. Schließlich werden in Anlehnung daran weitere Datenbanksysteme aus der gleichen Leistungsklasse vorgestellt.

In Kapitel 5 wird auf die Java-Schnittstelle eingegangen. Die Verbindungen zur Datenbank werden erklärt. Aufgetretene Probleme bei der Entwicklung werden zusammen mit deren Lösungen genannt. Abschließend wird in Kapitel 6 der Nutzen aus der Datenbank und der Java-Schnittstelle für H1 gezogen. Es wird festgestellt, welche Vor- und Nachteile sich ergeben haben, und ob eventuell andere Lösungen in Frage kommen.

Abschließend wird eine Bewertung des neuen Systems durchgeführt, welches aus der Java-Schnittstelle, deren Front-End und der Datenbank besteht. Hier gehen die Vor- und Nachteile dieser Komponenten in Verbindung mit Rückmeldungen von Benutzern ein. Zusätzlich wird ein Modell aus der Fachliteratur zur Bewertung von Informationssystemen auf Anwendbarkeit geprüft und mit dem System kombiniert. Die Diplomarbeit schließt mit einer zusammenfassenden Schlußfolgerung und einem praktischen Ausblick auf die nahe Zukunft der objekt-orientierten Entwicklungsarbeit von H1.

In Anhang A werden die Arbeitsweise und die Module der Java-Schnittstelle beschrieben. Er ist in 4 Abschnitte unterteilt: die Benutzer-Dokumentation führt in die Anwendung der Schnittstelle ein, die Administrator-Dokumentation erklärt Installation und Wartung der Schnittstelle, die Entwickler-Dokumentation beschreibt die Programm-Module, und eine Liste von zukünftigen Aufgaben wird beschrieben. Der gesamte Anhang der technischen Dokumentation wurde wegen der internationalen Belegschaft von H1 in Englisch verfaßt.

1.3 Der H1-Detektor

Die Elektronen und Protonen werden in Linearbeschleunigern[3] erzeugt und in ein System von Vorbeschleunigern[4] injiziert. Dort werden sie auf höhere Energien gebracht. Nach der Einspeisung in den HERA-Speicherring werden die Teilchen dort auf ihre Soll-Energie be-

[3] Linac 1 bis 3
[4] PIA, DESY II, DESY III, PETRA, DORIS. PETRA ist der Haupt-Vorbeschleuniger.

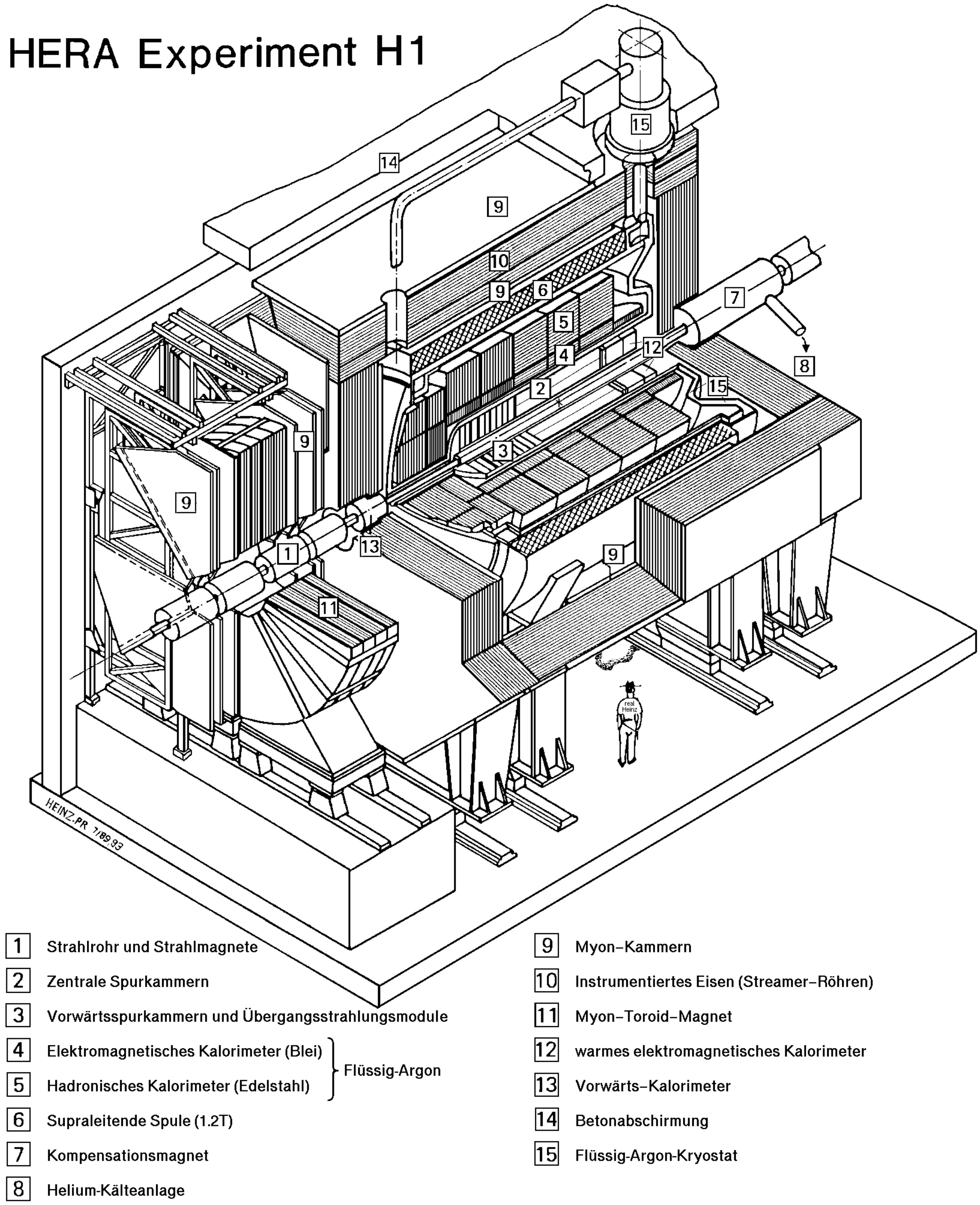


Abbildung 1.1: Aufbau des H1-Detektors

schleunigt.

Die Teilchen werden in Bündeln[5] im Ring transportiert. HERA wurde für die Speicherung von 210 Bündeln entwickelt. Ein Bündel hat eine Ausdehnung von etwa 1.4ns, das entspricht einer Breite von etwa 40cm. Die Kollisionen finden im H1-Detektor mit einer konstanten Rate von 10.4 MHz und einer Wahrscheinlichkeit in der Größenordnung von 0.1% statt. Diese Zeitspanne von 96ns wird bei H1 als 1 BC beschrieben[6]. Der H1-Detektor besteht aus neun wesentlichen Subdetektoren zur Messung der Energien und Teilchenbahnen von Elementarteilchen, die bei den Kollisionen entstehen (Abb. 1.1). Die gemessenen Rohdaten bestehen z.B. aus Energien, Vierervektoren, Bitketten oder Indizes für viele Meßzellen. Zu den Subdetektoren gehören Kalorimeter[7], Aufspür-Detektoren[8] und Myon-Detektoren. An diese Subdetektoren sind Schalteinheiten ("Trigger") angeschlossen, die mit logischen und arithmetischen Operationen entscheiden, ob die Meßdaten ein physikalisches Ereignis ("Event") darstellen.

Es existieren insgesamt fünf Triggermechanismen. Level 1 und 2 operieren auf Hardwareebene. Sie beruhen auf elektronischen Schaltungen, da an sie sehr hohe Geschwindigkeitsanforderungen gestellt werden. Level 3, 4 und 5 sind vollständig durch Software (Abb. 1.2) realisiert. Die Trigger Level 1–4 erzeugen entweder ein KEEP- oder REJECT-Signal, während Level 5 ausschließlich die Aufbereitung und Vorbereitung für die Analysen der Forscher erledigt. Ein REJECT-Signal bedeutet, daß das Ereignis nicht von Interesse ist und somit verworfen wird. Bei einem KEEP-Signal wird das Ereignis weitergereicht, da es die Entscheidungskriterien des zugehörigen Trigger Levels erfüllt hat. Es gelangt anschließend zum nachfolgenden Trigger Level. Mit dieser Anordnung sollen ungültige oder verfälschte Ereignisse gefiltert werden.

Die Anzahl der Trigger wird durch die Abstufung der Berechnungen motiviert. Eine Trigger-Stufe alleine kann während der sehr kurzen Experiment-Intervalle nicht die immensen Berechnungen durchführen. Daher sind bei H1 mehrere Berechnungs-Etappen hintereinander aufgebaut, die von Zwischenspeichern begleitet werden. Level 1 und 2 führen elementare Berechnungen aus, durch die eine schnelle Entscheidung gefällt werden muß, um die Totzeit möglichst klein zu halten. Die Totzeit ist der Zeitraum, in dem keine Messungen stattfinden können, weil das aktuelle Ereignis direkt im Puffer ausgewertet wird. Sie ist aufgrund der hohen Teilchenbündel-Kollisionsrate von 10.4 MHz und den Eigenschaften der Auslese-Elektronik leider nicht zu vermeiden und soll daher so klein wie möglich gehalten werden. Durch die Auslese-Apparatur bedingte Systemprobleme sind Verzögerungen durch Kabellängen oder elektronische Schaltkreise. Einige Subdetektoren weisen außerdem eine relativ große Verzögerung bem Füllen der Datenpuffer auf, was allerdings durch die Eigenschaften des Aufbaus selbst oder des Materials bedingt ist. Andere Subdetektoren reagieren sehr schnell. Dadurch würden viele Ereignisse nur aufgrund der schnellen Subdetektoren

[5]Bunches, Mengen von Teilchen

[6]Bunch Crossing time, äquivalent zur HERA-Clock 10.4 MHz

[7]Kalorimeter messen Energien von Teilchen. Beispiele sind das Flüssig-Argon-Kalorimeter (LAr, Liquid Argon) oder das Spaghetti-Kalorimeter (SpaCal).

[8]Tracker, z.B. Vorwärts/Zentral/Rückwärts, Silikon. Tracker messen den geometrischen Verlauf von Teilchen und liefern meist Vierervektoren oder z-Vertexe.

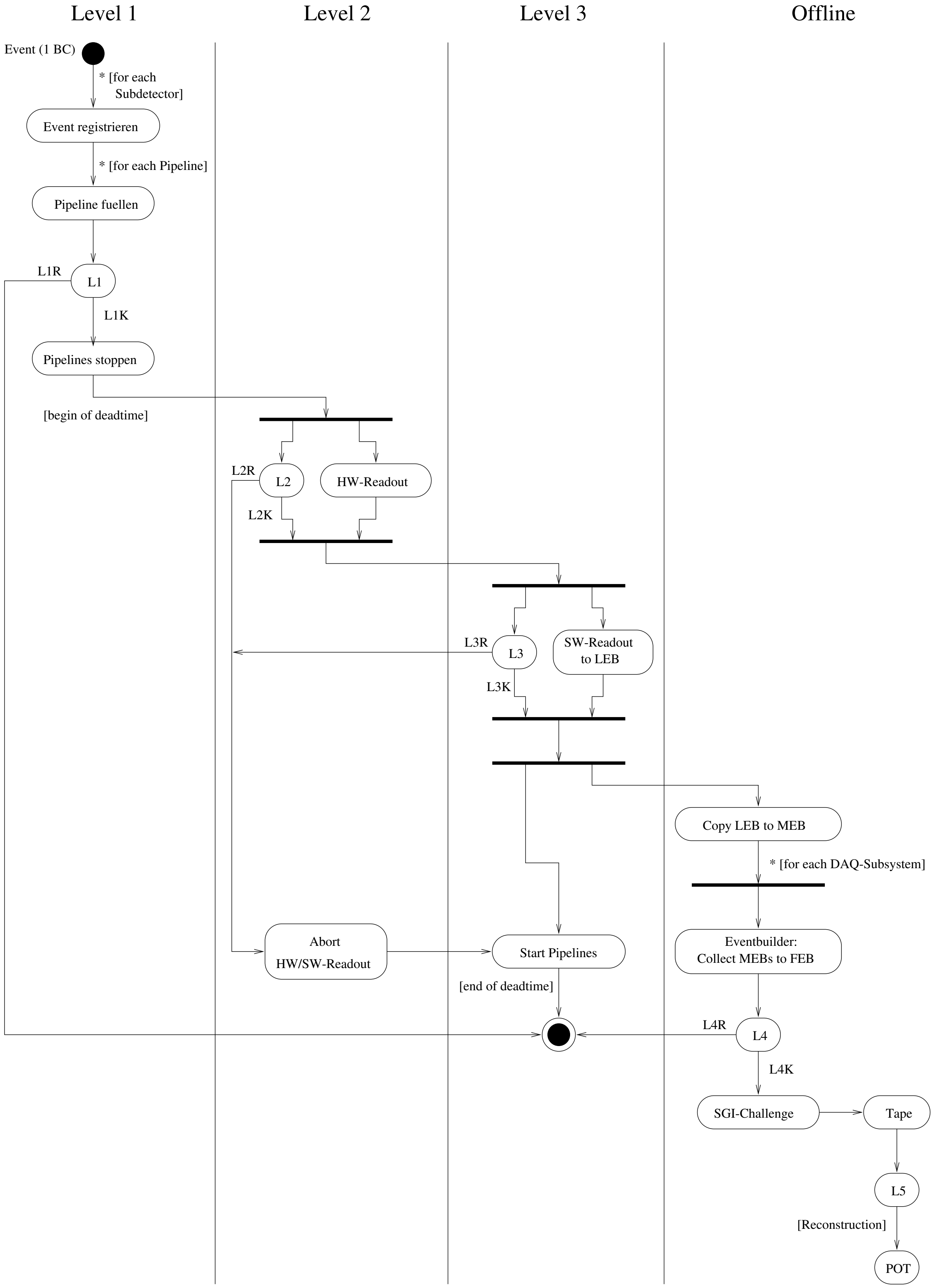


Abbildung 1.2: Trigger Levels des H1-Detektors

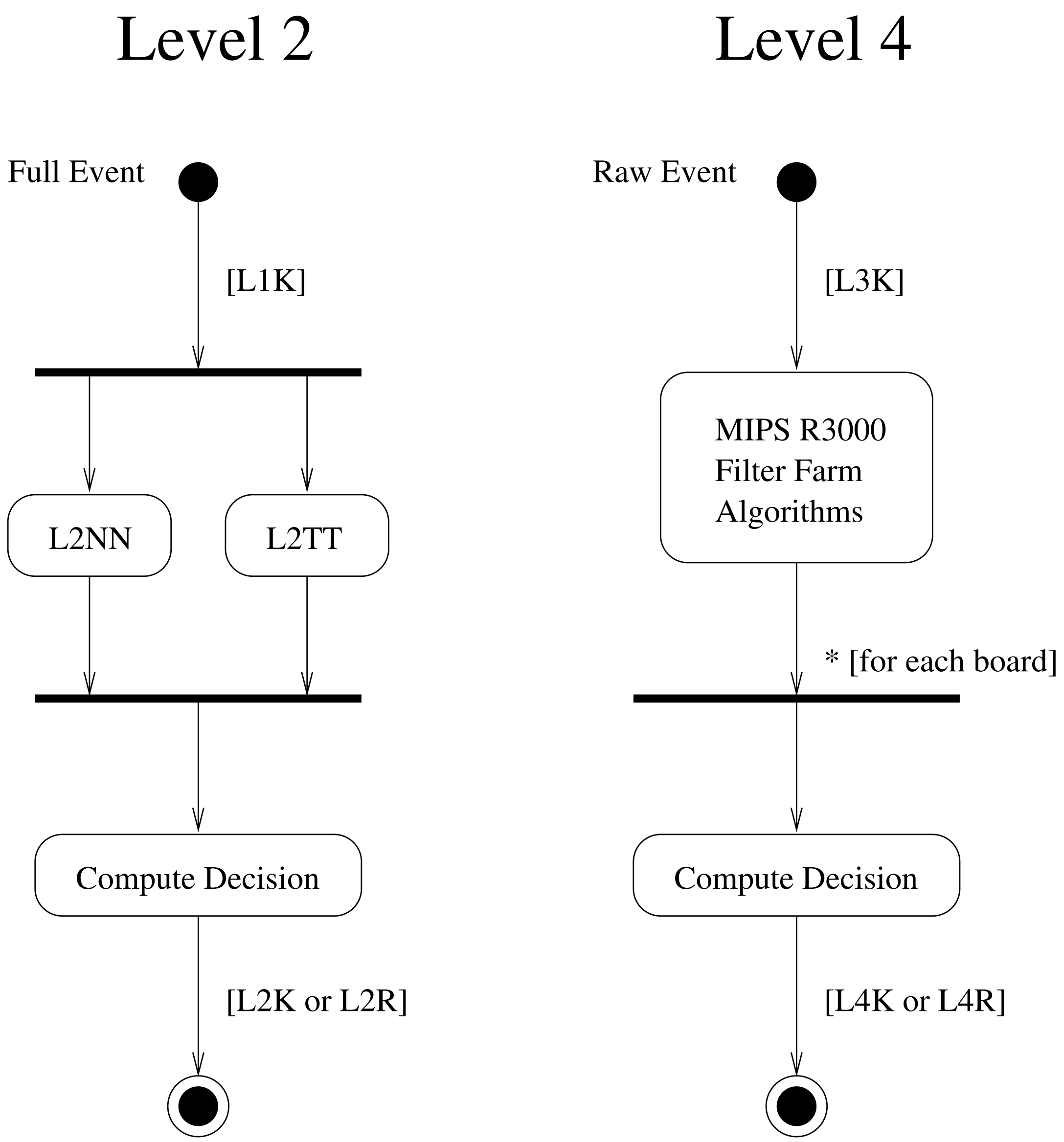


Abbildung 1.3: Level 2 und Level 4 Trigger

getriggert werden, und die sofort eintretende Totzeit würde andere, seltenere Ereignisse von anderen Subdetektoren verstreichen lassen. Daher werden die Subdetektoren, die sehr häufig Ereignisse erkennen, durch Prescale-Faktoren geregelt, welche nur jede n-te Ereignisaufnahme zulassen. Level 1 und Level 2 arbeiten innerhalb der Totzeit, weshalb bei ihrer Entwicklung besonderen Wert auf sehr kurze Antwortzeiten gelegt wurde. Die anderen Level (3–5) sind zeitintensiver, liegen aber außerhalb der Totzeit (Offline-Trigger). Die einzelnen Trigger-Mechanismen sollen im folgenden kurz erläutert werden.

Zu Level 1 gehören neun Subsysteme, die je einem Subdetektor zugeordnet sind. Jedes Subsystem verifiziert seine Meßdaten und füllt daraufhin die Datenpuffer[9] mit Trigger-Elementen[10]. Die Puffer können je nach Subsystem 27–35 BC Ereignisse speichern [Collab93]. Die Entscheidung des Level 1 Triggers beruht auf arithmetischen Operationen und boolescher Logik, die in Hardware realisiert ist. Dafür wird eine feste Zeit von 23 BC (2.5μs) benötigt. Die Datenpuffer sind also gerade groß genug, um die Daten für den Zeitraum halten zu können, der für eine Level-1-Entscheidung benötigt wird. Die zentrale Trigger-Logik[11] berechnet aus allen 256 Trigger-Elementen ein L1KEEP-Signal (L1K), wodurch die Datenpuffer in den Subsytemen angehalten und ausgelesen werden. Das Anhalten der Puffer ist notwendig, da die Subdetektoren die Ereignisse — unabhängig von Level 1 — kontinuierlich in die Datenpuffer schreiben. Falls das Ereignis unzulässig ist, wird statt L1KEEP ein L1REJECT (L1R) signalisiert. Ein Ereignis kann ungültig sein, wenn z.B. Reaktionen mit Restgas-Atomen oder Zusammenstöße mit der Wand stattgefunden haben. Damit die Ereignisdaten während der Bearbeitung durch die nachfolgenden Triggerstufen nicht verloren gehen, müssen die Datenpuffer im Falle eines erfolgreichen Ereignisses (L1K) gestoppt werden, und die Totzeit beginnt. Nachfolgende Ereignisse gehen während dieser Zeit verloren.

Level 2 arbeitet innerhalb der Totzeit und operiert daher synchron zu Level 1. Level 2 besteht hauptsächlich aus zwei parallelen Subsystemen: ein topologischer Trigger (L2TT) und ein neuronales Netzwerk (L2NN, Abb. 1.3). Parallel zum Start von Level 2 wird bereits das Auslesen der Subdetektoren gestartet ("Hardware Readout"). Dieser Vorgang dauert ca. 2.5ms. Nach 20μs fester Bearbeitungszeit durch Level 2 wird entweder ein L2K oder ein L2R signalisiert. Letzteres veranlaßt den Abbruch des Auslesevorganges und startet die Datenpuffer erneut. Ein L2K führt zu weiteren Ausleseprozessen ("Software Readout"), die die Ereignisdaten in einen Zwischenspeicher[12] sichern. Gleichzeitig dazu wird Level 3 gestartet. Hier arbeitet ein RISC-Prozessor und erstellt nach einigen hundert Mikrosekunden ein L3K- oder ein L3R-Signal. Seit 1997 ist Level 3 allerdings nicht in Betrieb, soll hier aber der Vollständigkeit wegen genannt werden.

Das Ereignis wird nun von Datennahmesystemen[13] in Ereignis-Puffer[14] geschrieben, die mehrere Ereignisse aufnehmen können. Die bei Level 1 begonnene Totzeit ist nun beendet,

[9] Pipelines
[10] Trigger Elements (TE)
[11] Central Trigger Logic (CTL)
[12] Local Event Buffer (LEB)
[13] Data Acquisition Subsystems
[14] Multi Event Buffer (MEB)

und die Datenpuffer der Subdetektoren sind wieder bereit zur Datennahme. Die Ereignis-Puffer sind mit der zentralen Datennahme[15] durch einen Lichtleiterring verbunden. Die Ereignis-Erstellungseinheit [16] verbindet schließlich alle partiellen Ereignisse zu einem zusammenhängenden Ereignis. Hier setzen viele Verbraucher an, die der Kontrolle und der Weiterverarbeitung der Ereignisdaten dienen — unter anderem der Level 4 Trigger. Dieser besteht aus 30 PowerPC-Prozessorboards[17], auf denen Programme zur Rekonstruktion und Filterung ausgeführt werden (Abb. 1.3). Level 4 wird vom Event-Builder bei einer Eingangsrate von 35–40 Hz [Levoni99] mit Ereignissen versorgt. Ereignisse, die Level 4 passieren, werden an einen entfernten SGI-Challenge Computer geschickt und dort auf Band geschrieben. Die Level-4-Ausgangsrate dieser Ereignisse liegt momentan bei 4–5 Hz [Levoni99].

Bei H1 werden die Ereignisse innerhalb von Staffeln oder Läufen[18] gemessen. Ein Lauf umfaßt einige tausend Ereignisse und ist in etwa 30 Minuten fertiggestellt. Dies deckt sich mit der Level-4-Ausgangsrate (s.o.). Zu jedem Lauf werden Informationen wie Konfiguration, Geometrie, Kalibrierungskonstanten und Trigger-Raten[19] in eine Datenbank geschrieben. Diese kann später bei der Rekonstruktion (s.u.) ausgelesen und auch verändert werden. Auf diese Oracle-Datenbank wird in Abschnitt 3.1.3 noch eingegangen. Sie ist jedoch unabhängig von der Objectivity-Datenbank, für die in dieser Diplomarbeit eine Programmschnittstelle entwickelt wurde.

Der Level 5 Trigger wird auf einem Mainframe-Computer (SGI-Challenge) mit 16 Prozessoren [Itterb97] ausgeführt und dient zur endgültigen Vorbereitung und Filterung der Ereignisse aus Level 4, die anschließend als rekonstruierte Ereignisse auf Band geschrieben werden[20]. Die durchschnittliche Verarbeitungszeit pro Ereignis liegt bei 1s. Im Gegensatz zu Trigger Level 1–4 produziert Level 5 kein KEEP-/REJECT-Entscheidungssignal. Das Ereignis wird nie verworfen, sondern stets nach der Aufbereitung gespeichert. In Level 5 findet der Hauptanteil der Rekonstruktionsarbeit statt. Die Rekonstruktion von Ereignissen ist ein wichtiger Bestandteil bei der Aufbereitung von Ereignisdaten aus Teilchendetektoren, da die am Detektor gemessenen Daten nicht direkt für die Forscher brauchbar sind. Diese Daten stellen Werte dar, die sich am technischen Aufbau, an den Meßaggregaten und an den Eigenschaften der Subdetektoren orientieren. Es ist notwendig, daß sie in physikalisch greifbare Größen transformiert werden, mit denen die Physiker schließlich arbeiten können. Dieser Aufbereitungsvorgang wird in der Elementarteilchenphysik Rekonstruktion genannt.

[15]Central Data Acquisition (CDAQ)
[16]Event Builder
[17]Stand 04/99: PPC CPUs 604e, 604r 166 MHz, 200 MHz, 300 MHz.
[18]Eine Staffel ist bei H1 als “Run” bekannt.
[19]bekannt als Scaler-Info
[20]Production Output Tape (POT)

Kapitel 2

Problemanalyse

2.1 Zielsetzung

Bei H1 wurde ein Projekt namens OOP ("Object Oriented Programming in H1", oder auch als Zweitname OOT, "Object Oriented Technology") gestartet, welches sich mit der Einführung von objekt-orientierter Entwicklung von Werkzeugen beschäftigt. In diesem Projekt ist die Inbetriebnahme des objekt-orientierten Datenbanksystems (ODBMS oder OODBMS) Objectivity/DB [Objy] vorgesehen. Für die Implementierung der Schnittstellen und Programme wurden die Sprachen Java und C++ gewählt. Unterstützung für Fortran-Software in H1 ist ebenfalls geplant, jedoch soll zunehmend C++ bevorzugt werden. Diese Tendenz zeigt sich auch in der HEP-Gemeinde[1] [ArdShi99].

Das Ziel der Diplomarbeit ist die Entwicklung einer Schnittstelle zur Anbindung der Objectivity/DB-Datenbank an ein Java-Software-Paket zur Analyse von Ereignisdaten durch die H1-Kollaboration. Das Programmpaket *Java Analysis Studio* (JAS) wird für diese Ereignis-Analysen verwendet werden. Da JAS bisher keine Schnittstelle zu Objectivity/DB besitzt, soll diese in der Diplomarbeit entwickelt werden. Es werden die Beweggründe für die Entscheidung zu Objectivity/DB diskutiert. Im Vergleich dazu werden auch andere Datenbanksysteme besprochen.

Es wird nun kurz darauf eingegangen, wie Ereignis-Analysen bei H1 bzw. in der Teilchenphysik getätigt werden. Zur Analyse von Elementarteilchenexperimenten werden Ereignisse oft in Histogramme gefüllt. Dort können Physiker anhand von Spitzenwerten oder Verläufen Zusammenhänge erkennen oder nach Teilchen suchen. Zum Beispiel können Streuwinkel von Teilchen histogrammiert werden, um Tendenzen zu erkennen, oder es werden Energiewerte in das Histogramm gefüllt, wobei charakteristische Energiespitzen für die Suche nach Elementarteilchen untersucht werden. Histogramme sind auch für Kontrolle und Wartung des Detektors nützlich, z.B. können defekte Komponenten erkannt werden, falls gewisse Parameter nicht mit den Soll-Werten übereinstimmen. Aus diesen Gründen ist es wichtig, mit der anvisierten Software Histogramme bearbeiten zu können.

Hier eine Aufzählung von Anforderungen an Software zur Analyse von Ereignissen, die für H1-Forscher wichtig sind. Da die Datenbank vornehmlich als Ersatz für Datenqua-

[1]High Energy Physics (HEP)

litätsprüfungen[2] gedacht ist, sind die Anforderungen relativ fest vorgegeben. Die Möglichkeit, Selektionskriterien sukzessive spezialisieren zu können, trägt zur Anforderungsdefinition bei.

- 1D-/2D-Histogramme
  Ein 1D-Histogramm zeigt von einer Funktion die vorkommenden Werte auf der horizontalen Achse an, wobei die vertikale Achse die Anzahl der jeweiligen Werte darstellt. Typischerweise werden Gruppen von Werten gebildet, so daß in der Grafik Balken entstehen, durch die das Bild besser zu verstehen ist. In einem 2D-Histogramm wird eine Funktion mit zweidimensionalem Wertebereich dargestellt. Jeder der beiden Achsen wird eine Dimension zugeordnet. Die Aufsummierung an den Koordinaten des 2D-Histogrammes kann z.B. durch Kästchen oder kleine Kreise mit unterschiedlich großen Kantenlängen oder Radien ausgedrückt werden. Mit 2D-Histogrammen ist es also möglich, Analysen in Abhängigkeit von zwei Parametern zu erstellen, womit sich Tendenzen oder Abhängigkeiten in der Physik erkennen lassen.

- Übereinanderlegen von Plots
  Zum Vergleich ähnlicher Histogramme sollen diese in demselben Koordinatenkreuz übereinandergelegt werden können. Damit kann man den Unterschied von Histogrammverläufen verfolgen.

- Variable Balkenbreite (Rebinning)
  Zur Identifizierung von Spitzenwerten oder Tälern wird die Balkenbreite verändert. Ein Balken repräsentiert zusammengefaßte Werte. Falls das Histogramm z.B. in 50 Balken aufgeteilt werden soll, werden die aufgenommenen Werte entsprechend auf die Balken verteilt. Die Anzahl der Balken im Histogramm ist umgekehrt proportional zur Balkenbreite.

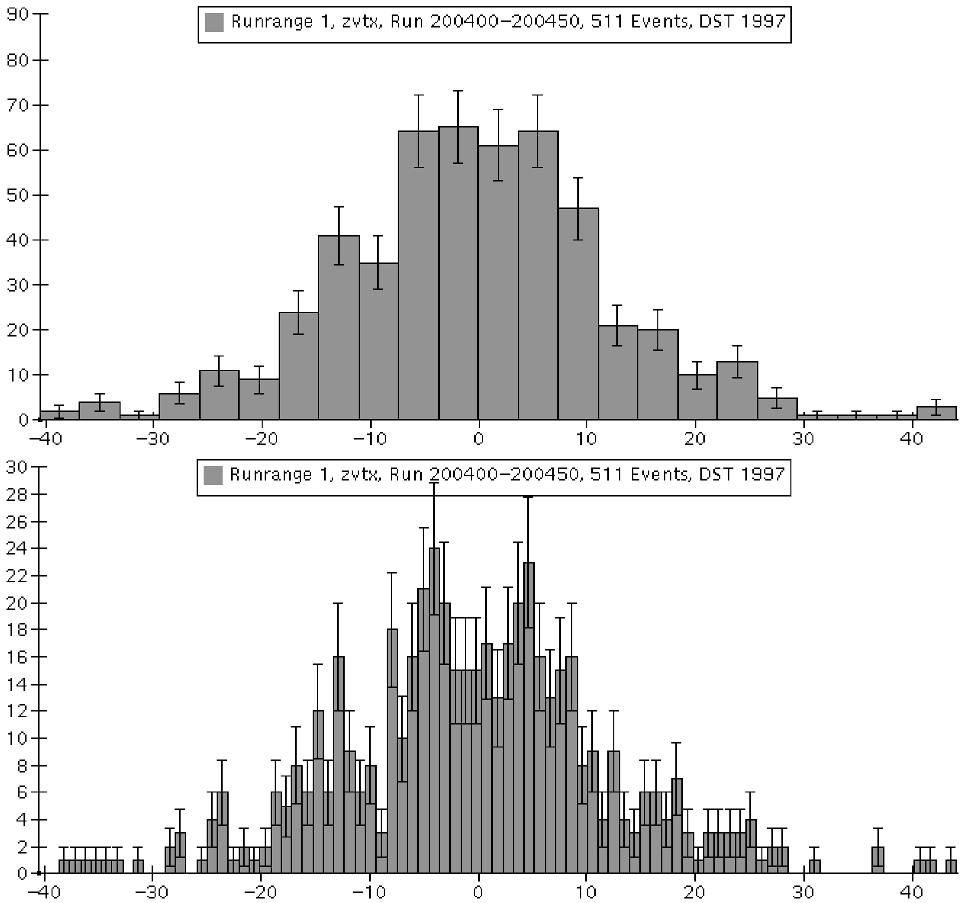
  Variable Balkenbreiten sind sehr wichtig für Physiker, da es gelegentlich vorkommt, daß sich Extrema mit ihren benachbarten Werten in denselben Balken aufheben oder abschwächen, so daß sie nicht mehr erkannt werden. Die Balkenbreite muß also verändert werden können, um interessante Stellen sichtbar machen zu können.

  In Abbildung 2.1 sind zwei Histogramme zu sehen, die unterschiedliche Balkenbreite-Parameter besitzen. Beiden liegt dieselbe Menge an Ereignissen zugrunde, aber die Spitzen bei dem Histogramm mit großer Balkenbreite sind nicht so deutlich wie bei dem anderen Histogramm.

- Angabe von Schnitt-Parametern (Predicate String)
  Manchmal ist es wünschenswert, nur einen Teil der Daten ins Histogramm einfließen zu lassen. Damit sollen interessierende Bereiche fokussiert oder hinderliche Werte ausgegrenzt werden. Dies wird mit der Angabe von Selektionskriterien erzielt.

- Anpassen von Funktionen (Fits)
  Manche Histogramme weisen Ähnlichkeiten mit kontinuierlichen Funktionen auf. Ein

[2] Data-Quality-Checks, Data-Quality-NTuples



Oben: breite (grobe) Bins, unten: schmale (feine) Bins

Abbildung 2.1: Zwei Histogramme mit unterschiedlichen Rebin-Parametern

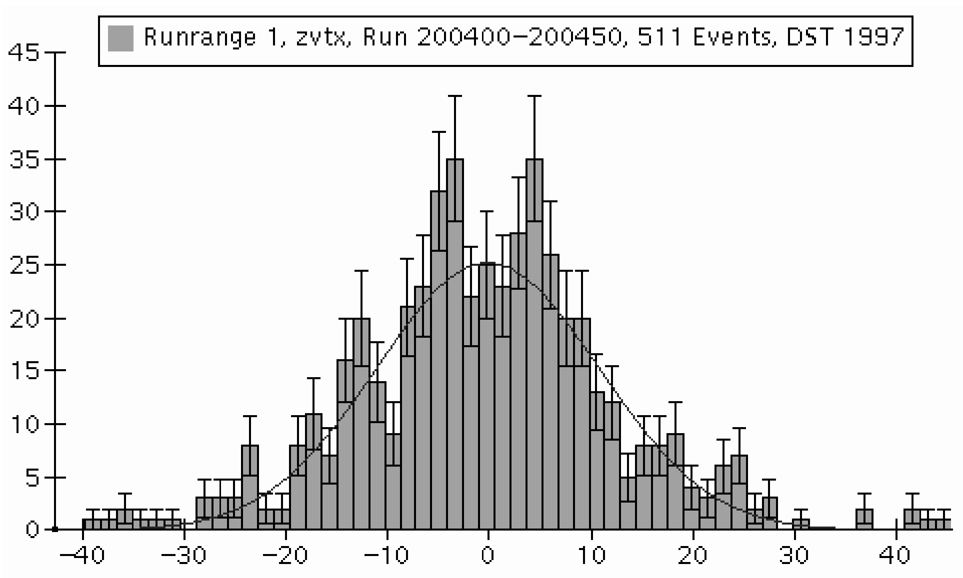


Abbildung 2.2: Ein Histogramm mit einer angepaßten Funktion

Fit stellt die Parameter einer solchen Funktion ein, so daß sie die Histogrammwerte möglichst genau nähert. Dieses Verfahren ist nützlich zum Auffinden von Abhängigkeiten oder Proportionalitäten zu bekannten Formeln in der Physik.

Abbildung 2.2 zeigt den Fit einer Gauß-Funktion an das Histogramm einer physikalischen Variable an.

Da die Objectivity/DB-Datenbank vollkommen objekt-orientiert aufgebaut ist, soll die Software zur Analyse von Histogrammen in einer objekt-orientierten Programmiersprache implementiert werden. In H1 bzw. im OOP-Projekt sind die Sprachen Java und C++ favorisiert, wodurch die Auswahl bereits eingeschränkt ist. Die Wahl fällt schließlich auf Java, und die Gründe hierfür sollen im folgenden verdeutlicht werden.

Ein wichtiges Argument für die flexible Gestaltung von Histogrammen sind die grafischen Schnittstellen von Java (AWT, Swing/JFC). Da es viele Möglichkeiten gibt (bzw. geben soll), eine Histogramm-Grafik zu bearbeiten, müssen die Grafikfähigkeiten der Entwicklungsumgebung sehr ausgeprägt sein. Bei OOP ist in Zukunft auch eine verkürzte Internet-Version der Analyse-Software geplant [Hadig98c]. Dies kann günstig mit einem Java-Applet realisiert werden, wodurch nahezu gleicher Quelltext benutzt werden kann. Beim Stichwort Internet steht natürlich auch die Frage der Datensicherheit im Vordergrund. Dieses Problem wird dadurch verschärft, daß es sich hierbei um eine Anwendung handelt, mit der auf möglicherweise wichtige Versuchsergebnisse zugegriffen werden kann. Auch hier genießt Java hohe Sicherheitsvorkehrungen, die ständig und schnell verbessert werden [Horstm99].

Neben Mechanismen zur Daten- und Zugriffssicherheit besitzt Java Techniken zum Datenschutz im Programm selbst. Dazu gehören viele Echtzeitprüfungen, z.B. wird bei jedem Zugriff auf ein Objekt dieses gegen Null geprüft. Dadurch sollen Fehler frühzeitig erkannt werden, die bei anderen Sprachen erst später entdeckt werden. Viele Vorteile von Java werden allerdings durch die Interpretierung des Java-Byte-Codes erkauft, wodurch die Ausführungsgeschwindigkeit im Vergleich zu Sprachen mit Übersetzern in Maschinen-Code langsamer ist. Es gibt aber auch JIT-Compiler (Just In Time), die Java-Byte-Code übersetzen kann, so daß das Programm anschließend aus maschinenabhängigem Code besteht, der von der CPU direkt ausgeführt werden kann.

Der vom Java-Übersetzer erzeugte Byte-Code ist auf allen Rechnersystemen lauffähig, auf denen sich eine Java-Umgebung befindet. Durch den Interpreter (Java Virtual Machine) wird der Code zur Laufzeit in Maschinencode umgesetzt und ausgeführt. Dieser Vorteil ist für H1 sehr wichtig, da dort eine heterogene EDV-Infrastruktur herrscht (Sun, SGI, PC, Macintosh). Mit dieser Systemunabhängigkeit, die für Java essentiell ist, geht einher, daß der Entwickler sich nicht um betriebssystem-spezifische Anbindungen kümmern muß (z.B. Threads).

Viele der oben genannten Argumente sprechen für Java. Die ähnliche Alternative bei H1, die Analyse-Software in C++ zu entwickeln, ist daher benachteiligt. Der wesentliche Vorteil aber für C++ als Wahl für die Implementierung der Analyse-Software läge in der Ausführungsgeschwindigkeit. Es hat sich aber herausgestellt, daß die Histogrammierung in Java für den Benutzer noch akzeptabel ist, da bei der Bearbeitung der Grafiken keine rechenintensiven Arbeiten erledigt werden müssen. Eine andere Frage ist, wie sich C++ bzw. Java auf den Zugriff auf die Datenbank auswirken würden. Mehr dazu befindet sich in Kapitel 4.5 über Benchmarks.

Die Objectivity-Datenbank soll im Rahmen des OOP-Projektes bei H1 in den Datenfluß eingebunden werden. Bisher wurden die Ereignisse auf Band oder Festplatte gesichert. Geplant ist, einen wichtigen Teil des Bestandes sowie zukünftig anfallende Daten in die Datenbank aufzunehmen. Dabei soll der bisherige Datenfluß weiterhin so existieren wie bisher. Die Datenbank ersetzt also keine bestehenden Datenbewegungen, sondern wird zusätzlich zum bisherigen Betrieb gefahren. Dadurch soll in Verbindung mit Java Analysis Studio der bisherige Analyse-Entwicklungsprozeß beschleunigt werden.

2.2 Java Analysis Studio

Das Softwarepaket *Java Analysis Studio* [Johnson98] wurde ab 1998 bei SLAC[3] entwickelt. Gegen Beginn der Diplomarbeit wurde eine öffentliche Vorabversion[4] freigegeben. Es handelt sich bei JAS um eine eigenständige Java-Applikation. Dadurch können die im oberen Abschnitt genannten Vorteile durch Java genutzt werden. JAS kann Histogramme erzeugen und darstellen, und es besitzt sehr viele Möglichkeiten, diese zu bearbeiten. Hier eine kurze Aufzählung der Fähigkeiten:

- Übereinanderlegen von Histogrammen
  Histogramme können in verschiedenen Farben und in gewünschter Reihenfolge übereinander dargestellt werden.

- Gitteranordnung mehrerer Histogramme möglich
  Mehrere Histogramme können in Raster-Anordnungen (z.B. 2×2 oder 2×3) nebeneinander betrachtet werden.

- Interaktive Manipulation der Achsen
  Die Achsen können in Echtzeit mit der Maus verändert werden (WYSIWYG). Je nachdem, an welcher Stelle die Achse mit der Maus angefaßt wird, kann man sie in beide Richtungen verschieben, stauchen oder strecken.

- Echtzeit-Rebinning
  Die Balkenbreite eines Histogrammes kann per Rollbalken verändert werden. Dies ist sehr nützlich für H1 (siehe Abb. 2.1).

- Grafikobjekte in Position und Größe frei veränderlich
  Histogramm, Titel und Legende sind grafische Objekte, die jederzeit verändert werden können.

- Anpassen von Funktionen (Fits), Berechnung von χ^2, Fehlerberechnung
  Zu Histogrammen können Funktionen verschiedener Art (Gerade, Polynom, Spline,

[3] Stanford Linear Accelerator Center
[4] Version 1.0beta von August 1998

Gauß) angelegt und angepaßt werden (siehe Abb. 2.2). JAS ist in der Lage, aus den Funktionen die χ^2-Verteilung zu berechnen.

Es besteht außerdem die Möglichkeit, eigene Fits in Form von Java-Klassen zu definieren. Die Anzeige der Fehler der Histogrammbalken ist voreinstellungsgemäß aktiviert.

- Datenformate PAW, HIPPO, StdHep, Flat-Text werden unterstützt
  Z.B. wird das PAW-Format bei H1 oft verwendet.

- Datenzugriff sowohl lokal als auch über das Netzwerk möglich
  Die Daten können sich lokal auf dem Rechner befinden. Auf anderen Rechnern kann jedoch auch ein Java-Data-Server installiert werden, der für JAS-Klienten Daten auf Anfrage zur Verfügung stellt.

- Einbindung eigener Module und Datenformate möglich
  Daten werden für JAS durch Data-Interface-Modules (DIMs) zur Verfügung gestellt. Jedes DIM liefert die Daten in einheitlicher Form an JAS. Es ist jedem DIM selbst überlassen, aus welcher Quelle es seine Daten bekommt. Dies können z.B. Meßstationen, lokale Dateien, Bandbestände oder auch Datenbanken sein.

- 2D-Histogramme in naher Zukunft geplant [Johnson99]
  Diese Art von Histogrammen soll in der kommenden Version von JAS möglich sein.

Es ist zu erkennen, daß bereits viele der in Abschnitt 2.1 genannten Anforderungen der H1-Mitarbeiter von JAS erfüllt werden. Ein sehr wichtiges Merkmal ist die Möglichkeit zur Erstellung eigener DIMs[5]. Mitgeliefert sind bereits DIMs für PAW-Dateien (N-Tuples), HIPPO, StdHep, Textdateien und Testdaten. Das StdHep-Format[6] wurde am Fermi National Accelerator Laboratory speziell für Monte-Carlo-Generatoren entwickelt. Monte-Carlo-Software dient der Simulation von Ereignissen in der Elementarteilchenphysik. Textdateien sollten nur einen Zahlenwert pro Zeile enthalten. Das Testdaten-DIM erzeugt Zufallszahlen mit reellen und ganzzahligen Zahlen, sowie andere fiktive Testwerte für andere Datenformate, die von JAS unterstützt werden, z.B. Zeichenketten oder Datumsangaben. Das PAW-DIM lädt Dateien im NTuple-Dateiformat ein. PAW wurde bei CERN[7] entwickelt und ist bei Teilchenphysik-Instituten sehr verbreitet.

Ein DIM, welches auf Objectivity/DB zugreift, wird in dieser Diplomarbeit erstellt. Es wird auch eine Benutzerschnittstelle haben, durch die die Forscher H1-spezifische Parameter festlegen können.

Die Fähigkeiten von JAS sind sehr vielfältig, so daß die Überlegung hinfällig wird, die Analyse-Software für H1 selbst zu entwickeln. In JAS sind bereits viele benötigte Eigenschaften enthalten, die man sich zunutze machen kann, anstatt sie im eigenen Projekt neu zu implementieren. Durch die Verwendung von JAS werden daher viele Manntage an Design

[5]Data Interface Module (DIM). Dieser Begriff wurde vom Projektteam von JAS in der Dokumentation eingeführt. Die Abkürzung DIM wird in der Diplomarbeit des öfteren benutzt werden.
[6]Genauere Informationen bei `http://fnpspa.fnal.gov/stdhep.html`
[7]CERN - European Laboratory for Particle Physics, www.cern.ch

und Entwicklung eingespart. So ist früh die Entscheidung gefallen, JAS als Analyse-Software zu wählen und hierfür eine Schnittstelle (ein DIM) zu Objectivity/DB zu entwickeln.

In verschiedenen Projekten und Experimenten bei CERN wird mittlerweile Java Analysis Studio eingeplant. Z.B. wird im Rahmen des Experimentes NA48 überlegt, JAS als Java-Frontend für Objectivity/DB zu benutzen [Hoschek98]. Im CMS-Experiment wurde bereits eine Schnittstelle zur Datenbank entwickelt (siehe Abschnitt 5.4.1), und es ist geplant, ein existierendes Java-Softwarepaket zur 3D-Darstellung von Events im Detektor mit JAS zu verbinden [Bunn98]. Insofern könnte ein Erfahrungsaustausch von H1 und CERN bezüglich JAS von Nutzen sein.

Kapitel 3

Datenfluß und Ereignisse

Im H1-Detektor werden durch die Kollision von Elementarteilchen Ereignisse ausgelöst. Die Ereignisdaten erfahren von der Entstehung bis hin zur Endverarbeitung durch das Programm eines Physikers einige Veränderungen, die in diesem Kapitel hinsichtlich Datengröße, Datendistribution und gemeinsamer Nutzung untersucht werden. Nachdem der Bearbeitungsablauf eines Ereignisses und dessen Daten dargestellt werden, sollen Parallelen zur Data-Warehouse-Architektur gezogen werden. Überlegungen zur Effizienzverbesserung werden angedacht und schließlich die eigene bzw. geplante Lösung vorgestellt.

3.1 Datennahme bei H1

3.1.1 Ereignisse aus Trigger Level 1–5

Wie bereits im Einleitungskapitel erwähnt, existieren fünf Trigger zur Klassifizierung und Verarbeitung eines Ereignisses (Abb. 1.2).

Nachdem ein Ereignis Level 1 passiert hat (L1K), werden die Puffer der Subdetektoren gestoppt. Hier beträgt die Größe der Daten insgesamt etwa 3 MB [Sefkow94].

Parallel zu Level 2 wird der Auslesevorgang der Datenpuffer initiiert. Nach einem L2K-Signal und weiteren Ausleseprozessen — Level 3 ist nicht aktiv — können die Datenpuffer gestartet werden und neue Ereignisse entgegennehmen. Das gewonnene Ereignis befindet sich in MEBs[1], die mit der zentralen Datennahme[2] verbunden sind. Dort werden die Ereignisdaten gesammelt und zu einem zusammenhängenden Stück vereinigt. Durch Kompression und andere Mittel wird die Ereignisdatengröße auf 50–100kB reduziert. Es handelt sich nun um ein Roh-Ereignis[3]. Unter Kompression ist hierbei nicht die Verdichtung durch Komprimierungsalgorithmen zu verstehen, sondern die Reduzierung um irrelevante Daten, z.B. werden bei der Rauschunterdrückung Kanäle ohne Signal entfernt. Level 4 benutzt das Softwarepaket H1REC [H1REC99] und nimmt bereits eine vorläufige Ereignisrekonstruktion vor. Dadurch werden ca. 80% der Ereignisse gefiltert und nicht zur Speicherung auf Band zugelassen [Itterb97].

[1] Multi-Event-Buffers
[2] Central Data Acquisition, CDAQ
[3] Raw Event

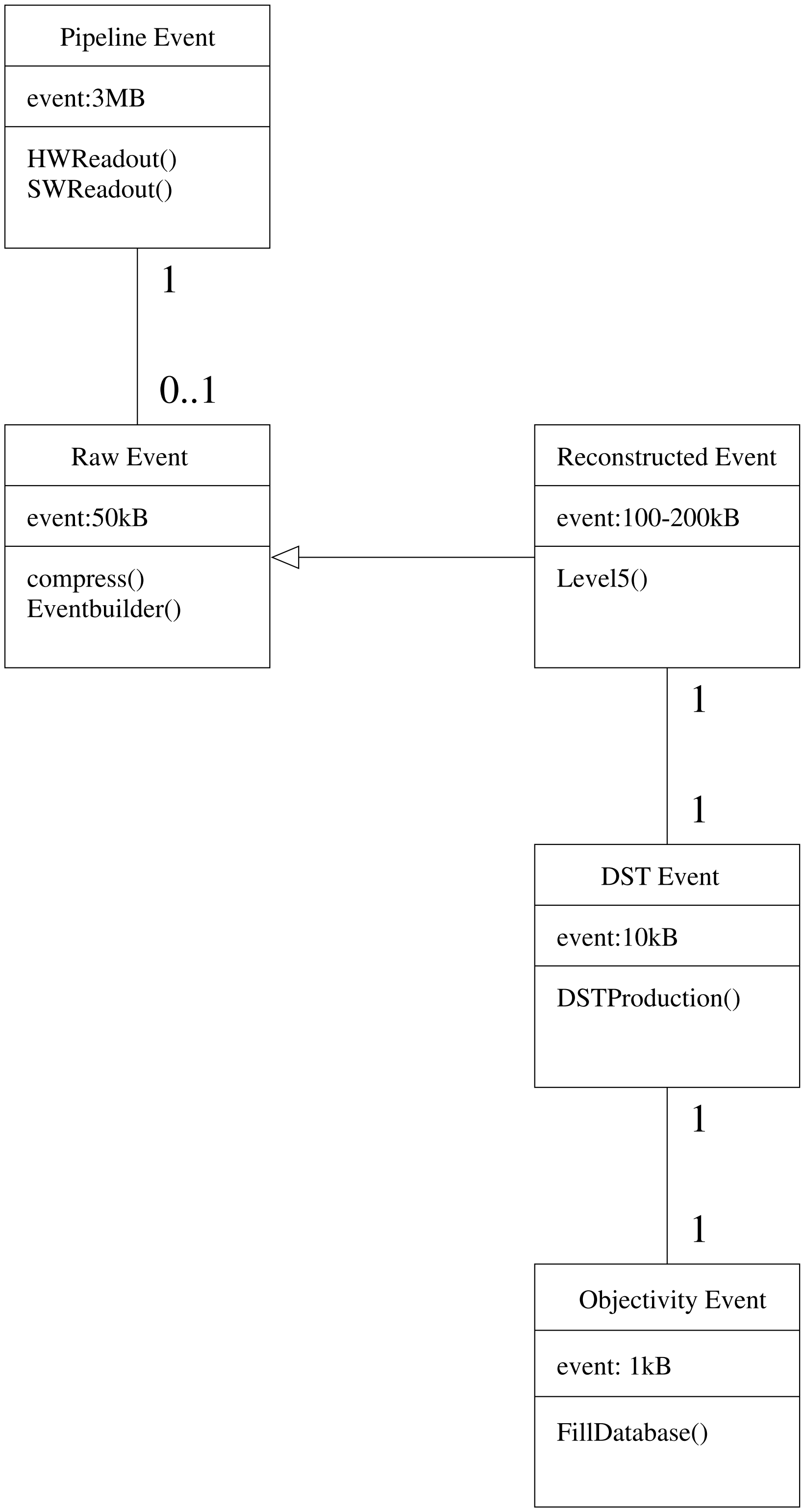


Abbildung 3.1: Event-Klassifizierung

Sobald die rohen Ereignisdaten auf Band gespeichert worden sind, werden sie von Level 5 dort ausgelesen und rekonstruiert. Eine Rekonstruktion bedeutet, daß Algorithmen aus dem bisherigen, physikalisch-technischen Inhalt der Ereignisdaten weitere Werte berechnen, die für die individuellen Berechnungen der Physiker wesentlich besser geeignet sind. Am Detektor werden im Laufe der Jahre ständig Hardwarekomponenten ausgetauscht, neu positioniert oder repariert, wodurch gewisse Parameter sich ändern (und sich normalerweise verbessern). Der Rekonstruktionsvorgang benutzt diese neuen Parameter und liefert dann im allgemeinen qualitativ bessere Ergebnisse. Rekonstruierte Ereignisdaten beinhalten auch die ursprünglichen Rohdaten, so daß ein Ereignis nun insgesamt 100–200kB Speicher benötigt. In Abb. 3.1 ist daher das rekonstruiertes Ereignis eine Spezifikation eines Roh-Ereignisses (Raw Event).

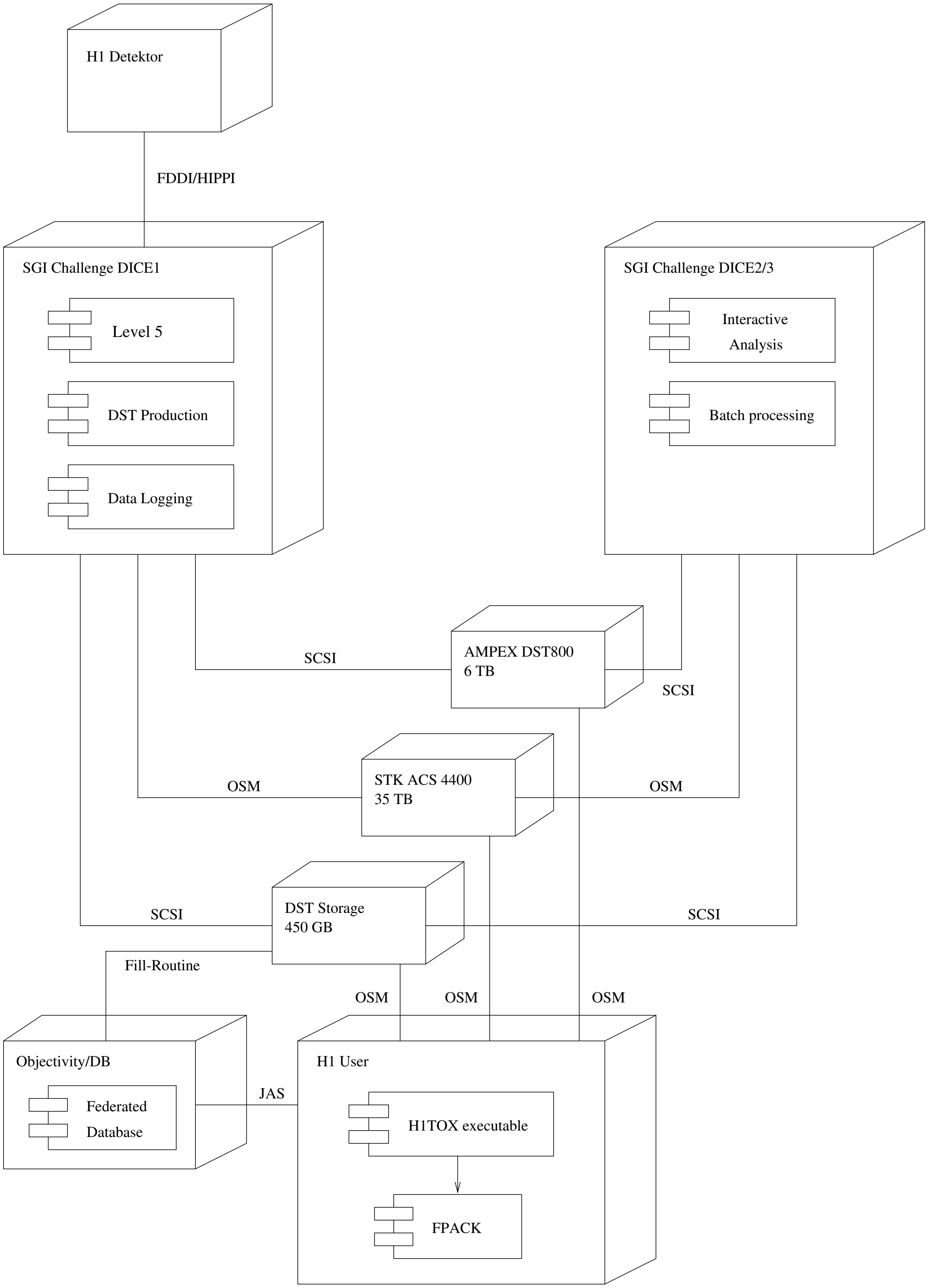
Das Ereignis wird durch Level 5 auf POTs geschrieben. Da im Normalfall nur ein Bruchteil der Ereignisdaten, wie sie auf POTs zu finden sind, von der Kollaboration zur Analyse verwendet wird, werden diese Daten durch einen Produktionsjob auf einen essentiellen Anteil reduziert. Dieser Teil wird wegen des schnellen Zugriffs auf Festplatten abgespeichert. Die resultierenden Dateien sind als DST[4] bekannt. Ein Teil davon wird auf Band gesichert, während der andere, häufiger genutzte Teil auf Festplatten abgelegt wird. Ein DST-Ereignis belegt etwa 10kB Speicher [Collab99] und ist genau einem rekonstruierten Roh-Ereignis auf POT zugeordnet (Abb. 3.1). Die DST, POT- und RAW-Bestände werden für etwa 4–5 Jahre behalten. Die Daten werden danach teilweise vom Bandspeicher entfernt und an einem anderen Ort ausgelagert. Mit welchen Daten dies geschieht, hängt hauptsächlich von der allgemeinen Nachfrage bei H1 ab.

Während der Datennahmezeiten von 1999 werden eingehende Ereignisse zunächst in SDST-Dateien[5] angelegt. Sobald 30 SDSTs beisammen sind, werden diese zu einer CDST-Datei[6] zusammengefaßt. SDST-Dateien sind etwa 14MB groß, CDST-Dateien liegen bei ca. 400MB. Nach aktuellem Stand ist der Grund für die SDST-Staffelung die instabile Software, durch die die SDST-Dateien beschädigt werden.

Die Objectivity-Datenbank, die bei H1 in Zukunft an diesem Datenfluß teilnehmen soll, speichert Ereignisse ab, die aus DST-Beständen stammen. Geplant ist, die Datenbank von einem Job füllen zu lassen, der DST-Daten einliest und nur einen Bruchteil eines DST-Ereignisses in die Datenbank schreibt. Dies soll mit Abbildung 3.1 dargestellt werden. Mehr über die Größe eines Objectivity-Ereignisses ist in Abschnitt 4.4 zu finden.

Sämtliche Datenflüsse außerhalb der in Kapitel 1 erklärte Totzeit werden durch den Oberbegriff „Offline-Betrieb“ (kurz: Offline) beschrieben. Dazu gehören die CDAQ, Trigger Level 4 und 5, sowie Protokollierung (Logging) und Beobachtung bzw. Überwachung. In Abbildung 3.2 ist der Offline-Betrieb grob skizziert. Die geplante Position der Objectivity-Datenbank ist dort abgebildet: Sie wird mit Daten aus DST-Beständen gefüttert, und die H1-Anwender können mit Java Analysis Studio darauf zugreifen.

[4]Data Summary Tape
[5]Small DST
[6]Compressed DST



OSM : Open Storage Manager
JAS : Java Analysis Studio

Abbildung 3.2: Level 5 Rekonstruktion, Offline-Datenfluß

3.1.2 H1-Programme

Die H1-Kollaboration verfügt über ein breites Spektrum an Software, die fast ausschließlich von H1-Mitgliedern selbst entwickelt wurde. Einige Programmpakete sollen hier genannt werden, damit klar wird, mit welchem Aufwand auf die Ereignisdaten zugegriffen wird.

Alle Ereignisse (Roh-, rekonstruierte und DST) bestehen aus BOS-Bänken [Blobel94]. Diese Bänke gruppieren die Ereignisvariablen nach deren Bedeutungen oder Zugehörigkeit, z.B. nach Subdetektoren, Elektron-/Proton-Daten, Myonen, Jets, oder nach verschiedenen Berechnungsmethoden oder Analysen. Eine BOS-Bank ist eine flache Struktur von Variablen. Die aktuelle komplette Liste aller BOS-Bänke in H1 umfaßt mehr als 1400 verschiedene Bänke[7]. Roh-Ereignisdaten bestehen aus detektorspezifischen Bänken mit direkten Meßdaten sowie unmittelbar berechneten Variablen. Rekonstruierte Ereignisse (POT) besitzen neben den rohen Ereignisdaten Bänke mit vielen Ergebnissen aus Algorithmen, die die Rohdaten selbst verwenden. DST-Ereignisse enthalten nur die gebräuchlichsten Bänke — wer Daten benötigt, die dort nicht zu finden sind, muß auf POTs oder sogar Roh-Ereignisse zurückgreifen.

Um auf die Bänke zugreifen zu können, wurde zu Anfang der 90er die Schnittstelle FPACK [FPACK98] entwickelt. Sie regelt den standardisierten Zugriff auf Ereignis-Bänke. Programmpakete zur Klassifizierung oder Analyse von Ereignissen wie H1REC [H1REC99], H1PHAN [H1PHAN97] oder H1TOX [H1TOX97] benutzen FPACK (Abb. 3.2). Je nach Anwendungsgebiet und den Bedürfnissen der Forscher selektieren diese Programme bestimmte Ereignisse, die anschließend in Form von NTuple-Dateien in Programme zur Erzeugung von Histogrammen einfließen (z.B. PAW [PAW99]).

3.1.3 Setup-Informationen und Slow-Control

Neben den DST-Dateien und POT-Bändern existieren bei H1 zwei Datenbanken [Kleinw99]: Die eine ist für Detektor-Konfigurationen zuständig, und die andere soll im nachfolgenden Abschnitt behandelt werden. Die Datenbank operierte bis 1996/97 auf komplett selbstentwickeltem Programmcode[8], der leider das Konsistenzprinzip der Datenbanktheorie[9] nicht unterstützt hatte. Dieses war der Hauptgrund, ein kommerzielles Datenbanksystem einzuführen. Da H1 zu diesem Zeitpunkt bereits Oracle-Lizenzen gekauft hatte, wurden diese Datenbank benutzt. In ihr werden nun die Konfigurationen zu den Ereignisstaffeln[10] in Form von BOS-Bänken gesichert, die bei der Rekonstruktion durch Level 5 (und teilweise schon durch Level 4) angefordert und aktualisiert werden. Abgespeichert werden Trigger-Konfigurationen aus Level 1 und 2. Level 4 führt sowohl Schreibe- als auch Lesezugriffe auf der Datenbank aus, und Level 5 arbeitet ausschließlich mit lesenden Funktionen. Hierbei werden ca. 200–300 BOS-Bänke pro Run gelesen. Die Oracle-Datenbank hat einen jährlichen Datenzuwachs von ungefähr 300MB (±100MB) und ist bei aktuellem Stand auf 1.5–2GB

[7] Die Liste befindet sich auf der internen "Computing and Software"-Internetseite von H1 ("Tools"-Untermenü).
[8] Die Datenbank war unter dem Namen MDB, später NDB bekannt.
[9] Objekte sind entweder gar nicht oder vollständig geschrieben, aber nicht teilweise.
[10] Runs

mit etwa 2 Millionen Einträgen (Bänken) angewachsen.

Die andere bei H1 geführte Datenbank wird mit sogenannten "Slow Control"-Daten gefüttert. Unter diesem Stichwort sind Detektor-Hardware-Parameter zu verstehen, die sich nur langsam ändern, zum Beispiel Luftdruck oder Hoch-/Niederspannung. Bei einer Änderung dieser Parameter wird ein "Slow"-Ereignis in die Datenbank geschrieben. Sie wurde bis 1996/97 als eine DB/2-Datenbank auf einem IBM-Mainframe-Computer geführt und ab dieser Zeit von Oracle abgelöst. Heutzutage wird aber beides noch benutzt — zumindest so lange, bis die IBM-Maschine außer Betrieb gesetzt wird (bereits in Planung). Die gesamte Größe der beiden Datenbänke summiert sich auf mehrere GB.

3.2 Data Warehouse

3.2.1 Einführung

Ein Data-Warehouse ist ein System von Technologien zur effizienten Organisation und Bereitstellung von Daten in OLAP-Umgebungen[11] [Jarke99]. Der gesamte Datenfluß in H1 kann durch ein Data-Warehouse dargestellt und mit den zugehörigen Methoden analysiert werden. Das Bedürfnis nach solchen Systemen ist allgemein in Zusammenhang mit der Zunahme an Komplexität und der Handhabung von großen Datenmengen gestiegen. Im folgenden wird der konzeptuelle Aufbau eines Data-Warehouses erklärt und anschließend auf die Datennahmeprozedur bei H1 abgebildet.

Das Data-Warehouse-Modell besteht aus vier Ebenen, die verschiedene logische Datenzustände von den technischen Datenquellen bis hin zum Endverbraucher darstellen (Abb. 3.3). Dies sind die Vorbereitungsebene (Preparation), die Integrationsebene, die Aggregationsebene und die Anpassungsebene (Customization). Die Vorbereitungsebene nimmt im Datenextraktionsschritt Daten von verschiedenen, möglicherweise heterogenen Datenquellen entgegen. Der anschließende Archivierungsschritt sorgt für eine eventuelle Synchronisierung der Datenströme. Bei der Datensäuberung (Data Cleaning) werden die rohen Daten auf ein Format vereinheitlicht, das von den nachfolgenden Schritten verarbeitet werden kann. Besonders bei heterogenen Datenquellen ist dies wichtig, da hier physisch verschiedene Datenstrukturen auf eine gemeinsame Ebene transformiert werden müssen. Die zweite Stufe ist die Integrationsschicht. Dort werden die anfallenden Daten gesammelt und im Operational Data Store (ODS) untergebracht. Dabei können neben Säuberungsaktionen auch Transformationen auf die Daten angewendet werden. Als nächstes folgt die Aggregationsschicht. Hier werden die Informationen aus dem ODS auf die Ansprüche des Unternehmens angepaßt. Dabei entstehen spezialisierte Datenkanäle, die im Corporate Data Warehouse (CDW) bereitgehalten werden.

In der obersten Schicht werden schließlich individuelle Anforderungen gestellt, die mit Hilfe des CDW erfüllt werden. Die Ergebnisse dieser Anforderungen sind sehr spezialisierte Teilmengen des gesamten Data-Warehouses. Diese werden als Data-Marts bezeichnet und sind auf die Bedürfnisse der einzelnen Benutzer oder Arbeitsgruppen angepaßt. Der Vorteil von Data-Marts liegt darin, daß Anfragen aufgrund des weitaus kleineren Datenvolumens

[11] On-line Analytic Processing (OLAP)

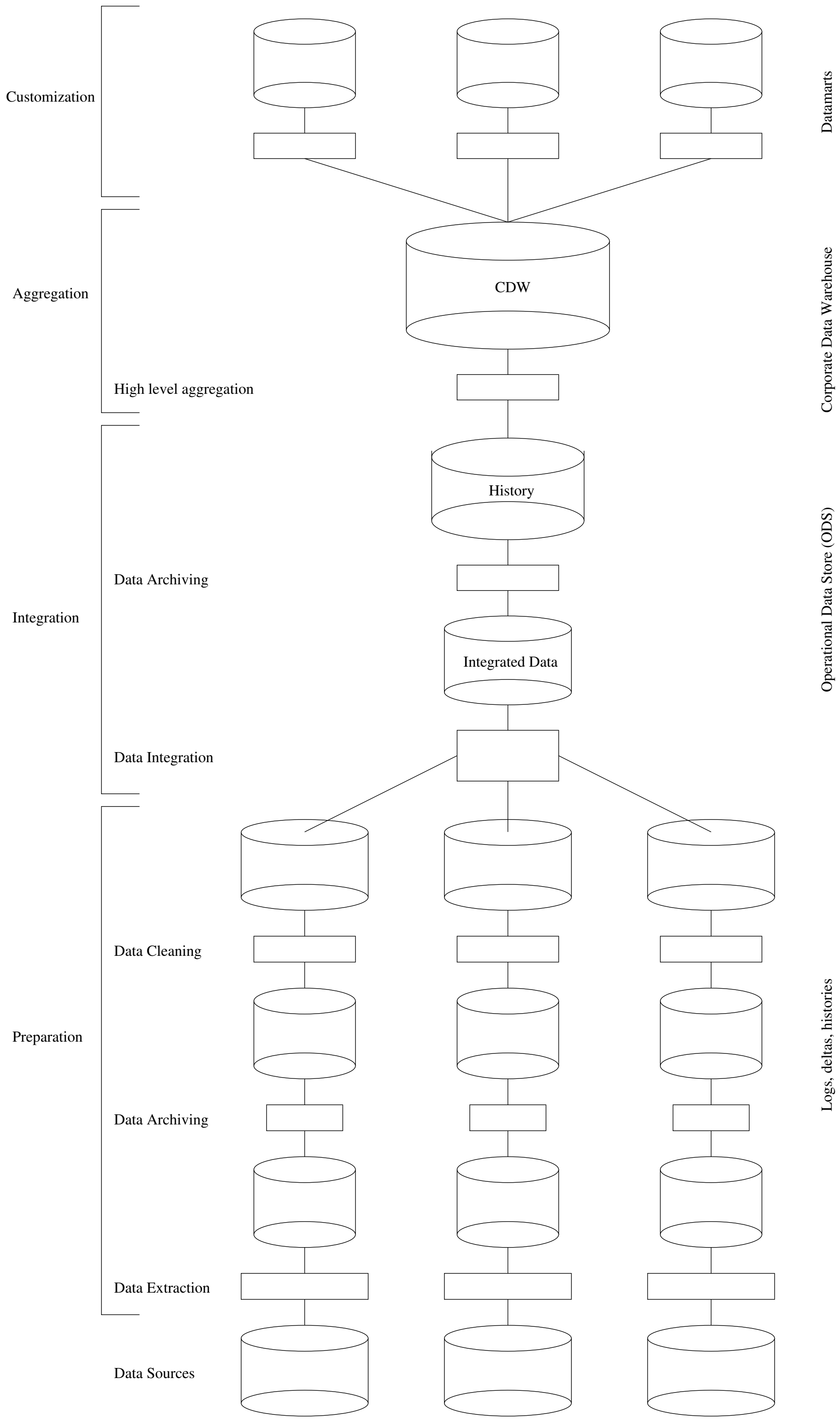


Abbildung 3.3: Data-Warehouse Modell

schneller bearbeitet werden können. Dies steht in Zusammenhang mit der Spezialisierung der Data-Marts, wodurch die angeforderten Daten näher beieinander liegen können als im CDW, was förderlich für die Zugriffsgeschwindigkeit ist.

Ein wichtiger Bestandteil von Data-Warehouses sind Metadaten. Diese Daten bestehen aus Informationen über die eigentlichen Daten des Data-Warehouses selbst. Metadaten sind typischerweise [Inmon96] [Jarke99]:

- Datenschemata oder Tabellendefinitionen für durchschnittliche Anwender und für Administratoren
- Informationen über die Datenquellen
- Transformation von Datenquellen ins DW
- Datenmodelle und deren Beziehung zum DW
- Aufzeichnungen (Logging, History) von Anfragen und deren Ergebnissen, Monitoring, Statistiken
- Allgemeine Routinen für Datenzugriff
- Sicherheitsinformationen und Richtlinien

3.2.2 Der H1-Datenfluß und Data-Warehousing

In Abbildung 3.4 ist der aktuelle Datenfluß bei H1 dargestellt. Die geplante Position der Objectivity-Datenbank ist dort eingearbeitet. Der Datenfluß wird auf der untersten Ebene des Data-Warehouse-Konstruktionsmodells durch den Speicherring HERA als Datenquelle begonnen, in dem das physikalische Experiment stattfindet.

Das Ereignis befindet sich noch unausgelesen in elektrischer Form in der Detektor-Apparatur. Im Extraktionsschritt werden die bei der Teilchenkollision entstandenen Produkte (andere Elementarteilchen, Wechselwirkungsteilchen, Quarks, Jets) z.B. von den Kalorimeterzellen oder von Trackern registriert. Sie hinterlassen dort Spuren, die anhand von Spannungen erkannt werden. Nach der Erfassung wird das Ereignis zunächst durch die Pipelines der Subdetektoren geschleust. Das Ergebnis wird dort durch Trigger Level 1 und 2 gefiltert und bereits parallel zu Level 2 ausgelesen. Die Informationen in den Pipelines bleiben maximal nur etwa 2.5ms erhalten, da dies der Zeitraum ist, der für den kompletten Auslesevorgang benötigt wird, sofern dieser nicht durch Level 1 oder Level 2 abgebrochen wird. Durch die Pipelines und die Prescale-Faktoren (siehe Kapitel 1.3) werden die Ereignisinformationen synchronisiert.

Die Detektoren besitzen unterschiedliche Ereigniserkennungshardware, so daß das Ereignis zunächst in eine digitale, elektronische Form vereinheitlicht werden muß, damit es durch die weiteren Datennahme-Stufen behandelt werden kann. Dieser Vorgang wird mit der Data-Warehouse-Prozedur “Data Cleaning” verbunden. Nach der Digitalisierung befindet sich das Ereignis in den Multi-Event-Puffern (MEB) der Subdetektorsysteme und steht hier erstmals in Form eines digitalen EDV-Speichers zur Verfügung[12], so daß es für die

[12]Zur Erinnerung: Das gesamte Ereignis ist an dieser Stelle ca. 3MB groß.

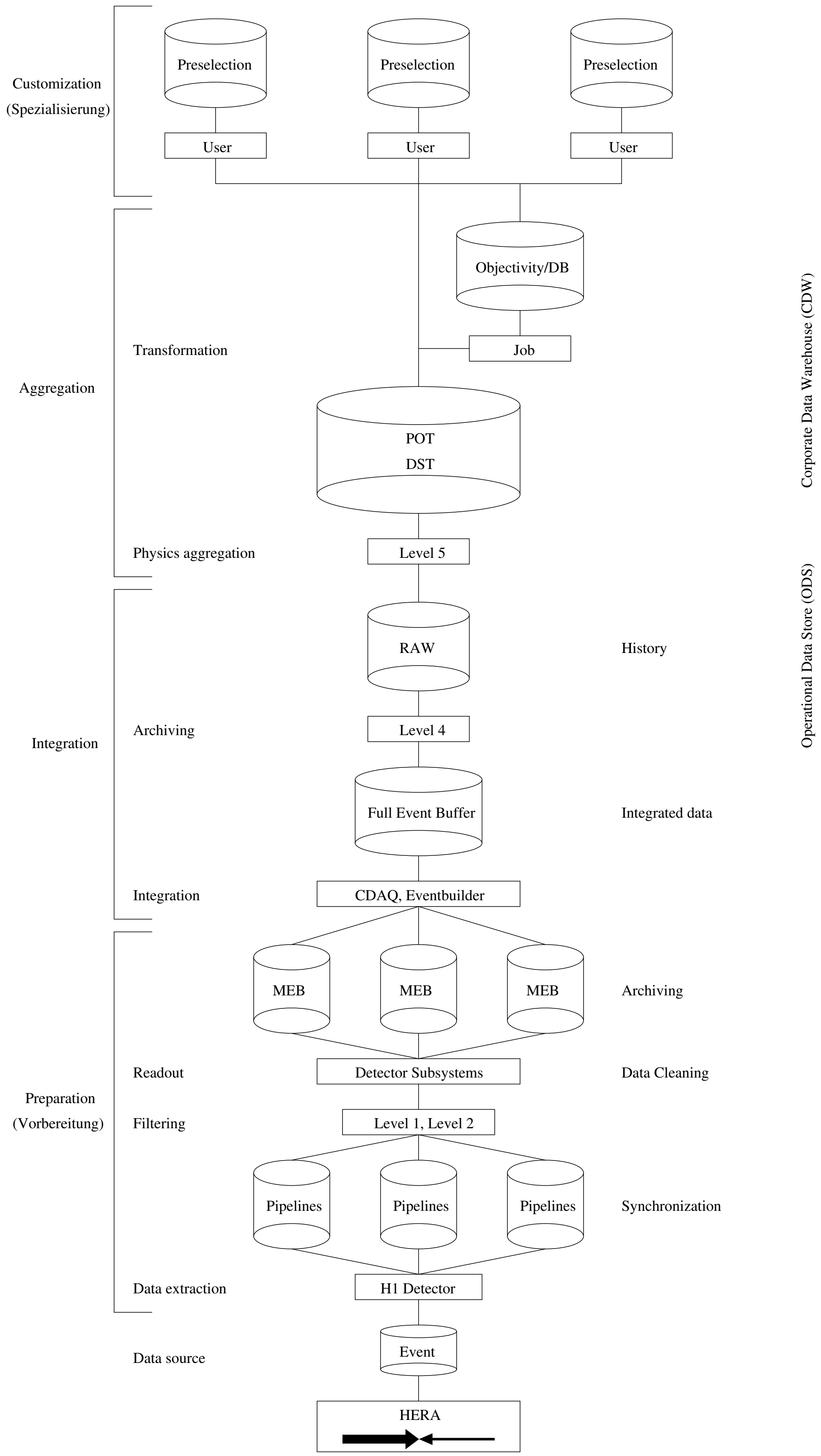


Abbildung 3.4: Data-Warehouse-Sicht bei H1

weitere Datenverarbeitung greifbar ist. Die Multi-Event-Puffer sind für mehrere Ereignisse ausgelegt, damit diese bei der Eingangsrate zur CDAQ von 40Hz für etwa 25ms zwischengespeichert werden können. Jeder einzelne Puffer enthält etwa 2MB Speicher. Diese kurzzeitige Aufbewahrung wird gleichzeitig zur Synchronisierung für den Eventbuilder genutzt. Daher ist hier der Archivierungsschritt aus dem Data-Warehouse-Modell zu finden.

Bis zu dieser Stelle existiert das Ereignis noch aufgeteilt in den Multi-Event-Puffern. Auf der Integrationsebene werden die Daten durch die CDAQ[13] aus den verschiedenen Subdetektorsystemen ausgelesen und vom Eventbuilder zu einem kompakten Ereignis zusammengefügt. Dieses ist nun ein Roh-Ereignis[14] und befindet sich im Full-Event-Puffer, welcher — ebenso wie die Multi-Event-Puffer — den Daten einen Aufbewahrungszeitraum von etwa 25ms gewährt (dies entspricht der 40Hz-Eingangsrate). Die Transformation der Detektor-Daten in ein Roh-Ereignis wurde bereits bei Abschnitt 3.1.1 beschrieben. Durch Level 4 wird eine letztere Filterung vorgenommen, und das gültige Ereignis wird mit einer Rate von 4–6Hz als Roh-Ereignis auf Band geschrieben, wo es über mehrere Jahre hinweg aufbewahrt wird. Der Bestand der Roh-Ereignisse wird auf den ODS-Speicher des Data-Warehouse-Modells abgebildet.

In der Aggregationsschicht werden die Ereignisse durch Level 5 mit intensiven Algorithmen für direkten Nutzen in der Physik aufbereitet. Diese Berechnungen entsprechen Rekonstruktionsvorgängen, die bereits in Abschnitt 3.1.1 erwähnt wurden. Die daraus gewonnenen POT- und DST-Daten entsprechen dem CDW-Speicher, woraus die Anwender ihre speziellen Auswahlen beziehen. POT- und DST-Bestände werden ca. 5–6 Jahre aufbewahrt und werden dann ausgelagert, da sie bis auf Ausnahmefälle nicht mehr genutzt werden, weil die neueren Daten von den Forschern bevorzugt werden. Dies gilt ebenso für die Roh-Ereignis-Daten.

Hinter der DST-Datenerzeugung wird die Objectivity-Datenbank angesetzt. Sie soll den Benutzern eine zusätzliche, bessere Möglichkeit geben, ihre Analysen anzufertigen. Die Datenbank wird damit Teil des Corporate Data Warehouse (CDW) werden. Sie soll als Hilfestellung zu den Analysen und für Datenqualitätsprüfungen dienen.

Die H1-Benutzer werden durch die Spezialisierungsschicht repräsentiert. Sie erstellen ihre eigenen Dateien (private DST-Selektionen und NTuples), deren Daten aus dem CDW oder aus dem ODS gewonnen wurden. Diese Dateien sind als Data-Marts anzusehen, da sie spezialisierte Informationen zu den Analysen der einzelnen Benutzer enthalten.

Für die Physiker reichen oft die Daten der DST-Bestände aus, um ihre Analysen zu erstellen. Manchen Anwendern genügt das nicht. Sie greifen daher auf POT-Ereignisse zu, die sehr viel mehr physikalische Daten als DST-Ereignisse enthalten. H1-Mitarbeiter, die mit der Hardware des Detektors beschäftigt sind (z.B. Administratoren, Experten, Schichtbetrieb), benötigen die rohen Ereignisdaten, um detektor-spezifische Aufgaben zu lösen. Die DST-, POT- und RAW-Datenbestände sind daher bei H1 jederzeit abrufbar. Die unterste Ebene, bis zu der H1-Anwender auf Ereignisdaten zugreifen können, ist der Level 4 Trig-

[13] Central Data Acquisition (CDAQ)

[14] Ein Roh-Ereignis (Raw-Event) ist ca. 50kB groß.

ger, wofür Software entwickelt werden kann. Diese wird von den Physik-Arbeitsgruppen[15] erstellt, damit Ereignisse nicht verloren gehen, die für die Gruppe interessant sind. Noch tiefer in die Datennahmeprozedur hinabzusteigen, ist nicht möglich und würde auch zu sehr die Verarbeitungsgeschwindigkeit beeinflussen. So dienen die Puffer, die auf dem Wege der Erstellung eines Roh-Ereignisses benutzt werden, nur als Zwischenspeicher und sind nicht für den Zugriff durch den Endbenutzer verfügbar.

3.2.3 Der Aktualisierungsfluß

Ein Bestandteil des Data-Warehouse-Konzepts ist die Aktualisierung (Refreshment) des Data-Warehouse. In [Jarke99] wird der DW-Lebenslauf in drei Phasen eingeteilt: Entwicklung, Initialisierung/Ladevorgang und Aktualisierung. Bei manchen DWs liegt die Betonung auf der Aktualisierungsphase. Im wissenschaftlichen — insbesondere physikalischen — Anwendungsgebiet weist diese Phase eine unterschiedliche Problematik im Vergleich zu anderen Gebieten auf. In der Physik stammen die abgespeicherten Daten aus Experimenten. Es liegt im Interesse der Forscher, diese Daten weder zu überschreiben noch zu ändern, sondern lediglich zu lesen. Sie werden dem aktuellen Bestand hinzugefügt, oder es werden überarbeitete Versionen existierender Daten erstellt und hinzugefügt, die von geänderten Korrekturfaktoren herrühren (Rekonstruktion).

Der Hauptanteil der Anfragen geschieht mit lesendem Zugriff, während Schreibzugriffe nicht oft und nicht gestreut auftreten, sondern relativ selten und großflächig. Die Aktualisierungsphase kann daher mit einer gewissen Abschwächung als eine in zeitlichen Abschnitten wiederkehrende Ladephase angesehen werden, während der bereits hinzugefügte Daten gleichzeitig angefordert werden können müssen. Auf das Problem des konkurrierenden Zugriffs auf die Objectivity-Datenbank wird in Kapitel 4 näher eingegangen werden. Gleichzeitige Lese- und Schreibzugriffe geschehen auch bei den DST-, POT- und RAW-Beständen. Diese sind nicht auf einer Datenbank, sondern als Dateien auf Bändern und Festplatten abgelegt. Die Anwender greifen direkt auf die Dateien zu. Neu hinzugekommene Daten werden in neue Dateien gefüllt, wobei die bisherigen Dateien unverändert bleiben. Danach wird die Aufnahme und die Verfügbarkeit der neuen Ereignisse bekanntgegeben, so daß darauf zugegriffen werden kann. Das Problem des konkurrierenden Zugriffs auf die Bestände in Dateiform ist daher — wenn auch auf triviale Weise — gelöst worden.

In Abbildung 3.5 ist der Daten-Aktualisierungsfluß bei H1 dargestellt. Der Aufbau des Bildes orientiert sich am Data-Warehouse-Modell aus Abbildung 3.3. Die vier Abschnitte von dort (Customization, Aggregation, Integration, Preparation) sind auch hier zu finden. Die Verbindungspfeile zwischen den einzelnen Teilabschnitten stehen für Ereignisse, deren Auftreten eine Änderungsfortpflanzung bewirkt. Damit sind etwa temporale, externe oder abschnittsabhängige Ereignisse gemeint. Diese sind nicht mit den Ereignissen aus den H1-Datenbeständen zu verwechseln. Das Ereignis, welches auf der untersten Ebene die Datenextraktion anstößt, tritt in feinster Granularität auf und ist ein temporales Ereignis: Die HERA-Bunch-Crossing-Rate von 10.4MHz ist konstant. Aufgrund der Kollisionswahr-

[15]Physics Working Group (PWG)

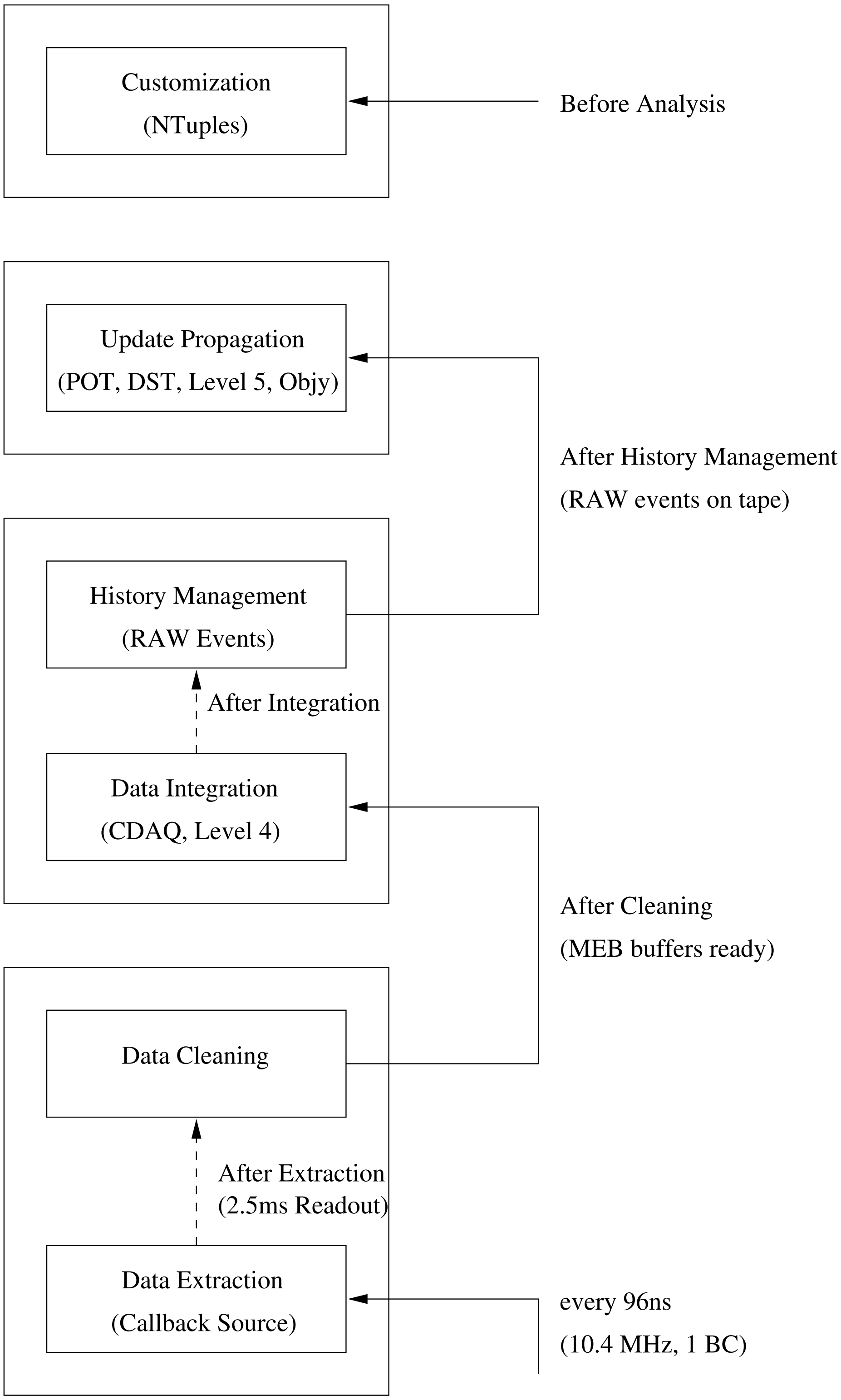


Abbildung 3.5: Arbeitsfluß der Daten-Aktualisierung bei H1

scheinlichkeit, der Totzeit durch Level 1–2 und der Prescale-Faktoren werden die Teilchenkollisionen jedoch nicht immer ausgewertet. Diese Gründe wurden in Abschnitt 1.3 erklärt.

In [Jarke99] wird der Aktualisierungsvorgang eines Data-Warehouses in drei Kategorien unterteilt: Die Aktualisierung geht von den Benutzern, vom Operational-Data-Store (ODS) oder von der Datenquelle aus. Für H1 gilt die letztere Kategorie. Nachdem die Ereignisdaten ausgelesen sind, durchlaufen sie sukzessive die Stufen Data-Cleaning (Level 1–2), Data-Integration (CDAQ, Level 4), History-Management (RAW-Daten) und Update-Propagation (POT-/DST-Aufbereitung, Level 5, Objectivity). Nach der Speicherung auf RAW-Bändern wird Level 5 angestoßen, wodurch das Ereignis zu POT- und DST-Daten aufbereitet wird. An dieser Stelle endet die Hintereinanderschaltung der Verarbeitungsstufen. Die Stufe "Customization" wird durch die Forscher von H1 dargestellt, die ihre eigenen Selektionen erzeugt haben. Es wurde bereits erwähnt, daß die Daten im ODS und im CDW nie verändert werden, sondern daß dort höchstens neue Daten hinzugefügt werden. Daher kann die Situation nicht vorkommen, daß die Daten, die durch eine Selektion des Anwenders ausgewählt wurden, im CDW geändert werden. Durch die Aufnahme von neuen Daten im ODS und CDW kann es jedoch geschehen, daß diese von größerem Interesse sind als die bisherigen Selektionen des Forschers. Dies kann z.B. durch aktuellere Parameter des Rekonstruktionsvorganges geschehen. Der Anwender muß seine in Eigenarbeit angelegten Dateien selbständig erneuern. Dieser Vorgang ist zeitlich unabhängig von der Aktualisierung des CDW. Daher besteht in Abb. 3.5 keine Verbindung zu den anderen Komponenten.

Die Objectivity-Datenbank ist Teil der Update-Propagation-Stufe. Ein Batch-Job soll dafür sorgen, sie mit Daten aus dem DST-Bestand zu füllen. Es ist zur Zeit noch nicht geklärt, ob dies in zeitlich regelmäßigen Abständen erledigt werden soll oder ob diese Prozedur nach der Erstellung neuer DST-Daten gestartet werden soll. Nach Möglichkeit wäre letzteres zu bevorzugen, um die Datenbank mit den zugehörigen DST-Daten konsistent zu halten.

3.3 Beispiele von physikalischen Analyse-Vorgängen

Hier sollen zwei Analyse-Szenarien demonstriert werden, die bei H1 real existieren. Dadurch wird der Umfang von Analyse-Prozessen beispielhaft verdeutlicht. Zusätzlich werden Probleme klar, für die im anschließenden Abschnitt Lösungen vorgestellt werden. Im ersten Anwendungsbeispiel wird ein von Grund auf neu entwickelter Analyse-Vorgang vorgestellt. Es wird die grobe Verfahrensweise von der Grobselektion bis hin zur verfeinerten Selektion erläutert. Dies ist ein eher allgemeines als konkretes Beispiel, welches die Prozedur an sich darlegen soll. Es beschreibt die Analyse-Entwicklung von Anfang an und ohne Mitwirkung von existierenden Forschungsarbeiten. Dagegen stützt sich das zweite Szenario auf bereits bestehende Arbeiten und Ergebnisse von anderen H1-Physikern.

3.3.1 Eine von Grund auf neue Analyse

Dieses Beispiel stellt eine physikalische Analyse vor, so wie sie von Grund auf durchgeführt werden kann. Dazu wurden Physiker befragt, die sowohl über ihre eigenen Analysen infor-

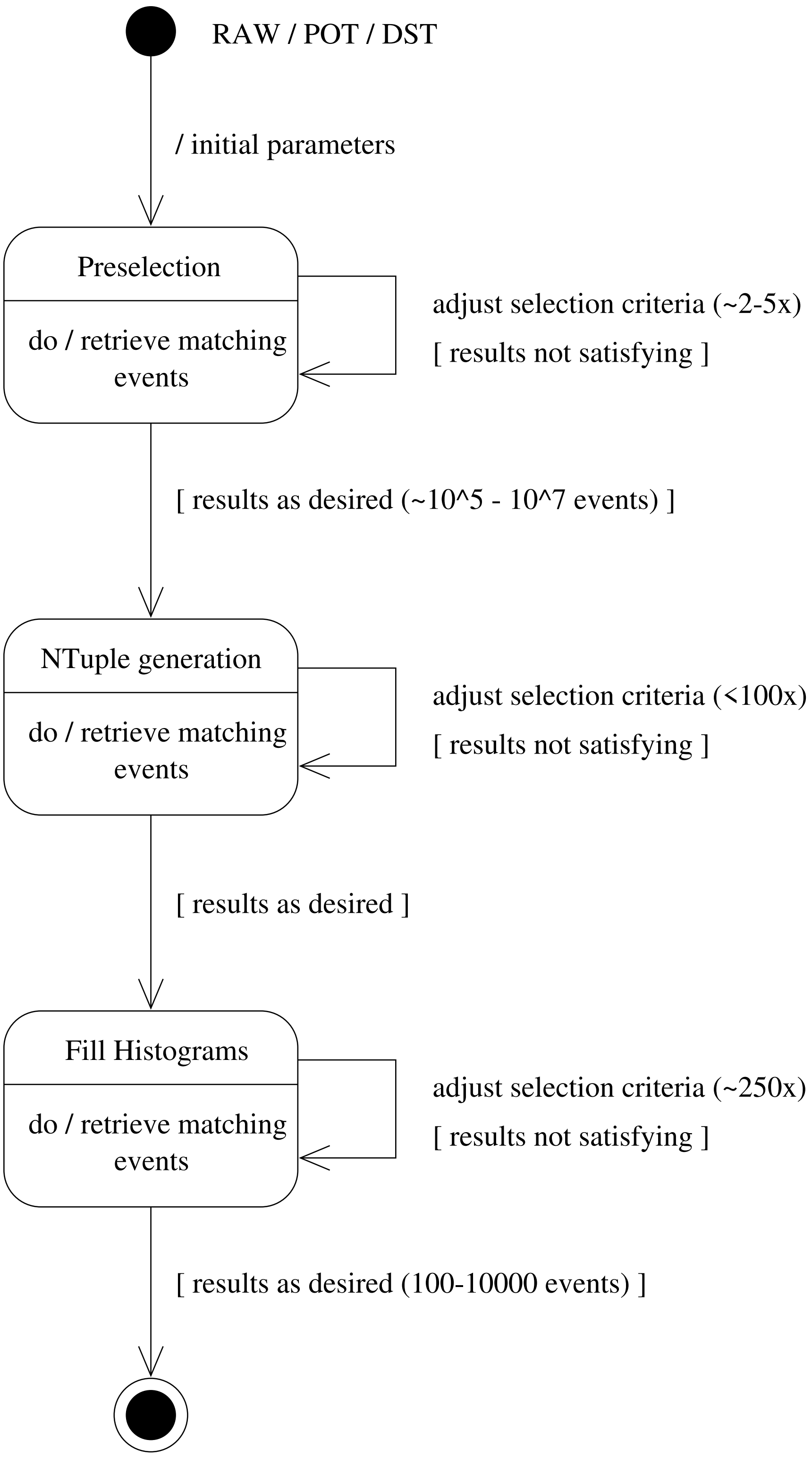


Abbildung 3.6: Schritte bei der physikalischen Analyse eines H1-Benutzers

miert haben, als auch allgemeinbezogene Angaben über den Analyseprozeß bei H1 gegeben haben. Dieser Prozeß ist in Abbildung 3.6 verdeutlicht und wird im folgenden erklärt.

Der Analyseprozeß wird mit der Erstellung einer Vorselektion und des dazugehörigen Datenbestandes begonnen. Hierfür werden „weiche“ Kriterien gewählt, durch die ein grober Filter für eine Vielzahl von Ereignissen hergestellt wird. Genaue Kriterien werden später festgelegt. In diesem Schritt sollten sie derart gewählt werden, daß ausreichend viele Ereignisse selektiert werden, und andererseits jedoch nicht zu viele, um Rechen- und Wartezeiten akzeptabel zu halten. Die Vorselektion wird je nach Spezialisierung auf die Ereignisse der DSTs oder POTs angewendet. Auf die RAW-Bestände kann ebenfalls zugegriffen werden, dies ist jedoch nur selten der Fall. Meistens ist der Informationsgehalt der DSTs oder POTs vollkommen ausreichend. Die Anzahl der durchlaufenen Ereignisse ist unterschiedlich: 10^6 ist ein Wert im Mittelbereich: Mengen in der Größenordnung von 10^5, aber auch 10^7 sind möglich. Dieser Schritt wird etwa 2–5mal durchgeführt. Die aus der Vorselektion resultierende Ausgabedatei dient dem Forscher als Aufsatzpunkt für die Verfeinerung seiner Auswahlkriterien. Das Dateiformat ist durch BOS-Bänke geprägt und ist dem Format der verwendeten Eingabe gleich (DST oder POT). Der Umfang der Ereignisauswahl ist sehr variabel: Größenordnungen um die 100 bis 10000 Ereignisse liegen vor.

Während der Selektion werden z.B. rechenintensive Elektron-Finder involviert, die individuell durch Anpassung der Implementierung spezialisiert werden können. Elektron-Finder sind spezielle Unterprogramme, mit deren Hilfe Ereignisse mit Elektronen ausfindig gemacht werden können. In Analysen, die mit Quarks und Jets zu tun haben, können Jet-Algorithmen angewendet werden. Diese sind ebenfalls sehr rechenzeitbedürftig und sollten nicht mit großen Ereignismengen verbunden werden. Jet-Algorithmen und Elektron-Finder sind von H1 entwickelte Softwarepakete, die stets auf die Bedürfnisse des Forschers zugeschnitten werden können.

Der nächste Schritt besteht darin, diese Eingabedatei in das NTuple-Format umzuwandeln. Dies ist notwendig, damit andere Softwarepakete und Algorithmen die Ereignisse behandeln können. Während der Konvertierung können weitere Schnitte oder Algorithmen vorgenommen werden. Dieser Schritt wird im Durchschnitt bis zu 100mal durchgeführt. Viele Analysen benutzen Simulatoren, die künstliche Ereignisse generieren. Diese werden für den Abgleich mit aus dem Experiment entstandenen Ereignissen benutzt. Die Simulations-Software ist unter dem Oberbegriff „Monte-Carlo“ bekannt. Bei H1 existieren etwa 10 Pakete, die von Physikern aus nationalen und internationalen Universitäten oder Instituten entwickelt wurden. Monte-Carlo-Benutzer müssen die bisherigen Analyse-Schritte noch ein zweites Mal durchwandern, und hierbei den Simulator einsetzen.

Schließlich dient das erstellte NTuple als Eingabe für den dritten Analyseprozedur-Schritt, bei dem nach endgültigen Schnitten die passierten Ereignisse in die Histogramme gefüllt werden. Dazu wird das Softwarepaket PAW eingesetzt, welches durch Makros gesteuert wird[16]. Dieser Schritt wird sehr oft mit geänderten Parametern wiederholt (200–250

[16]Diese Makros werden von PAW interpretiert und sind wegen ihres Suffixes (`.kumac`) auch als „Kumac“ bekannt. Die Langform von Kumac ist Kuip-Macro. Kuip ist der Name des Interpreters, der von PAW benutzt wird.

Iterationen). Dadurch werden die Selektionskriterien für die Schnitte sehr exakt justiert. Es werden nicht nur Schnitte, sondern auch Histogramm-Parameter eingestellt, z.B. Wertebereich der Achsen, Beschriftung, grafische Darstellungsvarianten (Balken, Punkte oder Kreuze) und weitere Attribute. In der Regel werden etwa zwischen 20 und 30 Histogramme mit verschiedenen Auftragungsvariablen durch ein Kumac-Makro gefüllt.

Der Grund, warum die Analyse in mehreren Stufen erfolgt, liegt in der Reduzierung der Wartezeiten, die mit den Rechenzeiten einher gehen. Sobald eine Anzahl an sicheren Kriterien feststeht, kann ein Schritt abgeschlossen werden, so daß im Hinblick auf die Entwicklung weiterer Schnitte die zu bearbeitende Ereignismenge auf ein Minimum gehalten wird [Holtm98]. Die in diesem Abschnitt angeführten Iterationshäufigkeiten der einzelnen Schritte und die daraus resultierenden Anzahlen der selektierten Ereignisse sind lediglich Größenordnungen. Diese Werte hängen stets von der zu tätigenden Analyse ab und sollen nur als Richtwerte für die quantitative Reduktion der Ergebnismengen dienen.

3.3.2 Wiederverwertung existierender Ergebnisse

In diesem Beispiel wird der Analyse-Vorgang eines Forschers dargestellt, der zunächst nach angeregten Elektronen sucht. Hierzu bedient er sich den Selektionskriterien einer Diplomarbeit, die bereits geschrieben wurde und in denen ebenfalls nach angeregten Elektronen gesucht wurde. Der Diplomand hatte wiederum auf eine bestehende Doktorarbeit zurückgegriffen. Es ist oft so in H1, daß es in der Vergangenheit Arbeiten gibt, die mit der eigenen Aufgabe Gemeinsamkeiten aufweisen. Der erste Schritt besteht darin, die Ergebnisse der Arbeit nachzuvollziehen, auf die sich gestützt wird. Sobald dies geschafft wurde, werden die neuen, eigenen Analyseschritte eingearbeitet. Der Forscher benutzt in diesem Beispiel die folgenden Selektionskriterien aus der anderen Diplomarbeit [Rottk98]:

1. Die folgenden Ereignisklassen werden selektiert:
   - SPSNMU — "special signatures, no muon"
   - QEDCOM — elastische QED-Compton-Ereignisse
   - NCHQSQ — "neutral current, high Q^2"
2. Die Spannungsversorgung für die Spurkammersysteme und für das Liquid-Argon-Kalorimeter muß gewährleistet sein. Die HV-Bits[17] für CJC1[18], CJC2 und das LAr müssen gesetzt sein.
3. Die über alle LAr-Kalorimeterzellen gebildete Summe $\sum(E-p_z)$ muß diese Bedingung erfüllen:
   $$38 \text{ GeV} < \sum(E - p_z) < 70 \text{ GeV}$$
4. Der rekonstruierte Vertex muß in z-Richtung zwischen -29 cm und 31 cm vom nominalen Vertex liegen.
   $$-29 \text{ cm} < z_{\text{vertex}} < 31 \text{ cm}$$

[17] High Voltage
[18] Central Jet Chamber

5. Es müssen mindestens zwei isolierte elektromagnetische Cluster im Ereignis gefunden werden, wobei die Energien der beiden Cluster jeweils größer als 2 GeV sein müssen:
$$E_e > 2 \text{ GeV} \qquad \text{und} \qquad E_\gamma > 2 \text{ GeV}$$

6. Die Summe der Energien der beiden elektromagnetischen Cluster muß größer als 20 GeV sein:
$$E_e + E_\gamma > 20 \text{ GeV}$$

7. Die gesamte im LAr-Kalorimeter summierte Energie muß abzüglich der Elektron- und Photonenergie kleiner als 5 GeV sein:
$$E_{tot} - E_e - E_\gamma < 5 \text{ GeV}$$

8. Wurden ein Elektron und ein Photon gefunden, so muß die Energie des nächsten rekonstruierten Clusters kleiner als 5 GeV sein:
$$E_{\text{next cluster}} < 5 \text{ GeV}$$

9. Die Lage von Elektron und Photon Cluster muß "back to back" sein:
$$|\Delta\varphi - 180°| < 20°$$

10. Der Polarwinkel für das Elektron muß in folgendem Bereich liegen:
$$10° < \theta_e < 150°$$

11. Der Polarwinkel für das Photon muß in folgendem Bereich liegen:
$$6° < \theta_\gamma < 150°$$

12. Die Anzahl der rekonstruierten Spuren im Detektor muß kleiner oder gleich 5 sein. Die Anzahl der rekonstruierten Spuren in einem Konus[19] um den γ-Cluster muß Null sein. Falls der Polarwinkel θ_γ kleiner als 35° ist, darf sie auch zwei oder vier sein.

13. Die invariante Masse des e-γ-Systems muß größer als 10 GeV sein:
$$m_{e,\gamma} > 10 \text{ GeV}$$

Als Standard-Elektronfinder wurde die H1-Software QESCAT benutzt. Diese Software besteht aus mehreren komplexen Unterprogrammen zur Auffindung von Elektronen und kann auch modifiziert werden. Es wurden alle H1-Ereignisse der Jahre 1995 und 1996 der Qualität "good" und "medium" miteinbezogen[20].

[19] Die exakte Beschreibung des Konus kann in der zitierten Diplomarbeit nachgelesen werden.
[20] Eine weitere, dritte Kategorie umfaßt Ereignisse schlechter Qualität.

Der Forscher versuchte zunächst mit Erfolg, das Ergebnis der Diplomarbeit nachzuvollziehen, auf die er sich stützt. Hier wurden die NTuple-Dateien des Autors der Diplomarbeit verwendet, es wurden also keine eigenen angelegt. Die Selektionskriterien wurden als Kumac-Makro für PAW implementiert. Die erhaltenen Ereignisse stimmten in etwa mit denen aus der Diplomarbeit überein. Der nächste Schritt bestand darin, das TOX-Executable[21], welches in der Diplomarbeit verwendet wurde, über die Ereignisse von 1995 und 1996 laufen zu lassen und damit eigene NTuples zu erzeugen. Ein eigenes PAW-Kumac mit den Schnitten aus der Diplomarbeit wurde auf das NTuple angewendet, und die Anzahl der selektierten Ergebnisse stimmte ungefähr überein. Nun wurden auf gleiche Weise mit dem TOX-Executable NTuples von 1997 erzeugt. Dazu wurden die Ereignisse der Vorselektion eines Kollegen benutzt, die jedoch dasselbe Ergebnis wie bei der Verwendung der DSTs liefern müßten. Es wurde ein eigenes TOX-Executable geschrieben. Im Abgleich der Daten von 1997 gegen die Anzahl der Ereignisse von 1995–1996 traten Probleme auf, denen nachgegangen werden mußte. Es wurden zuwenige Ereignisse in den 1997er Daten gefunden. Der Forscher hatte vermutet, daß diese durch Änderungen an den Track-Kriterien für das Elektron behoben werden könnten. Dazu wurde das TOX-Programm mit weicheren Kriterien versehen, und die NTuple-Dateien wurden neu erzeugt. Leider führte dies nicht zu einer Behebung des Problems. Die daraus resultierende Vermutung, daß nicht alle Photonen gefunden wurden, veranlaßte den Forscher zur Erstellung eines weiteren TOX-Programmes. Dabei wurde ein Unterprogramm eines Kollegen verwendet, welches unter anderem einen Jet-Algorithmus beinhaltet. Ein Jet-Algorithmus ist sehr kompliziert und benötigt daher relativ viel Rechenaufwand. Um nicht übermäßig viel Zeit zu investieren, mußte ein Kompromiß zwischen möglichst kleiner Eingabemenge und möglichst großer Ausbeute erzielt werden. Leider führte dies nicht zur Klärung des Problems. Nun wurde eine modifizierte Version des Elektron-Finders QESCAT eingesetzt, wodurch neue NTuples erzeugt wurden. Als dies auch nicht half, wurden die Ergebnisse von Kollegen eines französischen Institutes herangezogen. Eine Lösungsmöglichkeit wurde hier gefunden und befindet sich zur Zeit in Verifikation.

3.4 Möglichkeiten zur Leistungssteigerung

Der Datenfluß bei H1 funktioniert bisher auf der Hardware- und Offline-Seite gut. Damit sind die Hardware-Trigger und die Rekonstruktionsprozeduren gemeint. Das Data-Warehouse-Modell ließ sich unkompliziert darauf abbilden. Auf der Seite der Anwender kann jedoch einiges verbessert werden. Hierzu werden in diesem Abschnitt einige Vorschläge präsentiert.

In den physikalischen Analyse-Demonstrationen aus Abschnitt 3.3 wird von den Benutzern ein Vorselektionsbestand erzeugt, der als Basis für weitere Kriterien dient. Dieser Bestand ist durch Algorithmen und verschiedene weitere Selektionseigenschaften bereits spezialisiert. Die Kriterien sind jedoch noch relativ grob, da die genauen Selektionen erst

[21] Ein TOX-Executable ist ein ausführbares Programm, welches mit Hilfe der H1TOX-Software geschrieben wurde. Ein großes Programmgerüst in Fortran mit vielen Bibliotheken ist hier vorgegeben, so daß man lediglich seine Schnitte implementieren muß.

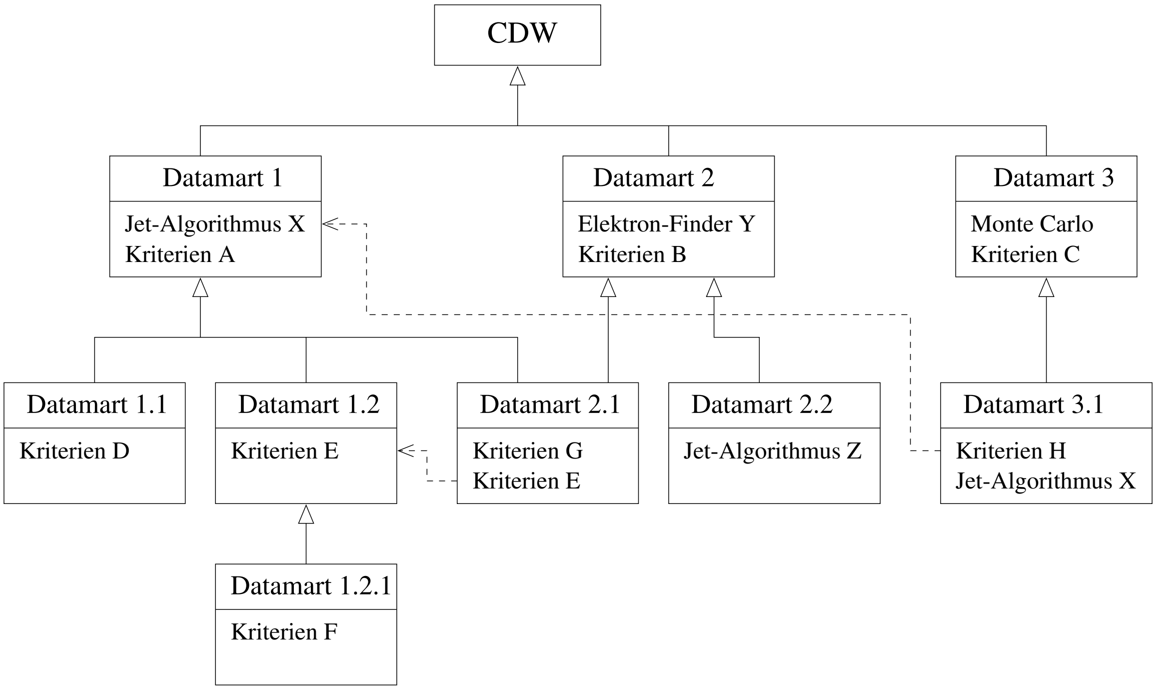


Abbildung 3.7: Data-Marts Hierarchie (Beispiel)

schrittweise ermittelt werden müssen. Die Erstellung des Vorselektionsbestandes ist sehr zeitaufwendig, da hier oft Bandstationen angesprochen werden müssen. Hinzu kommt, daß eine sehr große Anzahl an Ereignissen extrahiert wird. Die Benutzer müssen mehrere Tage warten, bis der Job fertig ist. Die H1-Kollaboration ist aus 39 internationalen Instituten zusammengesetzt; daher geschieht es oft, daß Physiker aus verschiedenen Instituten mit demselben oder einem ähnlichen Thema beschäftigt sind. Weiterhin sind in der Geschichte von H1 häufig gleichartige Themen auffindbar, die mit der aktuellen Arbeit des Forschers zu tun haben. Diese Voraussetzungen liefern eine gute Motivation für die organisierte gemeinsame Nutzung von Vorselektions-Datenbeständen. Dies entspricht dem Prinzip, welches den Data-Marts eines Data-Warehouses zugrundeliegt (siehe Abschnitt 3.2.1). Eine zentrale Anlaufstelle würde die öffentlich zur Verfügung stehenden Vorselektionsdateien verwalten und damit eine Übersicht schaffen. Die Benutzer würden dort neue Dateien anmelden, die anschließend für andere Physiker zur Einbeziehung in deren Analysen freigegeben werden würden. Die Objectivity-Datenbank wäre für diesen Zweck einsetzbar.

Die Data-Marts lassen sich durch weitere, geschickte Verfeinerungen der Selektionsparameter stärker spezialisieren. Die so erhaltene Auswahl kann wiederum als Vorlage für die Erstellung zusätzlicher, kleinerer und spezifischerer Data-Marts dienen. Durch die sukzessive Spezialisierung der Selektionskriterien wird eine Data-Mart-Hierarchie aufgebaut. Die Anwender suchen sich z.B. mit einer entsprechenden Oberfläche einen geeigneten Data-Mart, der als Basis für ihre weitere individuelle Analyse dient. Das Beispiel in Abbildung 3.7 zeigt eine mögliche Data-Mart-Hierarchie. Zwischen einigen Data-Marts existieren Abhängigkeiten: Ein Jet-Algorithmus aus einem anderen Data-Mart wird referenziert, und ebenso können auch auf Kriterien anderer Data-Marts verwiesen werden.

Da die Data-Marts selbst (die Vorselektionsdateien) nicht Bestandteil der Datenbank sind, würden lediglich die Verwaltungsdaten dort gesichert werden. Der Hauptbestandteil der Datenbank wäre wie bisher die Speicherung und Bereithaltung von Ereignissen, worauf in Kapitel 4 näher eingegangen wird. Die Verwaltungsdaten für die Vorselektionsdateien würden einen Teil der Metadaten der Datenbank darstellen. Sie würden vorschlagsweise folgende Elemente beinhalten:

- Standort der Datei auf DASD[22] oder Band
- Ausgangspunkt, Urbildmenge der Ereignisse: DST, POT oder RAW-Daten, oder auch andere Vorselektionsbestände
- Kriterien und Algorithmen zur Ereignisselektion, evtl. Aufnahme der Algorithmen mit Quelltext und ausführbarem Programm (z.B. modifizierte Elektron-Finder oder Jet-Algorithmen)
- Deskriptive Beschreibung der Kriterien und der Motivation für diese Vorselektion
- Logbuch über die Nachfrage dieser Vorselektion, Statistiken über Wiederverwertung
- Referenzen zu ähnlichen oder verwandten Selektionen
- Verbindungen zu Data-Marts, die auf dieser Selektion aufbauen
- Benachrichtigung von Benutzern, falls neue Assoziationen erstellt werden

Es kommt vor, daß Forscher Ereignisse nicht aus den DST-Beständen, sondern aus POT- oder RAW-Dateien anfordern wollen, da diese gewisse Ereignisattribute (BOS-Bänke) enthalten, die nicht in den DSTs vorhanden sind. Sie müssen sich dazu die Information holen, welche Bestände welche BOS-Bänke enthalten. Dies geschieht bisher durch die Befragung von Experten. In manchen Fällen ist es auch möglich, POT-Daten zumindest teilweise aus DST-Daten zu erstellen. Die Informationen über die bei H1 verwendeten BOS-Bänke und deren Auffindbarkeit in den DST-/POT-/RAW-Strukturen könnten durch Metadaten in der Datenbank abgespeichert werden. Eine entsprechende Applikation könnte Auskunft über den Aufbau der Strukturen geben.

Eine andere Verbesserungsmöglichkeit des Datenflusses bei H1 wäre die Speicherung von kompletten Analyse-Ergebnissen in der Datenbank. Der oben genannte Verbesserungsvorschlag zielt auf die Verwaltung und Wiederverwertung der Ausgangsselektion der Analysen ab, während der nun vorgestellte Lösungsansatz auf die Verwaltung und Speicherung des Endzustandes von Analysen eingeht. Ergebnisse, die in Form von Histogrammen in der Datenbank abgelegt sind, können von Forschern wiederverwendet werden, die an ähnlichen Themen arbeiten und versuchen, diese zuerst nachzuvollziehen. Genau dies war in einem der obigen Szenarien der Fall (siehe Abschnitt 3.3.2). Es wurde dort zunächst das Ergebnis einer früheren Arbeit praktisch wiederholt, um anschließend mit der eigenen Analyse darauf

[22]Direct Access Storage Device

aufzusetzen. Ein Histogrammeintrag in der Datenbank könnte etwa die folgenden Elemente enthalten:

- Alle Ereignisse, die ins Histogramm gefüllt wurden, oder Referenzen darauf (Assoziationen), falls sie sich in der Datenbank befinden
- Histogramm-Attribute, z.B. Achsenbeschriftung, Skalen, Achsenbereiche
- Vollständige Aufzählung der Selektionskriterien für dieses Histogramm, eventuell Unterteilung in Kriterien für die Vorselektion, für die NTuple-Erzeugung, für die Kumacs in PAW, und für die Selektion aus der Datenbank

Ein ähnliches Konzept wird in [Gupta97] vorgeschlagen. Dort wird davon ausgegangen, daß die Benutzer des Data-Warehouses häufig Zusammenfassungen von Daten anfordern. Daher können automatisch generierte Zusammenfassungen eingerichtet werden, so daß sie nicht bei jeder Anforderung neu angelegt werden müssen. In H1 könnte dies auf automatisch erstellte Selektionsbestände abgebildet werden, auf die die Benutzer vielfach zugreifen würden. Dies wäre für Monitore oder statistische Anzeigen nützlich, die in regelmäßigen Abständen diese Daten anfordern würden. Es wäre auch möglich, die Monitore dann zu aktualisieren, wenn neue Daten ins Data-Warehouse aufgenommen wurden. Ideen hierzu können in [StJa96] gefunden werden.

Die obigen Verbesserungsansätze tragen zur gemeinsamen Benutzung und Wiederverwendung von Ereignisdaten und Ergebnisdaten bei. Der folgende Vorschlag greift bei der Verarbeitung der Programmjobs während einer Analyseprozedur ein. In der Design-Phase eines Data-Warehouses sollten Überlegungen bezüglich der Granularität von Daten angestellt worden sein [Inmon96]. Hierunter ist das Detail der Daten zu verstehen. Bei hoher Granularität sind die Informationen sehr ausführlich vorhanden. Das hat den Vorteil, daß wirklich jede Anfrage erfüllt werden kann. Dies ist aber mit einem großen Datenvolumen verbunden, so daß Leistungseinbußen hingenommen werden müssen. Wenn aber die Daten wenig detailliert vorliegen und kompakte Informationen enthalten, so kann die Anfrage wegen der geringen Datenmenge relativ schnell durchgeführt werden. Die Kompaktheit drückt sich durch Zusammenfassungen oder partiellem Wegfallen von Informationen aus. Der Nachteil hierbei macht sich bemerkbar, sobald Informationen angefordert werden, die wegen des groben Details nicht mehr vorhanden sind. In [Inmon96] werden daher duale und einfache Granularitätsstufen vorgeschlagen ("dual or single levels of granularity"). Bei ersterem existieren die Daten sowohl im Detail als auch in grober Struktur. Einfache Granularität bedeutet lediglich ein einheitliches Datengerüst, bei dem den Anwendern keine Auswahlmöglichkeit zwischen verschiedenen Detailstufen gegeben ist.

Die bei H1 installierte Objectivity-Datenbank bildet zusammen mit den DST- und POT-Beständen den Corporate Data-Warehouse Datenspeicher, der bereits in Abschnitt 3.2.2 erläutert wurde. Bisher existieren Ereignisse in RAW-Beständen mit detektorspezifischem Gehalt, in POT-Bändern zusätzlich mit physikalischen und rekonstruierten Werten, und in DST-Dateien als zusammengefaßte Ereignisse. Damit existieren bereits drei verschiedene Granularitätsstufen. In Objectivity/DB sollen die DST-Ereignisse auf eine Teilmenge verkleinert werden, wodurch Analyseprozeduren und Datenqualitätsprüfungen effizienter gestaltet werden könnten. Nur die für die Forscher wichtigsten Informationen der DST-Ereignisse werden in die Datenbank übernommen, und die Ereignisgröße verringert sich auf etwa ein Zehntel eines DST-Ereignisses. Mit der Einführung der Datenbank wird also eine weitere Granularitätsstufe der Ereignisdaten geschaffen. Ausführliche Erklärungen über die Ereignisse in der Datenbank sind in Kapitel 4 zu lesen.

In [Kimball96] werden zukünftige Aspekte und Verbesserungsmöglichkeiten von Data-Warehouses angesprochen. Die Komprimierung von Daten ist bisher bei Datenbanken von wenig Interesse, weil dadurch die Transaktionen verlangsamt werden. Andererseits ist es durch Komprimierung möglich, mehrere Datensätze auf eine Datenbankseite (Page) zu speichern und somit den Zugriff zu beschleunigen. Dieser Gesichtspunkt des Einsatzes von Datenkompression ist allerdings auf das Leistungsverhalten der Datenbank bezogen und nicht auf die Effizienz des Datenflusses im Data-Warehouse.

Kapitel 4

Die Objectivity-Datenbank

4.1 Aufgaben und Fähigkeiten

Die H1-Physiker erhalten nach einer ersten, groben Analyse der Ereignisse vorläufige Ergebnisdateien, die durch Verfeinerung der Kriterien und Wiederholung der Analyse-Schritte verbessert werden. Aus jedem Zwischenergebis lassen sich weitere Maßnahmen zur Eingrenzung des Ziels schlußfolgern. Erst nach mehreren Schritten wird die finale Analysedatei erreicht, die präsentiert werden kann. Auf dem Wege dahin geht viel Zeit verloren, und Dateien müssen vielfach wiederholt gelesen werden. Durch den Einsatz einer Datenbank sollen die Schritte vereinfacht werden, die für eine endgültige Analyse nötig sind [Hadig99]. Des weiteren sollen Eingabedaten zentralisiert werden, die bisher in vielen Dateien an verschiedenen Stellen aufgehoben sind. Die Ereignisdaten sind somit umständlich zu finden. Da die Dateien durch Administratoren verwaltet werden, besteht stets ein Aufwand, diese zu verwalten, zu organisieren und geordnet zur Verfügung zu stellen. Dieser Aufwand kann durch den Einsatz der Datenbank verringert werden.

Weiterhin soll die Datenbank für Datenqualitätsprüfungen eingesetzt werden, die bisher auf herkömmlichem Wege gefahren werden, indem jeweils spezifische Daten von den einzelnen Arbeitsgruppen innerhalb Level 5 durch die erzeugten Datenqualitäts-NTuples verifiziert werden. Andere Qualitätsprüfungen werden bei Level 4 abgenommen, wodurch Detektorkonfigurationen regelmäßig durch Schichtmitarbeiter überprüft werden. In Level 5 finden nach dem ersten Rekonstruktionsvorgang Prüfungen durch Histogramme statt, die der Kalibrierung und der Verifikation von korrekten Bereinigungen dienen. Schließlich werden generelle, globale Qualitätsprüfungen in Level 5 vorgenommen, die von allen H1-Forschern eingesehen werden können. Viele dieser Prüfungen sollen durch die Datenbank abgelöst werden.

Es werden nun Datenzugriffseigenschaften vorgestellt, die der Teilchenphysik eigen sind. Auf die Ereignisse in Objectivity/DB wird von den Anwendern bei H1 ausschließlich lesend zugegriffen werden. Die Datenbank wird nur von ausgewählten Mitarbeitern (Administrator der Aufbereitung von DST-Dateien, Datenbankadministrator des OOP-Projektes) gefüllt werden. Wurde ein Objekt in die Datenbank geschrieben, wird es nie wieder aktualisiert, es sei denn, außerplanmäßige Korrekturen seien nötig. Neue Daten werden durch Batch-Jobs in großen Partitionen stets an das Ende des Datenbank-Bestandes hinzugefügt, sie werden also nicht zwischen bestehenden Daten eingefügt. Hinzukommende Daten sind gewöhnlicherweise später erzeugt worden als solche Daten, die bereits in der Datenbank enthalten sind. Dadurch ergibt sich eine chronologische Reihenfolge. Die Objekte in der Datenbank stehen in keiner Beziehung zueinander. Ihre Attribute werden ausschließlich zu statistischen Zwecken angefordert. Diese Eigenschaften sind charakteristisch für OLAP-Umgebungen [Hoschek99].

Die H1-Kollaboration stellt die folgenden Anforderungen an die Datenbank [Hadig98a]. Die Lösungen durch Objectivity/DB sind in den jeweiligen Punkten eingearbeitet:

- Gleichzeitiger Lese- und Schreibzugriff
  Mehrfache Lese-Operationen auf dasselbe Objekt sind bezüglich Datenkonsistenz unkritisch. Die Datenbank soll jedoch durch Administratoren gefüllt werden können, während Anwender darauf lesend zugreifen.

  Wie oben bereits erklärt wurde, werden Datenbankobjekte nie aktualisiert, sondern stets zum Bestand hinzugefügt. Da Objectivity auf Containerebene Lese- oder Schreibsperren einrichtet (mehr zu Containern siehe unten), steht das Problem zur Debatte, was geschieht, wenn ein Container mit neuen Objekten gefüllt wird und dieser gleichzeitig von anderen Benutzern gelesen werden soll. Der Container wird mit einer Schreibsperre versehen, so daß Objekte darin aufgenommen werden können. Für konkurrierenden Zugriff stellt Objectivity zwei Methoden zur Verfügung: Exklusiven Zugriff und die MROW-Technik[1]. Ersteres garantiert eine Schreibsperre auf den Container, falls dieser noch von keinem anderen Benutzer gelesen wird. Andererseits werden bei bestehender Schreibsperre alle lesenden Anfragen nicht zugelassen. Es kann also entweder ein Klient schreibend, oder mehrere Anwender lesend zugreifen. Die MROW-Methode macht beides möglich: Während der Container mit Objekten gefüllt wird, können lesende Anfragen bearbeitet werden. Die Klienten haben die Möglichkeit, jederzeit den aktuellen Stand des Containers abzurufen.

- Möglichkeiten zur Datensicherung
  Gerade während der Anfangsphase können Fehler durch Unwissenheit oder mangelnde Erfahrung geschehen, die die Datenbank in einen wüsten oder sogar irreparablen Zustand versetzen zu vermögen. Es gibt zwar mitgelieferte Werkzeuge, die eine Wiederherstellung von verlorenen Daten anhand datenbankinterner Sicherungstechniken versuchen, aber Datenverlust ist stets mit Aufwand und Risiken verbunden. Datenbankschäden können nicht nur durch Software, sondern auch durch Fehler bei den Speichermedien erzeugt werden. In diesem Falle kann mit Hilfe von Datensicherungen die Datenbank durch Einspielung der jüngsten Sicherung wieder auf den letzten gesicherten Stand gebracht werden.

  Der Objectivity-Datenbank liegt umfangreiche Software zur Wartung und Pflege bei. Das Werkzeug “oobackup” kann von der gesamten Datenbank eine Sicherheitskopie anfertigen. Da der Inhalt der Datenbank von H1 lediglich eine Kopie von DST-Beständen in Verbindung mit der Reduzierung von Ereignisdaten darstellt, ist selbst

[1] Multiple Read One Write (MROW)

der Totalverlust der Datenbank unkritisch. Die Datenbank kann jederzeit aus den DST-Daten wiederhergestellt werden.

- Datendistribution
  Die Daten sollen auf verschiedenen Rechnern auch außerhalb des Institutes und auf verschiedenartigen Speichermedien verteilt werden können.

  In der Tat ist Objectivity für dieses Problem sehr gut vorbereitet. Bei CERN ist geplant, das skalierbare Massenspeichersystem HPSS[2] mit Objectivity/DB zu verbinden. Die Test-Datenbank bei H1 ist bereits auf mehreren Festplatten verteilt.

- Schema-Entwicklung[3]
  Ein System zur Anpassung existierender Datenbankobjekte ist wichtig, da es mit Sicherheit in Zukunft vorkommen wird, daß sich die Bedürfnisse der Forscher ändern werden und somit neue Variablen in den Objekten hinzugefügt werden müssen.

  In der Datenbankwelt werden zwei Methoden zur Behandlung des Schemas praktiziert: Schema-Entwicklung und Schema-Versionierung[4]. Letzteres legt bei Objektstruktur-Änderungen neue Versionen an, so daß eine baumartige Historie entsteht. Das Konzept der Schema-Entwicklung erlaubt es den Anwendungen, Änderungen auf dasselbe Objekt durchzuführen. In [Kim90] werden zwischen zwei Aktualisierungsstrategien unterschieden: sofortige und verzögerte Änderung. Beim Ansatz der sofortigen Änderung des Schemas werden alle Objekt-Instanzen der gesamten Datenbank mit sofortiger Wirkung bearbeitet. Der Nachteil dabei ist, daß je nach Umfang und Verteilung der Datenbank diese Operation sehr lange dauern kann. Allerdings gibt es anschließend keine Objekte mit inaktuellen Informationen mehr. Schema-Entwicklung mit verzögerter Objektänderung bedeutet, daß der Inhalt der Datenbank nicht verändert wird, jedoch vor dem Transfer zum Klienten bearbeitet wird. So wird auf der Seite des Anwenders stets das neue Schema gesehen, welches während der Anfrage aus dem älteren Schema und der Differenz zum aktuellen Schema gewonnen wird. Ein Nachteil dieses Prinzips ist, daß die Datenbank immer noch das ältere Schema sowie die Differenzdaten enthält, bis ein Aktualisierungsvorgang vorgenommen wird.

  In Objectivity/DB wird das Konzept der Schema-Entwicklung mit verzögerten Änderungen verwendet. Ein mitgeliefertes Hilfsprogramm „ootidy“ kann die Datenbank partiell von alten Schemata säubern. Mit „partiell“ sind hier die Datenbanken gemeint, die der föderierten Datenbank untergeordnet sind. Mehr zur Struktur von Objectivity/DB kann in Kapitel 4.2 gefunden werden.

- Schreibschutz
  Nicht-autorisierte Benutzer dürfen keinen Schreibzugriff auf die Datenbank haben. Dies wird durch die betriebssystemspezifischen Zugriffsrechte des Benutzers geregelt.

[2]High Performance Storage System, `http://www.sdsc.edu/hpss/hpss.html`
[3]Mit „Schema“ ist der Begriff aus der Datenbank-Terminologie gemeint.
[4]Im Englischen “schema evolution” und “schema versioning”

4.2 Organisation der Objekte

Objectivity/DB besitzt eine vierstufige Hierarchie von Objekten. Die gesamte Datenbank ist die Federated-Database, die in Partitionen (Databases) unterteilt ist. Diese werden mit Containern gefüllt, die letztendlich eine Vielzahl von elementaren Objekten (Basic Objects) enthält. In Abbildung 4.1 ist diese Struktur dargestellt. In Objectivity/DB werden Java-Objekte persistent, indem sie Objectivity-spezifische Klassen (`ooFDObj, ooDBObj, ooContObj, ooObj`) durch Vererbung erweitern.

Die in dieser Arbeit erstellte Objectivity-Schnittstelle zu Java Analysis Studio greift noch auf Version 2 der Testdatenbank bei H1 zu. Die Organisation ist in Abbildung 4.2 dargestellt. Inzwischen existiert die Datenbankstruktur in Version 3 (Abb. 4.3). Die Partitionen, die Ereignisdaten enthalten, haben Namen der Form $yyyy$DSTn, wobei $yyyy$ die vierstellige Jahreszahl der Datennahmeperiode ist und n die Versionsnummer der rekonstruierten Daten innerhalb eines Jahres darstellt. Innerhalb der Partitionen befinden sich Container, von denen jeder einem Run zugeordnet ist. Eine Partition, deren Datennahme abgeschlossen ist, enthält etwa 20000–25000 Runs. Alle Container enthalten eine große Anzahl von Ereignissen (ca. 500–2000), die aus Meßdaten bestehen. Die Struktur der Ereignisobjekte ist in Abbildung 5.6 zu sehen.

Alle Objekte in Objectivity/DB werden durch eine Objekt-Nummer (OID) identifiziert. Diese Nummer benötigt 8 Bytes und setzt sich wie folgt zusammen: Partitionen werden in 16-Bit numeriert (64K), Container in 15-Bit (32K), jeder Container kann bis zu 64K Seiten aufnehmen, und jede Seite kann bis zu 64K Objekte verwalten. Ein Engpaß tritt hier bei der Anzahl von Containern pro Partition auf. Bei 20000–25000 Runs, die jeder einem Container entsprechen, ist die Grenze von 32768 Containern relativ nahe. Daher sollte im Laufe eines Jahres die Anzahl der Runs pro DST beobachtet werden, damit hier kein Überlauf entsteht.

Die Ereignisstruktur ist flach. Ein Ereignis besteht aus genau einem Objekt, welches aus elementaren Informationen aufgebaut ist. Die Analysen in der Teilchenphysik haben durch die Histogramme die Eigenschaft, von der Ereignisstruktur eine einzige Variable zu fordern. Zusätzlich werden einige weitere Felder zum Abgleich durch Schnittkriterien benötigt. Dies hat den Nachteil, daß ein Lesezugriff stets das gesamte Objekt anfordert, obwohl nur eine kleine Teilmenge letztendlich in der Analyse verwendet wird.

Es wird in Zukunft mit Sicherheit erforderlich sein, die Struktur der Ereignisobjekte anpassen zu müssen, weil sich die Bedürfnisse der Forscher sehr wahrscheinlich ändern werden. Zum Beispiel könnte eine bisher wenig untersuchte Variable interessant werden, die auf Wunsch der Forscher in die Datenbank aufgenommen werden soll. Daraufhin muß entschieden werden, an welchen Ereignissen diese Änderung vollzogen wird. Einerseits könnten alle Ereignisse der Datenbank angepaßt werden, so daß anschließend keine alte Strukturen mehr zu finden sind. Dieser Vorgang wäre jedoch mit dem Nachteil behaftet, daß der Batch-Job sehr lange dauern würde. Alternativ könnte abhängig von der Benutzernachfrage ein Jahr festgelegt werden (z.B. 1998), ab dem alle bisher bestehenden Ereignisse mit der neuen Struktur aktualisiert werden. Die Trennung der Ereignisse nach Jahren ist insofern zu empfehlen, da sie in der Datenbank nach Datennahme-Jahren gruppiert sind. Somit kann exakt

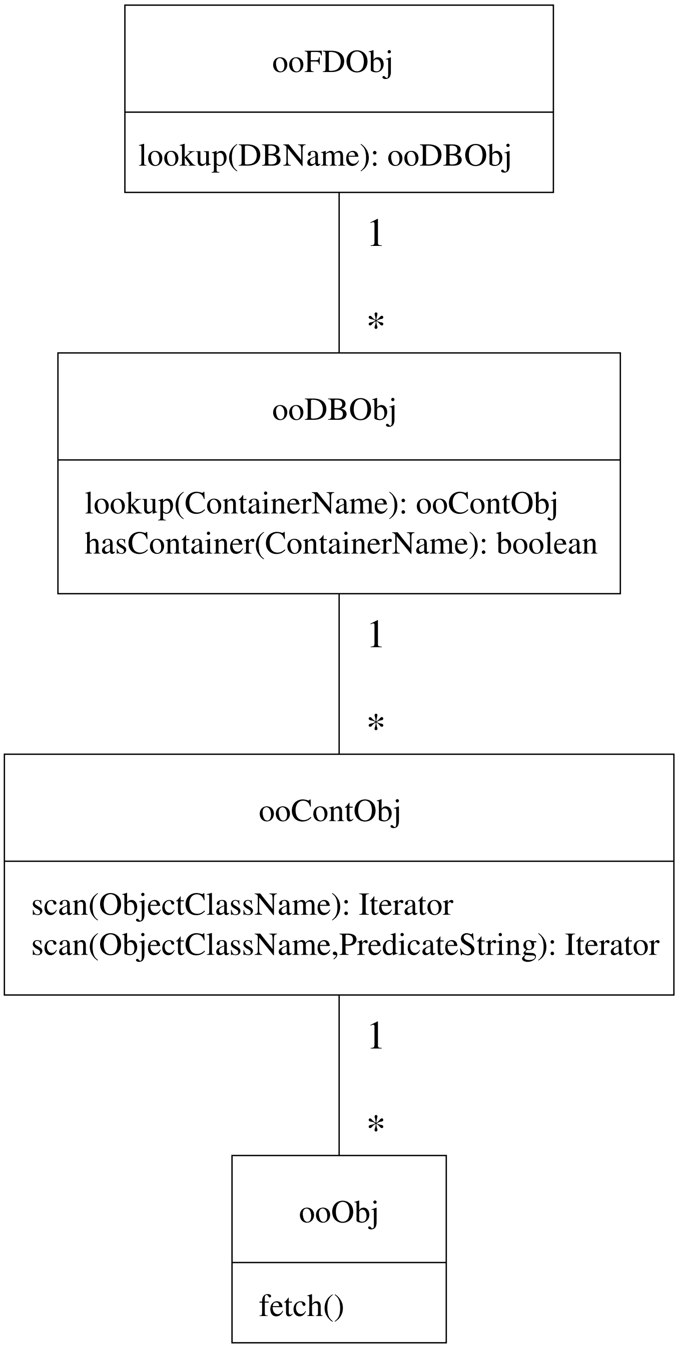


Abbildung 4.1: Objekt-Hierarchie in Objectivity/DB

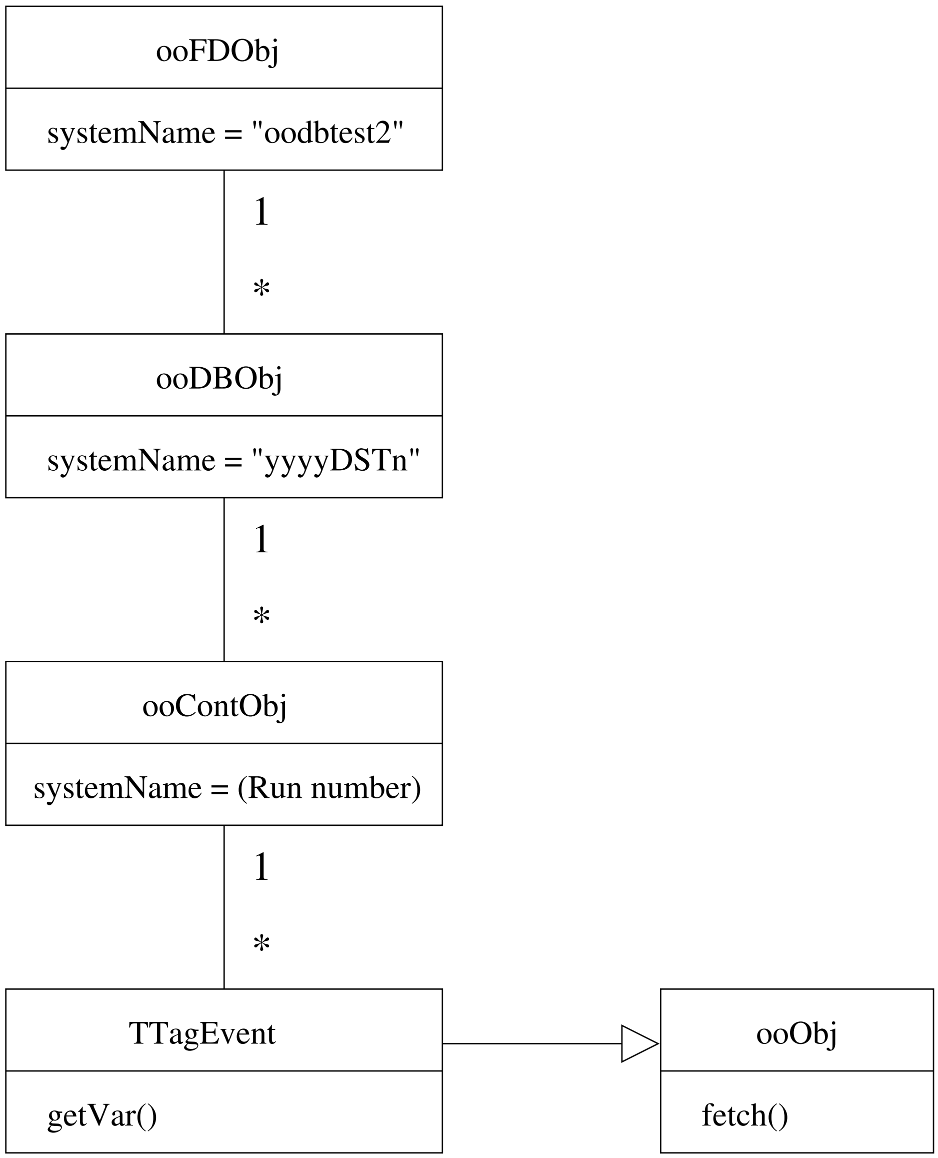


Abbildung 4.2: Aufbau der Test-Datenbank Version 2

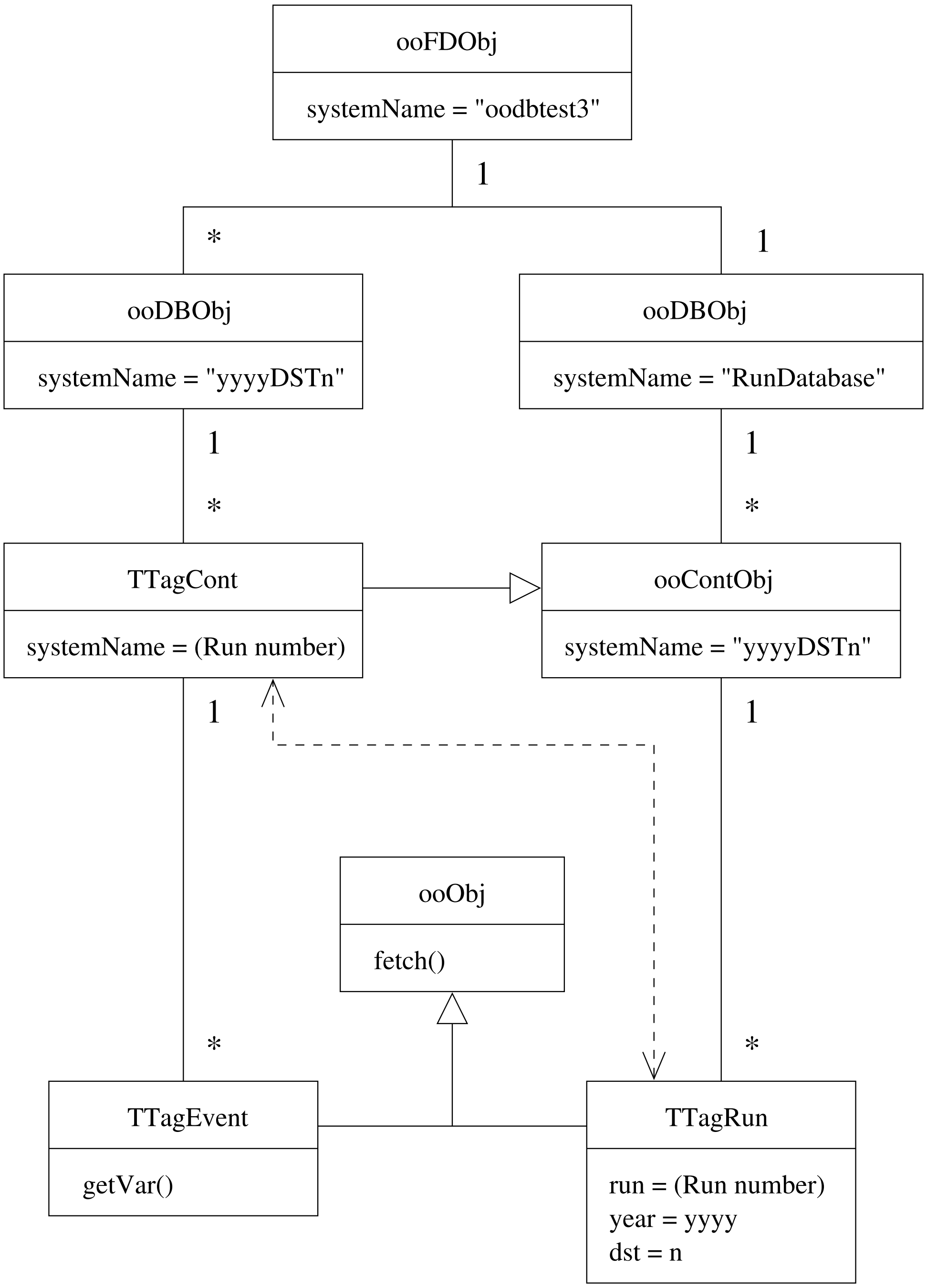


Abbildung 4.3: Aufbau der Test-Datenbank Version 3

TTagRun
public year: int public dst: int public run: int public fill: int public utime: int public qflag: int public phase: int public lumi: float public ecurr: float public pcurr: float public eenergy: float public penergy: float public xvertex: float public yvertex: float public xtilt: float public ytilt: float public hvbest: int[3] public hvworst: int[3] public l4events: int[18] public select: int public ctconfig: int[4] public nl2keep: int public runtime: float public l1time: float public tescal: float[192] public teiscal: float[192] public l1prefac: float[128] public l1scal: float[128] public l1scalp: float[128] public l1scal2: float[128] public l1scal3: float[128] public l2prefac: float[48] public l2ovrfac: float[48] public l2scal: float[48] public l2scalp: float[48] public l2scal3: float[48] public tocont <-> torun

Abbildung 4.4: TTagRun, ein Basic-Object in Objectivity/DB

anhand des Jahres unterschieden werden. Für die Aktualisierung der Ereignisse mit der neuen Struktur ist ein sehr großer Batch-Job notwendig, der die betroffenen Objekte einliest, ändert und mit der neuen Struktur wieder zurückschreibt. Da Objectivity/DB mit Schema-Entwicklung arbeitet, werden ausschließlich diejenigen Objekte aktualisiert, die angefordert wurden. Ereignisse, die nicht vom Batch-Job bearbeitet werden, verbleiben unberührt mit der bisherigen Struktur.

Neu in Version 3 ist die Verfügbarkeit von Run-Informationen. Neben den Partitionen mit Ereignisdaten gibt es in der Federated-Database eine Partition namens „RunDatabase". Diese enthält für jede DST-Partition genau einen Container, dessen Objekte Informationen zu den Runs aus den DST-Partitionen bereitstellen. Run-Informationen beziehen sich auf Eigenschaften des Runs selbst, z.B. Energien der kollidierenden Teilchen[5] oder Zustände von Trigger-Levels. Die Struktur der Run-Informationen wird in Abbildung 4.4 dargestellt. Es existiert eine bidirektionale Assoziation von jedem Run-Informationsobjekt zum zugehörigen Container, der die Ereignisobjekte dieses Runs enthält. Diese Verbindung wird durch die gestrichelte Linie in Abbildung 4.3 verdeutlicht. Die Assoziation hat den Vorteil, daß durch die Run-Informationen, die separat von den Ereignissen abgelegt sind, dennoch eine direkte Verbindung zu den Ereignissen hergestellt wird. Durch die Unterstützung von referentieller Integrität durch Objectivity/DB sind die Assoziationen sicher. Die assoziierten Objekte pflegen ihre Zeiger selbständig. Die Variablen des Run-Objektes bestehen aus elementaren Datentypen (Integer 32-Bit, Float). Die dynamischen Tabellen (Arrays) sind mit 48, 128 und 192 Elementen relativ groß. Das gesamte Objekt benötigt etwa 4–5kB Speicherplatz.

Die in Abb. 4.1 erklärte Hierarchie ist kein Standard, sondern die individuelle Organisationsform der Objectivity-Datenbank. Nachdem die ersten ODBMSs auf den Markt gekommen waren, gab es immer noch keinen Standard für objekt-orientierte Datenbanken. Daher wurde die "Object Data Management Group" (ODMG) von mehreren ODBMS-Herstellern gegründet [ODMG97], die seitdem in der Mitgliederzahl wächst. Der ODMG-Standard legt eine einheitliche Schnittstelle für objekt-orientierte Datenbanksysteme fest. Auch Objectivity ist Mitglied bei ODMG, unterstützt diesen Standard aber leider nur größtenteils [Shiers98]. Bei H1 wird der Standard nicht genutzt, sondern es wird auf die eigene Partitionierungs-Hierarchie von Objectivity/DB zurückgegriffen, da die flache H1-Ereignisdatenhierarchie besser darauf abgebildet werden kann als auf ODMG.

4.3 Aktueller Projektstatus

Das Programm zum Füllen der Objectivity-Datenbank existiert und wird im Laufe der verschiedenen Test-Datenbanken ständig verbessert. Es wird auch Linker-Bibliotheken für das Auslesen der Datenbank geben [Hadig98c]. Damit wird die Schnittstelle für Fortran-Programme von H1-Anwendern definiert.

Während der letzten Wochen wurde die dritte Test-Version der Objectivity-Datenbank besprochen und erstellt. Java Analysis Studio beruht noch auf Version 2 und ist damit noch nicht für die erweiterte Datenbankstruktur der neuen Version vorbereitet.

[5]Bei HERA sind dies ein Elektron und ein Proton. Je nach Run-Konfiguration variieren deren Energien.

Block	Blockgröße in Bytes
RUNEVT	12
TRIG	212
ELPH	252 ($84 \cdot 3$ Elektronen max.)
HFS	172
KINE	60
PRES	4
DIFF	72
JETS	20
H1Q2	8
VLQ	20
FNC	20
FPS	80
Summe	832

Datenfeld	Bytegröße
Integer	4
Float	4
Double	8
3-Vector	12
4-Vector	16

Tabelle 4.1: Eventgröße in Objectivity/DB bei H1.

Die installierten Testdatenbanken sind zur Zeit 100–110 MB groß (Versionen 2 und 3). Diese bescheidene Größenordnung kommt bei weitem nicht an die erwarteten Ausmaße heran, die in Abschnitt 4.4 (siehe unten) erwartet werden. Da allerdings bei CERN (Abt. RD45) bereits ein riesiger Datenbankumfang ausgiebig getestet wurde, brauchte man dies bei H1 nicht zu wiederholen. Dazu muß gesagt werden, daß die CERN-Datenbank im Mittelpunkt der Datennahme steht, während Objectivity/DB in H1 — verglichen mit den bisherigen Band- und Festplattenbeständen — nur ein relativ kleines Volumen einnimmt und parallel zum bisherigen Datenfluß betrieben wird.

In der Testdatenbank Version 3 sollte die Index-Fähigkeit von Objectivity/DB experimentell miteinbezogen werden, welches jedoch zunächst noch an Implementierungsproblemen scheiterte.

4.4 Größe der Datenbank

Bisher ist die Datennahme ab 1997 angedacht, damit die H1-Anwender auch diese Daten noch in ihre Analysen einbeziehen können. Der einmalige Ladevorgang bei Inbetriebnahme der Objectivity-Datenbank würde diese mit die Daten der Jahre 1997–1999 füllen. Das Verfahren der Migrierung ist bereits klar: Auf die BOS-Bänke der DST-Dateien wird mit einem Fortran-Unterprogramm zugegriffen, und die neuen, berechneten Daten werden dann als Blöcke durch ein C++ Unterprogramm in die Datenbank geschrieben [Hadig98c].

Die Ereignisse aus den DST-Dateien sollen bijektiv auf die Objekte der Datenbank abgebildet werden. Nicht alle DST-Ereignis-Informationen werden übernommen, sondern nur ein Teil davon, der zur Zeit noch bei der H1-Objectivity-Gruppe diskutiert wird, jedoch schon recht konkret festgelegt ist. Dadurch sollen Zugriffszeiten sowie Speicherplatz möglichst klein gehalten werden. Tabelle 4.1 gibt eine Berechnung der Ereignisgröße in Objectivity/DB an, die mit Hilfe des Proposal-Drafts für die Datenbank [Hadig99] erstellt wurde. Für die Berechnung der jährlichen Zuwachsrate von Objectivity/DB wird der jährliche

DST-Datenzuwachs von 500GB (ab Jahr 2000, lag bis 1999 bei 450GB) zu Hilfe genommen [Collab99], und die Objectivity/DB-Ereignisgröße wird auf 1kB aufgerundet (für eventuellen Overhead, zur Vereinfachung, und um einen Puffer für eventuelle zukünftige Änderungen zu haben):

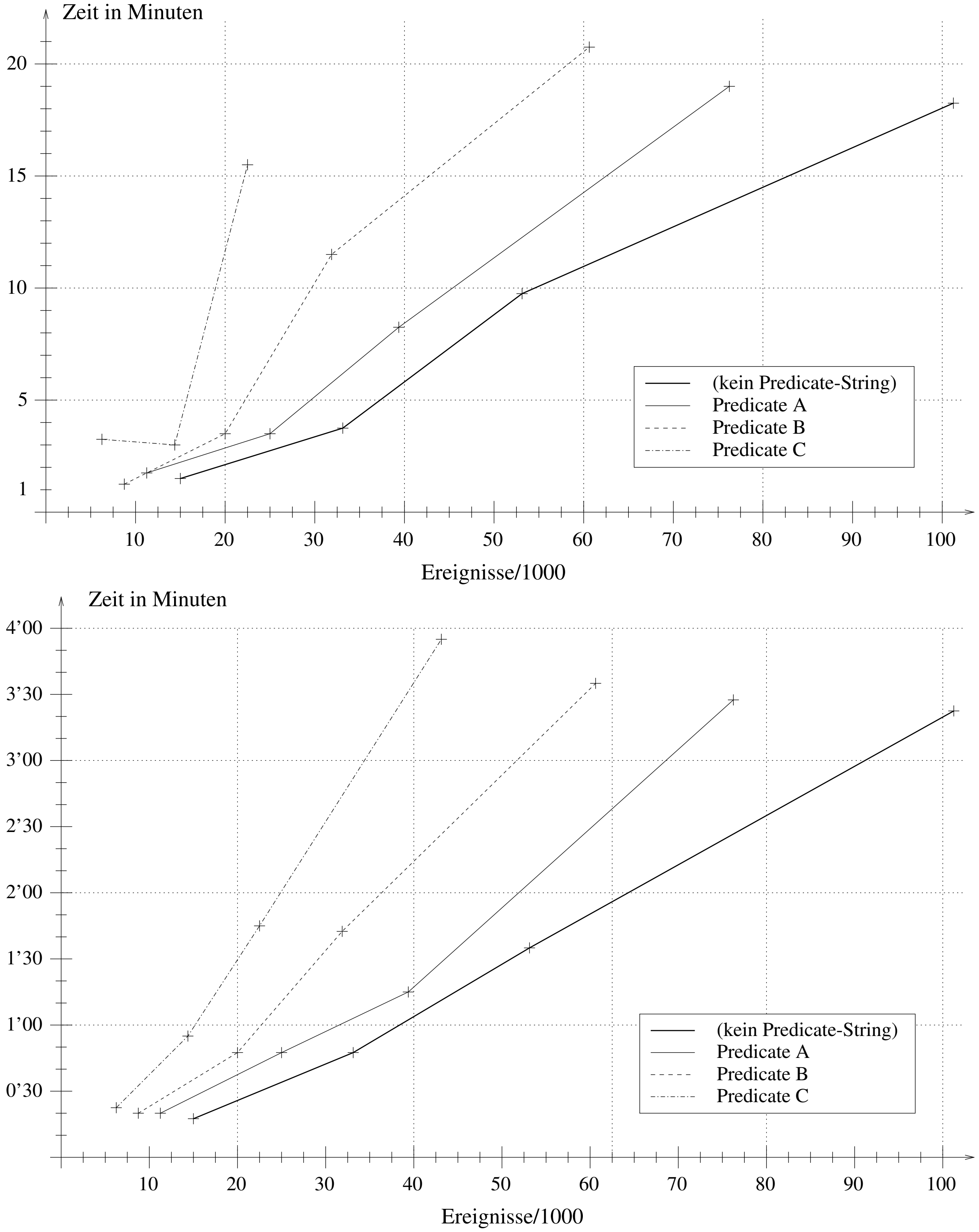
$$\begin{aligned} \text{Objy/DB-Zuwachs} &= \frac{\text{DST-Datenzuwachs}}{\text{DST-Eventgröße} \ / \ \text{Objy-Eventgröße}} \\ &= \frac{500\text{GB}}{10\text{kB} \ / \ 1\text{kB}} = 50\text{GB} \end{aligned}$$

In den Jahren 1997 und 1998 sind jeweils etwa 450GB an DST-Daten hinzugekommen. Bei den 1998er Daten gab es leider diverse Probleme, so daß dort wesentlich weniger Daten angefallen sind. Diese rechnerische Lücke kann man jedoch mit Daten von 1999 füllen. Die Summe von 900GB entspricht also der Datenmenge, die vor der offiziellen Einführung der Datenbank zu Objectivity/DB migriert werden müßte. Der Zeitplan [Hadig98a] hatte ursprünglich das Jahr 1998 als Testzeitraum vorgesehen. Jedoch hat sich dieser Plan verschoben, so daß ein Großteil von 1999 zur Testphase zusätzlich einkalkuliert werden sollte. Die DST-Daten der Datennahme-Periode von 1999 werden daher mit in die zu migrierende Masse eingehen. Insgesamt werden schließlich etwas mehr als 1TB an DST-Daten zu behandeln sein. Da die Ereignisdaten bei Objectivity/DB um den Faktor 10 kleiner sind als DST-Daten, kann eine initiale Datenbankkapazität von 100GB angepeilt werden. Der Plattenplatz dieser Datenmenge kann durch mehrere 18GB-Festplatten realisiert werden, wie es bereits bei der Kalkulation für die Plattenkapazität bei H1 getan wird [Collab99]. Auch die jährliche Aufstockung von Objectivity/DB um 50GB ließe sich durch dieses Verfahren bewerkstelligen. Die Realisierung der Datenbank müßte demnach nicht unter mangelndem Datenspeicher leiden.

Es besteht oft der Wunsch, partielle oder vollständige Sicherheitskopien von Datenbanken anzulegen. Falls dies für die Objectivity-Datenbank bei H1 entschieden wird, so muß — im Falle einer vollständigen Sicherung — der Speicherplatzbedarf in doppelter Ausführung bedacht werden. Da Datensicherungen auf Bändern gelagert werden, gibt es an dieser Stelle keine Platzprobleme. Durch den verhältnismäßig geringen jährlichen Bedarf von 50 GB können sogar mehrere Sicherungen aus vergangener Zeit gehalten werden. Da der gesamte Inhalt der Datenbank jederzeit aus dem DST-Bestand wiederhergestellt werden kann, ist zu überlegen, ob Datensicherungen dennoch notwendig sind.

4.5 Benchmark

Der Benchmark in Abbildung 4.5 wurde mit Java Analysis Studio und dem Objectivity-DIM erzeugt. Beiden Grafiken liegen die Meßwerte aus Tabelle 4.2 zugrunde. Die Graphen in den Diagrammen sind den Zeilen der Tabellen zugeordnet. Jeder Graph besteht aus vier Meßwerten, die den Ereignismengen der Run-Bereiche entsprechen, die aus dem unteren



Anfragebearbeitungszeiten an Objectivity/DB durch Java Analysis Studio (oben) und einem C++-Programm (unten); Schnittkriterien:

A (1 Bedingung): `rele>11`
B (3 Bedingungen): `rele>11 && relth>0.175 && relth<2.6`
C (9 Bedingungen): `rele>11 && relth>0.175 && relth<2.6 && zvtx>-29 && zvtx<30 && xe<0.05 && xso<0.1 && xjo<0.1 && q2jo<1000`

Abbildung 4.5: Benchmark: Java Analysis Studio, C++-Programm

Teil von Tabelle 4.2 entnommen werden können. Das obere Bild zeigt die Bearbeitungszeiten von verschiedenen Datenbankanfragen durch JAS mit unterschiedlichen Schnittparametern (Predicate-Strings). Diese Parameter bestehen aus mehreren miteinander verknüpften Bedingungen. Der unteren Grafik liegen die gleichen Anfragen wie aus dem oberen Bild zugrunde. Hier wurden die Anfragen durch ein eigens dafür geschriebenes C++-Programm umgesetzt, wobei Wert darauf gelegt wurde, den kritischen Programmteil möglichst gleichwertig zur Java-Routine zu halten. In der Java-Grafik fehlt leider ein Meßwert (bei Predicate C), der wegen einer Objectivity-Fehlermeldung nicht ermittelt werden konnte. Die Meßdaten verlaufen allgemein bei dem C++ Programm wesentlich regelmäßiger als bei JAS. Dies kann einerseits an der Speicherbereinigung von Java liegen. Andererseits ist die Bearbeitungszeit der Anfragen bei JAS wesentlich länger, so daß währenddessen mehr äußere Einflüsse auf das System einwirken können.

Der Lock-Server von Objectivity/DB läuft auf einer Sun-Ultra1-Workstation unter Solaris, und JAS wurde auf einem anderen, gleichwertigen Rechner gestartet, damit die Rechenzeit für den Lock-Server nicht durch das Client-Programm beeinflußt wird. Beide Systeme sind durch Ethernet mit TCP/IP verbunden und befinden sich in demselben Gebäude. Sowohl der Server- als auch der Client-Rechner waren zu der Zeitspanne der Tests durch andere rechenintensive Programme nicht belastet. Jedoch kann und wird es in Zukunft vorkommen, daß JAS parallel zu anderen Jobs läuft, und daß auch mehrere Anwender gleichzeitig auf die Datenbank zugreifen. In [HoltBunn98] wurden bei CERN/CMS umfangreiche Untersuchungen zur Skalierbarkeit bezüglich der Anzahl von Benutzern angestellt, die gleichzeitig zusammen die Datenbank benutzt haben. Hierbei wurde ein positives Ergebnis erzielt. Es wurde ein Engpaß beim Lock-Server von Objectivity/DB vermutet, da dieser die zentrale Stelle ist, durch die die Anfragen geschleust werden, doch diese Vermutung hat sich nicht bestätigt. Bei CERN bestehen zwar wesentlich größere Dimensionen als bei H1 hinsichtlich Rechenleistung und Benutzerumfang, aber die Ergebnisse der Skalierbarkeitstests verlieren deshalb nicht an Aussagekraft.

Java Analysis Studio wurde für jede Anfrage neu gestartet, da mehrfache Anfragen aus derselben Applikation die Antwortzeiten beeinflussen. Hierauf wird weiter unten eingegangen werden.

In den Anfragen werden vier verschiedene Run-Bereiche angegeben, wodurch die Anzahl der iterierten Ereignisse festgelegt ist (Abb. 4.2). Durch vier unterschiedliche Schnittparameter wird jeder der Run-Bereiche unterteilt. Dadurch werden Auswirkungen auf die Antwortzeiten untersucht, die durch Schnittkriterien entstehen könnten. Die Schnitte bestehen aus 0 (leerer Predicate-String, alle Ereignisse werden selektiert), 1, 3 und 9 atomaren Bedingungen, die in Abbildung 4.5 notiert sind.

Der Vergleich von JAS mit dem C++-Programm soll die Unterschiede in den Anfrage-Bearbeitungszeiten offenlegen, die ausschließlich durch die Java-Umgebung entstehen. Das Ergebnis ist deutlich: Die Antwortzeiten von JAS sind mindestens viermal länger als die gemessenen Zeiten des C++-Programmes. Bei den Run-Bereichen „190000–200000“ und „180000–201000“ steigt der Faktor auf sechs. Dies ist in Abb. 4.5 an einer Vergrößerung der Steigung der Graphen zwischen den beiden mittleren Meßpunkten zu erkennen. Bei JAS wird dieser Sprung besonders deutlich, was auf den verstärkten Einfluß der asynchronen Speicherbereinigung von Java[6] in Zusammenhang mit einer Überschreitung einer Speichergrenze durch eine zu große Ereignismenge zurückgeführt werden könnte. Jedoch ist dieser Sprung in sehr schwacher Ausprägung auch bei der Grafik zum C++-Programm zu erkennen. Daher ist zu vermuten, daß ein Cache bei Objectivity/DB übergelaufen ist oder ein anderer Grenzwert überschritten wurde.

Nach Auskunft von H1-Mitarbeitern ist der Umfang der NTuples sehr variabel, die mit Kumac-Makros verarbeitet werden. Kurze Analysen liegen bei etwa 20000 Ereignissen. NTuple-Dateien können aber auch bei 100000–150000 Ereignissen liegen. Manche Untersuchungen erstrecken sich sogar über mehrere NTuples, so daß eine Gesamtmenge von z.B. 500000 Ereignissen zustandekommt. Die Menge der zu verarbeitenden Ereignisse hängt stets von der Analyse ab und ist in der Regel sehr unterschiedlich. Die Abarbeitung von etwa 150000 Ereignissen nimmt bei 60% Rechnerauslastung etwa 3 Minuten in Anspruch. Bei halber Rechenleistung dauert eine Analyse mehrerer NTuples mit insgesamt 500000 Ereignissen ca. 20 Minuten. Im Vergleich dazu kann der Testlauf zu den Benchmarks aus Tabelle 4.2 herangezogen werden, bei dem 101666 Ereignisse verarbeitet wurden. Die Bearbeitungszeit beträgt 3'22", was in etwa der Zeitraum ist, der für eine Kumac-Analyse über 150000 Ereignisse benötigt wird. Dabei ist allerdings zu bedenken, daß NTuples flache, lokale Dateien sind und somit schneller geladen werden können als Ereignisse aus einer Datenbank auf einem anderen Rechner. Weiterhin ist die Größe eines NTuple-Ereignisses nicht allgemein festgelegt, sondern hängt von den Attributen ab, die in die NTuple-Datei geschrieben wurden. Es kann durchaus sein, daß ein Ereignis dort sogar größer ist als ein DST-Ereignis, da etwa zusätzliche Werte berechnet wurden.

Die hauptsächliche Erkenntnis aus dem Benchmark ist, daß die Sprache Java zusammen mit dem Java-Frontend von Objectivity/DB ein entscheidender Flaschenhals zur Datenbank ist. Eine weitere Aussage läßt sich auch bezüglich der Schnittparameter treffen. Durch sie wird die Ergebnismenge reduziert, die von Objectivity während der gesamten Anfrage an die Benutzer-Applikation geliefert wird. An den Werten aus Tabelle 4.2 ist zu erkennen, daß umfangreichere Schnittparameter nur geringfügige Auswirkungen auf die Antwortzeiten haben.

Bei den ersten Versionen des Benchmarks aus Abb. 4.5 entstand der Verdacht, daß die erste Datenbankanfrage in JAS schneller bearbeitet wurde als darauffolgende Anfragen. Daher wurde ein weiterer Benchmark gefahren, durch den dieses Verhalten untersucht wird. Es wurden die Antwortzeiten bei mehrfachen Datenbankanfragen innerhalb derselben JAS-Applikation gemessen. Die Ergebnisse sind aus Tabelle 4.3 ersichtlich. Es wurde sechsmal dieselbe Anfrage wiederholt, ohne JAS zu beenden. Die Antwortzeiten waren beim ersten Mal stets am kürzesten. Die erste Testreihe zeigt mit etwa 15000 Ereignissen einen Zeitzuwachs von 30–50%. Die Bearbeitungszeiten der Testreihe mit 34000 Ereignissen erfuhren eine Verschlechterung von etwa 100–200%, welches durch die Testreihe mit 53000 Ereignissen und 400–600% drastisch gesteigert wurde. Die Ursache dieses Problems könnte entweder mit der Speicherbereinigung von Java zu tun haben, oder auch — wie bei dem oben genann-

[6] Garbage-Collection

Antwortzeiten bei Java Analysis Studio

Predicate	198000–200000		195000–200000		190000–200000		180000–201000	
	JAS	C++	JAS	C++	JAS	C++	JAS	C++
(ohne)	01'30"	00'19"	03'43"	00'47"	09'49"	01'34"	18'21"	03'22"
A	01'53"	00'20"	03'35"	00'47"	08'14"	01'14"	19'02"	03'27"
B	01'17"	00'20"	03'32"	00'48"	11'36"	01'41"	20'53"	03'36"
C	03'16"	00'21"	02'56"	00'57"	15'26"	01'45"	n/a	03'55"
Anzahl der Ereignisse								
(ohne)	15384		33936		53475		101666	
A	11469		25132		39691		76136	
B	9108		19993		31658		60807	
C	6466		14109		22268		43159	

Legende zum Predicate-String siehe Abb. 4.5, weitere Spalten stehen für Run-Bereiche

Tabelle 4.2: Benchmark: Zugriffszeiten der Anfragen an Objectivity/DB durch Java Analysis Studio und ein C++ Programm

Mehrere Analysen in derselben Applikation

Appl.	Nr.	198000-200000 15384 Events	195000-200000 33936 Events	190000-200000 53475 Events
JAS	1	01'33"	03'28"	09'57"
	2	01'36"	07'34"	24'19"
	3	01'53"	07'37"	53'20"
	4	02'45"	07'32"	1°00'39"
	5	02'07"	07'35"	1°04'13"
	6	02'02"	10'40"	1°06'47"
C++	1	00'25"	00'55"	01'45"
	2	00'22"	00'53"	01'36"
	3	00'21"	00'52"	01'43"
	4	00'21"	00'53"	01'37"
	5	00'21"	00'58"	01'37"
	6	00'22"	00'52"	01'35"

Leerer Predicate-String, weitere Spalten stehen für Run-Bereiche

Tabelle 4.3: Benchmark: Mehrere Analysen in derselben Applikation ten Benchmark — mit Cache-Eigenschaften von Objectivity/DB. Um die letzte Vermutung auszuschließen, wurden dieselben Tests mit dem C++ Programm gefahren. Hier sind die Antwortzeiten bei allen sechs Anfragen in etwa konstant und erfahren keine Veränderungen. Die jeweils erste Anfrage benötigt sogar ein wenig mehr Zeit als die nachfolgenden Testläufe. Dies kann daran liegen, daß es sich dabei um „kalte“ Anfragen handelte — die erste Anfrage auf eine Datenbank stößt auf leere Caches, und beim ersten Zugriff können weitere Initialisierungen vorgenommen werden, so daß sich hier Verzögerungen ergeben. Bei den nachfolgenden Zugriffen ist die Datenbank „warm“, was bedeutet, daß die Cache-Speicher gefüllt und die Initialisierungsarbeiten bereits erledigt sind, wodurch die Anfrage eine effektivere Bearbeitungszeit erfährt. Insgesamt kann bestätigt werden, daß das Problem der Verschlechterung der Antwortzeit nicht an Objectivity/DB liegt.

Die oben genannten Benchmarks haben ein konkretes Bild davon gegeben, wie die Zugriffszeiten bei Objectivity/DB in H1 zur Zeit aussehen. In [Chaudhri95] wird eine Sammlung von Benchmarkstudien präsentiert, die sich mit objekt-orientierten Datenbanken aus verschiedenen Blickrichtungen auseinandersetzen. Darunter befindet sich der OO7-Benchmark [CDN94], der Leistungsergebnisse zum hauseigenen Datenbanksystem E/Exodus und drei kommerziellen ODBMS dokumentiert (Ontos 2.2, Versant 3.0 Beta und Objectivity/DB 2.1). Der Benchmark ist insofern interessant, da hier Objectivity/DB und Versant verglichen werden. Diese beiden Datenbanksysteme wurden bei CERN/RD45 auf der Suche nach einem skalierbaren DBMS-Hersteller in die engere Wahl gezogen [Shiers98]. Nach den ersten Skalierbarkeitstests wurde wegen Unterschieden in den Schnittstellen von weiteren, großflächigeren Tests zu Versant abgesehen [POMC99], und es wurde Objectivity/DB als ODBMS für LHC gewählt.

4.6 Verbesserungen

In Abschnitt 4.5 wurden einige Probleme aufgespürt, deren Ursache in Java und dessen Speicherbereinigung vermutet wird. In [Hoschek98] werden Ursachen genannt, die von Java herrühren, wie Objektduplizierung (Call-by-value), Speicherhandhabung, Speicherbereinigung und zusätzliche Abstraktionsstufen bei der Vererbung von Objekten. Der Java-JIT-Compiler ist momentan noch nicht sehr profitabel [Hoschek98]. Einige Operationen wurden sogar mit größeren Ausführungszeiten bei JIT-compilierten Programmen im Vergleich zum JDK gemessen [Heckner98]. Es wäre zu empfehlen, entsprechende Testläufe zu entwickeln, bei denen die Speicherbereinigung explizit im JAS-Objectivity-DIM angestoßen wird. Zusätzlich muß die voreingestellte asynchrone Speicherbereinigung per Kommandozeilenparameter zum Java-Interpreter ausgeschaltet werden. Es ist auch zu überlegen, ob die voreingestellte Obergrenze des zu bereinigenden Speichers von 16MB zu erhöhen ist[7]. Dies würde z.B. vom durchschnittlichen Umfang der Ereignismengen abhängen, die bei H1 von den Forschern verlangt werden.

Die Benchmarks haben ergeben, daß der Datenbankzugriff durch C++ um ein Vielfaches schneller erfolgt als durch Java. Die oben angeführten Möglichkeiten von Ursachen sind

[7] Im Sun-JDK 1.1.6 liegt die voreingestellte Speichergrenze für die Bereinigung bei 16MB.

mit Java selbst verbunden; eine andere Frage ist, inwieweit das Java-Frontend von Objectivity/DB den Zugriff ausbremst. Daher wäre es ein Experiment wert, herauszufinden, ob die Portierung der Datenbank-Zugriffsschleife nach C++ eine Leistungssteigerung bewirken würde.

Eine andere Möglichkeit, das Java-Frontend zu testen, ist die Durchführung einer Meßreihe zu den Antwortzeiten von JAS unter der JDK-Umgebung eines Windows-basierten PC's. Zwar werden bei H1 vornehmlich Unix-Systeme benutzt, aber ein Vergleich von JDK auf diesen beiden Systemen in Verbindung mit JAS und Objectivity/DB wäre interessant. Die Ergebnisse würden Aussagen über die Effizienz von JDK auf Windows/PC im Vergleich zu Solaris/UltraSPARC erzielen und die Leistung des Java-Frontends von Objectivity/DB aus der Sicht zweier verschiedener Java-Umgebungen betrachten lassen.

Die obigen Lösungsvorschläge sind lediglich für die Leistungssteigerung in Java anwendbar. Es ist jedoch auch effizient, die Datenbank hinsichtlich der Antwortzeiten zu verbessern, wovon sämtliche Datenbank-Anwendersoftware — unabhängig von der Implementierungssprache — profitieren würde. Während der Benchmark-Messungen mit Java Analysis Studio wurde die benötigte Rechenzeit gemessen. Der Rechner des Benutzers war mit etwa 60–80% und der Lockserver-Rechner mit 30–40% belastet[8]. Das C++ Programm benötige lediglich 40% Rechenzeit. Daraus läßt sich vermuten, daß hierbei Wartezeit für Ein- und Ausgabeoperationen zur Datenbank verbracht wurde. Bei CERN wurden bezüglich der Datenbankorganisation verschiedene Anstrengungen unternommen, da dort ebenfalls Geschwindigkeitsengpässe auftraten.

Objectivity/DB tauscht zwischen dem Server und den Clients keine Objekte, sondern ganze Objektseiten aus [Düllmann99]. Die angeforderten Objekte werden durch Selektionskriterien ausgewählt, wodurch ein unregelmäßiges Zugriffsmuster entsteht. Die Objekte werden der Reihe nach abgearbeitet, und nur diejenigen, auf die die Kriterien zutreffen, werden zum Client übertragen. Trotzdem aber müssen auf diese Weise auch diejenigen Objekte gelesen werden, die durch die Selektion ausgeschlossen werden. Um dies zu vermeiden, wird in [Holtm98] und [HSW98] vorgeschlagen, die Datenbankseiten so zu reorganisieren ("Reclustering"), daß sich auf den gelesenen Seiten ausschließlich die angeforderten Objekte befinden. Dazu werden zwei Methoden vorgestellt: Echtzeitreorganisation ("on the fly") und Batch-Reorganisation. Die erstere Methode findet während der Benutzeranfrage statt. Die betroffenen Objekte werden in eine Kollektion ("collection") kopiert, so daß sie dort direkt hintereinander angeordnet sind. Sobald später dieselbe oder eine ähnliche Anfrage gestellt wird, wird die Kollektion herangezogen. Das hat den unvermeidlichen Nachteil, daß bei der Erstellung einer Kollektion die Antwortzeit zur Anfrage länger dauert. Dies würde aber durch die wiederholte Kollektionsnutzung relativiert werden. Ein weiterer Nachteil ist, daß durch die Replikation von Objekten der Datenbankumfang vergrößert wird. Die Methode der Batch-Reorganisation basiert auf Zugriffsmuster-Statistiken und reorganisiert entsprechend die Objekte. Vorteilhaft ist hier die asynchrone Abarbeitung des Batch-Jobs, der zu beliebigen Zeiten laufen kann. Die Replikation von Objekten entfällt. Der Algorithmus zur

[8]Die Testrechner waren Sun Ultra1 mit Solaris. Die CPU-Belastung wurde mit dem Unix-Befehl „top“ gemessen.

Reorganisation birgt allerdings das NP-vollständige Problem der Mengenüberschneidung in sich, so daß diese Lösung nur bis zu 4–5 Zugriffskriterien skalierbar ist [HSW98].

Ein weiteres, in Abschnitt 4.2 bereits erwähntes Problem, hängt mit der Objektstruktur zusammen: Obwohl nur wenige Felder benötigt werden (z.B. 10 aus 150), wird das gesamte Objekt aus der Datenbank geladen. In [Hoschek98] und [Hoschek99] wird ein spaltenweise in Cluster angeordneter Index ("columnwise clustered index") als Lösungsvorschlag untersucht. Dieser Index ist eine Relation, durch die die Attribute in Spalten angeordnet sind und sich an der Seitengröße der Container in Objectivity/DB orientieren. Bei CERN wird für das Experiment NA48 eine sehr kompakte Objektstruktur vorgestellt, die etwa 200 Bytes umfaßt [Hoschek98]: Die Struktur "SuperCompactEventTag" besteht aus mehreren Teilobjekten, die den verschiedenen Detektorabschnitten zugeordnet sind. Es wurde festgestellt, daß bei einer Objektgröße, die der Seitengröße in der Datenbank entspricht, der Datendurchsatz ein Maximum erreicht. Die Tabellen des Indizes sollte sich daher an die Seitengröße in Objectivity/DB anlehnen.

Mit dem spaltenweise geclusterten Index entsteht eine Beziehung zum relationalen Datenmodell. Es ergibt sich eine Symbiose aus ODBMS- und RDBMS-Architekturprinzipien. Durch den Index soll verhindert werden, daß die irrelevanten Attribute der Objekte miteingelesen werden. Sogenannte „Proxy"-Objekte — Objekt-Platzhalter ohne Attribute, aber mit dem Wissen, wo diese sich befinden — sollen für den Benutzer als reale Objekte fungieren. Der Geschwindigkeitsgewinn des Indizes wäre hier von Vorteil, jedoch ist dieser Ansatz nicht so flexibel wie ein vollständig objekt-orientiertes Datenmodell, da ein größerer Aufwand der Objektverwaltung in Verbindung mit den Indizes anfällt.

4.7 Warum Objectivity/DB ?

Die Firma *Objectivity* [Objy] existiert seit 1988 und hatte im Mai 1997 Objectivity/DB Version 5 angekündigt. Diese wurde im Laufe des Jahres 1998 realisiert und ist seit einigen Monaten erhältlich. Neu in Version 5 ist unter anderem die Unterstützung von Java. Mittlerweile ist bereits die Aktualisierung 5.1 erhältlich.

Einer der schlagkräftigsten Gründe, warum sich die H1-Objectivity-Gruppe für diese Datenbank entschieden hatte, ist die Fähigkeit zur Handhabung von Datenmengen im Terabyte-Bereich. Objectivity wirbt mit diesem Plus, und bei CERN hatte man sogar ein eigenes Projekt zur Verifikation der Skalierbarkeit ins Leben gerufen (RD45), welches auf positive Ergebnisse gestoßen ist [Shiers98]. CERN geht davon aus, dass zu Beginn der Datennahme des LHC[9] im Jahre 2005 jährliche Datenmengen von etwa 5PB anfallen werden[10]. Auch hatte man für das BaBar-Experiment bei SLAC [Babar99] bereits eine voraussichtliche Datenzunahme von 85TB pro Jahr geschätzt [Quarrie96]. H1 profitierte von den Bemühungen dieser Institute zur Evaluierung von Objectivity/DB und entschied sich daher ebenfalls für dieses Datenbank-Management-System.

[9] Large Hadron Collider (LHC)
[10] 1 Peta-Byte = 1024 TB

4.8 Alternative Datenbanksysteme

4.8.1 Relationale Datenbanken

Zu Anfang der 1970er Jahre wurde das relationale Datenbankmodell entwickelt [Codd70], und gegen Ende des Jahrzehntes begann die Umsetzung in kommerzielle Produkte. Bereits vor den ersten Implementierungen waren die theoretischen Grundlagen festgelegt. Dies sicherte und förderte die eindeutige und daher rasche Entwicklung relationaler Datenbank-Management-Systeme (RDBMS) [McHenry93]. Die Etablierung von SQL (Standard Query Language) als einheitliche Anfragesprache hatte dazu beigetragen.

Im Laufe der Zeit sind die Ansprüche gewachsen, so daß die Hersteller von relationalen Datenbanken ihre Produkte immer wieder verbessert haben. Allerdings gibt es Nachteile, die durch das relationale Modell selbst entstanden sind. So können z.B. nur elementare Datentypen in die Tabellen eines RDBMS aufgenommen werden. Größere oder komplexe Objekte (Multimedia, CAD/CAM, variable Arrays) werden erst seit kurzem von einigen relationalen Datenbanksystemen unterstützt.

Würden die Anfragen an ein Datenbank-Management-System seriell bearbeitet werden, so wären die Wartezeiten der Anwender unzumutbar. Das System besitzt daher üblicherweise einen Mechanismus zur Sperre der Daten, so daß konkurrierende Zugriffe synchronisiert werden. Dadurch werden mehrere Datenzugriffe im gleichen Zeitraum ermöglicht, wobei die lesenden und die schreibenden Zugriffsarten unterschiedlich behandelt werden. Je feiner die Granularität der Datensperren ist, desto mehr Anfragen können parallel bearbeitet werden. In relationalen Datenbanksystemen können Sperren auf Tabellenebene erteilt werden. Da es aber auch vorkommt, daß auf verschiedene Zeilen zugegriffen werden soll, besitzen einige Systeme die Fähigkeit, zeilenweise zu sperren. Die feinste Einheit ist allerdings die Zelle in einer Zeile. Bei größeren Objekten, die auf viele Tabellen verweisen, müssen viele Sperren erteilt werden, wodurch Geschwindigkeitseinbußen die Folge sind.

Die Ereignis-Objekte in der Objectivity-Datenbank bei H1 besitzen eine flache Struktur. Diese setzt sich aus ganzen Zahlen, aus Fließkommazahlen sowie aus dynamischen Tabellen (Arrays) zusammen, deren Komponenten wiederum ganze oder Fließkommazahlen darstellen. Assoziationen mit anderen Objekten existieren insofern, daß die Run-Container und die Run-Objekte einander referenzieren. In Anbetracht der flachen Ereignisstruktur ist die Frage berechtigt, ob nicht besser eine relationale Datenbank eingesetzt werden kann. Es wäre sehr interessant, eine zum Objectivity-DIM äquivalente Schnittstelle zu einem RDBMS zu entwickeln, um so einen Vergleich vom objekt-orientierten zum relationalen Datenmodell ziehen zu können.

4.8.2 Objekt-relationale Datenbanken

Die heutzutage sich auf dem Markt befindlichen objekt-orientierten Datenbanken können in zwei verschiedene Evolutionen eingeteilt werden: als Erweiterungen von objekt-orientierten Programmiersprachen oder als Aufsatz auf eine relationale Datenbank [McHenry93] [Chaudhri93]. Einige Hersteller von relationalen Datenbanksystemen haben ihre Produkte bereits um das Konzept der Objekt-Orientiertheit erweitert, um einerseits dessen Vorteile bieten zu können, und um andererseits die sehr fundierte Relationalität zu bewahren. In diesem Fall handelt es sich um eine objekt-relationale Datenbank. Ein bekanntes Beispiel hierfür ist das Datenbanksystem von Oracle, welches bereits bei H1 für Slow-Control-Daten benutzt wird.

Objekt-relationale Datenbanken benutzen wie bisher das relationale Prinzip für die Speicherung der Daten. Das ORDBMS kann weiterhin mit relationalen Methoden angesprochen werden. Neu hinzugekommen ist eine Schnittstelle, die objekt-orientierte Operationen auf die relationale Ebene transformiert und die Konvertierung in Rückwärtsrichtung ebenfalls vornimmt [Wade98]. Eine herausfordernde Aufgabe dieser Schnittstelle ist es, komplexe Objekte in ihre elementaren Bestandteile aufzuspalten, so daß diese auf ein relationales Datenmodell abgebildet werden können. Ebenso müssen bei der Anforderung des gesamten Objektes die Teile wieder zusammengesetzt werden.

Falls ein DBMS ausschließlich für Umgang auf relationaler Basis konzipiert ist, so muß die Applikation auf der Seite des Klienten notgedrungen selbst für die Abbildung auf das relationale Datenmodell sorgen. Dies hängt von der Struktur des Objektes ab und braucht im allgemeinen viel Mühe und Zeit.

4.8.3 Objekt-orientierte Datenbanken

Im Gegensatz zu relationalen Datenbank-Management-Systemen gab es lange Zeit keine mathematischen Grundlagen für das objekt-orientierte Datenbankmodell. Aus diesem Grund gibt es auf dem Markt viele individuelle Ansätze mit unterschiedlichen Qualitäten. Daher wurde von verschiedenen Firmen der ODMG-Standard entworfen, der bereits in Abschnitt 4.2 erwähnt wurde. Dieser Standard versucht, eine theoretische Basis zu liefern, an der sich alle existierenden Datenbanksysteme mit objekt-orientiertem Konzept orientieren können. Als Konsequenz der verschiedenen existierenden Ansätze unterstützen einige Systeme nicht alle Prinzipien der Objekt-Orientiertheit, z.B. fehlende Verkapselung, welche ein grundlegender Bestandteil ist, oder keine Speicherung bzw. Persistenz von Methoden.

Abhängig von der Art der Anwendung können Objekte auch eine sehr große Struktur besitzen. Ein Vorteil von objekt-orientierten Datenbanksystemen ist der im Vergleich zu RDBMS oder ORDBMS weniger komplizierte Zugriff auf komplexe oder große Objekte. Die bei relationalen Systemen notwendigen Verfahren zur Komposition und Dekomposition von Objekten sind bei objekt-orientierten Datenbanken völlig redundant, da die Objekte nicht auf Tabellen umgebrochen werden müssen. Daher brauchen hier auch nur wenige Zugriffssperren erteilt werden. Ein ODBMS hat also allein durch seine Eigenschaft, Objekte atomar zu behandeln, einen Vorteil bezüglich komplexer Objekte im Vergleich zu einem RDBMS oder einem ORDBMS. Dieser Vorteil käme H1 mit Blick auf die in Kapitel 3.4 genannten Verbesserungsansätze im Rahmen eines Data-Warehouses zugute. Lösungen wie Data-Mart-Hierarchien, die Einführung von Metadaten oder die Abspeicherungen von Histogrammen würden durch die gegebene Freiheit der Komplexität der Objekte profitieren.

4.8.4 DBMS von anderen Herstellern

Während die Oracle-Datenbank ihre Wurzeln im relationalen Datenmodell hat, basieren die Konkurrenten auf dem objekt-orientierten Prinzip [Chaudhri96]. Seit kurzer Zeit hat Oracle davon ebenfalls Gebrauch gemacht und die Datenbank mit einem objekt-relationalen Front-End versehen. Objectivity Inc. hat ein White-Paper veröffentlicht [Wade98], in dem ein Benchmark zu einem RDBMS im Vergleich zu Objectivity/DB demonstriert wurde. Da das Schriftstück von der Firma selbst stammt, sind die gezeigten Resultate — wie nicht anders erwartet — sehr zum Vorteil des eigenen Produktes ausgefallen. Um sich selbst ein Bild von relationalen Datenbanksystemen neben Objectivity/DB machen zu können, sollten Benchmarks von Drittherstellern zu Rate gezogen werden, oder, soweit möglich, eigene Tests gefahren werden.

Das relationale DBMS Oracle existiert bei H1, daher wäre es die nächste Alternative zu Objectivity/DB. Allerdings sollte die Verwendung der Datenbank in der Hochenergiephysik wohlbedacht sein. In [POMC99] bei CERN ergab sich, daß Oracle wegen zu geringer Nachfrage bisher keine Unterstützung der Skalierbarkeit in den Peta-Byte-Bereich geplant hat. Für den LHC-Beschleuniger sei außerdem die Geschwindigkeit nicht effizient genug.

Kapitel 5

Die Schnittstelle zu Objectivity/DB

5.1 Anforderungen an die Schnittstelle

In Java Analysis Studio werden Daten zur Histogrammierung durch Data-Interface-Modules (DIM) zur Verfügung gestellt. Diese DIMs übergeben die Daten in einheitlicher Form an JAS. Jedem DIM bleibt es selbst überlassen, aus welcher Quelle es seine Daten bezieht. Es kann sie z.B. aus Dateien mit individuellem Dateiformat extrahieren, auf Bandstationen zugreifen, aus spezieller Hardware auslesen oder aus Datenbanken anfordern.

Das für H1 zu entwickelnde DIM soll dem Benutzer die Möglichkeit geben, Selektionskriterien für die Datenbankanfrage zu stellen und den gewünschten Datenbereich mit H1-spezifischen Angaben zu bestimmen. Die Anfrage soll an die Objectivity-Datenbank gestellt und das Ergebnis an JAS zurückgegeben werden. Die Selektionskriterien richten sich nach der Datenorganisations-Hierarchie von H1, die in Kapitel 4 auf Objectivity/DB abgebildet wurde. Ein Forscher will zunächst das Jahr der Datennahme angeben. In Frage kommen die Jahre von 1995 bis heute. Zu jedem Jahr gibt es verschiedene Versionen der Ereignisse: Version 1 (DST1) enthält die Werte, die nach den Experiment-Messungen rekonstruiert und gespeichert wurden. Nachdem Korrekturen und Qualitätsverbesserungen stattgefunden haben, werden diese Daten ein zweites Mal rekonstruiert und unter Version 2 (DST2) archiviert. Diese Version wird von den Anwendern oft bevorzugt. Innerhalb eines Jahres gibt es etwa 20000–25000 Runs, von denen eine Auswahl in die Analyse eingehen soll. Dies ist normalerweise ein zusammenhängender Bereich, der mit einem Start- und einem Endwert beschrieben werden kann. Runs werden mit einer laufenden Nummer benannt und identifiziert. Zum Beispiel wurden im Jahre 1997 die Runs 177920–195617 und 195667–200407 durchgeführt[1].

Für eine Analyse sind die Angaben über einen Run-Bereich, das Datennahme-Jahr und die DST-Version zu grob. Physiker wollen auch Schnitte und Eingrenzungsbedingungen angeben können, wodurch wesentlich detailliertere Analyse-Schritte möglich sind.

[1]Diese Information wurde der internen H1-DST-Internetseite entnommen.

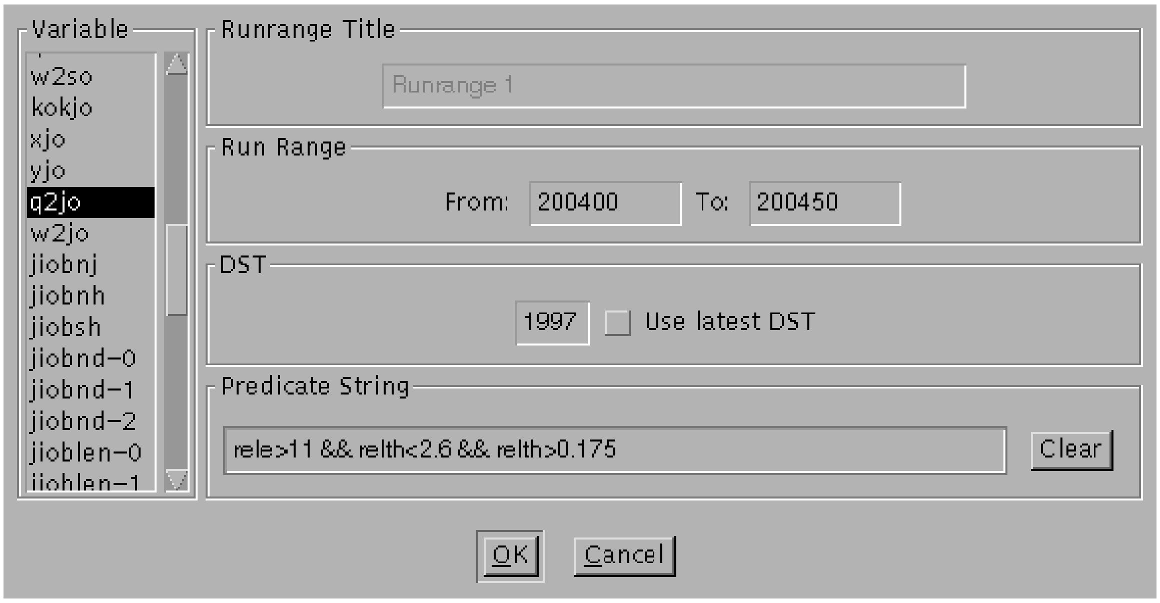


Abbildung 5.1: JAS-Objy-DIM: Grafische Benutzerschnittstelle

5.2 Funktionsweise des DIMs für Objectivity/DB

5.2.1 Visuelle Präsentation

Das JAS-DIM für Objectivity/DB besteht aus zwei größeren Teilen. Dies sind die Routinen, die die Datenbank ansprechen, und die grafische Benutzerschnittstelle, mit deren Hilfe der Anwender die Zugriffe parameterisieren kann. Diese wird im folgenden erklärt.

Die Run-Bereiche des Objy-DIMs sind im Baum auf der linken Seite des Hauptfensters von JAS zu finden. Jedem Run-Bereich wird genau eine Parameterisierung zugeordnet, die mit Hilfe des Dialog-Fensters gewählt werden kann (Abb. 5.1). Die Liste auf der linken Seite des Dialoges enthält alle physikalischen Variablen, die in den Ereignis-Objekten in der Datenbank vorhanden sind. Die Objekte können in Zukunft erweitert werden, und das Objy-DIM wird diese Liste an die Änderungen automatisch anpassen (mehr dazu siehe unten, Abschnitt 5.3.1). Die Variablennamen sind leider nicht selbsterklärend. Dies hängt damit zusammen, daß physikalische Größen bei vielen Forschern in H1 unter ihren Fortran-Variablennamen bekannt sind. Es ist aber technisch durchaus möglich, zumindest in dieser Liste, die dem Anwender präsentiert wird, aussagefreundlichere Bezeichnungen zu integrieren.

Es ist durch die Variablen-Auswahl im Options-Dialogfenster möglich, für jeden Run-Bereich die zu histogrammisierende Variable individuell einzustellen. Die Anwender können also in mehreren Run-Bereichen nicht nur verschiedene, sondern auch dieselben Variablen plazieren, um diese bei unterschiedlichen Eingrenzungskriterien (siehe unten) betrachten zu können. Dies ist zum Beispiel mit dem PAW-DIM von JAS nicht möglich.

Die Parameter "From" und "To" des Run-Bereiches beziehen sich auf die Nummern der Runs, deren Ereignisse in die Analyse einfließen sollen. In der Regel wird dieser Bereich auf mehrere tausend Runs ausgedehnt. Das DST-Jahr gibt das Jahr an, aus dem der Run-Bereich herangezogen werden soll. Die Anwender möchten oft möglichst aktuell rekonstruierte Ereignisdaten benutzen, daher gibt es einen Schalter, der das jüngste DST herausfinden soll.

Der Titel des Run-Bereiches kann zur Zeit nicht verändert werden. Das hat damit zu tun, daß die Run-Bereiche im Baum auf der linken Seite des Hauptfensters von JAS feste Elemente sind, die nicht im Nachhinein verändert werden können, sobald sie einmal erstellt wurden. Die allerersten Versionen des JAS-Objy-DIMs haben versucht, dynamische Baumelemente zu ermöglichen, so daß der Benutzer die Namen der Blätter verändern konnte. Dadurch waren jedoch tiefe Eingriffe in verschiedene Klassen des JAS-Paketes nötig, und das Ziel wurde letztendlich nicht ganz erreicht. Daher wurde bei dem DIM-Dialog das Eingabefeld des Run-Titels zunächst deaktiviert, in der Hoffnung, daß dies in einer zukünftigen JAS-Version möglich sei. Die Fähigkeit von JAS, den Run-Titel — unabhängig vom Selektionsbaum im Hauptfenster — nachträglich im Histogramm ändern zu können, war hilfreich.

Im Eingabefeld “Predicate String” können Selektionskriterien und Schnittparameter eingegeben werden. Sie bestehen aus arithmetischen und logischen Bedingungen, die miteinander kombiniert werden können. Die Syntax orientiert sich an der Programmiersprache C. Zeichenketten-Variablen können mit Konstanten verglichen werden oder durch reguläre Ausdrücke auf bestimmte Muster geprüft werden. Physiker möchten normalerweise ihre Analysen auf spezielle Spektren oder Teilbereiche von Ereignissen fokussieren, um ein möglichst genaues Ergebnis zu erzielen. Mit dem Eingabefeld für die Schnittparameter kann eine hohe Flexibilität erreicht werden. Der Nachteil dabei ist allerdings, daß dies keine intuitive Eingabemöglichkeit ist. Benutzer müssen statt dessen lernen, welche Syntax zu befolgen ist. Ein Beispiel für eine Selektions-Zeichenkette wäre:

```
rele>11 && relth>0.175 && relth<2.6 && ( zvtx<-29 || zvtx>30 )
```

Ein weiterer Nachteil ist, daß die im Dialogfenster präsentierte Eingabezeile schnell unübersichtlich wird. Sie ist momentan noch zu klein für umfangreiche Anfrage-Bedingungen, die bei H1 in der Größenordnung von 10–20 atomaren Bedingungen (Anfragedimensionen) liegen. Eine atomare Bedingung ist hier als eine arithmetische Relation zu verstehen. Das obige Beispiel besteht aus 5 atomaren Bedingungen. Das DIM könnte an dieser Stelle verbessert werden, indem die Textzeile durch ein mehrzeiliges Eingabefeld ersetzt wird.

Seit Mitte Februar 1999 wurde eine Alpha-Testversion von Java Analysis Studio zusammen mit dem Objectivity-DIM für die Benutzer der H1-Kollaboration installiert. Damit sollten die Anwender an die Bedienungsoberfläche von JAS vertraut gemacht werden. Das DIM kann leider nur zu Testzwecken und nicht für reale Analysen oder Data-Quality-Checks benutzt werden, da die Datenbank sich noch im Entwicklungs-Test-Zyklus befindet und die Ereignisstruktur erst zu etwa 25% implementiert ist (siehe unten, Abschnitt 5.2.3).

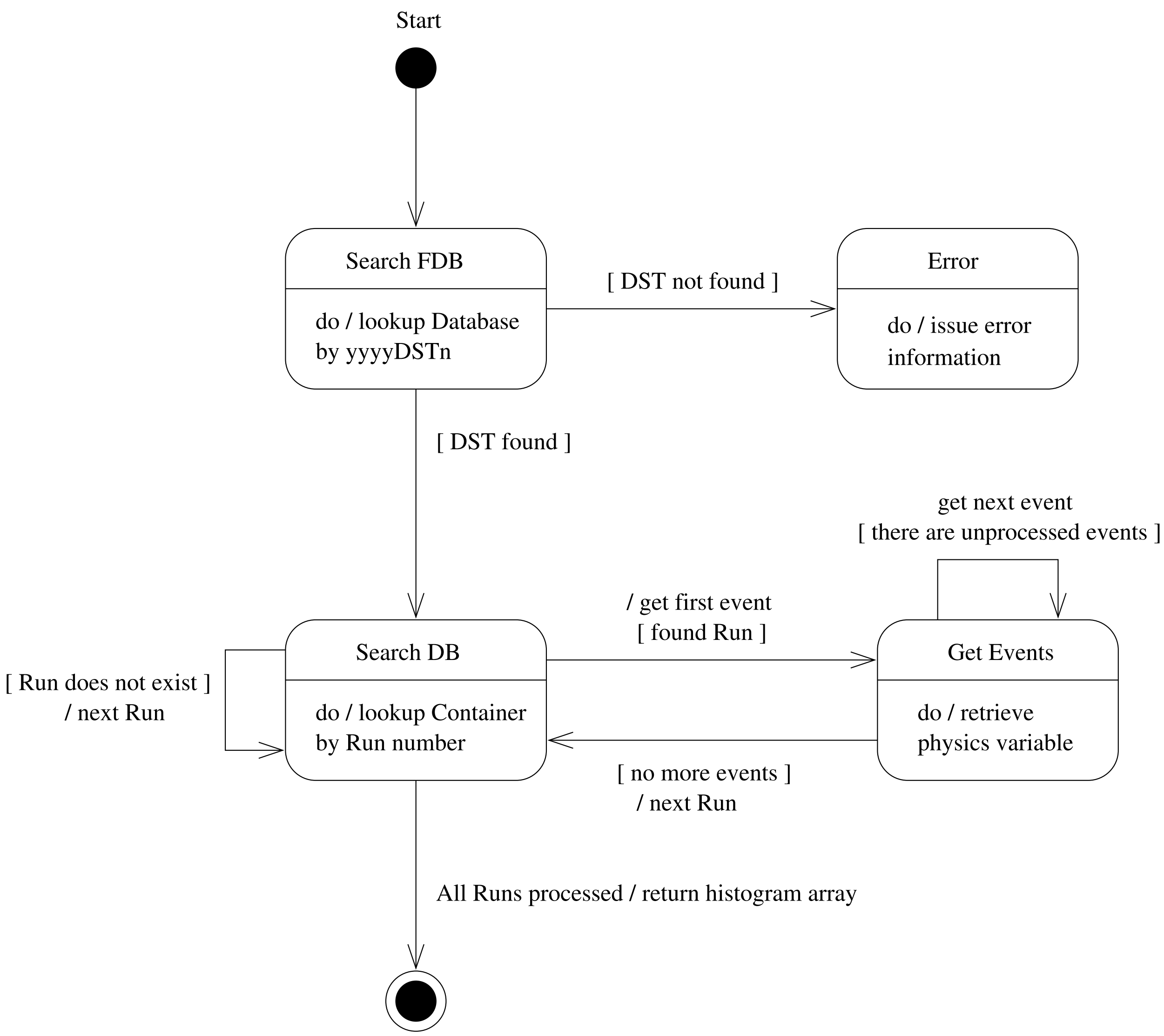


Abbildung 5.2: Zugriff des DIMs auf Objectivity/DB

5.2.2 Interne Aktivitäten

Sobald eine Datenbankanfrage durch das JAS-DIM initiiert wird, wird die Datenbank geöffnet und für Lesezugriffe vorbereitet. Aus dem gegebenen Jahr der Datennahme und der DST-Versionsnummer wird der Name des DST-Objektes aus der Datenbank zusammengesetzt. Zum Beispiel gibt es ein Database-Objekt namens „1997DST2“, welches die Runs aus dem Jahre 1997 in der zweiten Rekonstruktion enthält. Über diesen Namen wird das DST-Objekt in der Datenbank gesucht. DST-Objekte sind Instanzen der Database-Klasse in Objectivity (`ooDBObj`) und enthalten Run-Objekte.

Run-Objekte sind der Container-Klasse (`ooContObj`) zugeordnet. Der Namen eines Run-Objektes entspricht der Nummer des Runs selbst. Ein Diagramm der Klassen von Objectivity und deren Verwendung für H1 ist in Abbildung 4.2 zu sehen. Nachdem das DST-Objekt aus der Datenbank geladen wurde, müssen alle existierenden Runs innerhalb des von Benutzer gewünschten Run-Bereiches auf ihre Ereignisse untersucht werden. Dazu wird eine äußere Schleife über diese Runs gefahren.

Ein Ereignis entspricht einem elementaren Objekt in Objectivity (`ooObj`). Ein Run bzw. ein Run-Objekt enthält etwa 500–2000 Ereignisse. Das DIM iteriert nun in jedem Run über alle Ereignisse, auf die die gewünschten Selektionskriterien zutreffen. Diese Kriterien sind durch den Benutzer beim Eingabe-Dialog in Form einer Zeichenkette angegeben worden, welche nun ohne Bearbeitung direkt an Objectivity/DB weitergereicht werden. Die Abwicklung der Schnittkriterien wird ausschließlich von der Datenbank vorgenommen, diese Aufgabe braucht nicht vom DIM erledigt zu werden. In einer Schleife wird jedes einzelne Ereignis-Objekt aus der Datenbank gelesen, und der Inhalt der zu histogrammierenden Variable wird verarbeitet (Abb. 5.2). Leider bietet Objectivity/DB in dessen Java-Klassen keine Funktionalität, die das vollständige Run-Objekt ausliest und das Gesamtergebnis für die gewählte Variable an das DIM zurückgibt.

Direkter Zugriff oder RMI-Server

Die Objectivity-Datenbank wird zentral von einem Rechner aus gesteuert: der Lock-Server koordiniert die Zugriffsrechte der Applikationen auf die Datenbank. Eine Anwendung muß die Datenbankanfragen in jedem Fall an diesen Server richten. Die Objectivity-Schnittstelle für JAS wurde so entwickelt, daß sie in zwei verschiedenen Konfigurationen betrieben werden kann.

In der ersten Konfiguration greift JAS direkt auf den Lock-Server zu (Abb. 5.3). Dies ist für den Fall vorgesehen, daß der JAS-RMI-Server (siehe unten) aus beliebigen Gründen ausgefallen ist. Der Nachteil dieser Methode ist, daß von den angeforderten Ereignissen die vollständigen Daten — das sind alle physikalischen Variablen der Ereignisse — über das Netzwerk übertragen werden.

Die zweite Konfiguration der Objectivity-Schnittstelle ist so beschaffen, daß die Datenbank durch eine eigens dafür entwickelte, alleinstehende Java-Applikation angesprochen wird. Die Applikation wurde ebenfalls im Rahmen dieser Diplomarbeit erstellt und stellt im wesentlichen eine Hülle für die Objectivity-Zugriffe dar. Die Java-Applikation wird zur

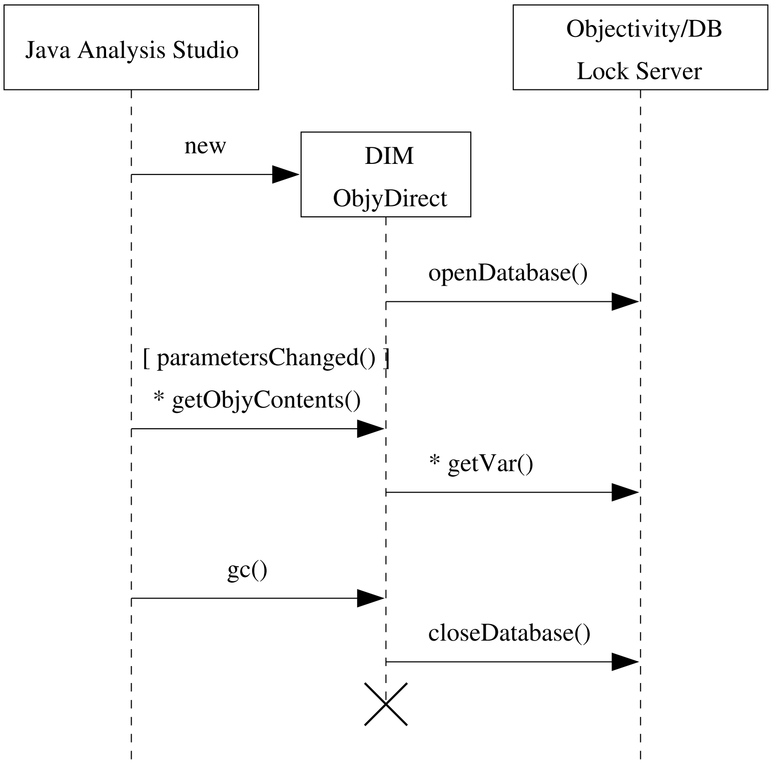


Abbildung 5.3: JAS-Objy-DIM: Direkter Datenbankzugriff

Datenaufbereitung zwischen JAS und den Lock-Server geschaltet (Abb. 5.4). Die Daten werden insofern bereinigt, daß anstelle der vollständigen Ereignisdaten von Objectivity/DB das reine Anfrage-Ergebnis wieder an JAS zurückgegeben wird. Damit werden die zu den JAS-Klienten transferierten Datenmengen wesentlich reduziert. Der Unterschied zur Konfiguration für Direkt-Zugriff liegt darin, daß hier die Datenbank-Anfragen mit Hilfe des RMI-Protokolls[2] von Java ausgeführt werden. Ein Java-Programm kann so beschaffen sein, daß es Prozeduren auf einem RMI-Server ausführen kann, der üblicherweise auf einem anderen Rechner gestartet ist. Der RMI-Server für JAS stellt lediglich auf seinem Rechner die Objectivity-Funktionsaufrufe zur Verfügung. Das bedeutet, daß ein JAS-Klient nun nicht mehr selbst auf die Datenbank zugreift. Statt dessen stellt er die Anfrage an den RMI-Server. Dieser sollte vorzugsweise auf demselben Rechner betrieben werden, auf dem sich der Lock-Server von Objectivity befindet, um die Datenbankanbindung so schnell wie möglich zu halten. Das Ergebnis der Datenbankanfrage wird schließlich über das Netzwerk in essentieller Form an JAS zurückgegeben. Die übertragenen Daten sind nun wesentlich kleiner als bei der Konfiguration zur Direktanbindung von JAS an die Datenbank, da nun nicht mehr alle Variablen eines Ereignisses transferiert werden, sondern nur die eine Variable, die durch das DIM in JAS zur Analyse gewählt wurde. Bei einem aktuellen Ereignisumfang von 58 Variablen — später etwa viermal so viel, mehr dazu in Abschnitt 5.2.3 — ergibt sich eine Ersparnis um den Faktor 58 (bzw. 26, da die angeforderte Variable als `double`

[2]Remote Method Invocation (RMI)

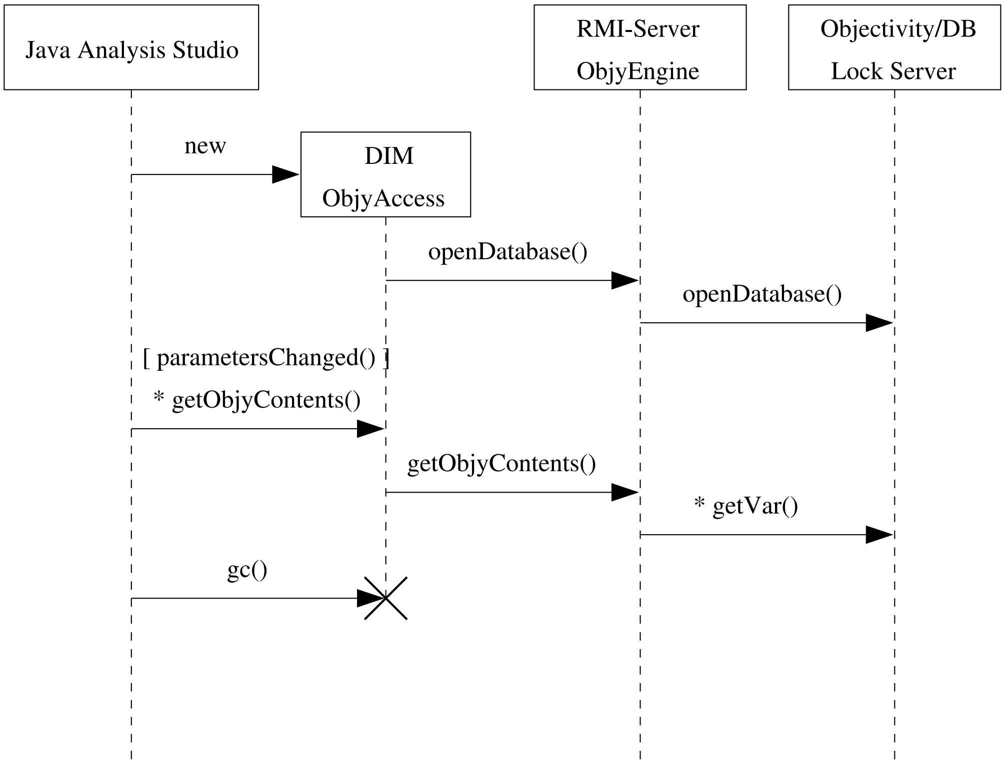


Abbildung 5.4: JAS-Objy-DIM: Datenbankzugriff über den RMI-Server

zurückgegeben wird statt `float` oder `int`).

Der Benutzer muß sich in JAS für eine der beiden Zugriffsarten entscheiden: entweder für direkten Zugriff auf die Datenbank oder für den Zugriff durch den RMI-Server. Das DIM stellt bei seiner Initialisierung eine Auswahlmöglichkeit für diese beiden Konfigurationen zur Verfügung. Die Oberfläche in JAS zur Bearbeitung von Histogrammen bleibt in jedem der beiden Fälle gleich. Die Datenbank-Zugriffsart ist also während der Histogrammierung für den Benutzer transparent.

5.2.3 Java-Klassen

In Abbildung 5.5 ist eine Übersicht aller wichtigen Java-Klassen des DIMs dargestellt. Die involvierten Klassen von JAS befinden sich im „jas“-Package, die in der Diplomarbeit entwickelte Objectivity-Schnittstelle besteht aus den beiden Packages „objy“ und „h1objy“, und das Java-Frontend der Datenbank ist mit dem Package „objy.db.app“ angedeutet. Die wichtigsten Klassen von Java Analysis Studio sind im „jas“-Package enthalten. Es gibt höchstens eine Instanz von `JASJob`. Jeder Knoten und jedes Blatt des Job-Baumes wird durch ein `TreeNode`-Objekt verkörpert. Dieser Baum ist grafisch auf der linken Seite des Hauptfensters von JAS zu sehen (Abb. A.4). Jedem Baumelement ist weiterhin genau ein `TreeAdaptor` zugeordnet. In dieser Klasse ist festgelegt, welche Aktionen der Benutzer mit den Baumelementen auslösen kann. Zum Beispiel kann er per Knopfdruck Histogramme anzeigen oder übereinanderlegen lassen, oder ein Popup-Menü mit weiteren Optionen kann

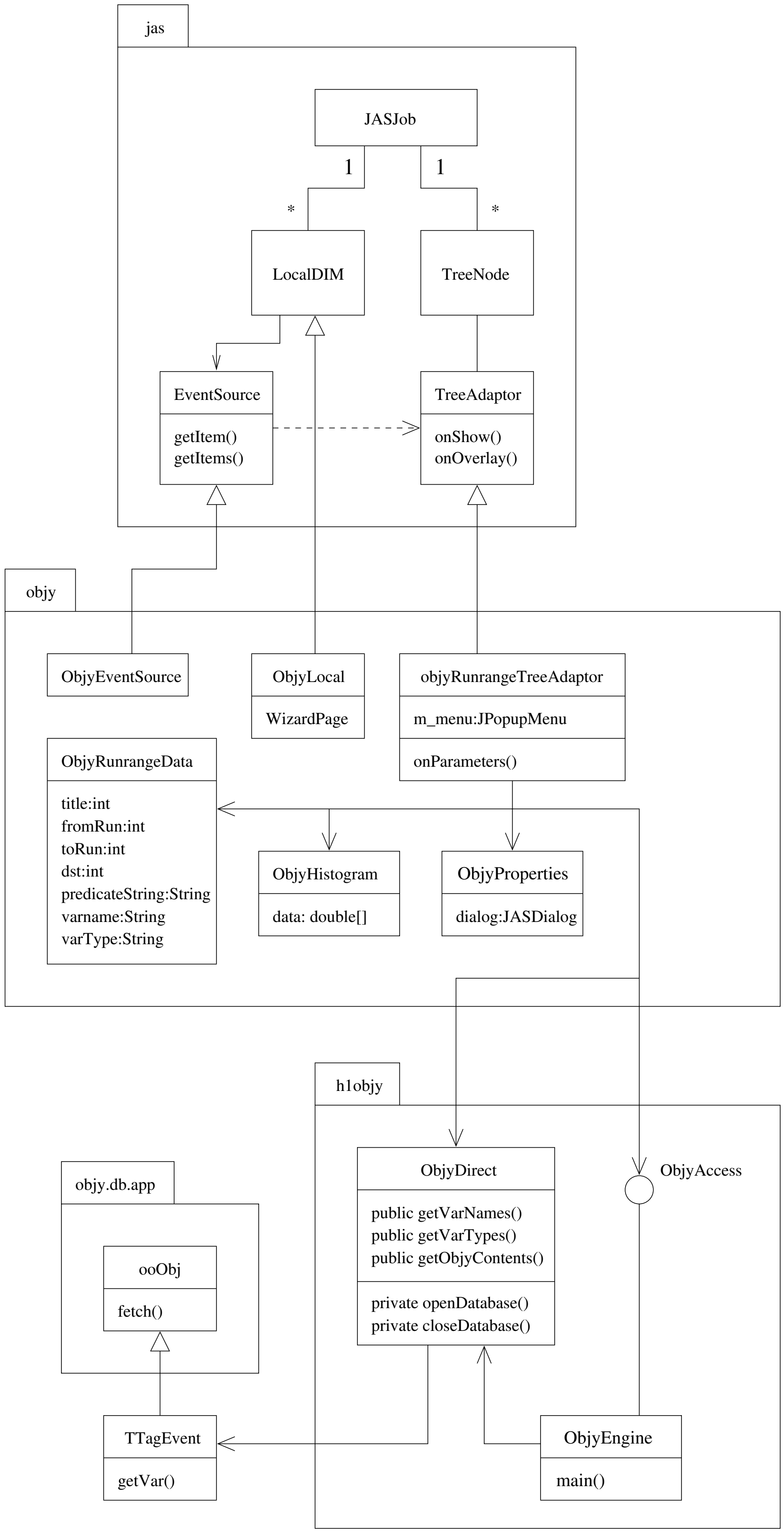


Abbildung 5.5: Die Java-Klassen des DIMs

TTagEvent

public irun: int
public ievent: int
public weight: float
public ibunch: int
public intvtx: int
public zvtx: float
public izvtx: int
public dzvtx: float
public zvtxno: int
public ieltyp: int
public rele: float
public relth: float
public relph: float
public koke: int
public xe: float
public ye: float
public q2e: float
public w2e: float
public kokso: int
public xso: float
public yso: float
public q2so: float
public w2so: float
public kokjo: int
public xjo: float
public yjo: float
public q2jo: float
public w2jo: float
public jiobnj: int
public jiobnh: int
public jiobsh: int
public jiobnd: float[3]
public jioblen: float[3]
public jioblpt: float[3]
public jioblph: float[3]
public jiobben: float[3]
public jiobbpt: float[3]
public jiobbrp: float[3]
public jiobbph: float[3]

getVar(): double
getVariableNames(): Vector
getVariableTypes(): Vector

Abbildung 5.6: Java-Klasse TTagEvent, ein Basic-Object in Objectivity/DB erscheinen. Die Klasse `objyRunrangeTreeAdaptor` im „objy“-Package ist daher ein Erbe von `TreeAdaptor`, damit von dort aus der Dialog für die Anfrage-Parameter aufgerufen werden kann.

Das DIM teilt sich in zwei Java-Packages: In Package „objy“ befinden sich die Klassen, die mit Java Analysis Studio verbunden sind. Dazu gehören `ObjyEventSource` zur Initialisierung des DIMs, `objyRunrangeTreeAdaptor` für die Handhabung des JAS-Baumes im Hauptfenster, oder `ObjyProperties` für das Dialog-Fenster mit den Anfrage-Parametern. Package „h1objy“ besteht aus Klassen, die sich um den Zugriff auf die Datenbank kümmern. Der RMI-Server befindet sich in `ObjyEngine`. Die Klasse `ObjyDirect` ist zentral für den Zugriff auf die Datenbank zuständig. Es fällt auf, daß diese Klasse sowohl vom RMI-Server als auch von `objyRunrangeTreeAdaptor` im JAS-Kontext benutzt wird. Dadurch ist das Problem gelöst worden, Datenbank-Zugriffe entweder direkt von JAS oder über den RMI-Server erfolgen zu lassen. Das Interface `ObjyAccess` wurde im Rahmen des RMI-Protokolls dem Package hinzugefügt und gewährleistet, daß das DIM im letzteren Fall die Zugriffsmethoden kennt. Durch dieses Konzept befindet sich der Code in `ObjyDirect`, der auf die Datenbank zugreift, an einer zentralen Stelle.

Die Ereignis-Klasse `TTagEvent` ist in den Run-Containern der Objectivity- Datenbank zu finden. Sie enthält die physikalischen Variablen eines Ereignisses. Eine aktuelle Auflistung ist in Abbildung 5.6 zu sehen. Die Summe des Speicherplatzes aller Variablen dieser Klasse beläuft sich momentan auf etwa 250 Bytes (58 Variablen). Bisher sind noch nicht alle geplanten Variablen in dieser Klasse enthalten, da das Füllprogramm für die Datenbank noch nicht komplettiert ist. Wenn alle gewünschten Variablen integriert sind, wird das Ereignisobjekt knapp 1kB umfassen (siehe Tabelle 4.1).

Ganzzahlige Größen sind in den auf Band gelagerten DST-Ereignissen mit verschiedenen Datentypen deklariert (8-/16-/32-Bit Integer). Sie werden in der abgebildeten Klasse bei Objectivity/DB durch den ganzzahligen 32-Bit-Datentyp `int32` vereinheitlicht, der sowohl bei Java als auch bei C++ auf `int` abgebildet wird. Ebenso ist der 32-Bit-Typ `float32` eine Kompromißlösung verschiedenartiger Fließkomma-Datentypen, die in den DST-Ereignissen auf Band vorkommen. Neben den beiden elementaren Typen `int` und `float` gibt es Tabellen (Arrays), die mit Hilfe des Objectivity-Typs `ooVArray` abgelegt sind. Dieser Typ wird in C++ auf das gleichnamige Objectivity-Handle abgebildet, und in Java kann auf ihn durch das gewohnte Array zugegriffen werden, welches dort dynamischer Natur ist. In der TTagEvent-Objektstruktur beinhalten diese Tabellen allerdings maximal 3 Elemente. Weitere Daten- oder Objekttypen sind nicht in der Ereignisstruktur vorhanden. Die Methode `getVar()` wird von `ObjyDirect` zur Extraktion der Histogramm-Variablen benutzt. `TTagEvent` muß ein Erbe der Objectivity-Klasse `ooObj` sein, um in der Datenbank als elementares Objekt persistent gemacht werden zu können.

5.3 Schwierigkeiten und Lösungen

5.3.1 Erweiterung der Ereignisdaten

Mit Sicherheit wird es in der Zukunft des OOP-Projektes vorkommen, die Variablenmenge der Ereignis-Objekte in der Datenbank zu erweitern. Gerade während der Testphase, die zur Zeit noch aktuell ist, werden die Variablen noch nicht hundertprozentig festgelegt sein. Bei objekt-orientierten Datenbanken gibt es zwei Methoden zur Erweiterung von Objekten: Schema-Entwicklung und Schema-Versionierung. Die erstere Methode verändert anhand verschiedener Objektstrukturen der Benutzer dasselbe Objekt in der Datenbank. Mit Schema-Versionierung wird bei jeder Änderung eines Objektes ein neues Objekt in der Datenbank erzeugt. Man würde folglich nach mehreren Änderungen eine baumartige Versionsstruktur erhalten.

Die Objectivity-Datenbank benutzt das Prinzip der Schema-Entwicklung. In der zugreifenden Anwendung kann festgelegt werden, ob die Änderungen am Objekt in der Datenbank durchgeführt werden, oder ob sie sich ausschließlich auf das Objekt innerhalb der Anwendung auswirken. Bei Objectivity/DB wird dies in der Schema-Politik[3] der aktuellen Transaktion festgelegt. Es sei darauf hingewiesen, daß bei den Objectivity-Java-Klassen die Änderung von Objekten in der Datenbank durch Schema-Entwicklung voreingestellt ist. Die JAS-Schnittstelle hatte in der ersten Testversion eine zu Testzwecken verkürzte Objektstruktur, so daß — trotz explizitem Lesezugriff (!) — anschließend die Objekte in der Datenbank entsprechend verändert waren. Die Datenbank mußte vom Administrator neu erzeugt werden. Es ist daher sehr wichtig, bei einer Applikation mit Lesezugriff (wie etwa JAS) dennoch die Schema-Politik derart einzustellen, daß Objekte in der Datenbank nicht angepaßt werden, falls sich die Strukturen unterscheiden.

In der aktuellen Testinstallation von Objectivity/DB bei H1 gehören Ereignis-Objekte der Klasse `TTagEvent` an (siehe technische Dokumentation am Ende der Arbeit). Sie enthält momentan 31 Variablen und 9 dynamische Tabellen (ooVArrays), welche einen Umfang von maximal 3 Elementen haben. Diese Menge von Variablen soll zukünftig erweitert werden können, so daß eine Lösung gefunden werden mußte, die den Aufwand einer Erweiterung oder Änderung so gering wie möglich halten soll.

Das JDK-Klassenpaket `java.lang.reflection` stellt Methoden zur Verfügung, die es erlauben, zur Laufzeit Informationen über unbekannte Variablen und Methoden von Klassen zu erhalten. Diese Technik wird in dem Objectivity-DIM verwendet, und durch sie ist der Aufwand, Ereignisvariablen der Objektstruktur hinzuzufügen, auf ein Minimum reduziert worden. Der Administrator muß lediglich die gewünschte Ereignisvariable an einer Stelle innerhalb der Ereignisklasse deklarieren. Die neue Struktur wird automatisch erfaßt und dynamisch in JAS eingebunden werden. Leider stellt das Java-Frontend von Objectivity/DB keine Möglichkeit zur Verfügung, das Ereignisschema aus der Datenbank zu extrahieren. Aus diesem Grund muß die Ereignisklasse manuell angepaßt werden.

[3] In der Dokumentation unter "schema policy" bekannt

5.3.2 Optimierung des Java-Zugriffs

Die Ereignisse werden durch eine Schleife einzeln aus der Datenbank gelesen (Abb. 5.2). Für jede extrahierte physikalische Variable wird ein Objekt der Klasse `java.lang.Double` neu angelegt, welches in einer dynamischen Liste (`java.util.Vector`) abgelegt wird. Die Neuanlegung der Variablenobjekte und deren Speicherung in der Liste sind zwei Operationen, die optimiert werden könnten. In JDK 1.2 gibt es eine neue Klasse `java.util.ArrayList`, die laut [Heckner98] einen Geschwindigkeitsvorteil von 16% im Vergleich zu `Vector` bringt. Dazu müste aber zuerst JDK 1.2 installiert werden. Die Kompatibilität zu Java Analyis Studio würde dann in Frage stehen und müßte überprüft werden. Im Vergleich zu den Datenbankzugriffen ist Java jedoch relativ schnell. Geschwindigkeitssteigerungen sind daher eher bei der Organisation von Objectivity/DB angebracht.

5.3.3 Java-Data-Server oder RMI-Server

Eine von den Autoren angepriesene Fähigkeit von Java Analysis Studio ist die Bereitstellung von Daten sowohl aus lokalen Quellen als auch von entfernten Rechnern. Letzteres kann durch einen Java-Data-Server (JDS) realisiert werden, der im JAS-Paket enthalten ist. Damit erheben sich die Fragen, ob dies nicht günstig für das DIM zu Objectivity/DB wäre, und warum statt dessen ein eigener RMI-Server entwickelt wurde. Es hatte sich herausgestellt, daß die internen Java-Klassen von JAS, die sowohl von JAS als auch vom Java-Data-Server verwendet werden, nicht serialisierbar sind, sobald sie Klassen von Objectivity/DB enthalten. Der Java-Data-Server verwendet das RMI-Protokoll; dadurch ist Serialisierbarkeit für einige Klassen in Java Analysis Studio unbedingt notwendig. Somit war ein Java-Data-Server für Objectivity/DB nicht machbar. Der RMI-Server dagegen weist dieses Problem nicht auf, da hier nur das Anfrage-Ergebnis zur Übertragung durch das Netzwerk serialisiert wird, nicht aber die Objectivity-Klassen. Dies war der Hauptgrund, den eigenen RMI-Server zu entwickeln.

5.3.4 Histogrammierung von Elementen aus dynamischen Arrays

In der Ereignis-Klasse `TTagEvent` gibt es dynamische Tabellen (Arrays) der Java-Elementartypen `int` und `float` (Abb. 5.6). Dynamik bedeutet hier, daß die Tabelle jederzeit auf eine feste, aber beliebige Größe initialisiert werden kann. Es gibt in der Klasse momentan neun tabellarische Variablen mit einer maximalen Kardinalität von 3. Sie können also auch 0, 1 oder 2 Elemente enthalten. Das sich hierbei ergebende Problem tritt auf, wenn das n-te Element einer tabellarischen Variable histogrammiert werden soll, aber in einigen Ereignissen diese Variable nur höchstens bis zu $n-1$ Elemente enthält. Das DIM würde versuchen, auf das n-te Element zuzugreifen, was bei Java zu einer Ausnahmebehandlung während der Laufzeit (Runtime-Exception) führt.

Ein Beispiel soll das Problem verdeutlichen: In einem Ereignis A habe die Tabelle `jiobnd` drei Elemente. In einem Ereignis B habe sie dagegen nur ein Element. Der Physiker stellt eine Anforderung über eine Menge von Ereignissen an die Datenbank, deren Kriterien auf die Ereignisse A und B zutrifft. Das zweite Element der Variable `jiobnd` soll in dem Histogramm aufgetragen werden. Dazu versucht das DIM, aus allen Ereignissen genau dieses Element zu lesen. Bei Ereignis A funktioniert dies, denn `jiobnd` besitzt hier ein zweites Element, da `jiobnd` in Ereignis A insgesamt drei Elemente hat. In Ereignis B hat `jiobnd` jedoch nur ein Element. Das gesuchte zweite Element dieser Variable existiert nicht, und Java antwortet mit einer Fehlermeldung.

Dieses Problem wurde mit Hilfe der Selektionskriterien bei der Anfrage von Ereignissen gelöst. In Objectivity/DB werden diese Kriterien in Form einer Zeichenkette übergeben, deren Inhalt aus Bedingungen und Verknüpfungen mit Objektvariablen besteht. Die Notation orientiert sich an der Sprache C. Die Ereignis-Objektstruktur von H1 enthält eine Variable `jiobnj` (Abb. 5.6), die die Anzahl der Jets darstellt. Diese ist gleichbedeutend mit der Anzahl der Elemente in den tabellarischen Variablen, deren Namen das Präfix `jiob` besitzen. Die Variable `jiobnj` wird daher zum Ausschluß solcher Ereignisse eingesetzt, die nicht genügend Elemente in den Tabellen enthalten. Dies wird durch die Erweiterung der Zeichenkette für Selektionskriterien durch eine entsprechende Bedingung erzielt.

Die Lösung soll mit einem Beispiel erhellt werden. Die Tabellen eines Ereignisses A seien alle 2 Elemente groß. Die Variable `jiobnj` würde also den Wert 2 enthalten. Der Benutzer möchte das dritte Element der Variable `jiobnd` mit den Selektionskriterien X histogrammieren. Beim Zugriff auf dieses Variablenelement in Ereignis A würde das DIM eine Fehlermeldung erfahren. Um dies zu umgehen, werden die Selektionskriterien mit dem Ausschluß aller Ereignisse erweitert, die weniger als 3 Elemente in den Tabellen aufweisen:

(X) `&& jiobnj > 2`

Auf diese Weise werden alle Ereignisse gefiltert, die zu einer Fehlermeldung beim DIM führen würden. Davon ist auch Ereignis A aus dem Beispiel betroffen, da dort `jiobnj==2` ist. Die Manipulation der Kriterien wird stets dem Anwender vorenthalten.

5.4 Andere Schnittstellen

5.4.1 Caltech/CMS

Bei Caltech[4] wurde bereits ein Objectivity-DIM für JAS in Zusammenarbeit mit Experiment CMS bei CERN entwickelt [Bunn98]. Dieses DIM ist jedoch auf ein völlig anderes Projekt bezogen und entspricht daher nicht den Anforderungen von H1 (siehe Kapitel 2.1). Zum Zeitpunkt der Überprüfung besaß es keine grafische Benutzerschnittstelle mit Konfigurierungsmöglichkeiten und konnte daher nur auf fest vorgegebene Datenbankinhalte zugreifen. Das DIM verwendet außerdem Objektstrukturen, die bei H1 nicht verwendet werden. Trotz seiner Unbrauchbarkeit für H1 konnte das Caltech-DIM dennoch als gutes Beispiel für Java-Zugriff auf die Objectivity-Datenbank dienen.

[4]California Institute of Technology, Caltech Particle Theory Group, http://www.theory.caltech.edu

Kapitel 6

Bewertung

6.1 Veränderungen durch Java Analysis Studio

In diesem Abschnitt werden die Neuerungen genannt, die durch Java Analysis Studio und das JAS-Objectivity-DIM im Vergleich zur bisherigen Analyse-Entwicklung hinzugekommen sind.

Vorteile

Die Fähigkeiten von JAS schaffen eine flexible WYSIWYG-Oberfläche. Im Gegensatz dazu erhält der Benutzer in der bisherigen Umgebung mit Fortran-Software nur statische Ausgaben, die durch mehrere, aufwendige Iterationen des Analysevorganges mit abgeänderten Parametern verbessert werden können. Durch JAS wird insbesondere der Vorgang der Veränderung der Balkenbreite in Histogrammen (Rebinning) drastisch beschleunigt. Dies bestätigten die Rückmeldungen von H1-Forschern, die bei einer etwa 80–100-köpfigen H1-Versammlung die Vorführung von JAS und dem Objectivity-DIM miterlebt haben.

Die Möglichkeit zur Parameterisierung der Datenbankzugriffe durch das Jas-Objectivity-DIM entlastet den Benutzer völlig von der Notwendigkeit, Programmpakete durch eigene Routinen zu spezialisieren. Selektionskriterien können direkt der Datenbankanfrage mitgegeben werden und brauchen nicht mehr durch Fortran-Quelltext implementiert zu werden. Damit entfallen Risiken und Software-Entwicklungsprobleme.

Ein weiterer Vorteil von Java Analysis Studio ist, daß das Projekt von Physikern bei SLAC entwickelt worden ist. Somit ist Kompetenz auf dem Gebiet der Elementarteilchenphysik gesichert, so daß das Gewicht und die Bedeutung von Anforderungen an die Software mit eigenem Interesse und eigenen Erfahrungen beurteilt werden können.

Nachteile

Einige Nachteile von Java Analysis Studio sind leider dadurch bedingt, daß die Software in Java implementiert wurde. Dem Benutzer fällt zuerst der langsame Programmstart auf. Es dauert etwa 20–40s, bis die Oberfläche benutzt werden kann[1]. Bei entsprechender

[1] Dies ist der ungefähre Zeitraum, den JAS vom Shell-Aufruf bis zum Öffnen des Hauptfensters benötigt. Das verwendete System war ein Sun-UltraSPARC Rechner.

Rechnerbelastung (z.B. durch Jobs oder andere JAS-Prozesse), die bei H1 nicht selten ist, dauert es entsprechend länger. Zu lange Wartezeiten verunsichern den Benutzer; er schöpft Verdacht, das Programm sei hängengeblieben. Dies wurde bei ersten Tests von JAS bei H1-Anwendern festgestellt. Dieser Nachteil ist jedoch als systematisches Problem zu betrachten, da er durch die Sprache Java bedingt ist. JAS besteht aus sehr vielen Klassen und muß daher viele Initialisierungen und umfangreiches dynamisches Laden vornehmen.

Die Benchmarks aus Kapitel 4.5, die durch JAS, der Schnittstelle und dem gleichwertigen C++-Programm gemessen wurden, haben ergeben, daß sich allein durch Java die Zeit einer Anfrage im Vergleich zu C++ mindestens vervierfacht. Es wurde erwartet, daß der Unterschied zwischen den Sprachen nur einen geringen Unterschied ausmachen würde, da die Anfrage größtenteils aus Eingabeoperationen besteht und während des Schleifenkörpers der Anfrage kein umfangreicher Programmcode ausgeführt wird. In der Praxis hat sich jedoch erwiesen, daß dem nicht so ist. Die Anfragezeit wird durch Java erheblich ausgedehnt.

Ein anderes Problem, welches aber auch mit Java zu tun hat, kann von der nicht ganz identischen Java-Bytecode-Ausführung durch den Java-Interpreter herrühren. Während der Testläufe von JAS auf Sun-Solaris-Systemen traten an verschiedenen bestimmten Stellen Nullzeiger-Ausnahmezustände (Nullpointer Exceptions) auf, die scheinbar auf der Entwicklungsplattform des Autors nicht entdeckt oder angezeigt wurden. Weiterhin kommt es auf nicht reproduzierbare Weise vor, daß JAS während einer Grafikoperation festhängt und nur durch den Abbruch des Java-Interpreters beendet werden kann. Dieses Phänomen ist mit JDK-Versionen bis zu 1.1.6 aufgetreten, jedoch häufiger mit 1.1.5. Dies könnte auch mit der Implementation von JDK zusammenhängen. Der Verdacht hierauf bekräftigt sich mit einigen Änderungen im JAS-Quelltext, welche damit begründet wurden, daß Sun-Solaris-JDK stellenweise unterschiedliche Effekte zeigte, verglichen mit JDK auf anderen Plattformen.

6.2 Was bringt Objectivity/DB für H1 ?

Vorteile

Mit Objectivity/DB gelingt es in erster Linie, den Zugriff auf Ereignisdaten zu zentralisieren. Dies ist durch den Lock-Server der Datenbank gegeben, ein Programm, welches die eingehenden Anfragen koordiniert. Damit brauchen keine Dateien mehr lokalisiert zu werden, so wie es bei bisherigen Analysen erforderlich war.

Das Objectivity-DBMS wurde bereits ausführlich bei anderen, sehr großen Forschungseinrichtungen wie CERN oder SLAC auf Skalierbarkeit und Distribution untersucht. H1 kann daraus Nutzen ziehen und die positiven Ergebnisse dieser Untersuchungen als Voraussetzung für die Verwendung von Objectivity/DB einsetzen.

In Kapitel 3.4 wurde der Einsatz von Metadaten in der Objectivity-Datenbank vorgeschlagen, z.B. ganze Histogramme oder Informationen zur Auffindbarkeit von BOS-Bänken. Durch das objekt-orientierte DBMS ist der Zugriff auf komplexe Objekte im Vergleich zu relationalen DBMS einfacher. Dadurch ist eine starke Flexibilität in bezug auf Größe und Struktur der Objekte gegeben.

Nachteile

Objectivity soll bei H1 für Datenqualitätsprüfungen und bei Ereignisanalysen verwendet werden. Die in Kapitel 4.5 gemessenen Benchmarks zeigen, daß die Datenbankzugriffsleistung noch zu schwach ist, um Anfragen in akzeptabler Zeit zu erfüllen.

Ein technischer Nachteil ist, daß das Java-Frontend von Objectivity/DB keine Möglichkeit zur Extraktion von Schemata zur Verfügung stellt. Dadurch müssen Java-Anwendungen stets eigene Schema-Deklarationen mit sich bringen, was mit Wartungsaufwand verbunden ist. Bei H1 würde diese Arbeit noch tolerierbar sein, da bisher nur wenige Java-Anwendungen geplant sind. Es wäre jedoch im allgemeinen vorteilhafter und einfacher, das Datenbankschema dynamisch extrahieren zu lassen.

6.3 Resonanz von Benutzern bei H1

Bei H1 finden verschiedene regelmäßige Besprechungen statt, an denen H1-Physiker teilnehmen, deren Forschungsgebiet sich mit dem wissenschaftlichem Gebiet der jeweiligen Besprechung überschneidet. Die H1-Kollaborationsversammlung findet alle drei bis vier Monate statt. Im Februar 1999 wurde dort ein Vortrag über Analysierung in H1 im Rahmen des OOP-Projektes (siehe Kapitel 2) gehalten, der von einer Demonstration von Java Analysis Studio mit dem Objectivity-DIM begleitet wurde. Es wurden sowohl die Fähigkeiten von JAS gezeigt als auch ein kurzes Analysebeispiel vorgeführt, so daß die etwa 80–100 Versammlungsteilnehmer einen realen Bezug zu physikalischen Analysen hatten. Die Möglichkeit in JAS, die Bins (Balkenbreite) des Histogrammes in Echtzeit zu ändern, wurde von den Zuschauern als besonders positiv betrachtet. Der bisherige Analyseablauf erlaubt den Forschern keine Echtzeit-Rebinning-Funktionalität. Um die Bins zu variieren, muß das PAW-Kumac-Makro verändert und damit das gesamte Makro — inklusive aller dort eingetragenen Schnitte und Histogramme — erneut vollständig abgearbeitet werden. Die möglichst geeigneten Bins herauszufinden, ist ein langwieriger, sich oft wiederholender Vorgang, dessen Bearbeitungszeit in Zukunft mit JAS von Stunden auf Minuten verkürzt werden kann.

Eine weitere positive Bemerkung zu JAS war, daß die Bedienungsoberfläche einen einfach lernbaren Eindruck machte. Dies war besonders den älteren Mitarbeitern wichtig, da für sie Umstellungen auf neue Software oder neue Prozeßabläufe schwieriger handzuhaben sind als für jüngere Menschen.

In Kapitel 5 wurde erwähnt, daß Februar 1999 eine frühe Testversion von Java Analysis Studio und dem Objectivity-DIM bei H1 installiert wurde. Die Anwender sollten sich einerseits mit der Histogrammierung durch JAS auseinandersetzen, und andererseits die Handhabung und das GUI des DIMs in Hinsicht auf Benutzerfreundlichkeit bewerten. Hierzu ist noch ein expliziter Plan in Richtung Benutzerbefragung auszuarbeiten, der zu dem Zeitpunkt gestartet werden sollte, sobald die Testumgebung genügend fortgeschritten ist. Diese ist momentan noch nicht in dem Zustand, daß sie in die realen Analyseprozeduren der Forscher integriert werden kann. Die Ereignisstruktur muß vervollständigt werden, die Struktur der Datenbank muß gefestigt werden, und der Zugriff auf die Datenbank muß

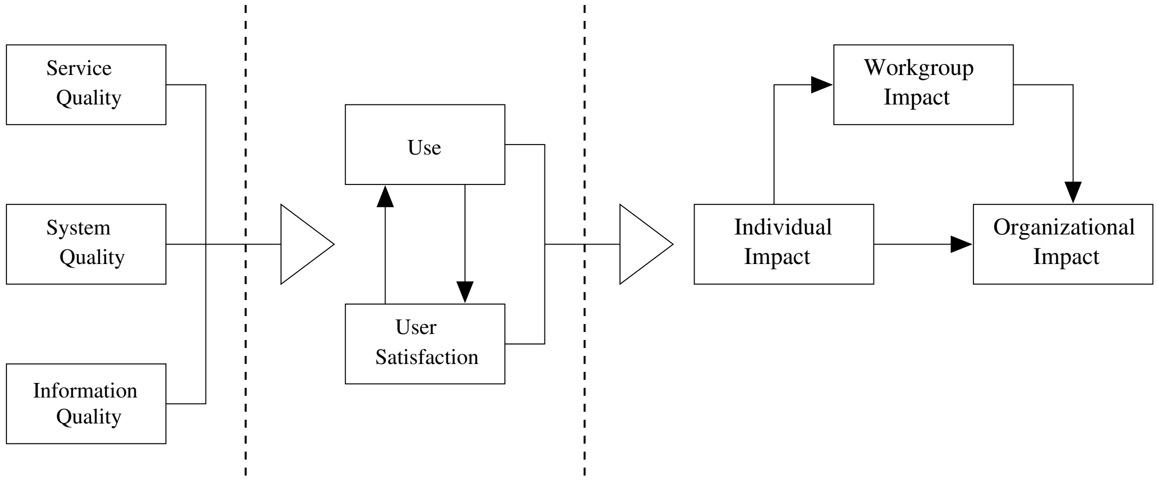


Abbildung 6.1: Bewertungsmodell für Informationssysteme nach DeLoan und McLean

noch erheblich verbessert werden. Erst dann sollte ein Befragungsbogen erstellt und unter Interessierten verbreitet werden. Die Gruppe der Beteiligten darf nicht zu klein gewählt werden, da sonst die Gefahr besteht, daß bei zuwenigen Teilnehmern das Testergebnis nicht aussagekräftig genug ist.

Die Aachener Lehrstühle Physik I und Physik III gehören zu H1 und pflegen je nach Übereinstimmung der Forschungs- und Anwendungsgebiete ihre Zusammenarbeit. April 1999 ist von beiden eine Vortragsreihe „H1-AC" ins Leben gerufen worden, deren Zweck die engere Zusammenarbeit der Lehrstühle ist. Alle drei Wochen finden ein bis zwei Vorträge statt, die von den Mitarbeitern oder Diplomanden der Lehrstühle gehalten werden. Damit sollen die zu bearbeitenden wissenschaftlichen Themen der Kollegen besser bekannt sein und somit eine gemeinsame Nutzung der Arbeitsergebnisse gefördert werden. Der Autor dieser Diplomarbeit hat am ersten Termin der Vortragsreihe über seine Arbeit referiert und somit einige Rückmeldungen zu JAS und Objectivity/DB bekommen. In Kapitel 4.4 wurde eine Schätzung der voraussichtlichen Speicherplatzanforderungen der Datenbank vorgenommen, die von den Zuhörern als akzeptabel empfunden wurden. Die Zugriffsgeschwindigkeit des JAS-DIMs auf Objectivity dagegen war erwartungsgemäß für die Anwesenden nicht tolerierbar. In diesem Kritikpunkt muß unbedingt eine starke Verbesserung vorgenommen werden, ansonsten wird kein Forscher JAS mit Objectivity/DB benutzen wollen.

6.4 Modell zur Bewertung

6.4.1 Einführung

Im obigen Abschnitt 6.3 wurde erwähnt, daß eine integrationsfähige Testumgebung noch fertigzustellen ist, und daß anschließend eine Bewertung durch die H1-Benutzer erfolgen sollte. In [MKP97] werden viele Modelle zur Bewertung von Informationssystemen untersucht, wobei dort der Ansatz von DeLoan und McLean ([DM92]) favorisiert wird, der durch weitere, in [MKP97] genannte Erweiterungen ausgebaut wurde. Das Resultat ist in

Abbildung 6.1 dargestellt. Die einzelnen Komponenten des Modells werden unten vorgestellt. Anschließend wird erörtert, inwieweit die Einführung von Java Analysis Studio, der Objectivity-Schnittstelle und der Objectivity-Datenbank bei H1 anhand dieses Bewertungsmodells beurteilt werden kann.

Service-Qualität (Service Quality)
In dieser Perspektive sind Bewertungskriterien angesiedelt, die am Umgang mit dem Benutzer beteiligt sind. Dazu zählen Zuverlässigkeit, Verantwortung, Kompetenz, Zugänglichkeit, Höflichkeit, die richtige Kommunikation, Glaubwürdigkeit, Sicherheit und Vertrauen.

Systemqualität (Systemquality)
Zu dieser Dimension gehören die Zuverlässigkeit des Systems, Antwortzeiten, Bedienbarkeit, Nützlichkeit, Flexibilität und Verfügbarkeit.

Informationsqualität (Informationsquality)
Die Informationsqualität setzt sich aus Begriffen wie zeitliche, örtliche und strukturelle Genauigkeit, Prägnanz und Relevanz der Informationen zusammen.

Benutzungshäufigkeit (Use)
Hier geht ein, ob und wie oft ein System genutzt wird.

Benutzerzufriedenheit (User Satisfaction)
Die Benutzerzufriedenheit setzt sich aus den individuellen Meinungen der Benutzer zusammen. In [MKP97] werden viele Quellen genannt, die verschiedene Methodiken entwickelt haben, um zu aussagekräftigen Bewertungsmaßstäben der Benutzerzufriedenheit zu kommen.

Individuelle Auswirkungen (Individual Impact)
Die Verwendung eines neuen Systems hinterläßt Auswirkungen auf den einzelnen Benutzer. Zu untersuchende Aspekte sind Effizienzsteigerungen, qualitative Verbesserung von Entscheidungen, verbesserte Kommunikation und bessere Kontrolle von Operationen.

Einfluß auf Arbeitsgruppen (Workgroup Impact)
Die Auswirkungen auf Arbeitsgruppen sind sehr bedeutsam, da Gruppenarbeit sich stark auf die gesamte Organisation auswirkt. Faktoren wie Terminlichkeit und Vorbereitung von Gruppentreffen, Verteilung der Versammlungsergebnisse, EDV-Verträglichkeit der verschiedenen Teilnehmer, sind zu beachten.

Auswirkungen auf die Organisation (Organizational Impact)
Der Einfluß auf die gesamte Organisation spiegelt sich durch Ziele wie Kostenersparnisse, erhöhte Produktivität, verbesserte Betriebsführung, Geschwindigkeits- und Qualitätssteigerung wieder. Nicht zuletzt soll die Verfeinerung des Kundenservices und des Kundenverhältnisses beachtet werden.

Systemqualitätsfaktoren

	Antwortzeit	Zuverlässigkeit	Einfachheit	Flexibilität
JAS, Java	–	–	+	+
	Java-Interpreter, Speicherverwaltung, Garbage-Collection, Sun-JDK	Sun-JDK	OO-Konzept, Garbage-Collection, intuitives GUI	Plattform-unabhängig, Applet möglich, DIM-Schnittstelle
JAS-Objy-DIM	–	?	?	+
	siehe JAS+Java	Breiter Test erforderlich	Dialogfenster	Java-Reflection, zentralisierter DB-Zugriff, Direct-/RMI-Zugriff
Objy/DB	–	+	+	+
	200K/s – 2M/s, Java-Frontend	Skalierbarkeit, Distributionalität, Tests bei CERN	Persistenz durch Vererbung, Integration in OO-Sprachen	Schema-Evolution, viele Plattformen, komplexe Objekte

– negative Eigenschaft
\+ positive Eigenschaft
? ist noch herauszufinden

Tabelle 6.1: Systemqualitätsfaktoren bei JAS und Java, dem JAS-Objy-DIM und Objectivity/DB

Die Faktoren System-, Service- und Informationsqualität beeinflussen direkt die Benutzer. Davon hängt ab, wie oft und wie intensiv das System verwendet wird. Gleichzeitig prägt sich eine subjektive Meinung des Benutzers über das System. Dadurch ergeben sich Änderungen bezüglich der Arbeitsweise und der Ergebnisse des Benutzers, wodurch unmittelbar die Arbeitsgruppen betroffen sind, welche letztendlich die gesamte Organisation repräsentieren und damit als Ganzes beeinflussen. In einigen Fällen existieren auch direkte Einflüsse auf eine Organisation durch den individuellen Benutzer.

6.4.2 Anwendung des Modells auf H1

Das Bewertungsmodell von DeLoan und McLean soll in diesem Abschnitt auf die Benutzung von Java Analysis Studio, des JAS-Objectivity-DIMs und der Objectivity-Datenbank angewendet werden. Dies ist jedoch stellenweise nicht realisierbar, da das gesamte Projekt (OOP, siehe Kapitel 2.1) sich noch in der Testphase befindet.

Zuerst ist zu überprüfen, inwieweit die Qualität zu den oben angeführten Dimensionen Service, System und Informationen gesichert ist. Die Service-Qualität kann nicht zur Bewertung herangezogen werden, da sich die Kriterien auf Personen beziehen, welche aber nicht Teil des zu bewertenden Systems sind.

Die Systemqualität wird anhand von Antwortzeiten, Zuverlässigkeit, Einfachheit und

Flexibilität bestimmt. In Tabelle 6.1 sind diese Faktoren in bezug zu Java Analysis Studio zusammen mit Java, zum JAS-Objy-DIM und zur Objectivity-Datenbank eingetragen. Es wird nachfolgend kurz erläutert, inwiefern durch sie die drei Komponenten bewertet werden.

Die Antwortzeiten wurden in Kapitel 4.5 gemessen. Die schlechten Ergebnisse wurden durch die Java-Speicherverwaltung, die Speicherbereinigung und das Sun-JDK begründet. Auf der Seite von Objectivity/DB könnte das Java-Frontend eine weitere Ursache sein. Dies ist jedoch noch ausführlich zu testen.

Die Zuverlässigkeit von JAS ist — zumindest bei Sun-JDK — nicht zufriedenstellend, da bei seltenen Fällen der Interpreter abgebrochen ist und bei grafischen Operationen gelegentlich Java-Ausnahmefehler (Exceptions) auftraten. JAS wurde unter JDK auf einem Windows-System entwickelt. Es wurden beim Laufzeitverhalten einige Differenzen zu Sun-Solaris-JDK festgestellt, wodurch dieses Verhalten erklärt werden könnte. Um die Zuverlässigkeit des JAS-Objy-DIMs zu bestimmen, wäre ein großflächiger Test mit einer realistischen Anzahl von Benutzern durchzuführen. In [HoltBunn98] wurden bereits Tests bei CERN zur Skalierbarkeit von Objectivity/DB bezüglich der Anzahl von Benutzern gefahren; dennoch sollte das DIM in der H1-Umgebung getestet werden, um sichere Werte zu haben.

Unter Einfachheit ist für den Forscher zu verstehen, daß die Benutzeroberfläche von JAS und vom Dialogfenster des DIMs schnell und leicht zu bedienen ist. Dies wurde in Abschnitt 6.3 durch Rückmeldungen von H1-Mitgliedern bereits bestätigt. Eine gezielte Umfrage würde ein genaueres Bild der Einschätzung der Anwendungsfreundlichkeit ergeben. Das Kriterium der Einfachheit soll auch bezüglich Software-Entwicklung bewertet werden. Allein durch Java ist eine übersichtliche, leicht erlernbare Entwicklungsumgebung gegeben. Durch die konsequent angewandte objekt-orientierte Programmiertechnik in Java werden viele Fehlerquellen ausgeschlossen. Ebenso ist fremder Java-Quelltext (wie der von JAS) relativ schnell zu verstehen. In Objectivity/DB ist das objekt-orientierte Konzept durch die Integration der Datenbankfunktionalität in objekt-orientierte Programmiersprachen wie C++, Java oder Smalltalk stark vertreten. Dadurch ist die Anbindung von Objectivity/DB relativ leicht zu vollziehen. Zum Beispiel werden Objekte durch Vererbung persistent, sie müssen nicht explizit in die Datenbank geschrieben werden.

Das Kriterium der Flexibilität wird von allen drei Komponenten — JAS/Java, das DIM und Objectivity/DB — zufriedenstellend erfüllt. Java besitzt eine hohe Kompatibilität zu Rechnerplattformen. Durch Java ist es auch möglich, Applets für den Internetgebrauch zu entwickeln, und für JAS ist eine Applet-Version bereits von SLAC angedacht. Für die HEP-Gemeinde wäre Online-Histogrammierung mit einigen Funktionalitäten von JAS zweifelsfrei attraktiv. Durch die Spezialisierung von individuellen DIMs können viele Dateiformate und Datenquellen für JAS zugänglich gemacht werden. Das JAS-Objy-DIM ist programmtechnisch insofern flexibel, daß die Datenbankzugriffs-Routine zentral gelegen ist, wobei zu bedenken ist, daß diese sowohl bei direktem Zugriff durch JAS als auch durch den RMI-Server des DIMs herangezogen wird. Für den Entwickler bzw. für den Administrator soll die Erweiterung des Ereignisobjektes in der Datenbank sehr einfach sein, was durch die Anwendung des Java-Reflection-Packages erreicht wird. Die Objectivity-Datenbank zeigt sich flexibel durch das Prinzip der Schema-Evolution und durch die Verwaltung von komplexen Objekten. Letzteres konnte bisher in den Testversionen bei H1 noch nicht weitreichend verifiziert werden. Dies kann sich in Zukunft jedoch ändern, falls Änderungen des Datenflusses im Benutzerdaten-Bereich eingeplant werden, wie sie in Kapitel 3.4 vorgeschlagen wurden (z.B. Abspeicherung von Histogrammen oder Metadaten).

Nach den obigen Kriterien der Systemqualität aus dem DeLoan-McLean-Bewertungsmodell wird nun auf die Informationsqualität eingegangen. Die Genauigkeit der Daten wird auf die Ereignisobjekte aus der Objectivity-Datenbank bezogen. Die Felder dieser Objekte werden aus den DST-Beständen gewonnen. Zu den Datentypen der DSTs gehören ganze Zahlen in 8/16/32 Bit, Fließkommazahlen in einfacher und doppelter Genauigkeit (float, double) sowie Bitketten. Bei der Erstellung der Datenbankobjekte werden diese Typen zu ganzen 32-Bit-Zahlen und Fließkommazahlen einfacher Genauigkeit konvertiert. Diese Umwandlung ist vertretbar, da die Analysen in erster Linie für Datenqualitätsprüfungen[2] gedacht ist und daher keine extreme Genauigkeit notwendig ist. Die Relevanz von Daten ist ein wichtiges Kriterium beim Aufbau eines Datenbankobjektes. In Kapitel 4.4 wird gezeigt, daß die Größe eines Ereignisobjektes aus der Datenbank etwa ein Zehntel des Umfanges der DST-Ereignisstruktur ausmacht. Dies wird durch den Ausschluß von Informationen erreicht, welches durch H1-Mitarbeiter besprochen wurde.

Die Bewertungsschlüssel Benutzungshäufigkeit und Benutzerzufriedenheit werden durch die Anwender des Systems — die Physiker, Forscher und Mitarbeiter von H1 — geformt. Das erstere konnte noch nicht erfaßt werden: Es gab bisher keinen expliziten Benutzertest. Dieser Test könnte auch Fragen nach der Anzahl erzeugter Histogramme oder selektierten Ereignissen beinhalten. Auch ist eine Fragestellung denkbar, die den Benutzer nach seiner voraussichtlichen Anwendungshäufigkeit des Systems interviewt. Die Benutzerzufriedenheit könnte in der gleichen Umfrage ermittelt werden. Hier eine Auflistung möglicher Aspekte für eine Benutzerumfrage:

- Persönliches Empfinden
  - Antwortzeiten von Anfragen
  - Intuitivität des Dialogfensters der Anfrage
  - Verständnis der DIM-Parameter bei der Erstellung eines neuen JAS-Jobs
  - Komfort bei der Bearbeitung von Histogrammen
- Statistische Angaben
  - Wofür wird die Analyse benötigt?
  - Wieviele Runs oder Ereignisse werden angefordert?
  - Eigenschaften der Schnittkriterien (Komplexitätstiefe, Anzahl von Bedingungen)
  - Wie oft wird das System genutzt (am Tag, in der Woche, im Monat), bzw. wie oft würde das System genutzt werden wollen?

[2]Data Quality Checks

- Wiederverwendung von Daten
  - Wie oft werden Daten von anderen Mitarbeitern genutzt?
  - Welche Art von Daten werden mitbenutzt (Vorselektionen, NTuples) ?
  - Wie groß sind die mitbenutzten Datenmengen?

Die Benutzungshäufigkeit und die Benutzerzufriedenheit beeinflussen direkt die Auswirkungen auf den Einzelnen. Damit sind individueller Nutzen und Verbesserungen angesprochen. Eine weitere Benutzerbefragung nach Ende der öffentlichen Testphase würde Auskunft über diese Sachverhalte geben. Zum Beispiel wäre es wichtig zu erfahren, ob der Analyseprozeß des Benutzers beschleunigt oder qualitativ verbessert werden konnte.

Zum Einfluß auf die H1-Arbeitsgruppen kann zum jetzigen Zeitpunkt noch keine feste Aussage getroffen werden. Mit der Voraussetzung, daß Pläne zur gemeinsamen Nutzung von Datenbeständen entwickelt und realisiert werden würden, könnte festgestellt werden, inwieweit dies innerhalb der Arbeitsgruppen zu Verbesserungen geführt hat.

Die Auswirkungen auf die Gesamtheit aller H1-Arbeitsgruppen sind wesentlich später zu beobachten und sind zur Zeit noch nicht absehbar.

6.5 Schlußbetrachtung

Die Wahl von Java als Implementierungssprache für JAS hat sich von Vorteil erwiesen. Dadurch hat sich das Softwarepaket trotz seines relativ großen Umfangs ein beachtliches Maß an Überschaubarkeit und Verständlichkeit bewahrt. Für das JAS-Objectivity-DIM ist Java zwar komfortabel, aber die Antwortzeiten der Anfragen lassen stark zu wünschen übrig.

Der langsame Zugriff auf die Objectivity-Datenbank wirft einen großen Schatten über Java Analysis Studio. Dies liegt weniger an dem JAS-DIM, sondern vielmehr an dem Java-Frontend von Objectivity/DB und an der Organisation der Datenbank. Bei H1 müssen zu letzterem noch ausführliche Tests mit dem Ziel der Zugriffsverbesserung durchgeführt werden. An dieser Stelle sind die Index- und die Cluster-Methoden von Objectivity/DB intensiv zu prüfen. Java Analysis Studio und das Objectivity-DIM sollten nicht mit der aktuellen Datenbank-Zugriffsgeschwindigkeit publiziert werden, da dies die Benutzer lediglich abschrecken würde und die Software sich nicht etablieren würde. Optimierungen an der Datenbank sollten möglichst vor der Freischaltung für H1 stattfinden. Hinzu kommt, daß JAS bzw. JDK auf Sun-Solaris noch nicht stabil genug sind. Unvorhergesehene Programmabbrüche geschehen zu oft, als daß sie toleriert werden können.

Das Data-Warehouse-Konzept konnte von den Datenquellen bis zum Corporate-Data-Warehouse ohne große Schwierigkeiten auf den H1-Detektor und die zugehörige Datennahmeprozedur übertragen werden. Bei der Erstellung von Data-Marts können Verbesserungen hinsichtlich der Organisation und des Zeitaufwandes von individuellen Analyseprozessen erzielt werden, indem Analyse-Ergebnisse von Forschern systematisch aufbewahrt und für andere Analysen wiederverwertbar gemacht werden. Es ist noch zu diskutieren, ob die Forschungsergebnisse in der Datenbank abgelegt werden sollen, denn es ist zu bedenken, daß

Objectivity/DB bisher für Datenqualitätsprüfungen und Analysen in kleinem Rahmen gedacht ist und deshalb nur reduzierte Ereignisse enthält. Da die ursprünglichen Ereignisse sich noch in flachen Dateien befinden, können diese nicht von der Datenbank aus referenziert werden, da keine direkte referentielle Verbindung besteht. Die Ergebnisse von Analysen würden sich lediglich auf die Ereignisse innerhalb der Datenbank beziehen.

Es kann trotz der Datenbank-Zugriffsverbesserungen dazu kommen, daß in einigen Fällen die Datenextraktion aus DST-Beständen mit herkömmlichen Methoden schneller durchgeführt werden kann als mit JAS. Hier kann seitens der Benutzer überlegt werden, zunächst mit JAS eine Stichprobe der gewünschten Ereignisdaten anzufordern und anhand dieser Probe Kriterien für die Gesamtmenge zu bilden (z.B. Schnitte oder Balkenbreite). Dieser Vorgang entspricht vollkommen dem Zweck von JAS und Objectivity/DB — der qualitativ verbesserte Entwicklungsprozeß einer physikalischen Analyse.

6.6 Ausblick

An dieser Stelle wird ein Überblick über einige Ideen und Aufgaben gegeben, die im Rahmen des OOP-Projektes bei H1 in Zukunft diskutiert oder realisiert werden können.

Was das JAS-Objectivity-DIM betrifft, so ist der konzeptionelle Aufbau der Anbindung an JAS und an Objectivity/DB getätigt und implementiert worden. Der Ausbau zur Anbindung an die Testdatenbank Version 3 ist noch zu bewältigen. Weitere geplante programmtechnische Änderungen können aus der technischen Dokumentation (siehe Anhang A.4) entnommen werden. Optional kann die Implementierung der Zugriffe auf die Datenbank in C++ umgesetzt und in Form von nativen Funktionen in das JAS-Objectivity-DIM integriert werden.

Im Rahmen des Data-Warehouse-Modelles wäre es interessant und nützlich, Analyse-Ergebnisse in Form von Histogrammen in der Datenbank abzulegen, die von Forschern als Aufsatzpunkt oder zur Orientierung dienen können. Um die Datenmengen in Grenzen zu halten, und um übersichtlich zu bleiben, bräuchten nur diejenigen Ergebnisse gesichert zu werden, deren Aussage von physikalischem Interesse für mehrere Forscher oder mehreren Arbeitsgruppen sind. Zum Beispiel werden bisher in Diplomarbeiten, Doktorarbeiten, veröffentlichten Schriftstücken oder sogar im Internet immer wieder Histogramme präsentiert. Diese könnten zusätzlich in elektronischer Form mitsamt der dazugehörigen Ereignisse in der Datenbank abrufbar gemacht werden, wodurch die Reproduzierbarkeit erheblich vereinfacht werden würde.

Es ist sehr empfehlenswert, die Zugriffszeiten der Objectivity-Datenbank durch organisatorische Maßnahmen zu beschleunigen. Im ersten Anlauf sollten Versuche zur Seitengröße innerhalb der Datenbank entwickelt sowie die Objectivity-internen Index-Funktionen miteinbezogen werden. Das Prinzip der Data-Marts und einer damit verbundenen Hierarchie kann sehr vorteilhaft für H1 sein, auch wenn die entsprechenden Datenbestände als Dateien und nicht in der Datenbank als Partitionen vorliegen. Informationen in Form von Metadaten sollten jedoch in der Datenbank abgelegt werden. Die in Kapitel 4.6 vorgestellten Verbesserungsvorschläge (Reclustering, Columnwise Clustered Index) sind wertvoll, bedürfen aber aufgrund ihrer theoretischen und spezifikationsintensiven Hintergründe viel Vorarbeit. Hier empfiehlt es sich, zunächst die Resultate der Testimplementierungen in CERN zu beobachten und sich diese zunutze zu machen.

Appendix A

Technical Documentation

A.1 User Documentation

A.1.1 Basic Usage in Java Analysis Studio

Creating a new job: Once you have started JAS, you should create a new job in order to see histograms at all. Choose the "Job/New Job" menuitem to do so, or click the appropriate toolbar icon. A dialog will show up. You might want to prefer "A local job for data analysis." for a start. Choose your desired DIM and follow its further instructions. (See below for a guide to using the Objectivity DIM.)

Showing histograms: After you have finished creating a new job, you can select the tree's leaves on the left. You can show data, erasing previous plots, or you can overlay data to existing plots. To have more than one histogram pages at a time, choose a page layout from the "File/New/Histogram Page" menuitem. The histogram pages can be shown as tabbed pages or as internal windows ("View/Window Style" menuitem).

Manipulating histograms: What you might not have noticed — you can actually drag a histogram around. You can even shrink or expand its axes just by dragging them. With the mouse, grab the axes at their ends to resize the scales, or grab them in the middle to just shift them.

Rebinning a histogram: The presently used software at H1 (PAW) has to reread all events in order to rebin a histogram. This might take about 20–30 minutes, depending on how many events are to be read and how many cuts are applied from the kumac. In JAS, there is a rebin-slider, enabling you to rebin the histogram in realtime as you move it. Select "View/Rebin Slider" and it will show up right to the icon toolbar. Your JAS window might not be wide enough for the rebin slider to fit. To solve this, enlarge the window horizontally or remove the toolbar ("View/Toolbar" menuitem) to have more space.

Multiple DIMs in one job: This feature will come in handy if you want to compare several PAW-files, or to compare PAW-files with Objectivity-Data, and so on. You can actually have several DIMs in one job. To accomplish this, open a job and a

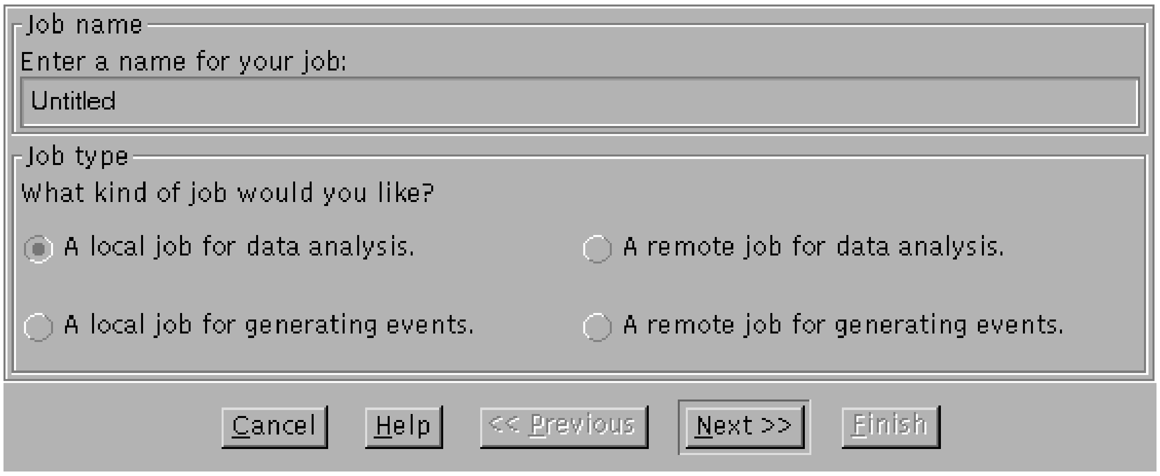


Figure A.1: New Job Wizard: Type of Job

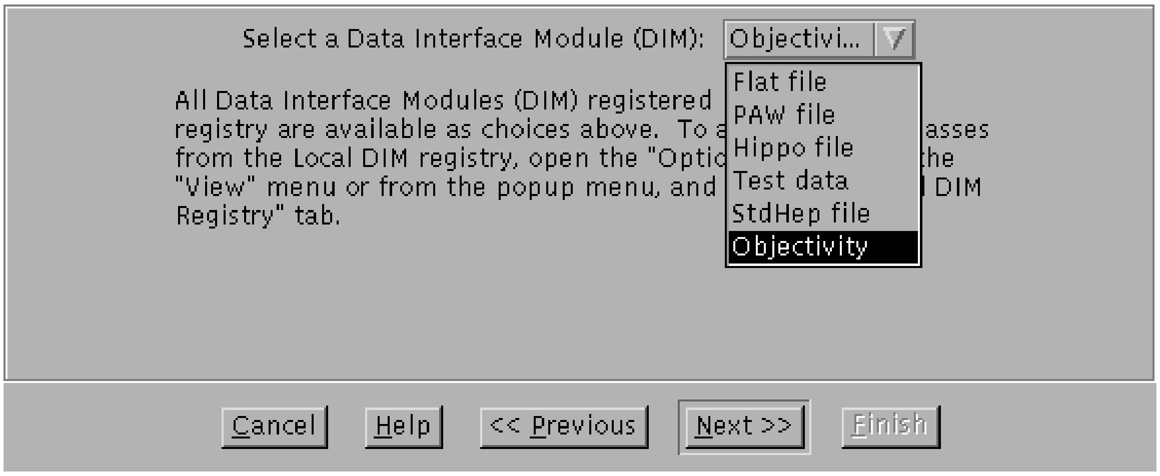


Figure A.2: New Job Wizard: Select a DIM

DIM, just as described above (see "Creating a new job"). The tree at the JAS main window will show a hierarchy with your job name, "Data" and a name describing the DIM. Now choose "Open Dataset" from the "Job" menu. A wizard dialog appears, letting you select another DIM which will be added to your job's DIM list. You can add multiple DIMs of the same type to your job, or you can add different DIMs, too.

More information and documentation about using JAS can be found at its homepage [Johnson98].

A.1.2 How to use the Objy-DIM

Proceed the following steps in order to access the Objectivity database:

1. **Open a new job.**
   Select the "Job/New Job" menuitem, or hit the appropriate toolbar icon.

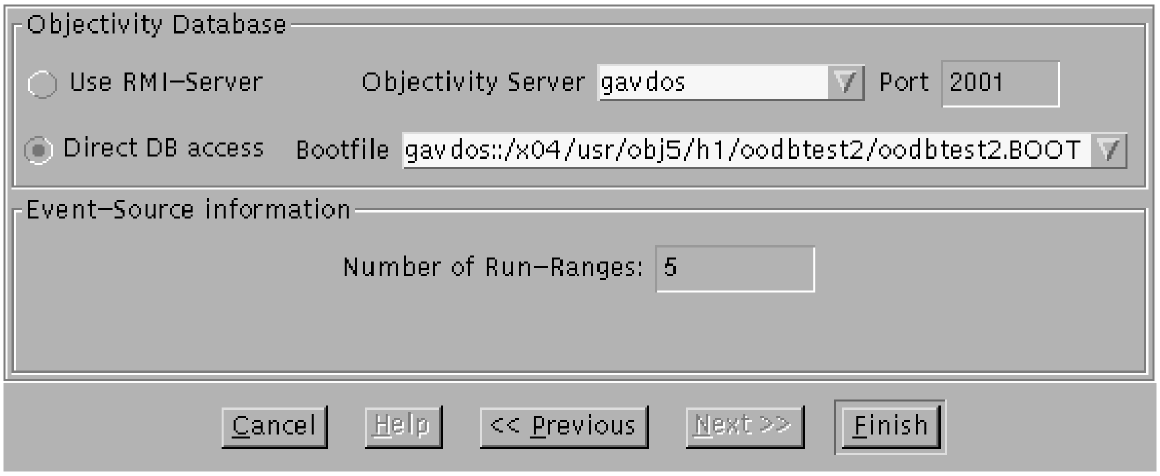


Figure A.3: New Job Wizard: Objectivity DIM Settings

2. **Select the type of job you want** — A Local Job for Data Analysis.
   A remote job is not recommended. Figure A.1 is showing the dialog.

3. **Choose the desired eventsource** — Objectivity.
   As you can see in figure A.2, the DIM can be selected at this page of the New Job Wizard of JAS. If it does not appear in the list, you can add it by yourself: Cancel the dialog, then choose the "Options/View" menuitem. Activate the "Local DIM registry" tab, then add the following line to the existing DIMs:

   `jas.jds.module.objy.ObjyLocal`

   Now you should see the Objectivity DIM when creating a new job.

4. **Choose RMI access or direct access**
   In figure A.3 you can see the settings page of the Objy-DIM. If you decide to use the RMI server for database access, you must supply the computer name and a port number where the RMI server is running. Default values are already supplied. If you want to access the database directly from JAS, give a bootfilename (there should be already a default name). The filename should be prefixed with the computer name running the Objectivity lock server, followed by a double colon. You can omit the computer name if you are running JAS on the lock server itself.

   The direct-access-method is an alternative way to access the database, in case there is no RMI server available.

   The number of runranges is the number of different histograms you want to create. You will be presented this many runrange treeitems at the JAS main window later on.

Now that you have setup a job, you are ready to retrieve data from the database. On the left of the JAS main window, you will see a tree. Expand it until its leaves. It will show the "Objectivity" eventsource, and a fixed number of runranges (see figure A.4). If

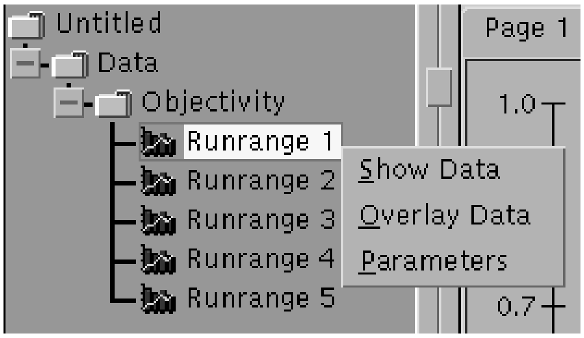


Figure A.4: Java Analysis Studio: Selection Tree

you need more runranges, please open a new job and enter your desired value when the Objectivity-DIM displays its panel. The runranges are called “Runrange n”. This cannot be changed by now, but that does not harm your histogram. Now you probably want to make JAS display your desired data. It must know the parameters then. Right-click one of the runrange tree items, hold down the mouse button and choose “Parameters”. A dialog window will appear — it might take longer at the first time, but it'll pop up quicker from the second time on[1]. See figure 5.1 for a picture of the dialog. On the left you can choose your desired variable from a listview. This will be the variable for evaluation in the histogram. Furthermore, choose the runrange you want. While retrieving data from this runrange later on, the DIM checks for non-existent runs and skips to the next run immediately.

The DST-year selection field is currently somewhat semi-functional. The DIM always accesses DST2 on the year given from the input field. This is because at the time of development of the DIM, there was no clear design of the DST/Run relationships yet. Please read below about the required future changes to the DIM for more information. Finally, there is a possibility to supply a predicate string. You can enter C-style conditions here, for example, “`relth<2.6 && relth>0.175`”. Hit the OK-button to confirm your parameters.

By choosing “Show” from the context-menu or by double-clicking the runrange tree item, the database is accessed with the corresponding parameters. Choose “Overlay” to add the histogram to the current plot. If you want to show or overlay a histogram again, without changing its parameters, the data held in memory will be used. So, the database is not being reloaded for every display request. It will be accessed if and only if any parameter of a runrange has been changed.

You can create histograms showing any variable in any runrange. This can be advantageous if you want to overlay more than one histograms of the same variable but different selection criteria. For example, you can select variable `zvtx` at one runrange, and variable `zvtx` on another. Give the same “from” and “to” values to each runrange, and also the same DST. But then, supply different predicate strings in order to yield two different histograms

[1] This is because it accesses the database briefly if opened for the first time. The DIM remembers its data so the dialog doesn't need to access the database for subsequent user requests, so the window will open more quickly when used more than once.

of the same variable.

When you have created your histogram, you might want to see the legend. The legend will show you information of the histogram. It is intended to show up if you have overlayed at least one more histogram, in order to explain the characteristics of each histogram. However, you can make the legend visible for only one histogram, too. Choose "Properties" either from the "Histogram" menu or from the histogram's context menu. There is a switch which is set to "Automatic", making the legend show only if at least two histograms are overlayed. Change this setting to "Show Legend", and the legend will stay, regardless of how many histograms are there.

A.2 Administrator Documentation

A.2.1 Installing JAS and the Objy-Interface

First, install the Java Analysis Studio package to the system. You can copy it from this AFS-path:

```
/afs/slac.stanford.edu/public/software/jas/v10beta/
jas-1.0Beta-solaris.tar.gz
```

Only packages for Linux, WindowsNT/Windows95 and Sun Solaris systems are available from there, but since it is a Java application, you can start the Solaris JAS installation on a SGI computer as well. The only restriction is that you cannot use the supplied libraries containing native code. Also note that currently JDK 1.1.3 is used on SGI, while 1.1.5 is used on Sun computers.

After you have successfully installed JAS, you want to add the Objectivity-DIM. In your JAS-installation, you will find a `lib` directory with all needed JAS classes in it. Copy the file ~`svent/public/jas/objy.jar` to this directory. The original batch script to launch JAS is located in the `bin` directory of your JAS installation. An updated starting script has been prepared and can be copied to the `bin` path from here: ~`svent/public/jas/bin/jas`.

If you examine the JAS starting script, you might have noticed that the Java interpreter command launching the JAS main class is using the `-native` parameter. This is strongly recommended if the Objy-DIM shall be usable. The Java classes supplied by Objectivity invoke native code which will crash the program if no native threads are used. For some reason, JDK 1.1.6 on Sun UltraSPARC Solaris computers doesn't have native threads included, that's why the previous release 1.1.5 is used there. So, if you want to change the starting script or if you want to start JAS from another script, don't forget to add the `-native` command line parameter.

A.2.2 The RMI-Server

You can skip this paragraph if you want to use the Objy-DIM for direct connections only. Otherwise, go on reading if you want to make use of the RMI-Server system.

Choose a `bin` directory where the starting script for the server shall reside. A suggestion is the directory with the JAS script in it. Copy the file ∼`svent/public/jas/bin/-objyserver.public` to your chosen directory and rename it to `objyserver`. Copy the file ∼`svent/public/jas/bin/rmireg` to the same directory. The `objy.jar` archive is required from the RMI-Server. Please edit the file `objyserver` you just copied. There are a few lines setting up the `CLASSPATH` environment variable for Java. Currently, the `objy.jar` file is accessed from the `lib` directory in parallel to the `bin` directory from where `objyserver` is started. You can change the `CLASSPATH` variable to match your location of the `objy.jar` package.

The `objyserver` starting script has one parameter which is the port number of the RMI connection. It defaults to 2001. The script can only be started by users who have set their `OO_FD_BOOT` environment variable. It also checks whether it is being launched on a SGI-IRIX or a Sun-Solaris computer, and it will abort if it is not.

If you take a look at the lines within the script where the server is started, you will see a HTTP URL leading to the author's webspace. Soon in the future the account of the author of the Objy-DIM will be deleted. By then, it is essential to install another webspace for RMI access, and to change this URL. Once there is an officially installed JAS-package with the Objy-DIM in conjunction with its RMI-Server, it is considerable to find a more sophisticated solution than the URL pointer to a person's webspace.

A.2.3 How to update or to change the TTagEvent variables

In the future of Objectivity/DB in H1, it might be desirable to change the event-holding objects in the database. Usually, physicists want to add one or more variables they consider important to reside in event objects. To change these objects, edit its class which is `TTagEvent`. The class methods are described below. `TTagEvent` has a list of public variables which are persistent in the database simply because they are public. Other variables will follow, but they are declared protected, that's why they are neither stored nor retrieved from the database. Now, if you want to add new variables to the database schema, add them to the public declared ones. Recompile `TTagEvent`, and copy the new class to the public location where the previous version resides. Of course, you should have agreed upon the variable change with your colleagues at H1 first.

There are several types of variables in `TTagEvent`. Variables with elementary types such as float or inteer are easy to add. Other, more complex but still simple types are arrays. Java arrays are always dynamic. Their size is unknown at declaration unless combined with an expression. Since there are also C++ programs at H1, accessing the database, there is a problem. C++ arrays are of fixed sizes, unlike in Java. This has been solved in Objectivity/DB with the introduction of dynamic arrays within the schema. These dynamic arrays are named "ooVArray" in C++, and they are mapped by the schema to Java arrays in Java programs. Supposed you want to add a new array in `TTagEvent`, add its declaration just like the other Java arrays which are already present. You MUST add an ooVArray-typed field to C++ programs, otherwise these fields will not work when accessing them from Java programs.

A.2.4 JAS Updates

Java Analysis Studio is in steady development, and soon there will be a new version available. It will be recommended that you match your current starting script for JAS with the one supplied by the new JAS package. However this is a topic that cannot be discussed here simply because there is no new JAS version yet.

A.3 Developer Documentation

A.3.1 Package `jas.jds.module.objy`

- `ObjyLocal`

  This file is loaded by the JAS program if a local job with the Objectivity-DIM is created. It extends `LocalDIM` so JAS knows it'll open another wizard page. The `toString`–method returns the name given in the cycle-gadget which contains other types of data too. The method `openDataSet` finally creates the eventsource which is explained below.

- `ObjyServer`

  This class creates an `ObjyEventsource` for the Java Data Server (`jas.jds.JDSMain`). It's just there to enable people starting a JDS if they want, but this does not really make sense, since the Objy-DIM either is using its own RMI server or is accessing the database directly.

- `ObjyEventsource`

  This is the eventsource which is instantiated by `ObjyLocal` and `ObjyServer`. Here's a description of its important methods.

  - `ObjyEventSource()`

    Its constructor initializes a fixed array of strings which are used later. In case the user chose to access the database via the RMI server (at the new job initialization procedure), `ObjyEventsource` sets its name (the field `m_name`) to the computername, followed by a colon and the portnumber. The name field will be shown as dataset name in the left-side tree of JAS. This is the place where `objyRunrangeTreeAdaptor` will retrieve it.

  - `getItems()`

    This method generates an array of `TreeItem` which is parsed by `jas.job.-AbstractJob:setEventSource()` in order to create subitems of the displayed tree. Note the `type` field of these `TreeItem` class. It is used to calculate at runtime the name of the class which will handle mouseclicks on the tree items. See `jas.swingstudio.JASJob`, class `TreeNode`, method `getTreeAdaptor` for the code to accomplish this.

  - `getItem()`

    This method is called from the TreeAdaptor which classname is derived from the treeitem's type. A TreeAdaptor in JAS will call its method `getAssociatedObject()` (which calls finally `getItem()`) to get a copy of the object associated with the treeitem. E.g. the Test-DIM (which is part of the JAS package) supplies fixed histograms as objects here. The TreeAdaptors in JAS usually call this method whenever the treeitem of its type is clicked with the mouse.

    In `ObjyEventsource`, this method returns an instance of `FakeHistogram`.

- `ObjyRunrangeData`

  Here is all information stored dealing with a run-range. There are methods to set the variables to desired values. To retrieve the values of these variables, access them directly. An exception is the title string which is retrieved by a method.

  The method `isValid()` returns a boolean value which is `true` if all run-range-variables are set to valid values. It returns `false` if the current instance is not yet initialized.

- `ObjyHistogram`

  This class implements the interface `jas.hist.Rebinnable1DHistogramData`, enabling it for being shown by `objyRunrangeTreeAdaptor`. The constructor has two parameters: a data array, and runrange-data. The data array is examined to compute its minimum and maximum values which are retrieved by JAS when the histogram is displayed. The runrange-data is examined in the method `getTitle()`, in order to construct the string which is displayed in the legend.

- `FakeHistogram`

  This class is no real histogram, as the name might imply. It only has the purpose to deliver the database bootfilename from `ObjyEventsource` to `objyRunrangeTreeAdaptor` in case a direct database connection is desired by the user. It is generated at `ObjyEventsource.getItem()`. All functions are without any use except `getTitle()` which is abused to return the bootfilename. "Why does `getItem()` not return the bootfilename itself?", you might ask. The problem is that this does not work when using a JDS server. You may want to trace `objyRunrangeTreeAdaptor.getAssociatedObject()`. Then, you will come across `JASJobAdaptor:getItem()`. At this function, there are only specific return types allowed. That is why a pseudo histogram (fake) is used as the returned class.

- `ObjyException`

  A subclass of `Exception` and should be instantiated with a string as parameter.

A.3.2 Class `jas.swingstudio.objyRunrangeTreeAdaptor`

Now, this class is actually the user interface of the Objy-DIM. Every runrange treeitem in JAS' display refers to one instance of `objyRunrangeTreeAdaptor` (which is created when the user clicks the according treeitem for the first time). JAS constructs the name of this class during runtime, as explained above (see `ObjyEventsource`).

This class is a subclass of `histogramTreeAdaptor`, profiting of all its methods, especially those dealing with mouseclicks. There are two methods responsible for showing or overlaying a histogram, `onShowData()` and `onOverlayData()`, respectively. Each `objyRunrangeTreeAdaptor` has exactly one runrange-data field containing the information for the current histogram. If data is requested to be displayed but is not available because it has not yet been loaded from the database, the properties dialog is shown. The dialog is represented by the class `ObjyProperties` which is included in the same source code file as `objyRunrangeTreeAdaptor`. It can be activated via doubleclicking the treeitem entry for the first time, or by selecting "Parameters" from the context-sensitive menu on each treeitem. If any of the data from the dialog is changed, the data is selected to be reloaded at the next show- or overlay-request.

To reload Objy-DB-data, `objyRunrangeTreeAdaptor` must know the computer name (and port) where the RMI-Server for Objy is installed. It yields this information by looking up the tree and retrieving the name of the eventsource. This is done by the method `getObjyReference()`. Then `ObjyEventsource` must have set its name accordingly, which has been described within its own paragraph already. After all data have been reloaded, they are stored within an instance of `ObjyHistogram` (see above), holding the data array until it is displayed.

A.3.3 Package `h1objy`

- `ObjyAccess`

  This is the interface used by both the RMI-Server `ObjyEngine`, and its client, `objyRunrangeTreeAdaptor`.

- `ObjyEngine`

  This class is the RMI-server for clients who wish to access Objectivity DB contents. It is a Java-application on its own. At startup, it opens the database bootfile given from a shell parameter. Please look up the documentation to `ObjyDirect` for an explanation of the methods `getVarNames()`, `getVarTypes()` and `getObjyContents()`.

- `ObjyDirect`

  It is possible for JAS to access the database either directly or via the RMI server. This class contains all methods for accessing the database. It is used in `ObjyEngine` and in `objyRunrangeTreeAdaptor` as well.

  - `ObjyDirect`

    The constructor opens the database specified by its bootfilename, supplied from a parameter.

  - `closeDatabase()`

    Closes the database.

– `getVarNames()`

  A `Vector` of strings is returned, containing all the variable names within the Objy-DB. This is used by the dialog window of `objyRunrangeTreeAdaptor` to show a list of variables available.

– `getVarTypes()`

  A `Vector` of strings is returned, containing all the variable types. The Vector is guaranteed to be of the same size like the Vector returned by `getVarNames()`. Each entry represents the type of the variable from the same index of the Vector containing the names. Valid types are `Integer` and `Float`.

– `getObjyContents()`

  Invoke this method to retrieve data from the database. Parameters are a runrange-data field, holding the information which data to retrieve, and a string containing the username invoking this method. `getObjyContents()` creates a new `ReloadThread` which then will access the DB. After the thread has successfully completed, an (often huge) array holding the results of the client's request is returned.

  The rationale behind spawning a thread is as follows: Objectivity demands that a thread can have access to one session at a time only. If `getObjyContents()` would access the DB right ahead, it would lead to confusion if this method is called by multiple clients. Running at overlapping times, the method would try to open a new session while another session is still active, causing the other session to be cancelled! To avoid this, a thread is created, started, and waited for until it is finished.

A.3.4 `TTagEvent`

This is the class holding all the variables being stored in Objy. Thus, it is a subclass of `ooObj`. Its methods are invoked by `ObjyEngine` only. An explanation follows:

- `TTagEvent()`

  This constructor is used just to create an instance. You can then call `getVarNames` or `getVarTypes`. Do not call methods accessing DB variables! You may access these with `getVar()` only after having used this constructor `TTagEvent( String )`.

- `TTagEvent( String )`

  Use this constructor if you are about to enter a loop, retrieving a specific variable from the DB. Here, TTagEvent prepares itself for retrieving the variable, which name is supplied via the parameter string. Use the constructor only once: just before you enter the retrieving loop, not inside of it (see `ObjyEngine.java`).

- `getVar()`

  Invoking this method is the usual way to retrieve a variable from the DB. Be sure you have called the constructor with the variable name as parameter before you call this method. The constructor initializes access to the variable of interest in order to optimize the number of calls to JDK's reflection package.

This method will access the variable using Java's reflection technique. By doing so, this function is usable for all types and names of the variables. Since the number or the names of the variables might be changed in future (they will!), this class should be as easy to expand as possible.

- `getVariableNames()`

  This method returns a `Vector` of strings, representing the names of all variables stored in the DB. Arrays show up as their variable names, concatenated with a hyphen and their indices. E.g. `jiobnd-0, jiobnd-2, jiobnd-2`. There's a special, private variable `jiob_MaxLength`, indicating the maximum size of all these arrays.

- `getVariableTypes()`

  A `Vector` of strings is returned, representing all the variables' types. They are either `Integer` or `Float`.

A.4 Future Changes

Here is a list of changes which are still to be applied on the Objectivity-DIM.

- Support for test database version 3
  The Objy-DIM has been developed for the test database version 2. A few months before completion of this diploma thesis a new test version has been created, introducing a new way to handle run information. Unfortunately, there is no time left within this thesis to upgrade the DIM in order to handle version 3. This has still to be done. However, the DST-databases structure has not changed, and it should be easy to add support for the new "RunDatabase"-database.

  A new object has been introduced in oodbtest3: `TTagRun`. It inherits the `ooObj` object and contains information about the run. H1 users will want to supply predicate strings over variables of `TTagRun`. In the Runrange dialog window, another string input field is suggested, which should hold the predicate string being applied on `TTagRun`.

  The test database version 3 has introduced indexing techniques supported by Objectivity/DB. This must be taken into account when applying the V3 changes to the Objy-DIM.

- Access correct DST and DST-year
  While the DIM has been developed, the design of DSTs within the database is still unclear. Therefore, the DIM always accesses DST 2 because there are many data in the database's 1997 contents (talking about version 2 of the database, oodbtest2). Meanwhile, version 3 of the test database (oodbtest3) introduces objects for handling run-information, like TTagCont and TTagRun. The JAS-DIM needs to be changed

for accessing these. For the same reason, there is no input field for the DST-file yet, e.g. the user might want to choose "2" for DST2.

The GUI-switch "Use latest DST" will set the year fixed to 1998 which has to be changed, too. The DIM shall retrieve information from the database and compute the most recent DST.

- Change a runrange title
  The runrange properties dialog window has a string field for the runrange title, which is disabled. The reason is behind the fixed selection tree on the left side of the JAS main window. Once its items are set, they cannot be changed (except initializing the entire tree again which would collapse it and so disturb users). Perhaps a future version of JAS will support dynamic tree items. The field can be re-enabled then.

- Enhanced predicate input field
  Currently, the runrange properties dialog window supports a predicate string in one wide input line only. Since large conjunctions would be too complex here, it is advisable to expand the input field. It is suggested to replace the Java-textfield with a multi-line field. It should cope with 10–20 atomic conditions at the same time, in such a way that users still see them in a convenient presentation.

Anhang B

Literaturhinweise

B.1 Referenzen aus der Teilchenphysik

[**ArdShi99**] Eva Arderiu, Jamie Shiers: "Very Large Object Databases: from Terabytes to Petabytes", VLDB '99, Edinburgh, Scotland, September 1999

[**Babar99**] The BaBar Homepage, http://www.slac.stanford.edu/BFROOT/

[**Blobel94**] Volker Blobel: "The BOS System - Dynamic Memory Management", 14th November 1994, http://www1.cern.ch/JuliaLIGHT/BosManual/bos.ps

[**Bunn98**] Julian Bunn: "Building a ∼1 TByte database of CMS Events Using Java for Event Viewing and Analysis — The GIOD Project (Globally Interconnected Object Databases), A Joint Project between Caltech (HEP and CACR), CERN and Hewlett Packard", RD45 Workshop, October 1998,
http://wwwinfo.cern.ch/asd/cernlib/rd45/workshops/oct98/presentations/bunn/giod/ppframe.htm

[**Collab93**] H1 Collaboration: "The H1 Detector at HERA", DESY preprint 93–103, July 1993, http://www-h1.desy.de/h1/www/h1det/detpaper/contents.html

[**Collab99**] H1 Collaboration: "Computing now and into the HERA2000 Era", Hamburg, 19th January 1999,
http://www-h1.desy.de/h1/iww/ipublications/int_notes_list.shtml

[**Düllmann99**] Dirk Düllmann: "Petabyte Databases", "Object Databases as Data Stores for HEP", Proceedings of the 1999 ACM SIGMOD, International Conference on Management of Data, Philadelphia, Pennsylvania.

[**FPACK98**] Fortran Package for Input/Output, FPACK Manual Version 1.00/00,
http://www-h1.desy.de/icas/imanuals/fpack/fpack.v10000.manuals.html

[**H1PHAN97**] H1 Physics Analysis Package, Version 2.03/11, May 15, 1997,
http://www-h1.desy.de/icas/imanuals/h1phan.1.txt

[H1REC99] H1REC — The H1 Reconstruction Program,
http://www-h1.desy.de/icas/imanuals/h1rec/h1rec9/h1rec.html

[H1TOX97] H1 Physics Analysis Tool Box,
http://www-h1.desy.de/icas/imanuals/h1tox.html

[Hadig98a] Thomas Hadig: "An Object Oriented Database For H1 — Feasibility of Tools, Time Scale, Design", October 1998,
http://www-h1.desy.de/~hadig/obj/prorep1.ps.gz

[Hadig98c] Thomas Hadig: "An Object Oriented Database For Physics Analysis, Report III — Program Design and Status", October 1998,
http://www-h1.desy.de/~hadig/obj/oodbrep3.ps.gz

[Hadig99] Thomas Hadig, J. P. Phillips: "A Tag Database for H1", Draft 1.1,
http://www-h1.desy.de/~hadig/obj/proposal3.ps.gz

[HoltBunn98] Koen Holtman, Julian Bunn: "Scalability to Hundreds of Clients in HEP Object Databases", September 1998, CHEP '98
http://home.cern.ch/k/kholtman/www/

[Holtm98] Koen Holtman: "Data Management in High Energy Physics Applications", Eindhoven University of Technology, 7th May 1998, CWI database colloquium
http://home.cern.ch/k/kholtman/www/

[Hoschek98] Wolfgang Hoschek: "Objy/Java for Online Analysis of NA48 Event Tags", Talk on 30th Oct 1998 at RD45 workshop
http://www.cern.ch/CERN/Divisions/EP/HL/Papers.html

[HSW98] Koen Holtman, Peter van der Stok, Ian Willers: "Automatic Reclustering of Objects in Very Large Databases for High Energy Physics", July 1998, IDEAS '98
http://home.cern.ch/k/kholtman/www/

[Itterb97] Heiko Itterbeck: "Techniques and Physics of the Central-μ-Trigger System of the H1-Detector at HERA", Dissertation at RWTH Aachen, 1997,
http://mozart.physik.rwth-aachen.de/Diplom-Doktor.html

[Johnson98] Java Analysis Studio by Tony Johnson/SLAC,
http://www-sldnt.slac.stanford.edu/jasweb

[Johnson99] Tony Johnson (Projektleiter von JAS): Vortrag am 11. Januar 1999 bei DESY, Persönliches Gespräch

[Keuker97] Claus Keuker: "The Central Muon Data Acquisition of the H1 Experiment and its Application", Dissertation at RWTH Aachen, December 1997,
http://mozart.physik.rwth-aachen.de/Diplom-Doktor.html

[Kleinw99] Gespräch mit Claus Kleinwort (H1 Oracle-DB Administrator) am 14. April 1999

[Levoni99] S. Levonian: "L4 Event Classification and Rates", 12-Apr-1999, http://www-h1.desy.de/itrigger/L4Farm/data/l4_rates_99.html

[Nicholls97] T. Nicholls, M. Charlet, J. Coughlan, E. Elsen, D. Hoffmann, H. Krehbiel, H.-C. Schultz-Coulon, J. Schütt, F. Sefkow: "Concept, Design and Performance of the Second Level Triggers of the H1 Detector", IEEE 1997 Nuclear Science Symposium, Albuquerque, New Mexico, USA, November 1997, IEEE Trans. Nucl. Sci., Vol 45, No. 3 (1998) 810,
http://www-h1.desy.de/trigger

[PAW99] Physics Analysis Workstation,
http://wwwinfo.cern.ch/asd/paw/index.html

[POMC99] "Persistent Object Manager Choices", white paper, CERN/RD45/99/01, January 1999,
http://wwwinfo.cern.ch/asd/rd45/recommendations.htm

[Quarrie96] David R. Quarrie: "BaBar: Requirements & Plans for Data Storage and Access", February 1996,
http://www.slac.stanford.edu/BFROOT/www/Computing/Offline/Databases/Event/Talks/20Feb96/Talk20Feb96-1Up.ps

[Rottk98] Wilhelm Rottkirchen: „Weitwinkel-Bremsstrahlung und angeregte Elektronen im H1 Detektor", Mai 1998, Diplomarbeit an der RWTH-Aachen, Physik I

[Sefkow94] F. Sefkow, E. Elsen, H. Krehbiel, U. Straumann: „Experience with the First Level Trigger of H1", 1994 Nuclear Science Symposium, Norfolk, Virginia, USA, October 1994, IEEE Trans. Nucl. Sci. Vol. 42, No. 4 (1995) 900,
http://www-h1.desy.de/trigger

[Shiers98] Jamie Shiers: "Building a Database for the LHC - the Exabyte Challenge", CERN Geneva 1998, Switzerland,
http://wwwinfo.cern.ch/asd/rd45/papers/15ieee.html

B.2 Andere Hinweise

[CDN94] Michael J. Carey, David J. DeWitt, Jeffrey F. Naughton: "The OO7 Benchmark", Technical Report, 21st January 1994, Computer Sciences Department, University of Wisconsin-Madison

[Chaudhri93] Akmal B. Chaudhri: "Object Database Management Systems: An Overview", BCS OOPS Newsletter. No. 18 (Summer 93), pp. 6–15,
http://www.soi.city.ac.uk/~akmal/html.dir/home.html

[Chaudhri95] Akmal B. Chaudhri: "An Annotated Bibliography of Benchmarks for Object Databases", SIGMOD Record, Vol. 24, No. 1 (March 1995), pp. 50–57, http://www.soi.city.ac.uk/~akmal/html.dir/home.html

[Chaudhri96] Akmal B. Chaudhri, Peter Osmon: "A Comparative Evaluation of the Major Commercial Object and Object-Relational DBMSs: GemStone, O_2, Objectivity/DB, ObjectStore, VERSANT ODBMS, Illustra, Odapter and UniSQL", Technical Report, 27th August 1996, The City University, London

[Codd70] E. F. Codd: "A Relational Model of Data for Large Shared Data Banks", Communications of the ACM, vol. 13 #6 (June, 1970)

[DM92] W. H. DeLone, E. R. McLean: "Information systems success: The quest for the dependent variable.", 1992, Information Systems Research, 3(1), 60–95.

[Fowler98] Martin Fowler, Kendall Scott: "UML Distilled", Addison-Wesley, 9th Printing 1998

[Gupta97] Vivek R. Gupta: "An Introduction to Data Warehousing", System Services Corporation, August 1997,
http://system-services.com/dwintro.htm

[Heckner98] Hannes Heckner: „Das neue JDK 1.2 — Ein Überblick", Java™ Spektrum, Ausgabe 05/98

[Horstm99] Cay S. Horstmann, Gary Cornell: "Core Java 2, Band 1 - Grundlagen", pp. 19–27, 1999 Prentice Hall

[Hoschek99] Wolfgang Hoschek: "Partial range searching in OLAP data warehouses", 25th VLDB Conference Edinburgh-Scotland-UK 1999,
http://www.cern.ch/CERN/Divisions/EP/HL/Papers/europe88.pdf

[Inmon96] W. H. Inmon: "Building the Data Warehouse", Second Edition, John Wiley & Sons, 1996

[Jarke99] Matthias Jarke, Maurizio Lenzerini, Yannis Vassiliou, Panos Vassiliadis: "Fundamentals of Data Warehouses", Springer-Verlag 1999

[Kim90] Won Kim: "Architectural Issues in Object-Oriented Databases", Journal of object-oriented Programming, May/June 1989/1990, Vol. 2 No. 1

[Kimball96] Ralph Kimball: "The Data Warehouse Toolkit", John Wiley & Sons, 1996

[McHenry93] Stephen McHenry: "RDBMSs vs. ODBMSs for Product Information Management Systems", Software Institute Report for Autofact, 08th November 1993

[MKP97] Barry L. Myers, Len A. Kappelman, Victor R. Prybutok: "A Comprehensive Model for Assessing the Quality and Productivity of the Information Systems Function: Toward a Contingency Theory for Information Systems Assessment" http://www.year2000.unt.edu/kappelma/framisrc.htm

[Objy] Objectivity Inc., http://www.objectivity.com

[ODMG97] Object Data Management Group, The Standard for Storing Objects, http://www.odmg.org

[StJa96] Martin Staudt, Matthias Jarke: "Incremental Maintenance of Externally Materialized Views", Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB), 1996

[Wade98] Andrew E. Wade: "Hitting the relational wall", White Paper, Objectivity Inc., 1998.

Abbildungsverzeichnis

1.1 Aufbau des H1-Detektors 3
1.2 Trigger Levels des H1-Detektors 5
1.3 Level 2 und Level 4 Trigger 6

2.1 Zwei Histogramme mit unterschiedlichen Rebin-Parametern 11
2.2 Ein Histogramm mit einer angepaßten Funktion 11

3.1 Event-Klassifizierung 18
3.2 Level 5 Rekonstruktion, Offline-Datenfluß 20
3.3 Data-Warehouse Modell 23
3.4 Data-Warehouse-Sicht bei H1 25
3.5 Arbeitsfluß der Daten-Aktualisierung bei H1 28
3.6 Schritte bei der physikalischen Analyse eines H1-Benutzers 30
3.7 Data-Marts Hierarchie (Beispiel) 35

4.1 Objekt-Hierarchie in Objectivity/DB 43
4.2 Aufbau der Test-Datenbank Version 2 44
4.3 Aufbau der Test-Datenbank Version 3 45
4.4 TTagRun, ein Basic-Object in Objectivity/DB 46
4.5 Benchmark: Java Analysis Studio, C++-Programm 50

5.1 JAS-Objy-DIM: Grafische Benutzerschnittstelle 62
5.2 Zugriff des DIMs auf Objectivity/DB 64
5.3 JAS-Objy-DIM: Direkter Datenbankzugriff 66
5.4 JAS-Objy-DIM: Datenbankzugriff über den RMI-Server 67
5.5 Die Java-Klassen des DIMs 68
5.6 Java-Klasse TTagEvent, ein Basic-Object in Objectivity/DB 69

6.1 Bewertungsmodell für Informationssysteme nach DeLoan und McLean . . . 78

A.1 New Job Wizard: Type of Job 88
A.2 New Job Wizard: Select a DIM 88
A.3 New Job Wizard: Objectivity DIM Settings 89
A.4 Java Analysis Studio: Selection Tree 90

Tabellenverzeichnis

4.1 Eventgröße in Objectivity/DB bei H1. 48
4.2 Benchmark: Zugriffszeiten der Anfragen an Objectivity/DB durch Java Analysis Studio und ein C++ Programm 53
4.3 Benchmark: Mehrere Analysen in derselben Applikation 53

6.1 Systemqualitätsfaktoren bei JAS und Java, dem JAS-Objy-DIM und Objectivity/DB 80