Persistent ATLAS Data Structures and Reclustering of Event Data

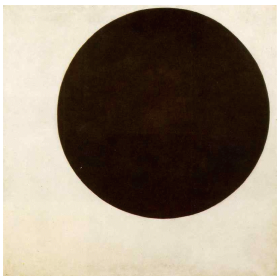
DISSERTATION

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften

eingereicht an der
Naturwissenschaftlichen Fakultät der
Leopold-Franzens-Universität Innsbruck

von
Dipl.-Ing. Martin Schaller

Genf, September 1999



Kasimir Malevich
Black Circle, 1913,
Oil on canvas, 105.5 x 105.5 cm
State Russian Museum, St. Petersburg

Dedicated to the little Eskimo woman
and to the tiger in the jungle.

Acknowledgement

This work was carried out within the scope of the Austrian Doctoral Student Program in the EP/ATC group at CERN, Geneva, Switzerland.

I would like to express my thanks to my supervisors Prof. Dietmar Kuhn and Dr. Jürgen Knobloch. I would also like to express my thanks to all members of the ATLAS computing group, especially RD Schaffer who was a great help in the beginning.

I want to thank Christopher Williams for correcting the English language of this thesis. Steven Goldfarb corrected the English of one conference paper. Thanks also to him. This thesis profited from numerous discussions I had with Johannes Gutleber, Koen Holtman, Wolfgang Hoschek, and Dietmar Schweiger. I want to thank Pierre Bauer for his interest in the foundations of quantum mechanics and for the related discussions we had together with Thomas Toifl.

I will always remember the people with whom I shared the house Ferme de l'Espeneux. My stay in Geneva was pleasant due to the friendship with many people, too many to list them all here. Especially I want to thank all the people with whom I had the pleasure to explore the climbing sites in the neighborhood of Geneva.

Contents

Abstract

The ATLAS experiment will start to take data in the year 2005. The amount of experimental data forms a serious challenge for data processing and data storage. About 1 PB (10^{15} bytes) per year has to be processed and stored. Currently, a paradigm shift in High-Energy Physics (HEP) computing is taking place. It is planned that software is written in object-oriented languages (mainly C++). For data storage the usage of object-oriented database management systems (ODBMSs) is foreseen.

This thesis investigates the usage of an ODBMS in the ATLAS experiment. Work was done in several connected areas. First, we present exhaustive benchmarks of the commercial ODBMS Objectivity/DB that is today the most promising candidate for the storage system. We describe the ATLAS 1 TB milestone that was performed to investigate the reliability and performance of an ODBMS storage solution coupled to a mass storage system. Second, we report about the design and implementation of the persistent ATLAS data structures, both in the detector description and event domain. We describe the implementation of the AMDB (ATLAS Muon Database), the design of the raw event model and the implementation of several event collection classes.

The most important result and the main novel contribution of this thesis is a new reclustering algorithm for the event data. Clustering describes the object placement on disk. It is one of the most effective performance enhancement techniques for ODBMSs. We describe a reclustering algorithm for objects contained in several (possible overlapping) collections. The algorithm works by decomposing the collections into a set of non-overlapping atomic regions. The objects within each of these finer subsets are stored together; thus the issue is how to order the subsets. The problem is mapped to a weighted graph resulting in an instance of the traveling salesman problem. A standard heuristic can be chosen to find an approximate solution. We can show that under a set of realistic and natural assumptions the algorithm reduces the number of disk seeks almost to the theoretical lower limit. We describe the design and implementation of a prototype. The reclustering algorithm is qualitatively and quantitatively analyzed. The experimental results are presented and compared with the theoretical model.

Zusammenfassung

Das ATLAS Experiment geht im Jahr 2005 in Betrieb. Die Menge der zu erfassenden Daten erreicht im Vergleich zu vergangenen Hoch-Energie Physik (HEP) -Experimenten neue Grössenordnungen und stellt eine ernsthafte Herausforderung an die Planung der Datenverarbeitung und Datenspeicherung dar. Pro Jahr werden 1 PB (10^{15} bytes) an Daten verarbeitet und gespeichert. Momentan findet in der HEP-Datenverarbeitung ein Paradigmenwechsel statt. Die Software wird statt in Fortran nun in objekt-orientierten Programmiersprachen (hauptsächlich C++) geschrieben. Als Speicherlösung findet ein objekt-orientiertes Datenbank-Management-System (ODBMS) Verwendung.

Die vorliegende Arbeit untersucht die Verwendung von objekt-orientierten Datenbanken im ATLAS Experiment. Wir präsentieren ausführliche Benchmarks einer kommerziellen Datenbank, deren Verwendung geplant ist. Ergebnisse des 1 TB-ATLAS-milestones sind inkludiert. Weiters beschreiben wir das Design und die Implementierung persistenter ATLAS Datenstrukturen, sowohl im Detektor- als auch im Eventbereich. Wir beschreiben die Implementierung der AMDB (ATLAS Muon Database), das vorläufige Design des Event Modells, das zur Speicherung der Rohdaten herangezogen wird, sowie die Implementierung und den Vergleich verschiedener Klassen von Event-Kollektionen.

Das Hauptresultat der vorliegenden Arbeit ist ein neuer Clustering-Algorithmus für (möglicherweise überlappende) Event-Kollektionen und die von ihnen referenzierten Events. Clustering beschreibt die Objekt-Plazierung von Objekten im sekundären Speicher. Event-Kollektionen sind Container-Objekte, die die zu analysierenden Events beinhalten. In Anbetracht der großen Datenmengen werden sehr hohe Anforderungen an die Performance der Datenbank gestellt. Das gezielte Clustering von Objekten ist eine der effizientesten Maßnahmen, um die Performance zu erhöhen.

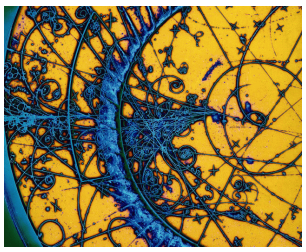
Der vorgeschlagene Clustering-Algorithmus nutzt die Tatsache aus, daß die Ordnung der Events innerhalb einer Kollektion vom physikalischen Standpunkt aus unerheblich ist. Der Algorithmus arbeitet in zwei Schritten. Zuerst wird die Menge der Events in nichtüberlappende atomare Regionen aufgeteilt. Danach werden die atomaren Regionen auf einen gewichteten Graphen abgebildet. Die Knoten des Graphen entsprechen den atomaren Regionen. Die Distanz zwischen zwei Knoten ist die Anzahl der Event-Kollektionen die genau eine der beiden atomaren Regionen beinhalten.

Es wird in der vorliegenden Arbeit gezeigt, daß das Clusteringproblem damit zum Problem des Handlungsreisenden äquivalent wird. Weiters wird bewiesen, daß unter einer Anzahl von realistischen Annahmen die kürzeste Tour im Graphen derjenigen Objektanordnung entspricht, die die geringste Anzahl von Festplatten-Suchzugriffen benötigt, um die Kollektionen zu lesen.

Ein Prototyp, der diesen Reclustering-Algorithmus implementiert, wird beschrieben. Die experimentellen Ergebnisse werden mit einem theoretischen Modell verglichen. Die Arbeit vergleicht auch experimentell zwei Heuristiken für das Problem des Handlungsreisenden in Hinblick auf ihre Eignung für dieses Problemgebiet.

Chapter 1

Introduction



BEBC (Big European Bubble Chamber)
particle tracks color treated picture
CERN photo

This chapter gives a brief introduction to high-energy physics (Section 1.1) that sets the background of the present work. Section 1.2 describes the Large Hadron Collider (LHC) currently being built at CERN. In Section 1.3 we describe ATLAS, one of the LHC experiments. Section 1.4 gives an overview of data flow and data analysis in a high-energy physics experiment. It serves as motivation for the ATLAS data structures described in Chapter 4 and the description of a new reclustering algorithm in Chapter 5.

1.1 High-Energy Physics

The purpose of this chapter is to give a brief introduction to high-energy physics. High-energy physics investigates elementary particles and their interactions. The reductionist's point of view is that in principle every physical phenomena can be derived from the interaction between these elementary particles. The particles are elementary in the sense that there is no experimental hint or convincing theoretical argument that they have internal structure. The elementary particles fall into three different classes.

- *Quarks.* The atomic nucleus is built out of protons and neutrons. These particles are not themselves elementary, they are composed of smaller particles, called

Table 1.1.1 Quarks (Spin $\frac{1}{2}$) [26]

Flavor	Charge	Mass (MeV)
u	$+\frac{2}{3}$	1.5 – 5
d	$-\frac{1}{3}$	3 – 9
c	$+\frac{2}{3}$	1.1 – 1.4
s	$-\frac{1}{3}$	60 – 170
t	$+\frac{2}{3}$	173.8 ± 5.2
b	$-\frac{1}{3}$	4.1 – 4.4

quarks. Particles that consists of quarks are either built out of two or three quarks. Particles with three quarks are called *baryons*. The proton and the neutron are the lightest baryons. Today more than 100 baryons are known. Baryons other than the proton are not stable and decay into lighter particles. Baryons have spin 1/2 and belong therefore to the class of particles with fractional spin $n + 1/2$ where n is a natural number. These particles obey the Pauli-principle saying that no two of them can have the same set of quantum numbers. Particles with spin $n + 1/2$ are called *fermions*.

The six known quarks, *up (u), down (d), charme (c), strange (s), top (t)* and *bottom (b)* fall into three *families* or *generations*

$$\begin{pmatrix} u \\ d \end{pmatrix}, \begin{pmatrix} c \\ s \end{pmatrix}, \begin{pmatrix} t \\ b \end{pmatrix}.$$

The two families to the right are heavier versions of the family to their left. The quarks carry fractional electrical charge. The u, c, and t quark carry a charge of 2/3 of the elementary charge e = 1.602×10^{-19} C , whereas the other three quarks d, s, and b carry the charge $-1/3$ e. One peculiarity of the quarks is that a single quark was never observed in isolation. This fact is called *quark confinement*. The quarks have an additional characteristic. Each quark comes in one of three different "colors": red, green, and blue. Quark color is an abstract property. One of the reasons for this naming is the following: All naturally occurring particles are colorless. Antiparticles (see below) carry the corresponding "anti-color." Either the total amount of each color is zero or all three colors are present in equal amounts. Table 1.1.1 depicts the charge and the speculative mass of the quarks.

- *Leptons*. The six known leptons are arranged in three families as well. The *electron (e)*, *muon (μ)* and the *tau (τ)* carry a charge of -1. The neutrinos *electron neutrino* (ν_e), *muon neutrino* (ν_μ), and the *tau neutrino* (ν_τ) are electrically neutral and have no or very little mass. We arrange the leptons in three families

$$\begin{pmatrix} e \\ \nu_e \end{pmatrix}, \begin{pmatrix} \mu \\ \nu_\mu \end{pmatrix}, \begin{pmatrix} \tau \\ \nu_\tau \end{pmatrix}.$$

The leptons have spin 1/2 and belong therefore to the fermions. The μ and τ decay

Table 1.1.2 Leptons (Spin $\frac{1}{2}$) [37]

Lepton	Charge	Mass (MeV)	Lifetime (s)	Principal decays
e	-1	0.511003	∞	–
ν_e	0	0	∞	–
μ	-1	105.659	2.197×10^{-6}	$e\nu_\mu\overline{\nu}_e$
ν_μ	0	0	∞	–
τ	-1	1784	3.3×10^{-13}	$\mu\nu_\tau\overline{\nu}_\mu, e\nu_\tau\overline{\nu}_e, \rho\nu_\tau$
ν_τ	0	0	∞	–

Table 1.1.3 Mediators (Spin 1) [37]

Mediator	Charge	Mass (GeV)	Lifetime (s)	Interaction
gluons $g_i, i = 1, \ldots, 8$	0	0	∞	strong
photon (γ)	0	0	∞	electromagnetic
$W^\pm$	± 1	81.800	unknown	(charged) weak
Z^0	0	92.600	unknown	(neutral) weak

into the electron. In contrast to the quarks the leptons may exist for themselves. Table 1.1.2 gives an overview of the leptons.

- *Mediators.* The mediators allow the interactions among quark and leptons. They are *gauge bosons* of the field which enables fundamental interactions such as scattering, binding or decay. The gauge boson of the electromagnetic interaction is the photon γ. The two *W's* and *Z bosons* are the gauge bosons of the weak interaction. The strong interaction is realized by eight different *gluons*. The carrier of the gravitational interaction is the hypothetical *graviton*. Table 1.1.3 gives an overview of the mediators.

Antiparticles are a profound and universal feature of quantum field theory. For every kind of particle there exists an *antiparticle*, with the same mass but opposite electric charge. This leads to 6 antiquarks and 6 antileptons. Only the gauge bosons are their own antiparticles. The fact that there is much more matter than antimatter in this universe is not understood yet. Antiparticles are denoted by a bar, for example the antiparticle of the electron e is the anti-electron or *positron* $\overline{e}$.

In classical theories, e.g. Maxwell's theory of electromagnetism, particles interact through a field. Today it is believed that all interactions of nature are a result of particle exchange. Figure 1.1.1 shows two electrons that interact by exchanging a single photon. Either attractive and repulsive forces can be described by photon exchanges. The photon is used to describe the electromagnetic interaction, the photon itself can not be observed, it is a *virtual* photon. All electromagnetic phenomena are ultimately reducible to the absorption or emission of a (virtual) photon. Interactions between

Table 1.1.4 The interactions treated by high-energy physics [40].

Interaction	Range	Typical Lifetime (sec)	Typical Cross Section (mb)	Typical Coupling α_i
Strong	$1F \simeq \frac{1}{m_\pi}$	10^{-23}	10	1
Electromagnetic	∞	$10^{-20} \sim 10^{-16}$	10^{-3}	10^{-2}
Weak	$\frac{1}{M_W}$	10^{-12} or longer	10^{-11}	10^{-6}

particles can be represented by *Feynman diagrams* like the one shown in Figure 1.1.1. The fundamental interactions between between elementary particles are:

- *Gravitation.* The gravitational interaction plays a minor rôle in high-energy physics. The gravitational force is very weak compared to the others. The fact that the gravitational interaction is the one most clearly observable in every days life results from its cumulative character and its long range. The classical theory of gravity is Newton's law of universal gravitation. Its relativistic generalization is Einstein's general theory of relativity. A complete satisfactory quantum theory of gravity has yet to be worked out.

- *Electromagnetic Interaction.* It describes the electromagnetic interaction between charged particles. The quantum mechanical formulation of electrodynamics is called quantum electrodynamics (QED).

- *Strong Interaction.* The binding of protons and neutrons in an atomic nucleus is described by the strong interaction. The strong force is described in quantum chromodynamics (QCD). The range of the strong interaction is about the size of the nucleus itself. Only the quarks experience the strong interaction.

- *Weak Interaction.* In contrast to the strong interaction, the weak gauge bosons interact with acts on both, quarks and leptons. For example it is responsible for the beta-decay
$$n \rightarrow p^+ + e^- + \overline{\nu}.$$
The Glashow-Weinberg-Salam (GWS) theory unifies the electromagnetic and weak interaction using the concept of gauge bosons.

Figure 1.1.2 shows the Feynman diagrams of typical interactions. Table 1.1.4 shows differences in the behavior of the interactions.

The Standard Model is a set of related theories that describes the interactions between the elementary particles. One prediction of the standard model is the existence of a further particle called the *Higgs particle*. The Higgs particle is used to explain that the $W^\pm$ and Z^0 mediators have mass. The LHC will provide such high energies that allow physicists[1] to search for the Higgs particle over a wide mass range.

[1]We do not want to imply a particular gender using the word physicist. "Physicist" should be read as "female or male physicist" throughout this thesis.

Figure 1.1.1 Two electrons interact by exchanging a photon.

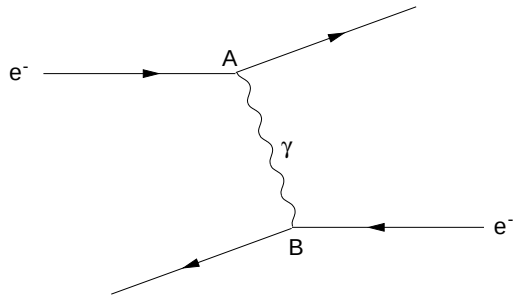
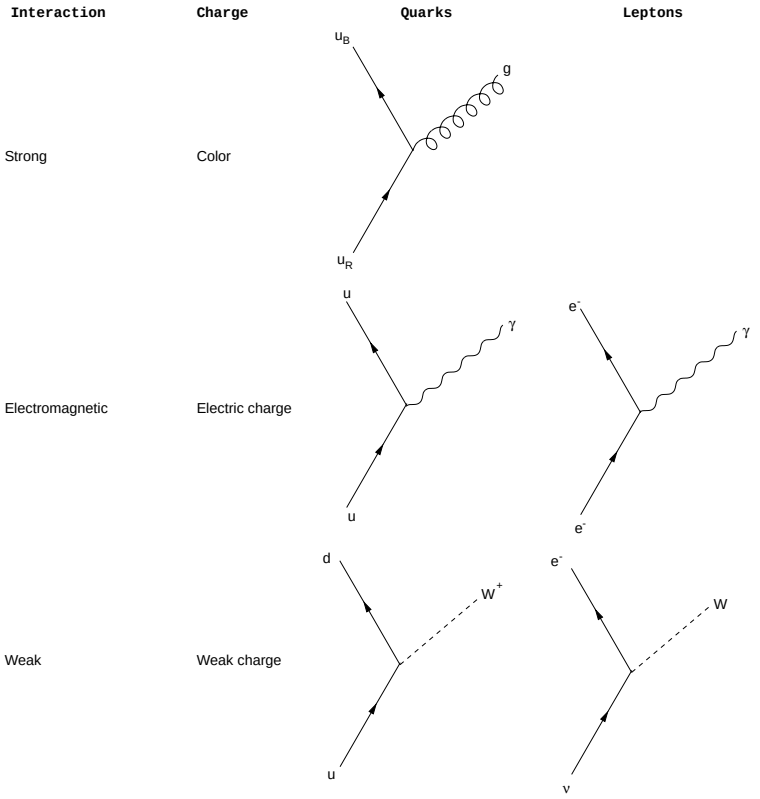
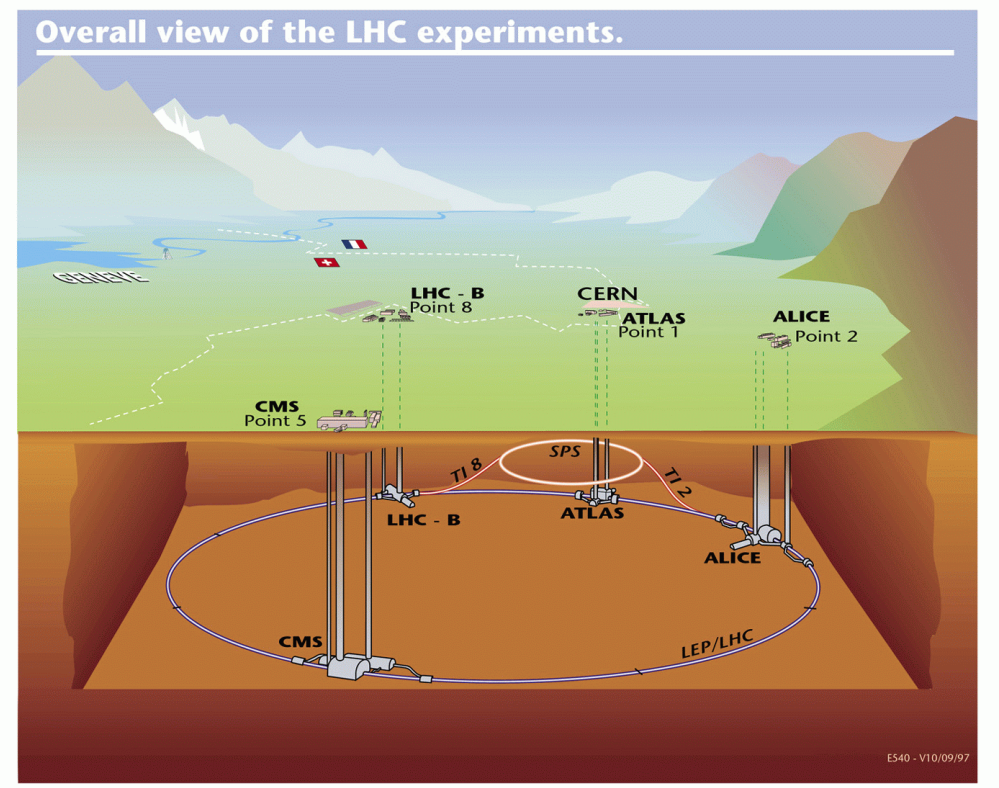


Figure 1.1.2 Diagrams showing typical interactions



Due to the high energies required for creating new, yet undiscovered particles, high-energy physics has become a synonym for particle physics. The other reason to justify the high energies stems from Heisenberg's *uncertainty relation*: $\Delta p \Delta x \geq \frac{h}{4\pi}$. To explore dimensions in the range Δx momenta greater than $\frac{h}{4\pi\Delta x}$ are required where h denotes Planck's constant: $h = 1.055 \times 10^{-34}$ Js.

Figure 1.2.1 LHC Overview, CERN photo



For a more detailed introduction to high-energy physics consult e.g. [37], [40], [62] or [56].

1.2 The Large Hadron Collider

The Large Hadron Collider (LHC) [39] is a new particle accelerator that is currently being built at CERN and scheduled for operation in the year 2005. It will be installed in the 27 km tunnel at CERN, Geneva that currently houses the Large Electron Positron Collider (LEP). The LHC is a particle accelerator which brings protons or ions into head-on collisions at higher energies than ever achieved before. It has two beam pipes where particles are accelerated in opposite directions before they collide. 1232 superconducting dipole bending magnets keep the beams in the beam lines. Each magnet has a magnetic field strength of 8.4 T. The protons are bunched such that counter-rotating bunches of about 10^{11} protons cross another every 25 ns in the interaction regions of the experiments built at the LHC. The bunch-crossing rate is therefore 40 MHz. With filling, they will undergo acceleration up to their nominal collision energies in about 20 minutes. The protons in these bunches carry each an energy of 7 TeV, yielding an energy of 14 TeV in the centre-of-mass frame. When the two counter-moving proton

bunches cross, protons from the bunches can collide, producing new particles in inelastic interactions. Such inelastic interactions are also referred to as events.

The probability for such inelastic collisions to take place is determined on one hand by the cross section for proton-proton interactions, on the other hand by the density and frequency of the proton bunches. The latter quantity, which is a characteristic of the collider, is the beam luminosity $\mathcal{L}$. The targeted luminosity of the LHC is $\mathcal{L} = 10^{34}$ $\mathrm{cm}^{-2}\mathrm{s}^{-1}$. The proton-proton inelastic cross section σ_{inel} is energy-dependent. Due to the fact that such high energetic particle collisions were never systematically studied before, the cross section is not exactly known. It can, however, be predicted by extrapolation from measured values at lower energies and theoretical models. The total inelastic, non-diffractive cross-section at LHC energies will be about 100 mb, corresponding to an interaction rate of 10^9 Hz. The number of inelastic interactions per second, the interaction rate, is given by the product of cross section and luminosity:

$$\mathcal{L} \times \sigma = 10^{34}\ \mathrm{cm}^{-2}\mathrm{s}^{-1} \times 100\ 10^{-27}\ \mathrm{cm}^2 = 10^9\ \mathrm{s}^{-1}.$$

This interaction rate corresponds to 23 interactions per bunch crossing. This interactions are also sometimes called events but we preserve the word "event" for all the interactions that take place in a bunch crossing.

Most interactions are due to collisions at a relatively large distance between incoming protons where protons interact as "a whole". These interactions lead to a small momentum transfer. The particles in the final state have large longitudinal momentum but small transverse momentum. These are called minimum-bias events ("soft events"). They are the large majority but are not very interesting.

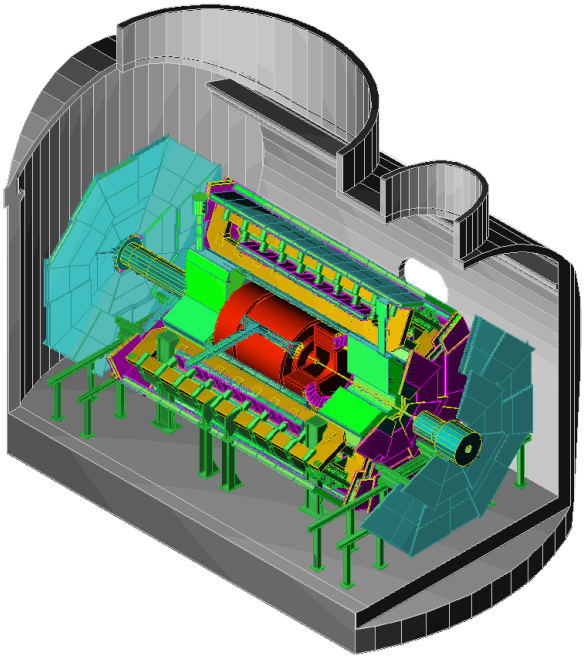
The monochromatic proton beam can be seen as a beam of quarks and gluons with a wide band of energy. Occasionally hard scattering ("head on") between constituents of incoming protons occur. These interactions take place at a small distance leading to a large momentum transfer.

This high number of particles generated poses a serious challenge to the detector and its data[2] analysis. The task of the LHC experiments is to identify and select the interesting events on top of this background. For example, in the case of the Higgs particle with mass 500 GeV, about 17 K events will be produced per year, compared to a total of 1.7×10^{16} events from the inelastic interactions.

The LHC can also be used to collide beams of heavy ions, e.g. lead ions, with a center-of-mass energy of about 6 TeV per nucleon, but with lower luminosity ($\sim 10^{27}$ $\mathrm{cm}^{-2}\mathrm{s}^{-1}$) and interaction rate of $\sim 10^4$ s^{-1}, compared to 10^9 s^{-1} for pp collisions.

LHC will host four experiments. Figure 1.2.1 shows the LHC and the location of the four experiments. ATLAS and CMS are general-purpose pp experiments. LHCb is a pp experiment dedicated to b-quark physics and CP violation. ALICE is a heavy-ion experiment (Pb-Pb collisions) dedicated to quark-gluon plasma studies.

Figure 1.3.1 The ATLAS Detector



1.3 The ATLAS Experiment

ATLAS is the acronym for A Toroidal LHC Apparatus. Figure 1.3.1 shows the ATLAS detector. The ATLAS detector [16] is a spectrometer, containing detector components for calorimetry, particle identification and particle tracking. The detector is composed of individual sub-detectors, which specialize in different capacities of particle identification and measurements of signatures, plus structures for the provision of magnetic fields, cooling and mechanical support, and electronic instrumentation. The detector is a general purpose proton-proton detector which was designed to exploit the full discovery potential of the LHC. A major focus of interest is the origin of mass at the electroweak scale. It is sensitive to the largest possible Higgs mass range. Other important goals are the searches for heavy W- and Z-like objects, for supersymmetric particles, for compositeness of the fundamental fermions, as well as the investigation of the CP violation in B-decays, and detailed studies of the top quark. The overall detector will have a length of 50 m, a diameter of 25 m and a weight of 7000 tons. The ATLAS detector includes an inner tracking detector inside a super-conducting solenoid which generates a 2 T magnetic field along the beam axis, electro-magnetic and hadronic calorimeters outside the solenoid and muon spectrometers in the forward, barrel and end-cap regions. A spectrometer measures momentum and energy. The particle energy is measured in calorimeters. The momentum is measured from the bending radius r in the magnetic

[2]We will use the word *data* in both singular and plural, which is common in database literature. Context will determine whether it is singular or plural. In standard English, *data* is used only as the plural; datum is used as the singular.

field. The particle tracking system must measure at least three points in order to determine the radius. The ATLAS inner detector from the outside to the inside consists of a Transition Radiation Tracker (TRT), the Semi-Conductor Tracker (SCT, barrel and forward direction) and an inner pixel detector for tracking. The detector is exposed to high radiation doses, its components have to be designed using radiation-hard technology.

1.4 Data Analysis

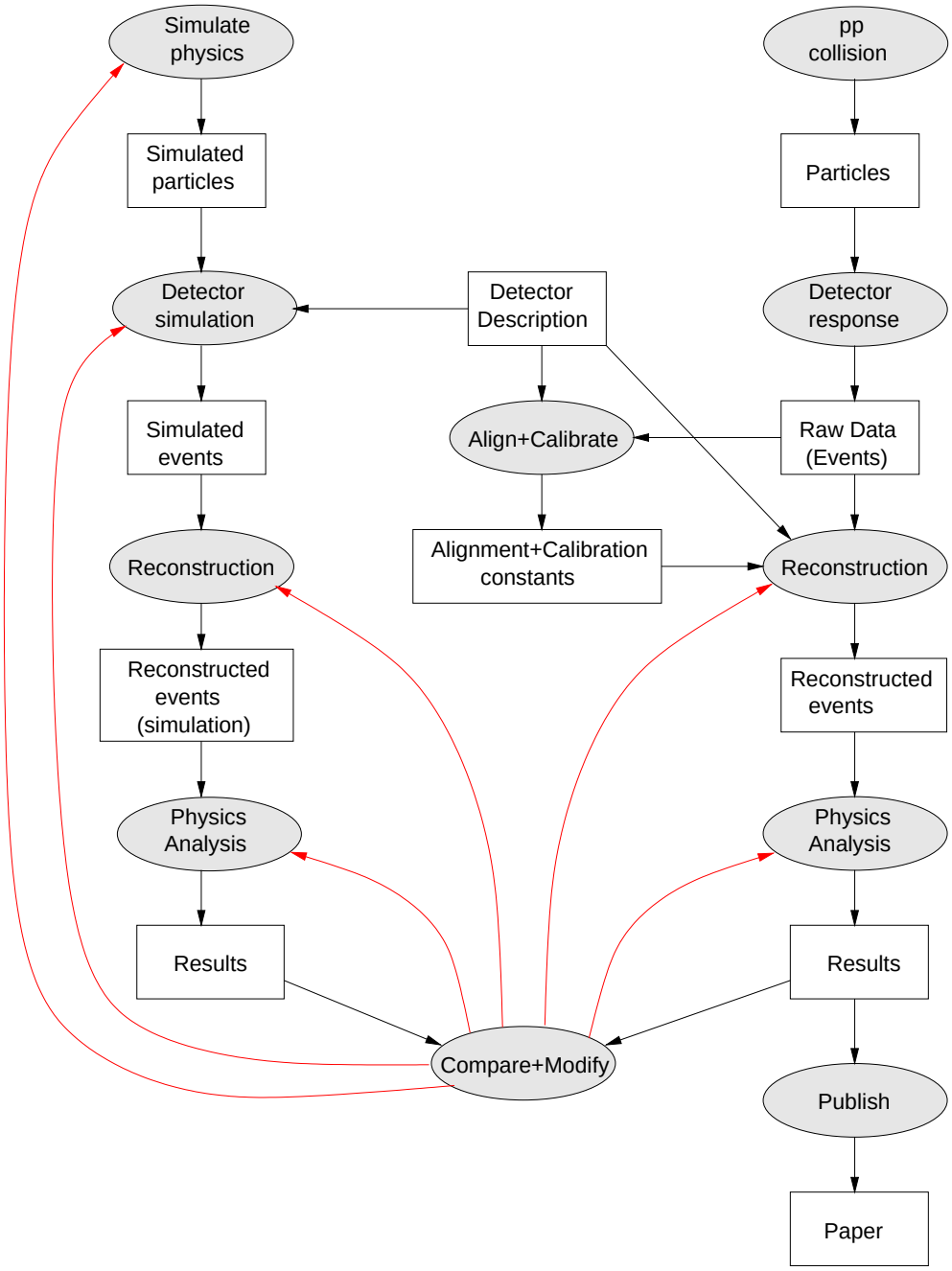
Figure 1.3.1 illustrates the data flow in a high energy physics experiment, see also [5], Figure 3-3. The high-energy particle collisions (or events) inside the detector lead to the creation of new particles. These particles interact with the detector and produce electrical signals. Not all the events are stored. A chain of so-called *trigger levels* selects the interesting events. The three trigger levels reduce the event rate from 40 MHz to 100 HZ.

The data associated with the collision is called *raw data*. It consists mainly of a set of *digits*. A digit is an address-value pair, where the address specifies the detector element that took the reading and the value is the outcome of the electronics. The reconstruction process uses the raw data to identify the particles and their properties. There are a number of facts that make this task quite challenging. First the passing of a particle through the detector only leads to a number of discrete space-time points, each subject to unavoidable measurement errors. High numbers of created particles make it difficult to assign the digits to the right particles. Even worse, electrically neutral particles do not interact with the detector at all. Their existence can be inferred indirectly , e.g by considering the energy or momentum balance or by their decays into charged particles. The reconstruction process can be divided into *pattern recognition* (i.e *track finding*), *track fitting* and *track matching*. Track finding is the problem of partitioning the set of the detector's hit-position measurements into subsets whereby all hits in a partition are caused by the same particle. Track fitting assigns trajectories to the identified tracks. The track matching problem arises from the utilization of different detector subsystems. Tracks found in an outer detector have to be matched with the tracks found in an inner detector. The detector is a highly complex measuring instrument. Software simulation and testbeam results are indispensable tools and form the basis for the understanding of the complete detector.

The ATLAS collaboration will investigate proton-proton collisions at very high energies. The largest part of the data stored in the database is the raw data observed from the ATLAS detector. The data is split up into chunks, where each data chunk corresponds to a particle collision, called an event. After data taking important event attributes are precomputed to speedup the query execution, e.g. the trajectories of the particles found in the event.

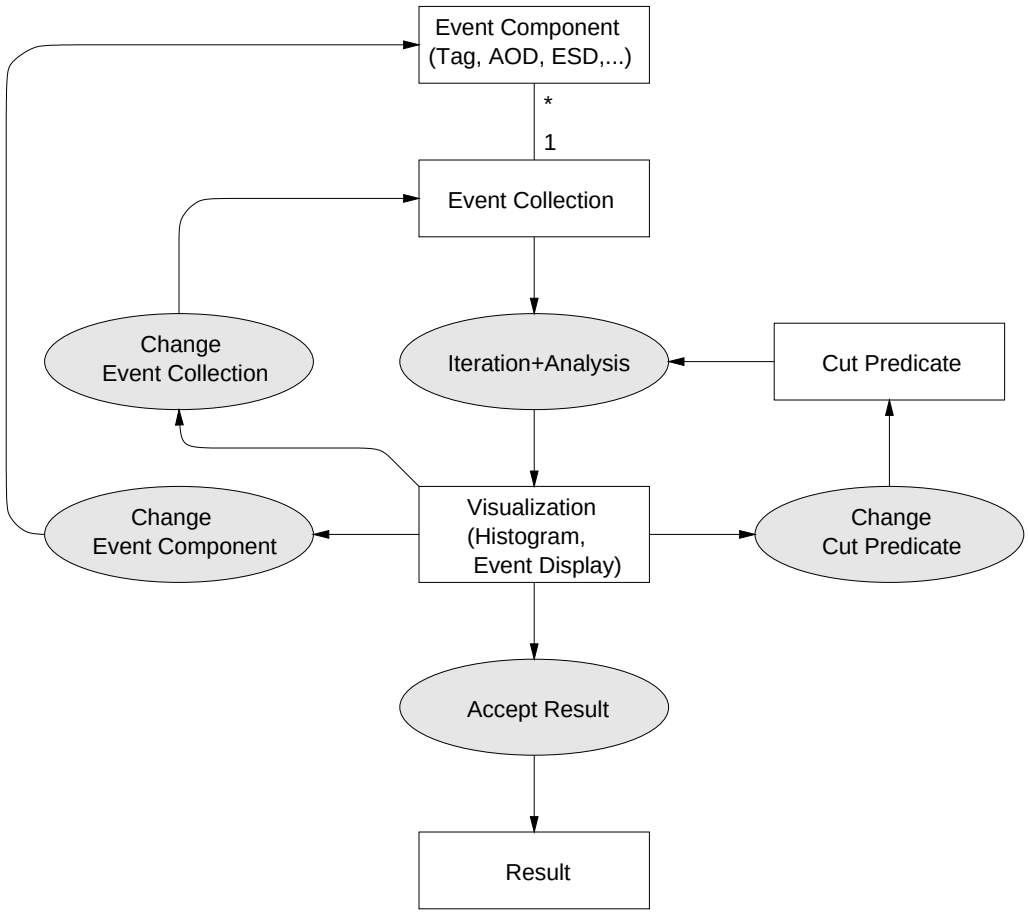
The event data is hierarchical in nature with the bulk of the data being rarely accessed, and additional levels with varying reduction factors being accessed more frequently. The largest part is the raw data having a size of about 1 MB. The other object groups are the Event Summary Data (ESD) with a size of 100 KB, the Analysis Object Data (AOD) with a size of 10 KB, and the Event tag with a size of 100 bytes. The data and the access to the data possess some special properties that simplify the storage management. First, the data itself is mainly write-once, read-many. Furthermore, each

Figure 1.4.1 The Data Flow in a High Energy Physics Experiment



event is independent of another – there are no relationships between individual events. The data will be stored in a two-level storage hierarchy, consisting of a multi-100 TB disk farm and a mass storage system. Analysis jobs will traverse collections with up to 10^9 events. ATLAS will have hundreds of physicists analyzing data, many of whom

Figure 1.4.2 Overview of the Analysis Process



require different collections of events and different data objects from each event for their analysis. The analysis scenario consists roughly of the following steps ([5], p. 21,22):

1. Looping over the collection of events of interest, applying a current set of selection criteria and producing output which can be viewed interactively;

2. Histogramming the results of 1. to understand the effects of the selection criteria on the data, and to improve/vary the criteria.

The physicist is interested in a certain sample of all events in the database. A set of selection criteria, called *cut-predicate* define the sample. The analysis process is very performance intensive. Several techniques are commonly used in HEP to speed up the analysis process.

- *Event collections.* The analysis is not run against all the events in the database but against a preselected sample, the event collection. Since the cut-predicate changes slightly from analysis to analysis, the event collection has to be large enough to include all events that fulfil the current cut predicate. An event collection can be

regarded as a *materialized view*, see e.g. [1]. The concept of an materialized view stems from relational database theory. In relational database theory a view is a table that is derived from other tables. For example, a view can be the joint of two tables or it can be a sub-table of an existing table (i.e. a composition of a projection and selection). The view is called *virtual*, if only the definition of the view is stored in the database and not the table that corresponds to a view. For example if the view represents a join of two tables the join has to be executed every time a query is run against the view. The view is called *materialized* if the view exists in physical form. The materialization of the view trades some storage space for performance speedup. Event collections are absolutely essential for data analysis. The performance degradation that would result if every analysis would have to run against the whole database would be enormous.

- *Hierarchical event model*. The event information exists in different versions that differ in the level of detail. The event summary data (ESD) contains all information of the reconstructed event. The reconstructed event is the outcome of the reconstruction process. Since the reconstructed event can always be computed from the raw data there is no absolute reason to store it. The reason why it is done is again a performance enhancement. The reconstructed event is an example of *function materialization* [52]. The other versions of the event, the event tag and the analysis object data (AOD) contain a subset of the information of the ESD. These two versions can be seen as a materialized view. Choosing the smallest event version that contains sufficient information for an analysis can lead to tremendous speedups.

Figure 1.4.2 displays the analysis loop in more detail. Depending on the stage of the analysis the analysis job might not access the whole event but only a sub-component (e.g. event tag, AOD, or ESD). Not all of the events in the event selection are selected by the analysis job. The events are selected by selection criteria, called the *cut predicate*. Most of the time the physicist will slightly change the cut predicate and study the impact of these changes. In the HEP environment, data analysis is often increasingly selective: subsequent analyses often access smaller and smaller subsets of the data. From time to time the physicist will generate a smaller event collection taking the higher selectivity of the cut predicate into account. It may also happen that the currently used event component does not contain enough information to continue the analysis task. At this moment the physicist will change to an event component that contains more information.

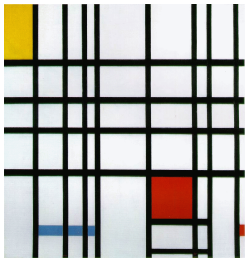
A possible analysis scenario for the ATLAS experiment at the LHC is as follows [77]:

- Some 20 working groups, each consisting of 10-30 people, are involved in the analysis for the data at any time.
- These working groups correspond to approximately 150 users accessing the data concurrently.
- Each analysis group looks at an event sample of some 10^9 events, producing subsamples for further analyses.
- These subsamples are then analysed 10-15 times per day.

A detailed analysis scenario for a past experiment can be found in Ref. [27]. Ref. [58], [77], and [76] describe the problems one has to face building a storage and analysis system for the coming LHC experiments.

Chapter 2

Object-Oriented Database Management Systems



Piet Mondrian
Composition with Gray and Light Brown
1918; Oil on canvas, 80.2 x 49.9 cm
Museum of Fine Arts, Houston, Texas

"The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination."

Frederick P. Brooks, Jr. [29]

Object-oriented databases will play a central role in the ATLAS storage system. Object-Oriented Database Management Systems (ODBMSs) are now adopted for data persistency not only by all the LHC experiments but by many others (BaBar, NA45, COMPASS, RHIC) ready to take data in 1-2 years. The research and development group RD45[1] at CERN was founded to investigate object persistency for HEP, see e.g. their status reports [66], [65], [67], and [68]. This chapter is organized as follows. Section 2.1 gives an overview of the object model. The next Section 2.2 introduces ODBMSs. Section 2.3 and Section 2.4 discuss ODBMSs in more detail. The last Section 2.5 treats the implementation of ODBMSs.

[1] http://wwwinfo.cern.ch/asd/rd45/

Figure 2.1.1 Block diagram of a conventional program [7]

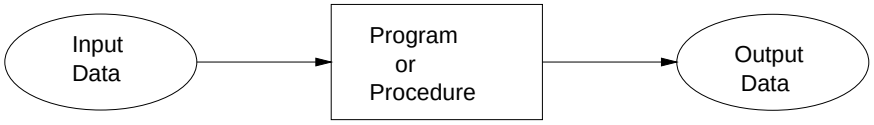
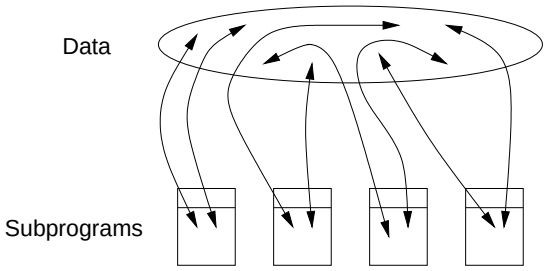


Figure 2.1.2 The Topology of Conventional Programming Languages [8]



2.1 The Object Model

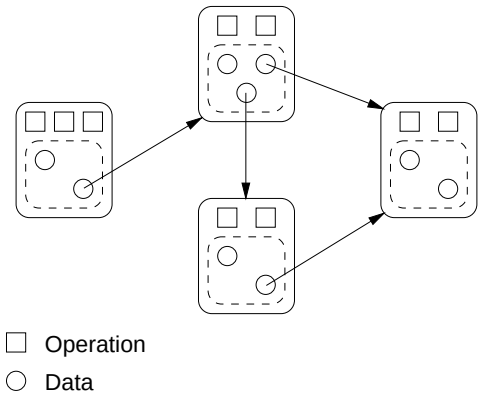
The object-oriented approach to problem solving provides a model that consists of objects sending messages to other objects. Object-oriented programming has gained enormously in popularity in the last decade. Expectations generally cited for object technology include:

- improved software quality, reliability, testability, and extensibility
- shorter development times
- increased programmer productivity
- greater reusability of code.

In this section we will give a very brief introduction to object-oriented programming and give definitions for the most frequently used concepts. Due to the limited space it can be no more than merely scratching the surface. For a more comprehensive introduction we refer the reader to the literature. The book of Booch [8] is one of the most cited ones in relation to object-oriented analysis and design. Introductions to object-oriented programming might also be found in the books that describe the individual object-oriented languages. The following ones are written by the creators of the respective languages themselves: Stroustrup [81] describes C++, Arnold and Gosling [3] Java, Meyer [60] Eiffel, and Blaschek [7] Omega.

Conventional programming is procedure-oriented. The focus is on the processing – the algorithm needed to perform the desired computation. This is best illustrated by a block diagram, as shown in Figure 2.1.1. The topology of a conventional programming language, say FORTRAN, is depicted in Figure 2.1.2. The basic physical building block

Figure 2.1.3 Block diagram of an object-oriented program [7]



is the subprogram. The arrows in this figure indicate dependencies of the subprograms on various data. An error in one part of a program can have a fatal effect across the rest of the system because the global data structures are exposed for all subprograms to see. Data can be fairly complex and highly structured, but this is primarily seen as an additional difficulty in getting the algorithm right. The program has to deal with many sorts of data with different properties. All these differences must be taken care of in the algorithm. Needless to say, just a minor change in the data structures (such as the introduction of a new kind of data) can easily cause the whole program to break down.

In contrast, data plays a central role in object-oriented programs. Different sorts of data are represented by different kinds of object, each with its own structure and behavior. Although the objects themselves are generally quite simple, complex structures can be built by connecting objects with each other. An *object* unites data and algorithms. There is little or no global data. The physical structure of an object-oriented program appears as a graph, called the *object graph*, see Figure 2.1.3.

All common higher-level programming languages have some kind of built-in data types (INTEGER, FLOAT, CHARACTER). Object-oriented languages allow the user to define types that behave in the same way as built-in types. Such a type is often called an *abstract data type*. An abstract type is created by defining a *class*. The class definition acts as a template for the objects. It lists the *data members* (also called *instance variables* or *data fields*) that are part of an object and the *member functions* (also called *operations* or *methods*) which the object can execute. The member functions define the *behavior* of the object and the value of the data members define the *state* of the object.

We illustrate these concepts on an example. Program 2.1.1 shows a class that implements a stack in C++.

The class `Stack` defines four member functions: `push()` to push an integer at the top of the stack, `pop()` that returns and removes the element at the top of the stack, and `size()` that returns the number of elements on the stack. The fourth member function `Stack()` has a special meaning. It is automatically called when an object of the class `Stack` is instantiated. A member function that has the same name as the class is called

Program 2.1.1 Class Stack (C++)

```
class Stack {
public:
  Stack() {
    top = 0;
    data = new double[max_size];
  }
  void push(double d) {
    if (top < max_size) data[top++] = d;
    else /* error handling */;
  }
  double pop() {
    if (top > 0) return data[--top];
    else /* error handling */;
  }
  int size() { return top; }
private:
  int top;
  double* data;

  static int max_size;
};

int Stack::max_size = 4096;
```

a *constructor*. The class defines also two data members: the integer `top` that holds the index of the next free entry in the stack and the pointer `data` that is used to hold the address of an array.

The `Stack` class can be used as follows:

```
Stack s;
s.push(1.0);
s.push(M_E);                // Euler's constant e = 2.1828182
s.push(M_PI);               // Pi = 3.1415926
cout << s.pop() << endl;    // prints 3.1415926
```

The declaration `Stack s` creates a new object of type `Stack` that can be accessed via the variable `s`. Each object gets it own copy of the member data. The next three lines push the three numbers $1, e$ and π on the stack. The last line pops the number π from the stack and sends the number to the standard output that is represented by the object `cout`. The keyword `endl` starts a new line and flushes the output.

It is also possible to share a data item between all objects of a class. A variable that is part of the class, yet is not a part of an object of that class, is called a *static member* (also a *class variable*). The data member `max_size` is a static member of the class `Stack`.

Classes not only define the structure and behavior of their objects, but also their interface (the set of operations provided for clients). In this sense, classes constitute abstract data types; they specify what can be done with objects and how these operations are invoked. Objects communicate with each other by means of *messages*. A message is

an abstract operation. The object is commanded to perform a certain task. In general, clients do not (and should not) know how the operation is performed. For example, an application might tell the `Stack` object `s` to push a number on the stack. The complete set of messages accepted by a (class of) objects is called the *protocol* of the class. The protocol of a class describes its interface; it not only defines the names of the messages but also the types of their arguments and the types of their results. In most object-oriented languages, messages are treated in a similar way to procedure calls. The main difference between a message and a procedure call are that messages always have a receiver that is responsible for execution of the operation and that the effect of a message is determined at run time by means of dynamic binding.

The keywords `public` and `private` in the class definition of `Stack` are *access modifiers*. They control the access to the data members and member functions. Public data members or member functions can be accessed by all functions, whereas private members can only be accessed by the class own's members. The data members of the class `Stack` are *encapsulated*. They can only be accessed through the public member functions which form the *interface* for the client objects (i.e. the objects that want to use this class). The purpose of *encapsulation* is *information hiding*. Clients should know only the interface, but not the implementation of the class. This allows changes to the implementation without invalidating programs that make use of this class. If later on the implementation of `Stack` is, for example, changed from an array to a list existing code does not break as long as the changed `Stack` class provides the same interface.

Another central object-oriented concept is *inheritance*. Classes can be constructed by deriving a new class on the basis of an already existing class. New properties can be added to the derived class, and certain existing properties can be changed in order to adopt the new class to specific needs. Inheritance simplifies software reuse. Existing functionality found in already existing classes can easily be extended or modified.

Inheritance is closely related to *polymorphism*, the ability of a variable to refer to objects of different classes. This feature is responsible for much of the flexibility in object-oriented programming. Consider a class `Shape` to be used in a graphical editor. The class itself is too general as that it would allow to provide meaningful implementations. Object-oriented languages allow the definition of abstract classes, that describe only an interface, but not their implementation.

```
class Shape {
public:
  void move(Point to) = 0;
  void rotate(int angle) = 0;
  // ...
};
```

The usage of the initializer `=0` makes the member function *pure virtual*. A class with a pure virtual function is *abstract*, i.e. it cannot be instantiated. An abstract class can only serve as a base class for a derived class.

```
class Rectangle : public Shape {
public:
  void move(Point to) { upperRight = to; }
  void rotate(int angle) { /* ... */ }
  // ...
private:
```

```
  Point upperRight;
  int height, width;
};

class Circle : public Shape {
  void move(Point to { center = to }
  void rotate(int angle)  { }
  // ...
private:
  Point center;
  int radius;
};
```

The class `Rectangle` and the class `Circle` inherit from the class `Shape`. Since the class `Shape` is an abstract class, they inherit only the interface. A variable of type `Shape` can point to an object of class `Rectangle` or `Circle` since both objects offer the methods declared in the class `Shape`. Assume that the variable `shape` points to an array of all the shapes (circles, rectangles, etc.) that currently exist in the graphical editor. Let the variable `nShapes` denote the number of shapes. To rotate all shapes by a given angle one could use a code snippet like this:

```
Shape** shape;
int nShapes;
int angle;
// ...
for (int i = 0; i < nShapes; ++i) {
  shape[i]->rotate(angle);
}
```

The message `rotate(angle)` is send to every object in the array. Since the object can be a `Circle` or `Rectangle` the method that is executed has to be different. The proper method cannot be determined at compile time (*static binding*), but has to be found at run time (*dynamic binding*). The method that is executed depends not on the variable (`shape[i]`) but on the object that is pointed to by `shape[i]`.

We summarize the description of an object. An object has

- *state*. The value of the data members compromises the object's state.
- *behavior*. The behavior of an object is defined by its member functions.
- *identity*. Two objects of the same class are different, even if they have the same values for their data members.

2.2 Introduction to Object-Oriented Databases

Textbooks that may serve as an introduction to ODBMS are e.g. [9], [11], [52] and [57]. C++ Object Databases are specially treated in [50]. General database systems are, for example, treated in [22], [84], [85].

Table 2.2.1 Stonebraker's application matrix

	Simple Data	Complex Data
No Query	File System	ODBMS
Query	RDBMS	ORDBMS

2.2.1 Background

In the early 80's, it became clear that relational database systems were not robust enough for non-administrative data-intensive applications of the day, e.g. CAD (Computer-Aided Design, CAM (Computer-Aided Manufacturing), CASE (Computer-Aided Software-Engineering), VLSI. It turned out that relational databases do not adequately support the complex data models found in these so-called *non-standard* applications. The development took two routes:

- Object-Relational Database Management Systems (ORDBMSs). This is the *evolutionary approach*. The relational model is extended to incorporate the object model.

- Object-Oriented Database Management Systems (ODBMSs). This is the *revolutionary* approach. Here, the object model is the fundamental data model of the database system.

Stonebraker gives a rough classification of storage systems depending on the application area. His application matrix [79] is drawn in Table 2.2.1. Many ODBMSs concentrate on managing complex data but have somewhat limited query capabilities. ORDBMSs do not integrate the object model nearly as well as ODBMSs do. Due to their different architecture they have also a performace degradation for the navigational access of large complex data.

A basis for definition of the object-oriented approach can be found in the Object Oriented Database System Manifesto [4], which describes an agreement between several leading researchers in the field on what constitutes an object-oriented database system. It presents a number of mandatory, optional and open features, supporting the use of an object oriented data model, as opposed to extending the relational model. Mandatory object oriented features included support for object identity, complex objects and single inheritance, among others.

The Third Generation Database System Manifesto of Stonebraker et. al. [80] also supports the introduction of a number of object oriented features, such as inheritance, support for complex objects and an extensible type system. However, Stonebraker, et. al. argue that these features should be incorporated on top of the existing relational model.

Object databases in many ways represent the marriage of object programming and database management technologies. Depending on which of these technologies is more dominant in a developer's or vendor's perspective, different aspects of object databases become more important than others.

2.2.2 The ODMG Standard

The success of complex technologies depends strongly on their standardization. Standardization helps to improve the portability and interoperability of applications. The major standardization organ in the field of ODBMSs is the Object Database Management Group (ODMG).[2] The ODMG defines an Object-Oriented Database Management System (ODBMS) to be a Database Management System (DBMS) that integrates database capabilities with object-oriented programming language capabilities [10]. An ODBMS makes database objects appear as programming language objects, in one or more existing programming languages. The ODBMS extends the language with transparently persistent data, concurrency control, data recovery, associative queries, and other database capabilities.

The standard defined by by ODMG concerns the data schema, programming language binding, data manipulation and query language. The goal of the ODMG is to put forward a set of standards allowing an ODBMS customer to write portable applications, i.e. applications that can run on more than one ODBMS product.

The major components of ODMG 2.0 [10] are:

- Object Model. The Object Management Group[3] (OMG) Object Model served as the basis for the ODMG Object Model.
- Object Specification Language. One of the specification languages for ODBMSs is the Object Definition Language (ODL). The ODL is used to define the types that can me made persistent.
- Object Query Language (OQL). A declarative language for querying and updating database objects.
- C++ Language Binding.
- Java Language Binding.
- SmallTalk Language Binding.

The language bindings are collectively called Object Manipulation Languages or OML. Some of the areas not specified in the standard are e.g. version and configuration management or the physical organization of the ODBMS.

It is possible to read and write the same database from C++, Smalltalk, and Java, as long as one stays within the common subset of supported data types.

Figure 2.2.1 provides a comparison of DBMS architectures. Rather than providing only a high-level language such as SQL for data manipulation, an ODBMS transparently integrates database capability with the application programming language. This transparency makes it unnecessary to learn a separate Data Manipulation Language (DML), obviates the need to explicitly copy and translate data between database and programming language representations, and supports substantial performance advantages through data caching in applications.

[2] http://www.odmg.org/
[3] http://www.omg.org/

Figure 2.2.1 Comparison of DBMS Architectures

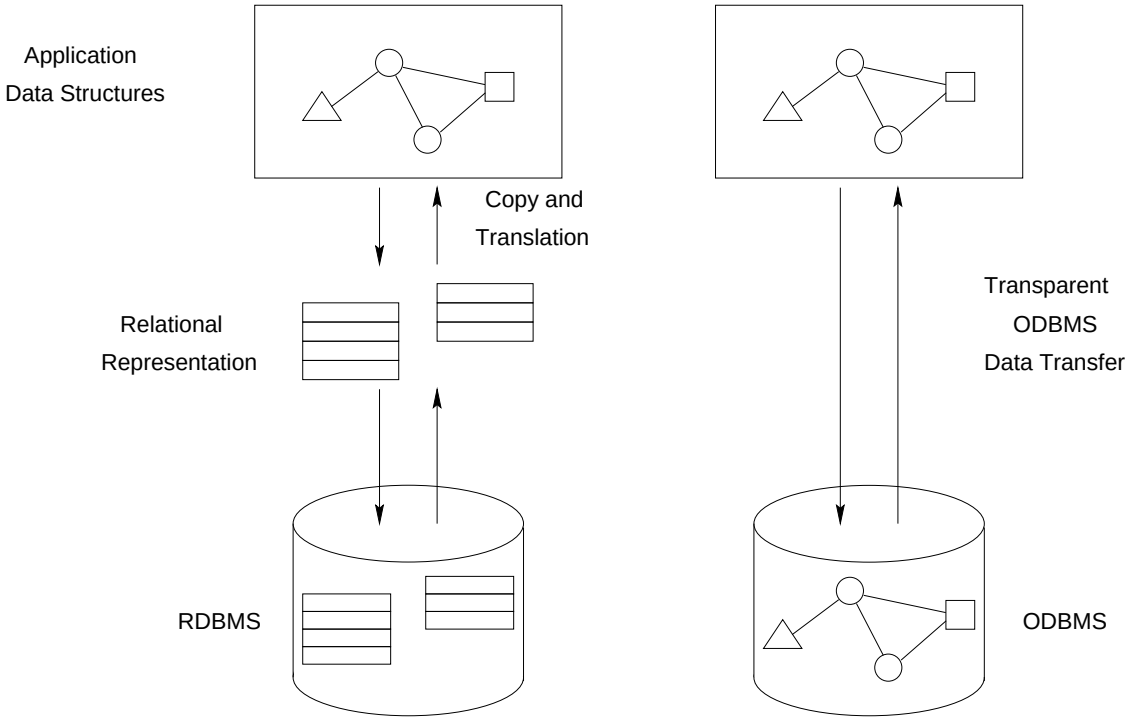
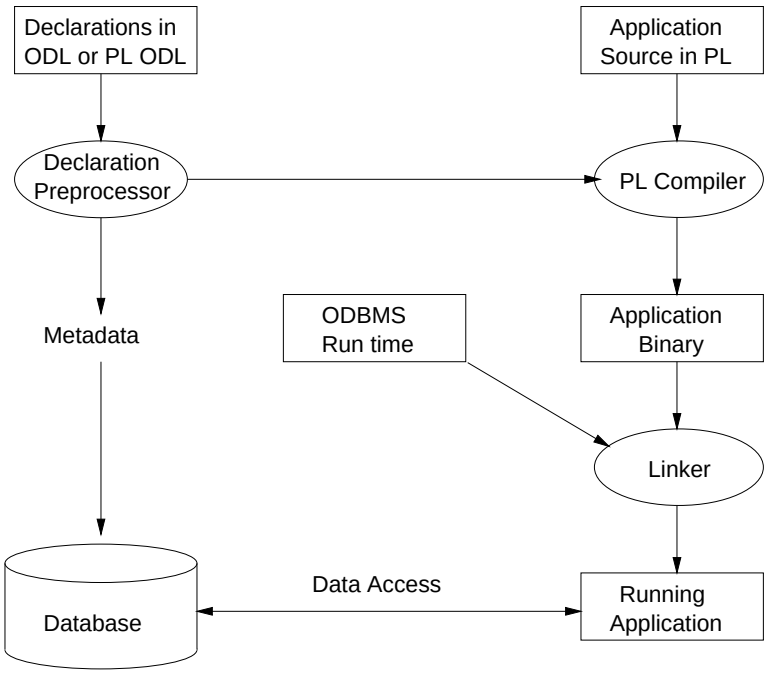


Figure 2.2.2 illustrates the use of a typical ODBMS application. The programmer writes declarations for the application schema (both data and interfaces), plus a source program for the application interface. The source program is written in a programming language (PL) such as C++, using a class library that provides full database OML including transactions and object query. The schema declaration may be written in an extension of the programming language syntax, labeled PL ODL in the Figure, or in a programming-language-independent ODL. The latter could be used as a higher level design language, or to allow schema definition independent of programming language.

The declarations and source program are then compiled and linked with the ODBMS to produce the running application. The application accesses a new or existing database, with types that must conform to the declarations. Databases may be shared with other applications on a network; the ODBMS provides a shared service for transaction and lock management, allowing data to be cached in the application.

An ODBMS distinguishes between persistent and transient objects. An object whose lifetime is *transient* is allocated memory that is managed by the programming language run-time system. The lifetime of an transient objects ends with the program termination. An object whose lifetime is *persistent* is allocated memory and storage managed by the ODBMS run-time system. These objects continue to exist after the procedure or process that creates them terminates. Persistent objects are sometimes referred to as *database objects*.

Figure 2.2.2 Using an ODBMS



2.3 C++ Binding

The C++ language binding specifies how ODL and OML constructs are mapped to C++ constructs. This is done via a C++ class library that provides classes and operations that implement the ODL and OML constructs. In addition to the ODL/OML bindings, a set of constructs called *physical pragmas* are defined to allow the programmer some control over physical storage issues, such as clustering of objects, utilizing indices, and memory management. The class library added to C++ for the ODMG standard uses the prefix `d_` for class declarations that deal with database concepts.[4] We illustrate the fact that the ODBMS access is transparently embedded into the programming languages for two areas: object creation/deletion and relationships between objects.

2.3.1 Object Creation and Deletion

The C++ to ODBMS language binding described by the ODMG standard is based on a smart pointer or "Ref-based" approach. Smart pointers are objects that have the same interface as native C++-pointers. The intercept indirection and dereferencing, perform various actions, and then continue with the indirection or dereferencing [6]. For each persistence-capable class `T`, an ancillary class `d_Ref<T>` is defined. Instances of persistence-capable classes are then referenced using parameterized references, e.g.,

```
d_Ref<Student> stud;
d_Ref<University> uni;
```

[4] Presumably, d_ stands for *database* classes.

```
uni = stud->university();
```

Objects are created in C++ OML using the `new` operator, which is overloaded to accept additional arguments specifying the lifetime of the object. If no additional arguments are passed to the new operator a transient object is created. To create a persistent object either the reference of another persistent object can be passed to serve as a clustering hint, or a reference to a database in which the object shall be stored.

```
void* operator new(size_t size);
void* operator new(size_t size, const d_Ref_Any& clustering,
                   const char* typename);
void* operator new(size_t size, d_Database *database, const char* typename);
```

Persistent objects can be deleted using the `d_Ref::delete_object` member function.

2.3.2 Relationships

Almost no object lives in isolation. Relationships allow to be built persistent navigatable object graphs that capture part of the information stored in the database. Relationships in transient objects are modeled using native C++ pointers or references. For persistent objects other approaches have to be chosen. The ODMG standard defines only the interface and the semantics of relationships between persistent objects, the implementation is up to the ODBMS implementer. Binary relationships are relationships between two classes of objects. Since relationships between more than two classes can be modeled using binary relationships, ODBMSs normally provide only binary relationships. The *multiplicity* of a relationship denotes with how many objects an object can be associated. Assume a relationship between the classes `A` and `B`. The relationship is

- 1-1, if every object of class `A` is associated with at most one object of class `B` and vice versa.
- 1-N, if every object of class `A` is associated with an arbitrary number of objects of class `B`. Each object of class `B` is associated with at most one object of class `A`. N-1 relationships are declared analogous.
- N-M, if every object of class `A` is associated with an arbitrary number of objects of class `B` and vice versa.

Relationships can be further classified into *one-directional* and *bi-directional* ones. A relationship between the classes `A` and `B` is one-directional if navigation is possible in only one direction. It is bidirectional, if navigation is possible in both directions.

The following template class allows one to specify a to-one relationship to a class `T`.

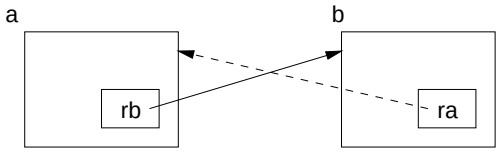
```
template <class T, const char* Member>
class d_Rel_Ref : public d_Ref<T> {};
```

The integrity of relationships is maintained by the ODBMS. Both to-one and to-many traversal paths are supported by the OML. A 1-1 relationship is defined as follows.

```
extern const char _ra[],_rb[];
class A {
   d_Rel_Ref<B,_ra> rb;
```

Figure 2.3.1 No relationship

Figure 2.3.2 1-1 relationship




```
};
class B {
   d_Rel_Ref<A,_rb> ra;
};
const char _ra[] = "ra";
const char _rb[] = "rb";
```

Assume `a` is a reference to an object of type `A` and `b` is a reference to an object of type `B`. The initial situation with no relationship between `a` and `b` is depicted in Figure 2.3.1. Then, adding a relationship between `a` and `b` via

```
a.rb = &b;
```

results in the situation depicted in Figure 2.3.1. The solid arrow indicates the operation specified by the program, and the dashed line shows what operation gets performed automatically by the ODBMS. 1-N and M-N relationships are specified via the following template:

```
template <class T, const char* M>
class d_Rel_set : public d_Set<d_Ref<T> > {}
```

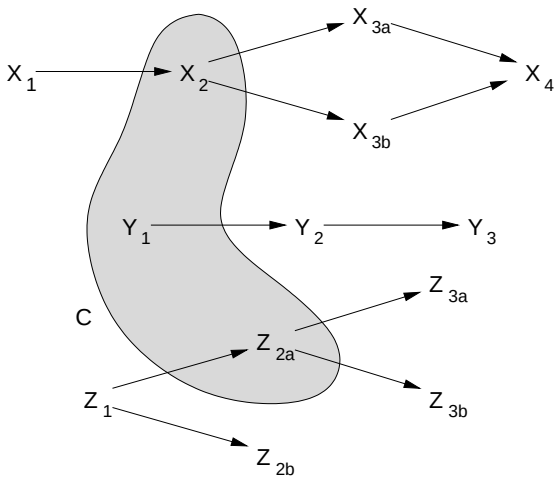
2.4 ODBMS Concepts

We discuss two concepts not found in traditional DBMSs in more detail.

2.4.1 Version Control

ODBMSs have been developed in order to support advanced applications, e.g. engineering applications. Most high-level entities, e.g. detector layout, VLSI design, CAD pictures, which are themselves composed of numerous objects exist in different versions. Version control manages these versions and the dependencies between them.

Figure 2.4.1 Version history of three objects X, Y, and Z in a database [9].



The most basic functionality required for version control is the creation and deletion of versions. Complex *version histories* can result for a particular object. The versioning graph is for three objects illustrated in Figure 2.4.1. Versioning can *branch*, two versions X_{3a} and X_{3b} are created from the version X_2. These versions may later be subsumed by a *merged* version X_4. The version history for an object forms always a directed acyclic graph (DAG). The database has to provide a mechanism to examine the version history. Some databases also offer the possibility to set a *default version*.

Versioning leads to new issues. If an object refers to another object, which version of the referred object should be used? We address this question next. A *configuration* is a collection of versions of objects in a database that are mutually consistent. The configuration C in Figure 2.4.1 represents a group of object versions that are a mutually consistent view of the database.

In HEP version control could almost be used for every data domain:

- Reconstructed event data. Different users will use different reconstruction algorithms that identifies the tracks and the particles out of the raw data. Furthermore, the calibration constants will change over time. Therefore the reconstructed event will exist in several versions.

- Detector description. The detector description and geometry will change slightly over time.

In all cases it is important to allow an easy and transparent navigation to the desired version. For more detailed information see e.g. [20], [51].

2.4.2 Schema Management

The set of type information defined in the ODL (object definition language) has to be persistent. This set is called the *schema*. A schema is a set of type definitions –

including their definitions of the structure and the behavior. A schema is not fixed once written but evolves over time in order to capture the ever-changing requirements. New types are introduced, old types deleted, operations modified, errors eliminated and so on. Changes can include:

- Changes to attributes: e.g. add or drop an attribute
- Changes to methods: e.g. add or drop a method, change the name of a method, change the implementation of a method, inherit a different method definition
- Changes to inheritance graph/tree: e.g. add a new supertype/subtype
- Changes to types themselves: add a new type, drop an exisiting type, change the name of a type

There are four main approaches to maintaining data instances under schema modification [9]:

- *Write-once types.* The simplest approach is to disallow type modification once instances have been created.
- *Immediate update.* In the immediate-update approach, a change to the schema immediately affects existing data of that type. For example, if a new attribute is added to an object type, all existing objects of that type are updated automatically to have null value for that attribute.
- *Lazy update.* The update of an object is deferred until the object is accessed by the application.
- *Schema mapping.* A mapping is maintained between all the versions of the schema. Users may view the same objects through different versions of the schema. Schema mapping is very difficult and is not implemented in any of the commercial systems.

2.5 Implementation Issues

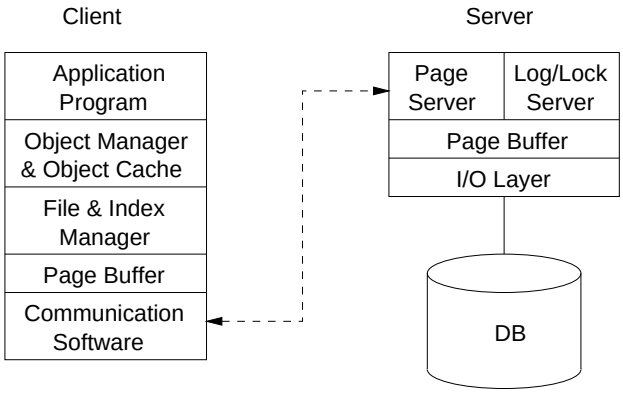
In this section, we focus on implementation issues for ODBMSs. Since object clustering is a central concept of the present work it will be described in the Chapter 5.

2.5.1 Architecture

In an important paper [19] three alternative client-server architectures are discussed. The most outstanding difference is the unit of transfer between client and server.

- *Object-Server Architecture.* In this architecture the server understands the concept of an object and is capable of applying methods to objects. Without going too much into detail, this design has some serious problems. There may be one remote procedure call per object reference. Another major problem is that that this architecture complicates the design of the server.

Figure 2.5.1 Page-Server Architecture [19]



- *Page-Server Architecture* Figure 2.5.1 shows one possible architecture for a page server. The server deals only with pages and does not understand the semantics of objects. In addition to providing the storage and retrieval of database pages, the server also provides concurrency control and recovery services for the database software running on the clients. This architecture is used by most commercial ODBMSs.

- *File-Server Architecture.* This approach represents a further simplification of the page-server architecture in which the client use a remote file service, such as NFS (Network File System), to access database pages directly. As with the page-server design, the server in this architecture provides concurrency control and recovery services.

Despite the fact that there is no clear winner, a page-server approach seems beneficial with good clustering and a large workstation buffer.

2.5.2 Object Identifiers

An ODBMS provides a unique identity to each independent object stored in the database. This unique identity is typically implemented via a unique, system-generated object-identifier (OID). Typically the value of an OID is not visible to the external user, but it is used internally by the system to identify each object uniquely and to create and manage inter-object references.

The full support of object identity is one of the most important features of ODBMSs. A number of approaches have been taken to OIDs [9], [21]. OIDs may be *physical* (that is, they may contain the actual address of the object) or *logical* (that is, they be mapped through an index to obtain a object location). Also OIDs that both contain a physical and logical component are possible. Because of the widespread use of OIDs, the choice for their representation can be a critical factor in the performance of an ODBMS. An object can be directly loaded from disk on the basis of a physical OID. On the negative side, the reorganization and reclustering of the database is difficult because an object

cannot simply be moved to another page or another disk if physical OIDs are used. Although the additional overhead of logical OIDs for object lookup is tolerable [21], most commercial and experimental systems still use physical OIDs.

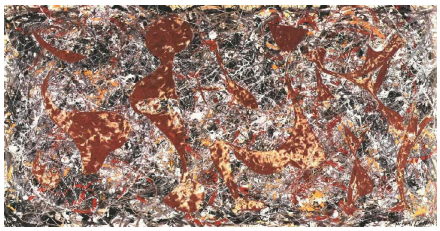
2.5.3 Object Buffering

An important factor in ODBMS performance is the technique for caching and accessing objects in main memory after they have been fetched from disk. There are several points where an ODBMS has performance advantages compared to a traditional DBMS.

- In a traditional DBMS disk pages are copied into a DBMS buffer, not directly into application memory. Individual fields or records are copied into application variables from the DBMS buffer, rather than entire complex objects being copied. In contrast, an ODBMS copies the data pages directly from disk to application memory, making any conversions necessary to the representation in place.
- In contrast to traditional DBMSs, ODBMSs also have a cache on the client site. If the same objects are repeatedly processed this leads to a large performance improvement.

Chapter 3

Objectivity/DB



Jackson Pollock
Out of the Web: Number 7, 1949
Oil and enamel on masonite
Staatsgalerie Stuttgart, Germany

Objectivity/DB is the most promising candidate under the commercial ODBMSs to be used for the LHC experiments. This chapter is organized as follows. Section 3.1 describes the architecture and usage of Objectivity/DB. Section 3.2 presents exhaustive benchmarks of Objectivity/DB. Section 3.3 describes the ATLAS 1 TB milestone: 1 TB of event data was successfully stored into Objectivity/DB coupled to the mass storage system HPSS.

3.1 Description

Objectivity/DB is a distributed ODBMS produced by Objectivity.[1] Objectivity/DB features a C++ and Java language binding.

3.1.1 Usage

Objectivity/DB usage follows the process illustrated in Figure 2.2.2. Persistent-capable class definitions are written in Objectivity/DB's ODL (called DDL for data definition language). The Objectivity/DB preprocessor runs on the DDL file. It creates three new files. For example let us assume the DDL definition of the persistent class `Event` is stored

[1] http://www.objectivity.com

in the file `Event.ddl`. The DDL processor creates the files `Event.h`, `Event_ddl.C`, and `Event_ref.h`. `Event.h` is the header file to be included in the application's source code. `Event_ref.h` declares the necessary OML classes and methods for the `Event` class, e.g. the declaration of `d_Ref<Event>` or the overloading of the new operator. `Event_ddl.C` implements all methods added by the preprocessor.

An object must be an instance of a derived class of the Objectivity/DB class `d_Object` to exist persistently in an Objectivity/DB federated database. A class that derives from `d_Object` is called *persistent-capable*. The creation of the object determines whether the object is persistent or transient. If the object is created with `new(0)` the object is transient, in the other cases it is persistent. The new operator is also used to specify a clustering hint. If an persistent object is passed as an argument to the `new` operator Objectivity/DB tries to store the newly created object near the physical location of the argument. If a container or database is passed as an argument, the new object is stored in the container or database, respectively.

Objectivity/DB does not fulfil all the requirements of the ODMG standard. Some of the differences are:

- Persistent capable objects cannot contain other persistent capable objects.
- The smart pointer types implemented in Objectivity/DB (`ooHandle`, `ooRef`, `d_Ref`) are not capable of referencing true transient objects that have been allocated on the normal program heap.

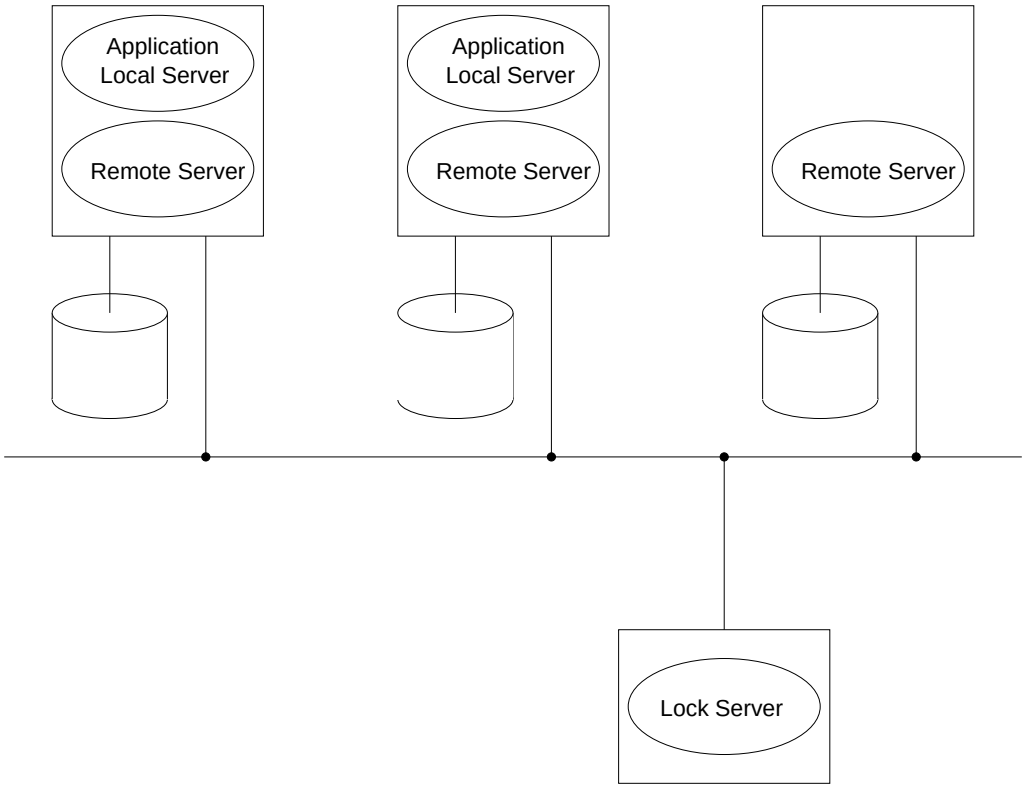
3.1.2 Architecture

The distributed architecture of Objectivity/DB is illustrated in Figure 3.1.1. The databases can be distributed over several machines. Only concurrency control which is done by the Lock Server is centralized. The advantage of a distributed architecture is the higher scalability due to the fact that the workload is distributed over the available servers. Objectivity/DB can employ a specialized data server daemon called Advanced Multithreading Server (AMS) that reads database pages on remote servers and transfers them to the client application. The AMS uses a set of standard POSIX.1 filesystem I/O routines to perform operations on directories (open, read, close) and database files (open, read, write, etc.). Hence the AMS assumes that it has access to a standard POSIX-compliant filesystem.

Objectivity/DB is logically and physically structured as follows.

- *Federated Database.* The federated database logically contains the catalog of databases and the data model (or *schema*) that describes the class definitions. An application cannot have more than one open federated database at the time. The federated database is physically stored in two files. The boot file is a small ASCII file that contain the configuration and setup parameters. For example it contains the page size, the federated database ID, the host and the pathname of the lock server. The schema and the database catalog is stored in the federated database file.
- *Database.* A database is logically a collection of containers. A database and all the container and objects in the database is stored as one file. Therefore the database forms the unit of storage with regard to the file system.

Figure 3.1.1 Distributed Database Architecture of Objectivity/DB



- *Container*. A container is a collection of objects. Locking of individual objects takes place at the container level.
- *Basic objects*. A basic object is the fundamental logical unit of storage.

The logical and physical organization of Objectivity/DB is illustrated in Figure 3.1.2.

Objectivity/DB is built as a page server. The page size is specified at creation time of the federated database and can vary between 512 bytes and 64 KB.

3.1.3 Object Identifiers

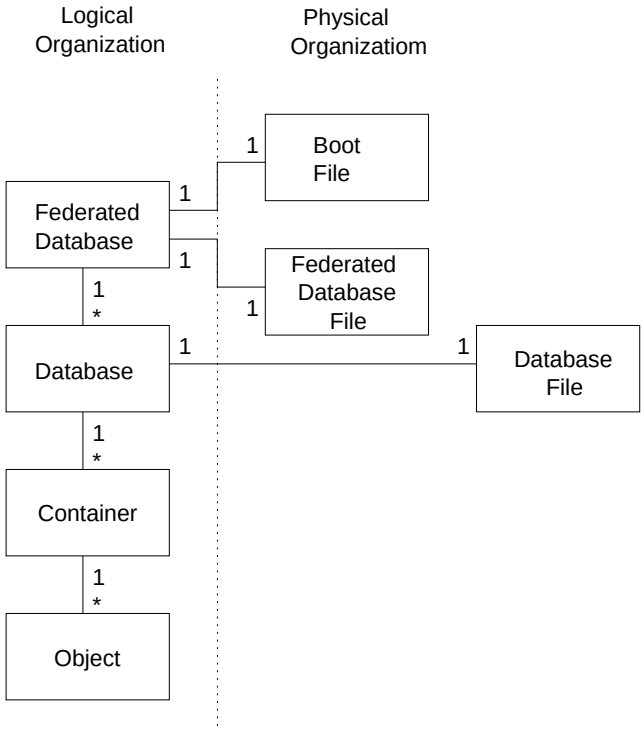
Objectivity/DB uses 64 bit-long physical Object Identifiers (OIDs). The OID is composed of four 16-bit fields in the following format:

`D-C-P-S`

- `D` Database identifier
- `C` Container identifier
- `P` Page number
- `S` Slot number on the page.

The length and the decomposition of the OID is a delicate choice. A long OID lead to

Figure 3.1.2 Logical and Physical Organization of Objectivity/DB



a higher maximum number of objects that can be stored in the database. On the other hand, since object relationships, indexes and container that hold objects by reference have to use the OID an unnecessarily long OID increases the storage overhead. The limits in scalability are for Objectivity/DB

Number of databases per federation	$2^{16}-1$
Number of containers per database	$2^{15}-1$
Number of pages per container	2^{64}

The database size is also limited by the maximum file size of the file system. Theoretically, maximum database size can be 128 TB, if they are stored on a true 64-bit filesystem:

$$2^{16}\frac{\text{byte}}{\text{page}} \times 2^{16}\frac{\text{page}}{\text{container}} \times 2^{15}\frac{\text{container}}{\text{database}} = 128 \text{ TB}.$$

Practical databases about 100 GB are expected to be difficult to handle and should be avoided. Taking this bound in consideration one obtains a maximum size of a federated database of $(2^{16}-1)\times 100$ GB $= 6.25$ PB. There is a clear requirement either to increase the length of the 64-bit OID or to change the mapping between the logical and physical organization.

Objectivity/DB offers additional short OIDs that occupy only 4 byte. Short OIDs identify a basic object within the scope of a single container.

3.1.4 Overview of other Objectivity/DB Features

Objectivity/DB Fault Tolerant Option Using Objectivity/DB Fault Tolerant Option the federated database can be divided into autonomous partitions residing on different servers. The break-down of a server or the interruption of the network makes only the local partition unavailable. The other partitions continue to work autonomously.

Objectivity/DB Data Replication Option This option allows the replication of databases across the network to improve the database performance and availability.

3.2 Benchmarks

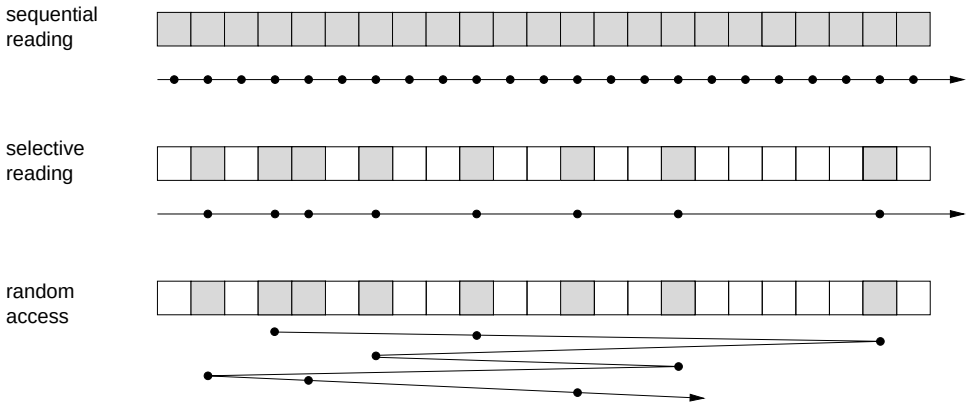
Most of the coming HEP experiments will use object-oriented database management systems (ODBMSs) as the data store. A detailed understanding of the the ODBMS performance is crucial to enable high-performance analysis scenarios. In an ideal world the object model would be independent of the underlying storage technology. In reality the object granularity and the layout of the database pages have a great impact on the database performance. Almost every technology has its sweet spots where it is performing very well and areas where the performance decreases drastically. This section presents performance measurements for different access patterns carried out on Objectivity/DB. Measurements were done for sequential reading, sequential writing and selective reading. Reading times for random access are estimated using some of the values of the hard disk specifications. Furthermore an experimental and analytical analysis of the storage overhead of Objectivity/DB is given.

The ODBMS performance depends on the underlying storage medium. At the moment there exist no alternatives to hard drives for secondary storage. Although the performance/price ratios of both processors and disks are improving, the rate of improvement is greater for processors. Hence, the disk subsystem is emerging as a bottleneck factor in some applications. Recent advances in high bandwidth devices (e.g. RAID, ATM networks) have had a large impact on file system throughput. Unfortunately, access latency still remains a problem due to the physical limitations of storage devices and network transfer latencies.

3.2.1 Related work

A broad range of read/write benchmarks and storage overhead measurements can be found in Ref. [67]. Sequential reading is specially investigated in Ref. [45]. Ref. [28] presents storage overhead measurements and some read/write benchmarks. Ref. [44] describes the CPU requirements necessary to write with a transfer rate of 100 MB/s into Objectivity/DB. These estimations are based on measurements carried out on several platforms. A general valuable reference for carrying out benchmarks is Ref. [48].

Figure 3.2.1 Different access patterns



3.2.2 Read/Write Performance

A coarse classification of read/write access patterns for hard drives is done by the following categories: sequential access, selective access and direct access. Figure 3.2.1 illustrates these access patterns.

Sequential access patterns describe read and write accesses where objects are read or written consecutively from or to the drive. If the application is not CPU-bounded the maximal transfer rate of the hard drive can be achieved.

Selective reading reads not necessarily consecutive objects in ascending physical ordering. The disk head moves in only one direction. If the gaps between the objects are not too big, the next object that is read in is typically on the same track as the object previously read in. Therefore the head need not to be moved in most cases.

Sequential access and selective reading are characterized by the fact, that the disk head moves in only one direction. This is no longer true for *random access* (also called *direct access*). In this case the order in which the objects are read differs from their physical order. The time needed to position the head is typically in the range of milliseconds up to the maximum seek time of the hard disk.

We use the following terminology to describe the benchmarks. The *attribute size* specifies the amount of user information stored in an object. For example, an object containing an array of 100 integers has an attribute size of 400, assuming that an integer occupies 4 byte. The *write_rate_object* gives the write rate on an object basis

$$write\_rate\_object = \frac{attribute\_size \times num\_of\_objects}{write\_time}.$$

The *read_rate_object* is defined in an analogous way. We distinguish the *write_rate_object* from the *write_rate_page*, which gives the transfer rate on a page basis. It measures the writing speed for the storage actually written, independently from the amount of user information stored in the database

$$write\_rate\_page = \frac{database\_size}{write\_time}.$$

Table 3.2.1 Data specifications for the Seagate Elite hard disk.

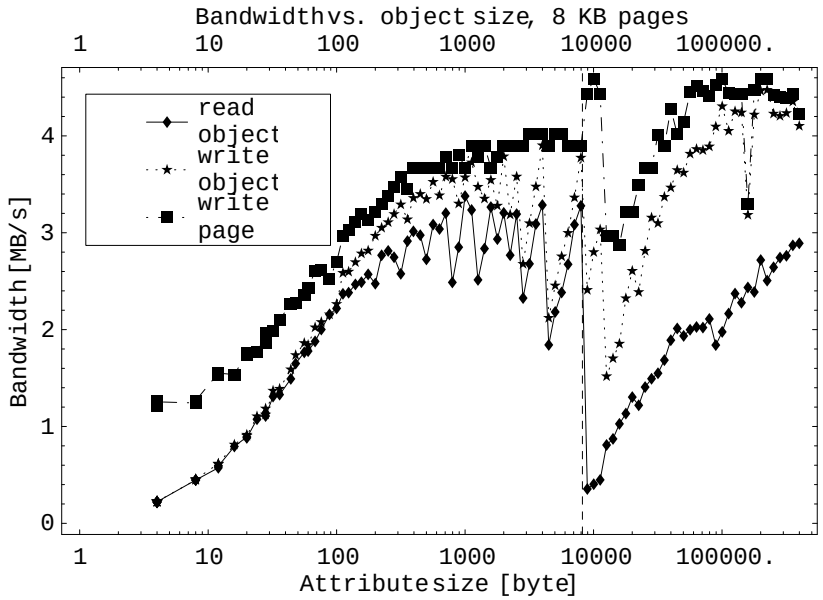
Formatted Gbytes	23.20
Interface	Ultra SCSI
Internal Transfer Rate, ZBR (Mbits/sec)	86 to 124
External Transfer Rate, 8/16 (Mbytes/sec)	20/40
Multisegmented Cache (Kbytes)	2.048
Track-to-Track Seek, Read/Write (msec)	1.1/2.1
Average Seek, Read/Write (msec)	13/14
Maximum Seek, Read/Write (msec)	28
Average Latency (msec)	5.56
Spindle Speed (RPM)	5,400
Cylinders	6,882

One may interpret the *write_rate_page* as the writing rate one would obtain, not taking the storage overhead of Objectivity/DB into account. The general outcome of the benchmarks is depending on the *page size*. The page size represents the unit of data transfer between Objectivity/DB and the secondary storage. It is specified on the creation time of the federated database. We split the range of object sizes into two subranges. We call an object that is small enough that it can be stored on a page a *small object*. On the other hand, we call an object that is greater than a page a *large object*.

For convenience we give the specifications for the Seagate Elite hard drive, that we used for most of the benchmarks, see Table 3.2.1. This hard drive uses Zone Bit Recording (ZBR). ZBR is a technology to increase the capacity of the disk. A drive that uses ZBR stores on the outer tracks more sectors than on the inner tracks. A consequence is that the transfer rate varies depending on the location of the track. By reading an outer track higher transfer rates are achieved than by reading an inner track. For performing benchmarks ZBR is a nuisance. Methods to take the different transfer rates into account are cumbersome and were not done for the benchmarks presented in this paper.

We describe briefly the different delays, involved in reading from or writing to the disk, see e.g. Ref. [78]. To read or write, the head must be positioned at the desired track and at the beginning of the desired sector on that track. If the head is not already at the desired track, it has to be moved. The time it takes to position the head at the track is known as *seek time*. Once, the track is selected, the system waits until, the appropriate sector rotates to line up with the head. The time it takes for the sector to reach the head is known as *rotational latency*. The sum of the seek time, if any, and the rotational latency is the *access time*, the time it takes to get into position to read or write. Once the head is in position, the read or write operation is then performed as the sector moves under the head. Ref. [72] provides a detailed description of disk drives and their modelling.

Figure 3.2.2 Read/Write transfer rate as a function of the attribute size on Windows NT. The plot has a logarithmic x-axis.



3.2.3 Sequential Reading and Writing

We present and analyse the benchmark results for sequential access.

Setup and Realization of the Benchmarks

The benchmarks were performed on a Sun Solaris and on a Windows NT machine. The following table contains the hardware and configuration description.

Windows NT

System :	Windows NT Version 4.0, PII/233 MHz processor
Objectivity/DB Version :	5.0
Compiler options :	/O2
Page size :	8 KB
Database size :	between 128 and 192 MB

Sun Solaris

System :	SunOS 5.5.1, Ultra-5_10, 270 MHz UltraSPARC
Objectivity/DB Version :	5.0
Compiler options :	-O2
Page size :	8 KB
Database size :	between 64 MB and 96 MB bytes

The objects consisted of a 4-byte identifier and a fixed-sized integer array, whose size was varied from run to run. Measuring points were chosen in progressive distances. One

Figure 3.2.3 Read/Write transfer rate as a function of the attribute size on SUN. The plot has a logarithmic x-axis.

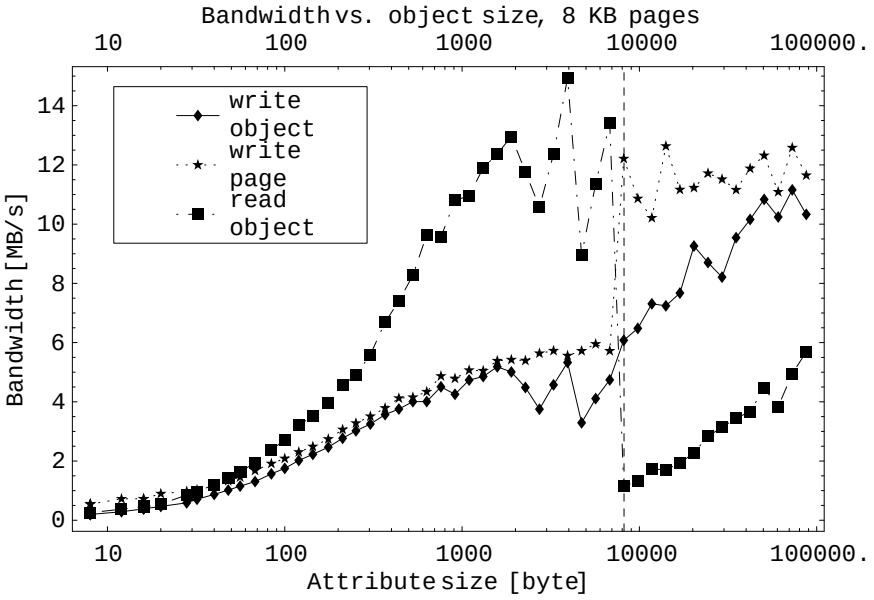


Figure 3.2.4 Write transfer rate as a function of the attribute size on SUN. This diagram extends Figure 3.2.3 by containing more measuring points. The plot has a logarithmic x-axis.

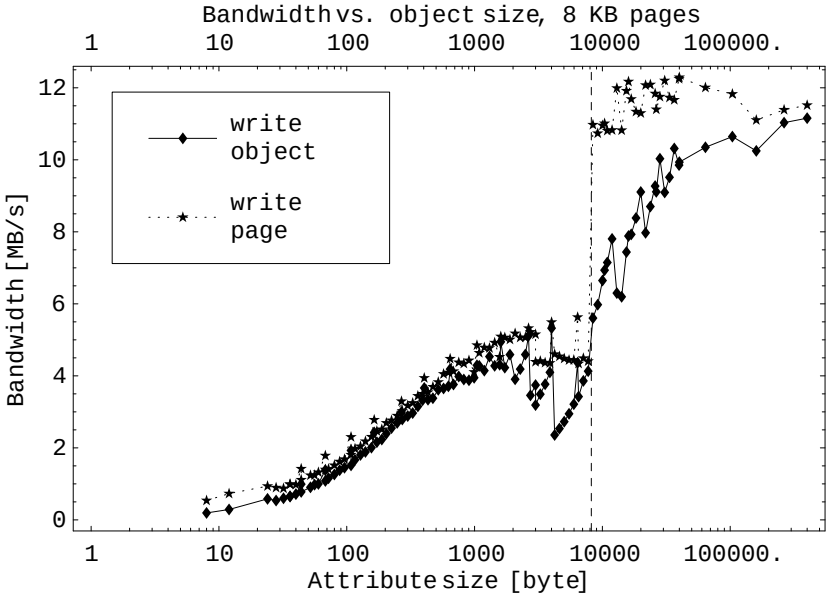
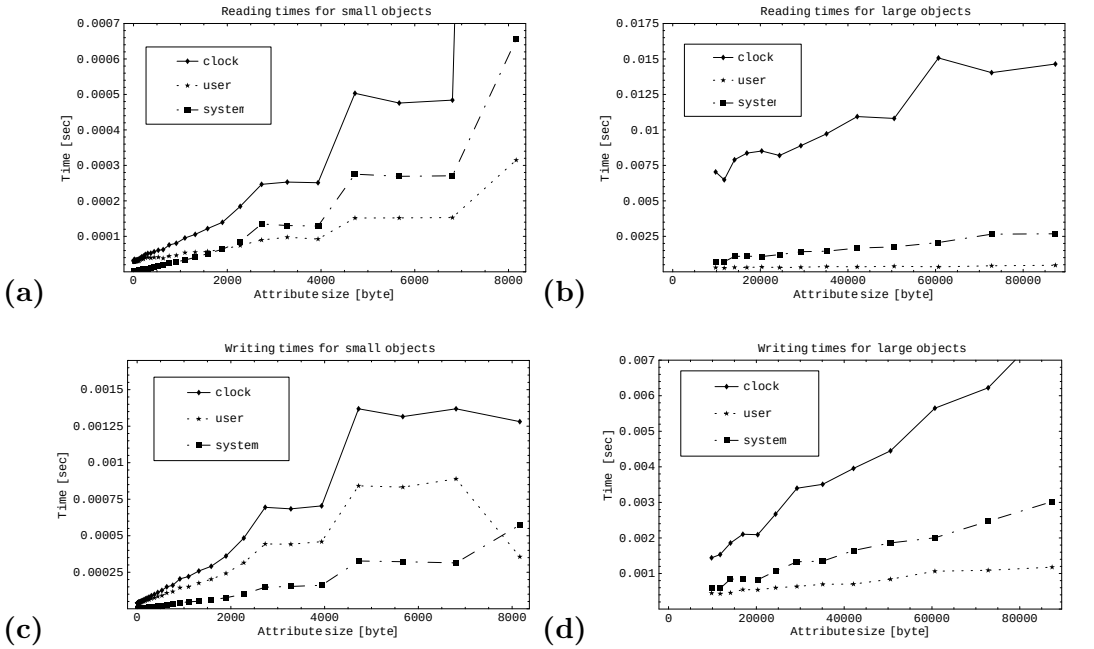


Figure 3.2.5 Read/Write times on SUN
(a) Read Times for small objects
(b) Read Times for large objects
(c) Write Times for small objects
(d) Write Times for large objects



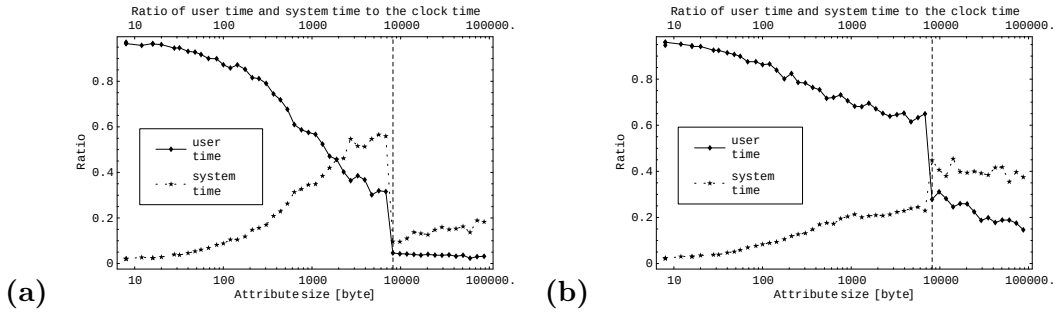
transaction and one container was used. The time which was spent to create the database and the container is also part of the total run time measured for the data writing. Figure 3.2.2 – 3.2.4 show the experimental results. All these figures use a logarithmic x-scale. Figure 3.2.5 distinguishes the measurements by object sizes. The plots for large and small objects are depicted separately. These plots use a linear x-scale. In Figure 3.2.6 the ratio of the CPU time to the total time is given. The CPU time is split up in user time (the time spent outside the kernel) and system time (the time spent in the kernel). Unfortunately it seems that Objectivity/DB is quite CPU-intensive. Reading and writing of small objects takes almost the whole CPU time.

Analysis of the experimental data

First we discuss properties common to both platforms. All plots exhibit a different behavior for small and large objects. Really astonishing is the bad read performance for large objects. We will come back to this peculiarity later on. The read and write curves for the objects show a more or less significant saw-tooth-behavior. This can be explained as being due to the storage overhead, see Section 3.2.6.

Let us now discuss differences in the plots between the two platforms. For Windows NT the write transfer rate is always higher than the read transfer rate. Solaris shows a different behavior. For small objects the read rate is higher than the write rate, for

Figure 3.2.6 Ratio User Time / Wall Clock Time and System Time / Wall Clock Time on SUN. The plots have a logarithmic x-axis.
(a) Reading
(b) Writing



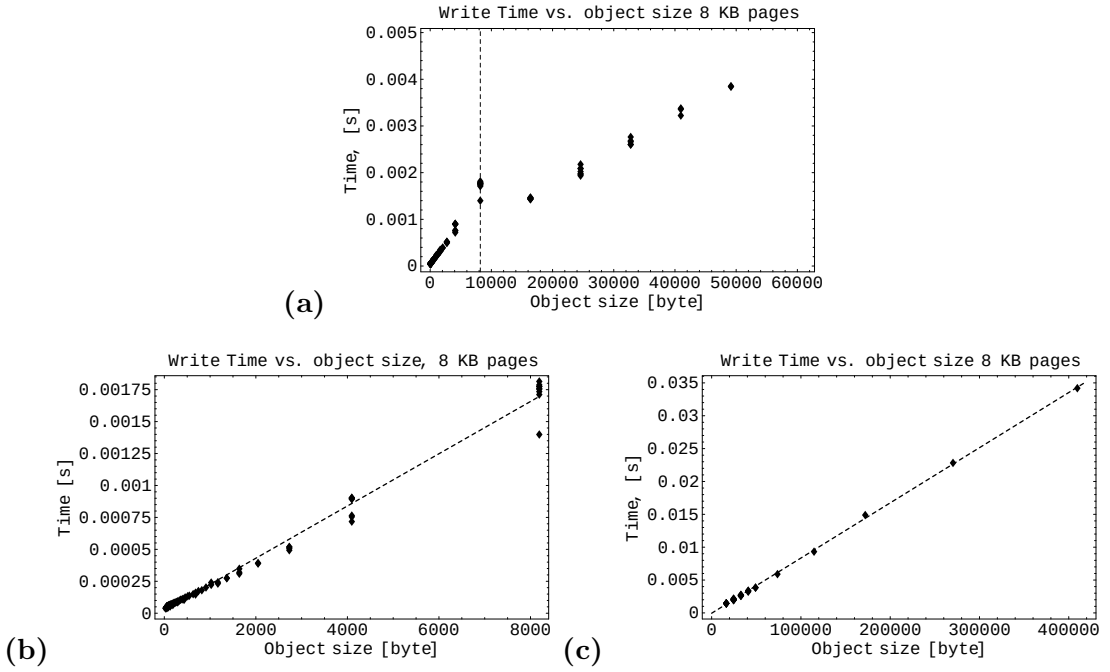
large objects the opposite is true.

The reason for the bad performance whilst reading large objects may explained by the following fact. The storage layout for large objects is such, that first the object data is stored and then the object descriptor.[2] If a large object is accessed the object descriptor has to be read first. If the object data is not in the drive's cache, almost a whole disk rotation is necessary till the disk head is positioned above the data and the data can be read. Unfortunately, the Seagate Elite cache is not large enough to hold the data of a complete track. The following values are from Table 3.2.1: $track\_size = capacity/cylinders = 3$ KB > 2 KB $= cache\_size$.

Next we will study the linearity of the write and read time for a single object. For this purpose, we depict the read/write time not as a function of the attribute size, but as a function of the object size thereby taking the space overhead into account. Due to the properties of the page layout and the storage overhead the object size can assume only certain values, see Section 3.2.6. For the benchmarks on SUN this is done for writing in Figure 3.2.7-(a). The same graph is also depicted for small and large objects separately, see Figure 3.2.7-(b) for small and Figure 3.2.7-(c) for large objects. These figures also show a linear least-square fit made for these data samples. These two fits were made independently for the small and the large objects. Figure 3.2.9-(a) – Figure 3.2.9-(c) show the read/write times for the benchmarks performed on Windows NT. The following function was fit into the data: $t = a + bs$, where t is the time, s is the object size and a and b are the coefficients of the polynomal. Table 3.2.2 shows the coefficients. The coefficients should be interpreted with caution. For small objects, the coefficients may express the split-up of a constant time part due the CPU overhead and due to the size-dependent part in a proper way. For large objects, an interpretation of the coefficients for reading is complicated due to the storage layout of the objects and the need to do a full disk spin to complete the reading of the object. The coefficients for the writing of large objects show that in principle the whole bandwidth of the disk can be reached.

[2] K. Holtman, private communication

Figure 3.2.7 Write Times as function of the object size on SUN
(a) small and large objects
(b) small objects
(c) large objects



The total time to read/write an object is the sum of a constant overhead and a linearly depending part on the object size. The fraction of the constant part in the total time is depicted in Figure 3.2.10 for Windows NT. One sees how the large constant part for reading is responsible for the bad performance reading large objects.

3.2.4 Selective Reading

Selective reading reads objects in their physical order but the objects are not necessarily contiguously stored. The percentage of objects read is given by the selectivity s where s is a real number form 0 to 1. The selectivity is a parameter of the benchmark and represents the probability that an object is read. The probability is for each object the same and independent whether other objects are read or not. Of course if the selectivity is 1 selective reading is equal to sequential reading.

Technically the benchmarks were carried out as follows. A vector was used that contained the object identifiers (OIDs) of all objects in physical ascending order. Before each benchmark a second vector was created, containing a subset of the first vector. The probability that an OID was contained in the second vector was equal to the selectivity s. After that the objects referred to by the second vector were read.

If N is the number of objects in the database, the number of objects actually read for a given selectivity s is about $s \times N$. Let T_s denote the time it takes to read the

Figure 3.2.9 Read/Write Times as function of the object size on WNT
(a) small and large objects
(b) small objects
(c) large objects

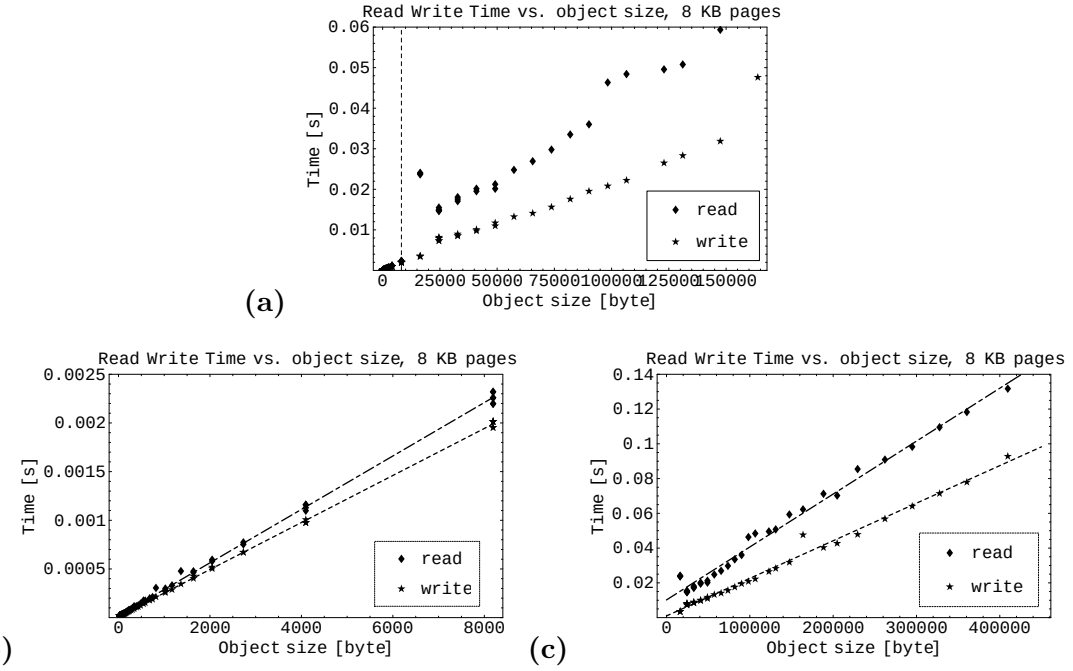


Table 3.2.2 time (y) to read an object as a function of the object size (x).

	write/	*small/*	$t = a + bs$		
platform	*read*	*large*	a [sec]	b [sec/Byte]	b^{-1} [Byte/sec]
SUN	write	small	2.16×10^{-5}	2.044×10^{-7}	4.89×10^{6}
SUN	write	large	-4.48×10^{-5}	8.40×10^{-8}	1.19×10^{7}
NT	write	small	1.11×10^{-5}	2.42×10^{-7}	4.14×10^{6}
NT	write	large	9.83×10^{-4}	2.16×10^{-7}	4.63×10^{6}
NT	read	small	1.44×10^{-5}	2.74×10^{-7}	3.64×10^{6}
NT	read	large	0.0102	3.05×10^{-7}	3.28×10^{6}

selected objects from the database. Let t_s denote the average time it takes to read a single object. The relation $T_s = s \times N \times t_s$ holds. The bandwidth B_s is the amount of data read per time unit. It is defined by

$$B_s = \frac{s \times N \times attribute\_size}{T_s} = \frac{attribute\_size}{t_s}. \tag{3.1}$$

Figure 3.2.12 shows the results of the benchmarks. For object sizes 400 byte and 800 byte a collection with 10^6 objects was used, for object size 8 KB a collection with

Figure 3.2.10 The fraction of the constant time part for Windows NT. The plot has a logarithmic x-axis.

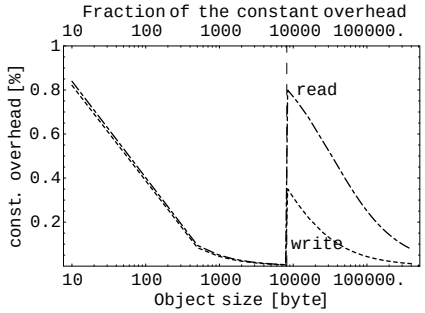
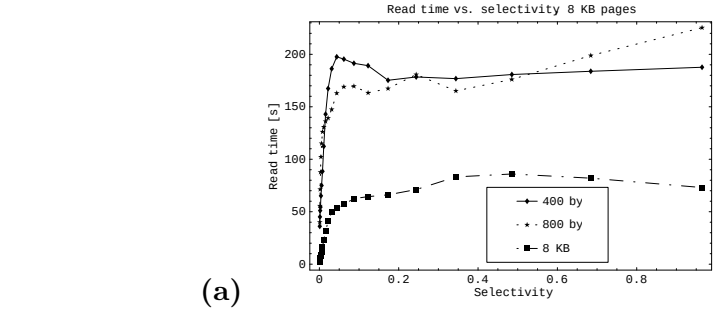
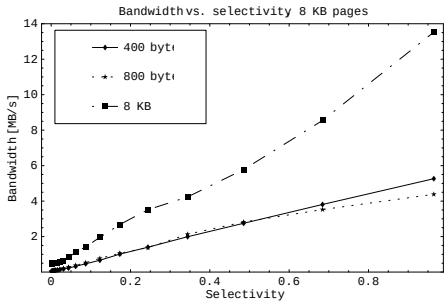


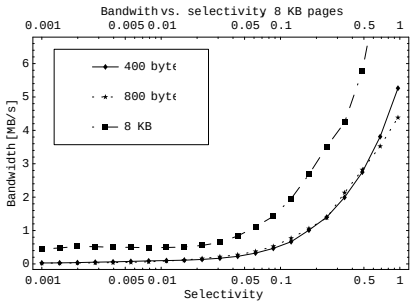
Figure 3.2.12 Selective Reading on SUN
(a) Reading Time vs. Selectivity
(b) Transfer Rate vs. Selectivity
(c) Transfer Rate vs. Selectivity, Logarithmic Scale



(a)



(b)



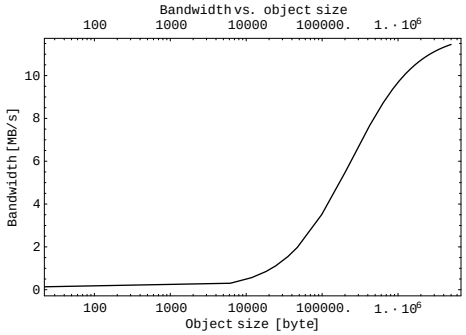
(c)

200'000 objects were used. Figure 3.2.12 depicts the time it takes to read $s \times N$ objects vs. the selectivity s. The two other pictures show the bandwidth vs. the selectivity. While Figure 3.2.12-(b) uses a linear x-axis Figure 3.2.12-(c) uses a logarithmic one.

The plots in Figure 3.2.12 feature three selectivity ranges with different properties.

1. $0\% \leq s < 2\%$. The bandwidth B_s is nearly constant, see Figure 3.2.12-(c). From equation 3.1 follows that t_s is also nearly constant and that T_s is a linear function

Figure 3.2.13 Upper bound for Random Access Reading Rate. The plot has a logarithmic x-axis.



of s. The disk seeks in this area are typically inter-track seeks.

2. $2\% \leq s < 5\%$. This is a transition area between the first and the third one.

3. $5\% \leq s \leq 100\%$. The reading time T_s is almost constant. There is no speedup in comparison to sequential reading that reads all objects in the database. From equation 3.1 it follows that B_s and t_s are linear increasing functions of s. The disk seeks in this area are mainly intra-track seeks.

3.2.5 Random Access

Due to the long seeks between read accesses random access should be avoided as far as possible. Instead of experimental measurements we treat the direct access qualitatively and derive a simple formula for the upper bound of the expected transfer rate. The reading time for an object determined by the hard drive is given by

$$t_{read} = t_{seek} + t_{latency} + \frac{object\_size}{transfer\_rate}. \tag{3.2}$$

The bandwidth for random access is given by

$$bandwidth = \frac{object\_size}{t_{read}}. \tag{3.3}$$

We assume that the data of interest fills the whole disk. Then we can use the values in Table 3.2.1 : $t_{seek} = 13.5$ msec, $t_{latency} = 5.5$ msec and $transfer\_rate = 12$ MB/sec. Figure 3.2.13 depicts the bandwidth vs. the object size. As mentioned above the graph is just an upper bound. Using an ODBMS like Objectivity/DB one has also to take the CPU and storage overhead into account. The bandwidth varies like $\frac{x}{1+x}$. Essentially it is a linear slowly growing function for small object sizes and constant for very large object sizes (> 10 MB). Even if the object size is as large as 8 KB the transfer rate is only 400 KB/s.

3.2.6 Storage Overhead

We investigate the storage overhead of Objectivity/DB. Additional measurements of the storage overhead can be found in Ref. [28] and Ref. [67]. The database size will always be larger than the amount of data stored in the database. The following factors contribute to the storage overhead:

- *Object header.* Objectivity/DB uses 14 byte per object for its object organization. This information includes, for example, the type information of the persistent object.
- *Compiler alignment of data members.* A certain compiler can enforce a particular alignment of the data members leaving some bytes between the storage layout of the data members unused.
- *Object alignment in slots.* Objectivity/DB allocates storage in slots to 8 bytes. The size of the object is therefore rounded up to a multiple of 8 bytes.
- *Half-page storage overhead for large objects.* Objects that are larger than the page-size have an additional overhead of a half-page size.
- *Remaining free space on page.* The storage layout of a small object cannot cross page boundaries. The remaining free space on a page is lost.

See also [67], Table 11-2 for a detailed explanation of factors that contribute to the storage overhead. We use the following definition of the storage overhead:

$$storage\_overhead = \frac{size\_database\_file}{size\_of\_input\_data} - 1.$$

In [67] the following definition of storage efficiency is used:

$$storage\_efficiency = \frac{size\_of\_input\_data}{size\_database\_file}.$$

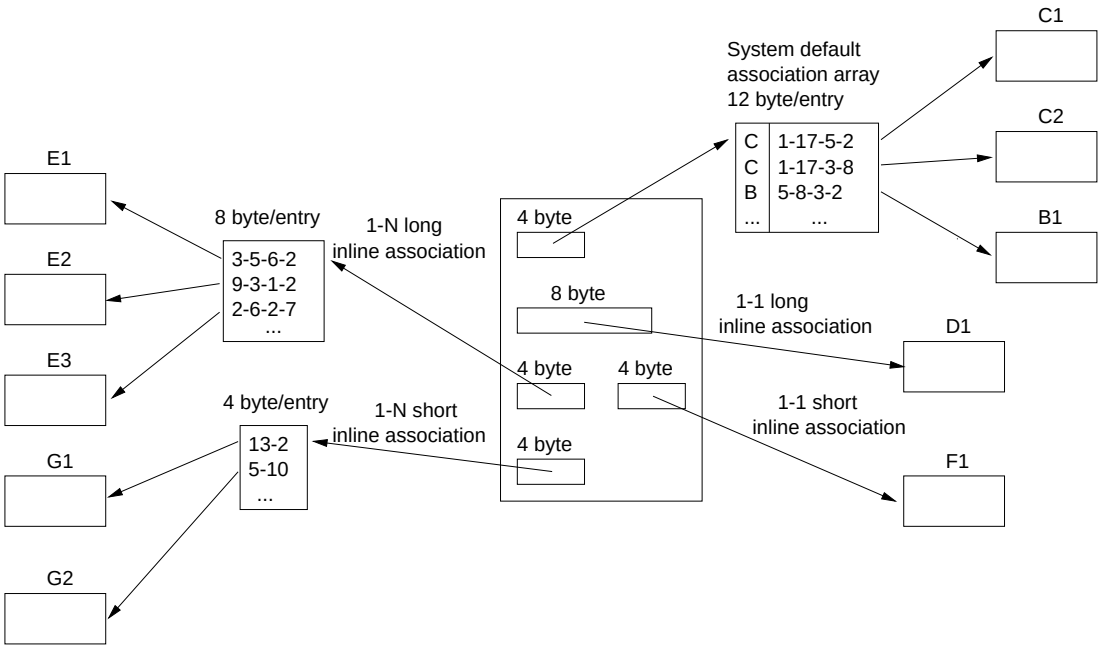
These two quantities are simply connected by

$$storage\_overhead = (1/storage\_efficiency) - 1.$$

Since object relationships capture parts of the information in the database, their storage requirements cannot simply be viewed as overhead. Objectivity/DB lets the user choose between three different implementations of associations.

- *System default association array.* By default associations are stored in a VArray. An entry in this array consists of two fields and consumes 12 byte. The first field specifies the association link ID and the second field the OID.
- *Long inline association.* An association can also be defined inline. If the association is 1-1 a 8 byte data member is added to the object that holds the OID of the associated object. For an 1-N association an own VArray is created where each entry consumes 8 bytes.

Figure 3.2.14 Different implementations of Objectivity/DB associations



- *Short inline association.* This implementation is identical to the long inline association beside the fact that only short OIDs are stored. Therefore it can only be used for associations where the associated elements are in the same container.

Figure 3.2.14 illustrates all three different possibilities. An embedded Objectivity/DB VArray is implemented as a separate object and has therefore a constant storage overhead of 14 byte. This is also true for the VArrays that are used to implement the associations.

We analyse the storage overhead for simple objects, i.e. objects that neither have VArrays nor associations to other objects. The *attribute size* is the aggregate size of the data members in the object. We assume for the sake of simplicity that the compiler alignment does not introduce an additional storage overhead. Therefore only the object header and the alignment into 8-byte slots contribute to the effective *object size*. We express these relationships now mathematically. We are using the floor (greatest integer) and ceiling (least integer) functions, which are defined for all real x as follows:

$\lfloor x \rfloor$ = the greatest integer less than or equal to x;

$\lceil x \rceil$ = the least integer greater than or equal to x.

The object size is then given by

$$object\_size = \lceil attribute\_size/8 \rceil * 8 + 14. \tag{3.4}$$

We now analyse the storage overhead for objects of the same size. If the object size is smaller than the page size, a page is filled up with objects, till the remaining space is

smaller than the object size. The next object to be stored will start on the next page, e.g. objects less than a page size do not cross page boundaries.

On the other hand, if the object size is greater than the page size, a half page is added to the object overhead. Then the least number of pages to hold the object is allocated to this object. The remaining space on the last page is lost.

- $object\_size \leq page\_size$

$$objects\_per\_page = \lfloor \frac{page\_size}{object\_size} \rfloor \tag{3.5}$$

$$storage\_overhead = \frac{page\_size}{objects\_per\_page \times attribute\_size} - 1 \tag{3.6}$$

- $object\_size > page\_size$

$$pages\_per\_object = \lceil \frac{object\_size}{page\_size} + 0.5 \rceil \tag{3.7}$$

$$storage\_overhead = \frac{pages\_per\_object \times page\_size}{attribute\_size} - 1 \tag{3.8}$$

The formulas above make use of the equation for the object size (3.4).

3.2.7 Measurements

The measurements were done on the Solaris system described above. The DDL file consisted mainly of an fixed array.

```
class FixedSizeArray : d_Object {
public:
  int32 _oid;
  int32 _array[ARRAY_SIZE];
};
```

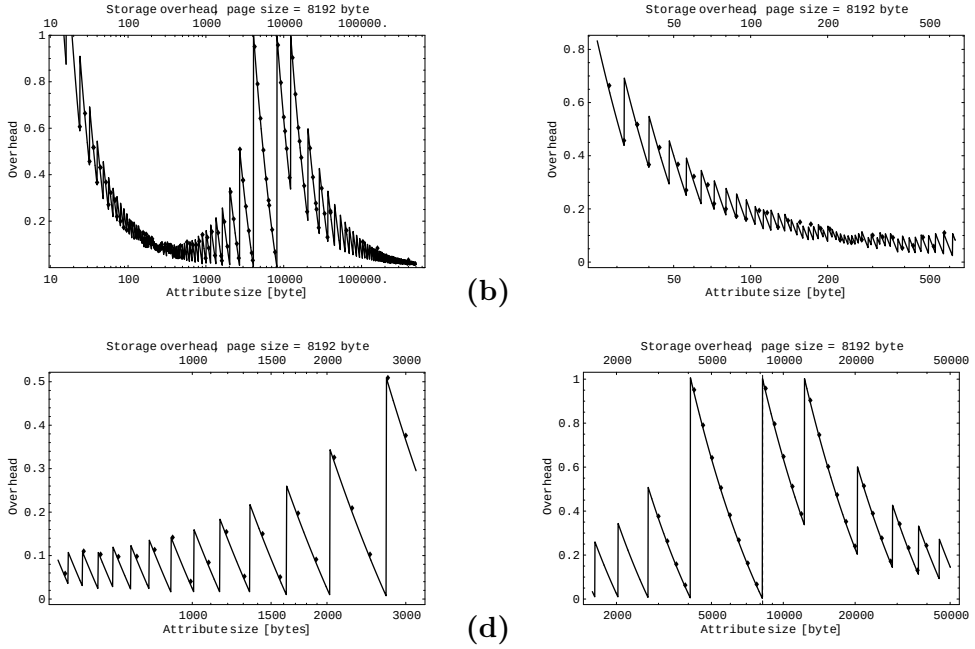
The usage of fixed-size arrays made it necessary to recompile the application for each measuring point. The benchmark program itself created a database and a container and populated the database with objects. The number of objects written in this container was chosen in such a way, that the DB size was in the range 60 – 90 MByte. .

Figure 3.2.15-(a) covers the whole range of measured array sizes. Figure 3.2.15-(b) – 3.2.15-(d) show certain subranges. Notice that the scales for the array sizes are all logarithmic. The sawtooth function was calculated using (3.6) and (3.8).

For very small objects (< 300 bytes) the constant overhead of 14 bytes dominates the storage overhead. This component varies like $1/attribute\_size$. The sawtooth behavior in this range is due to the ceiling function in equation (3.4). For larger objects (> 300 bytes) the overhead due to the alignment of the objects onto the pages is dominating. The sawtooth behavior in this range is now due to the ceiling function in equation (3.5). Crossing the page size border introduces a new overhead of a half page. This overhead varies also like $1/attribute\_size$.

For any given page size p there are three points with a storage overhead of 100%: $p/2, p$ and $3/2p$ (4 KB, 8 KB and 12 KB for $p = 8$ KB). We give the ranges where the

Figure 3.2.15 Storage Overhead. The plots have a logarithmic x-axis.
(a) Attribute Size: 10 byte - 500 KB
(b) Attribute Size: 20 byte - 600 byte
(c) Attribute Size: 500 byte - 3 KB
(d) Attribute Size: 2 KB - 50 KB



storage overhead is less than 10%, and 20%:

$storage\_overhead < 20\%$:	$attribute\_size$ in the range 80 byte - 2 KB or $attribute\_size > 30$ KB,
$storage\_overhead < 10\%$:	$attribute\_size$ in the range 200 byte - 800 byte or $attribute\_size > 80$ KB.

Due to the sawtooth behavior these values are only approximate. The list above is incomplete. There are also a couple of relatively small ranges that have a small overhead, e.g. 3584–4080 byte and 7168–8176 byte.

Most applications will have a distribution of objects spread over different sizes. Objectivity/DB tries to use the free space on pages by storing objects there that fit on the remainder of the page. Therefore, one may expect that the storage overhead peaks are smoothed out by a broader attribute size distribution.

3.2.8 Future Trends

One has to be aware that predictions in the computer industry are more uncertain than in other areas. Trying to predict price, performance, or price/performance more than five years into the future is one of the pitfalls described in Ref. [42]. Therefore the following numbers should be read with caution.

Ref. [38] gives a detailed analysis of future trends in hard disk drives. There the

following predictions are made. The internal data rate of hard disks is doubling every two years ([38], Fig. 5). This corresponds to an improvement of 40% per year. The access time is decreasing at a much lower rate. In 9 years the access time (seek time plus rotational latency) reduces by a factor 2 ([38], Fig. 6). This is an improvement of 8% per year. It is also interesting to note that hard drive prices fall faster than DRAM prices ([38], Fig. 4). It is interesting to compare the numbers above with the increase in CPU performance. Moore's law states that chip performance doubles every 18 months, e.g. [78]. This corresponds to an annual increase of 60%. This trends are also expected to continue in future.

Ref. [13] forms a second information source. It is illustrative to compare the predictions with the first reference. According to this reference, the capacity improves at 27% per year, the transfer rate improves at 22% per year and the seek time improves at 8% per year.

The improvement in the transfer rate is in both references significantly larger than the improvement in the seek time. Therefore the hard drive seek time will become a serious bottleneck in the future. This means that the speed gap between sequential reading and random reading will continue to grow.

There are several possibilities to deal with this development. On one hand good object clustering can reduce the total seek time for a transaction by a large factor. On the other hand, parallelization of disk reads can also help to improve the performance.

3.3 The 1 TB Milestone

The ATLAS 1 TB milestone is defined in the ATLAS Computing Technical Proposal ([5], Table 3-5):

End 1998 *Provide ∼ 1 Tbyte working prototype of database*

The main purpose of this milestone was to demonstrate the feasibility of the different elements of the ATLAS software using a first approximation of the raw data model of an ATLAS event. A secondary goal was to understand the performance of the various components of the system, although actually obtaining the maximum throughput was not the main focus of the test.

The ATLAS 1 TB milestone was successfully absolved by storing 1 TB of simulated event data in an Objectivity/DB federated database that was coupled to the mass storage system HPSS[3] (High Performance Storage System). Effective data flow started around 11. Dec. 98. The 1 TB milestone was passed on 1. Jan. 99. Half of the operation was unattended (Christmas period).

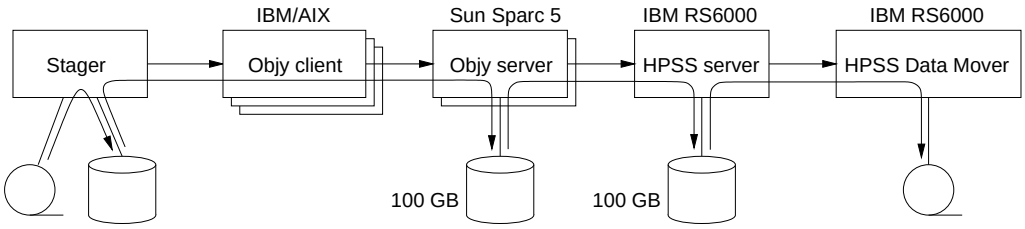
3.3.1 Configuration

Figure 3.3.1 illustrates the configuration of the hardware and the data flow. The configuration consists of:

- *Stager*. The stager reads ZEBRA files stored on tape and stores them in a disk pool.

[3] http://www.sdsc.edu/hpss/hpss1.html

Figure 3.3.1 Configuration of the 1 TB milestone



- *Objectivity/DB Clients.* The Objectivity/DB clients convert the ZEBRA files into objects and store them onto the Objectivity/DB servers. The machines are IBM/AIX machines. Typically 5 machines out of a cluster of 15 were used. The load sharing is done by Load Sharing Facility[4] (LSF). LSF is a general purpose distributed computing system from Platform Computing Corporation.

- *Objectivity/DB Servers.* The servers host the federated database. Two machines are used: a Sun Ultra 5, 300 MHz monoprocessor and a Sun E450, 300 MHz dual processor.

- *HPSS Server.* The HPSS server migrates the database files first from the server diskpool to its own diskpool and afterwards to tape or vice versa. The server runs on a IBM RS6000.

- *HPSS Data Mover.* The mover is responsible for transferring data from a source device to a sink device. A device can be a standard I/O device with geometry (e.g. a tape or disk), or a device without geometry (e.g. network, memory). The data mover runs on a IBM RS6000.

The application (running on the Objectivity/DB clients) reads ZEBRA events from a stage pool, converts the events into persistent objects and stores them in databases that are on one of the two Objectivity/DB server machines. The data that is to be stored in Objectivity/DB is simulated event data residing in ZEBRA files on tape. ZEBRA[5] is a data structure management package developed at CERN to overcome the lack of dynamic data structuring facilities in the Fortran 77 language. The stager brings in ZEBRA files from tape and stores them in a disk pool. This data is accessed by several Objectivity/DB applications that read the data, convert the data into objects and store these objects into an Objectivity/DB database. This database is stored in a diskpool. A migration daemon periodically scans the disk pools and copies new or modified files into HPSS. HPSS manages the data on the tertiary storage. It allows a client-transparent migration/staging of databases to/from tertiary storage. When the amount of free disk space falls below a predefined threshold, the migrated files are deleted from the pool. Ref. [41] describes the coupling between Objectivity/DB and HPSS in more detail.

[4] http://wwwinfo.cern.ch/pdp/pc/Doc/LSF/guides/html/01-intro.html
[5] http://wwwinfo.cern.ch/asdoc/zebra_html3/zebramain.html

Table 3.3.1 Size and Out-degree of the Composing Event Objects

detector	EventObjVector	PEventObj		PDigit	
	fan-out	size	fan-out	size	average VArray size (byte)
Si	1500	23	7	12	85
TRT	600	23	13	8	100
Calo	40	23	200	8	1600

3.3.2 The Event Model

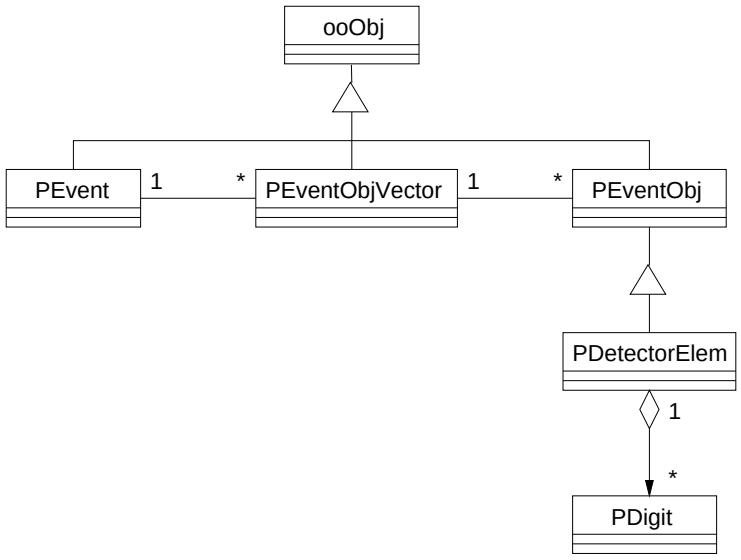
The data that was used for the 1 TB milestone were simulated events from the Jet Production in 1997[6]. As mentioned above this data resides on tapes in the ZEBRA format. Figure 3.3.2 illustrates the object-oriented event model that was used to store the data into Objectivity/DB. The event is a composite object. The constituting objects are hierarchical structured similar to a tree. Most data resides in the `PDigit` objects. The other objects serve merely to organize the data and allow easily navigation inside the event. `PEvent` forms the root of the tree. There is one `PEventObjVector` for each detector system. The events that are converted from the ZEBRA format to objects do not contain all the information. The converted data is limited to the Silizium, TRT and the Calorimeter detector subsystem. Table 3.3.1 gives numbers for the out-degrees of the internal nodes in an event object.

3.3.3 Results

The typical aggregate write speed was $\sim$ 1.5 MB/s with HPSS staging out and $\sim$ 3 MB/s without HPSS staging out. It took 19 days to write 1 TB. There were about 10 stoppages in the data flow due to several reasons: running out of the AFS[7] token, database locks held by no longer existing batch jobs, a lack of tapes in the IBM robot, one crash of the AFS server that hosted the lock recovery tools, and two crashes of the HPSS data server.

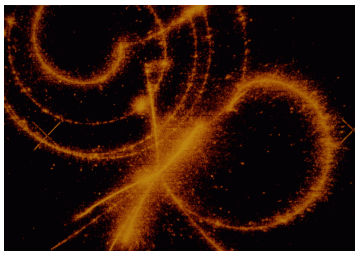
[6] http://atlasinfo.cern.ch/Atlas/GROUPS/SOFTWARE/HELP/jet_production.html
[7] A distributed filesystem

Figure 3.3.2 Event Model of the 1 TB milestone



Chapter 4

Persistent Data Structures in ATLAS



Streamer chamber
Decay of positive pions
CERN photo

The major storage needs of the ATLAS experiment are [73]:

- event data;
- calibration/alignment;
- detector description.

The data volume of the ATLAS experiment is given by the following numbers [5]:

- 100 Hz event rate out of Level-3 trigger, i.e. 10^9 events per year;
- 1 MB event size;
- $\sim$ 1 MB/h of calibration and alignment data.

The data volume is $\sim 1\mathrm{P}B$ (10^{15} bytes) of raw data per year.

The key software elements which directly concern the computing model are the management and the storage of the data [5]. The two central components for the data storage are:

- Object Database Management System (ODBMS)

- Mass Storage System (MSS)

RD45 [67] is investigating the usage of commercial ODBMSs and MSSs in HEP. It is expected that there will be about 500 physicists performing some analysis task with about 150 users simultaneously accessing the data.

4.1 Event data

The ATL-SW-Atlas-Requirements-V1[1] document gives the following definition for an event.

> An event is a container for all data which is considered to constitute a physics event throughout the phases of its processing: i.e. raw data collection or simulation, reconstruction, and analysis. This includes the raw data, the reconstructed data, the analysis objects, and the event tag information.

The various objects of an event are grouped together to define different event object groups used in the offline analysis: raw data, ESD, AOD and event tag. A given object might be part of one or more event object group(s). These definitions and an estimation of their size are given in Table 4.1.1. A preliminary raw data event model is depicted in Fig. 4.1.1, see also Ref. [73]. The raw data, i.e. the *digits*, are organised by the containing detector element. Each detector element provides an *Identifier*, to allow for both identification and data selection. The digit objects are accessed via an iterator. The general event model is still under development. The Computing Review document[2] distinguishes between two general models.

- Persistent Model. In a persistent model, objects are accessed typically by dereferencing smart pointers whose purpose is to fetch relevant data from the database when already present in the local cache. This leads to a situation where the software is embedded in the database environment (typically by deriving classes from Objectivity/DB base classes), and to a slightly modified C++ syntax. On the other hand, this approach allows one to fully exploit the object-oriented features of a product such as Objectivity/DB. BaBar has investigated this kind of solution at early stage of their Event Store development, but has failed to provide a workable prototype or a satisfactory design.

- Transient Model. In a transient model, the database is completely hidden from the client application (also called the transient world). There is a small layer (Converters) that copies back and forth transient objects to persistent objects into the database. This approach has the advantage of a fully independent development of the reconstruction software, but does not allow one to exploit the full possibilities of an object database. This is the solution chosen by BaBar for their Event Store.

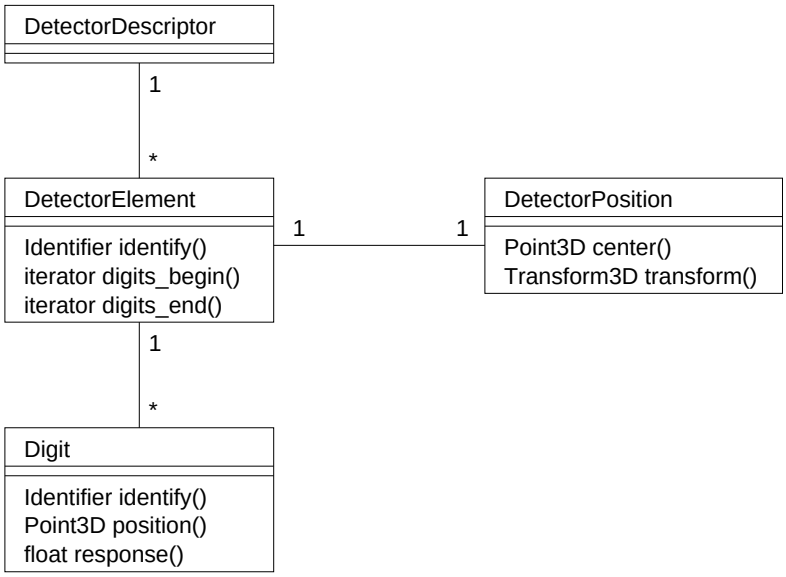
[1] http://www.cern.ch/Atlas/GROUPS/SOFTWARE/OO/deliverables/ATL-SW-Atlas-Requirements-V1/

[2] http://edmsoraweb.cern.ch:8001/cedar/doc.download?document_id=101560&version=1&filename=ReviewFinal.pdf

Table 4.1.1 The groups of objects for each physics event used in the off-line environment ([5], p. 8)

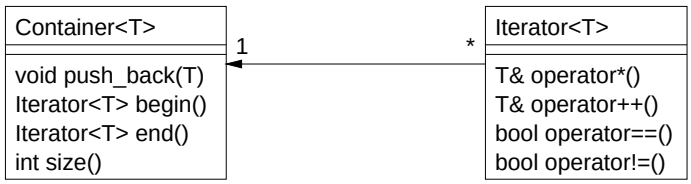
Event object group	*Size*	*Description*
Raw data	1 Mbyte	information coming out of Level-3
Event summary data (ESD)	100 kbyte	reconstruction information in enough detail to do event display, generate analysis objects, and redo most of the reconstruction.
Analysis object data (AOD)	$<$ 10 kbyte	physics objects, e.g. electrons, muons, etc., used for analysis
Event tag	$<$ 100 bytes	brief information allowing a rapid first-pass selection to find events of interest

Figure 4.1.1 Class diagram of the raw data structure, [73]



The distinction between a transient and persistent event model is closely related to the issue of a database access layer. Encapsulating database specific code in a separate layer has become good practice when using relational databases. However ODBMSs are much more integrated into the programming language. Ref. [15] describes the development of an ODBMS access layer. Other open issues are related to the navigation inside an event.

Figure 4.2.1 Iterator Pattern



4.2 Event Collections

A physicist doing analysis is investigating a preselected set of events (or components of the events). Occasionally new sets of events are created, new events are inserted into existing collections, or collections that are no longer needed are deleted. Clearly, persistent data structures are needed to store these sets of events. Event collection is the general term that denotes a container that store events (or components of them) or references to events (or components of them).

At a dedicated workshop held by RD45 the following list of requirements were made for event collections:[3]

- single class for the user interface;
- support for a description of the collection;
- STL-like interface, including a forward iterator;
- support for collections up to $10^9 - 10^{11}$ events;
- parallel processing;
- set-style operations based on a unique event identifier;
- synchronised iteration.

4.2.1 Containers and Iterators

A *container* is a object that holds other objects. Examples are lists, vectors, and associative arrays. In general, you can add objects to a container and remove objects from it.

An aggregate object such as a container should give you a way to access its elements without exposing its internal structure. Moreover, you might want to traverse the container in different ways, depending on what you want to accomplish.

The separation of object containment and object access fulfills these requirements. The key idea is to take the responsibility for access and traversal out of the container object and put it into an *iterator* object.

Fig. 4.2.1 shows a container and an iterator with a minimalistic interface. The container provides member functions to add and remove elements. The method

[3] http://home.cern.ch/dirkd/EventCollection-Workshop/EventCollections/ppframe.htm

`push_back()` inserts an element of type `T` into the container. The container interface also provides two methods that return iterators: `begin()` and `end()`. The iterator interface provides methods to access elements (`operator*()`) and to step ahead in the traversal (`operator++()`). The comparison operators `operator==()` and `operator!=()` can be used to test for the container end. The fundamental iteration loop looks like:

```
Container<Event> c;
Iterator<Event> i;
Histogramm h;
// ... set c;
for (i = c.begin(); i != c.end(); ++i) {
   h.insert(*i);
}
```

Event collections can become very large. It is often desirable to investigate only a small random sample and not the whole event collection. This would be an example for a different container traversal. The responsiblitiy of the random sample generation and the interaction over this sample can be encapsulated in a further iterator. Since the interface of this iterator is the same, the analysis code does not have to be changed.

The interplay between the container class and the iterator class is an example for a *design pattern*. A design pattern names, abstracts and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The term design pattern for object collaborations was introduced by Gamma et al. [30]. The authors also present a catalog of 23 different design pattern.

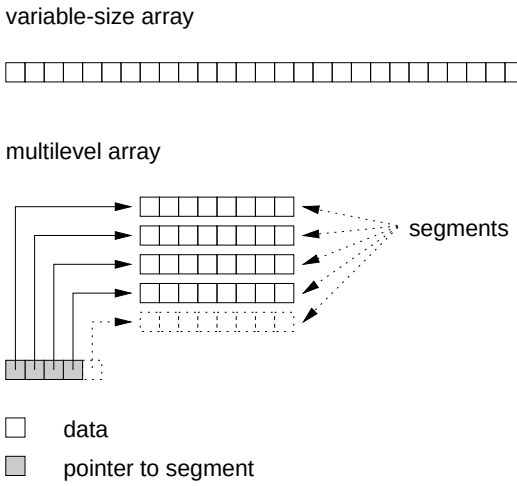
4.2.2 Multilevel Array

Objectivity/DB provides several persistent array data structures:

- *Fixed-size array.* The size of this array is specified in the DDL file and cannot be changed at run-time.
- *Variable-size array (VArray).* The size of the VArray can be changed at runtime. The contents of a variable-size array is guaranteed to be allocated contiguously in persistent storage.
- `vector<T>`. Support for a persistent version of the C++ Standard Library containers was added recently.

None of them is adequate to be used as an event container. An event container must be able to handle a number of events from some hundreds up to 10^9. Fixed-size arrays clearly disqualify. VArrays have some constraints that make their usage as an event container problematic. Contiguous allocation of storage offers faster element access than noncontiguous allocation but it might lead to expensive resize operations. If the size of the array is extended another larger contiguous block of storage has to be allocated and the contents of the original array has to be copied to the new place. The indexing type of the VArray is `uint32`, which means that a VArray can cold up to 2^{32} entries. This number is large enough to satisfy known HEP requirements. However, the effective limit in the size of a VArray comes from the fact that the entire VArray must be read into memory before an operation can be performed on it. Although this is useful if all

Figure 4.2.2 Variable-size array vs. multilevel array.



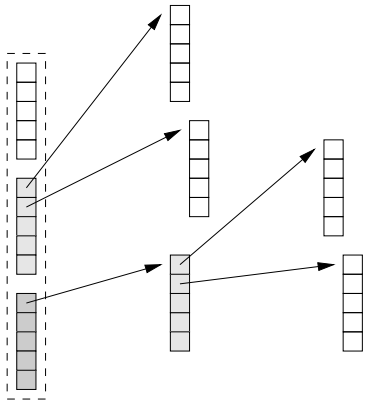
or most elements will be accessed, it is not always the optimal solution. This additional constraint represents a severe performace drawback if only one or a couple of array elements are accessed by the application. The last array class is a persistent version of `vector<T>`. This class still has severe performance problems.

We describe an array structure, the multi-level array, that overcomes the previous mentioned problems. We also implemented an event collection class that is based on the multilevel array. The multi-level array data structure splits a large array up into smaller fixed-size arrays called segments. A variable-size array refers to the segments, see Fig. 4.2.2. If an entry of the multilevel array is accessed only the variable-size array referring to the segments and the segment containing the entry have to be brought into memory. The size of the segments is a critical choice. Since the page size is the unit of data transfer between main memory and secondary storage there is no advantage to make the segments smaller than the page size. On the other hand, if the segment size is greater than the page size an additional space overhead in the size of a half page occurs, see section 3.2.6. This additional overhead is only acceptable if the segment size is large compared to the size of a half page, say if the segment is larger than five pages. In summary a reasonable choice of the segment size is either limited to the page size or the size should be significantly larger than the page size (say more than five pages).

A further optimization is possible by combining several levels of indirection into one data structure. The collection class contains several fixed-sized arrays, each having a different level of indirection to the data. The first array contains directly the data (called a data array). The second one contains pointers to segments of data (called first-indirection array). If this is still not enough a third array would contain pointers to first-indirection arrays. Of course more levels of indirection could be introduced. This structure is similiar to an i-node used in the UNIX file system, see e.g. [82].

We used the multi-level array to implement an event collection and provided the corresponding iterator. A subset of their interfaces are shown in Fig. 4.2.1. Objectivity/DB has also developed an unsupported multi-array `raArray`.

Figure 4.2.3 An array with further levels of indirection



4.2.3 Association- and Container-based Event Collections

We describe the implementation of two other event collections. The first one is based on a 1-N relationship (Objectivity/DB calls it an association), the other one is based on the Objectivity/DB container class.

The association-based event collection defines an 1-N association to the events. Objectivity/DB provides an iterator object `ooItr<Event>` that can be used to iterate over all the events in the association.

The iterator `ooItr<Event>` has some constraints that make the task of providing an STL-conforming interface troublesome.

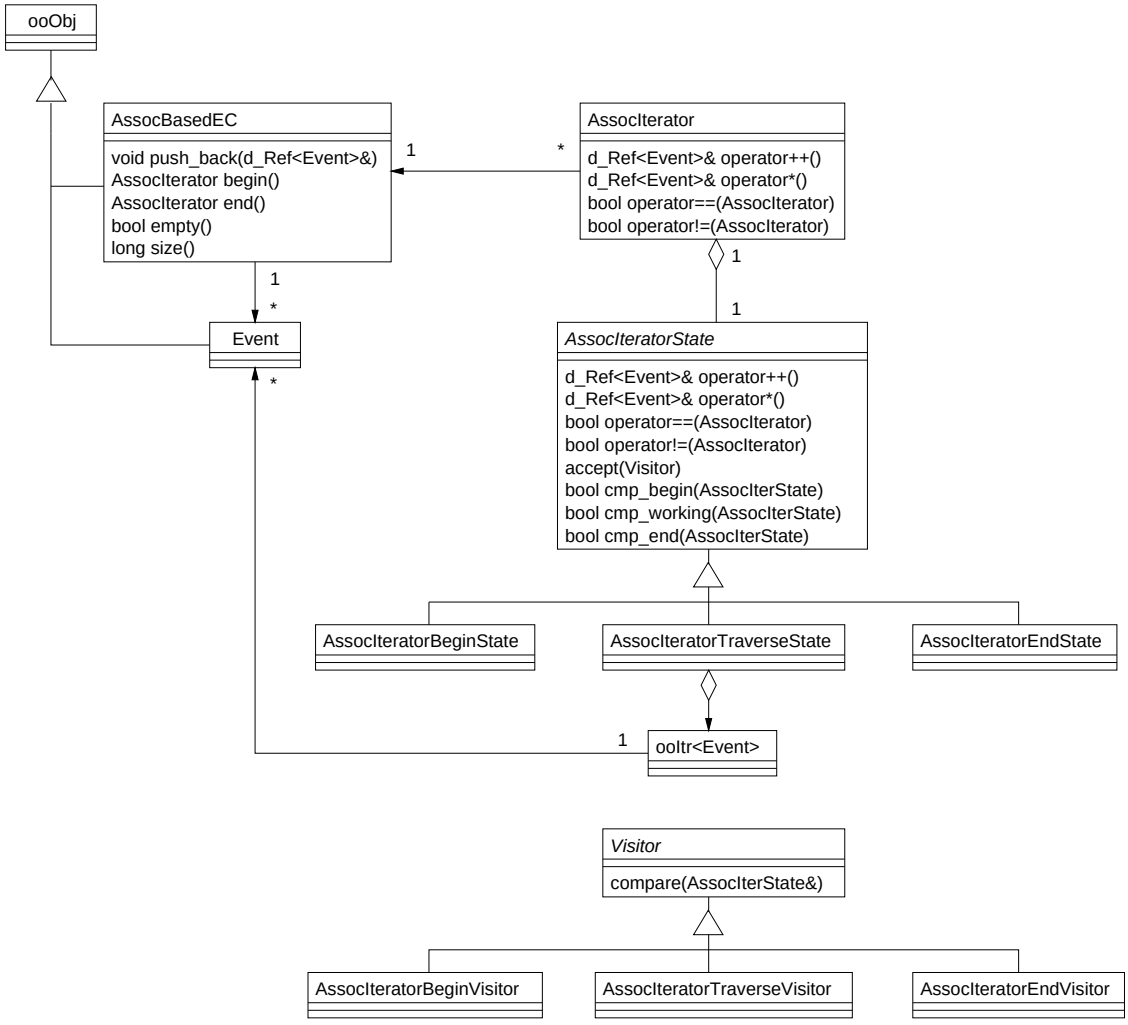
- `ooItr` has a different interface than a C++ standard library iterator. Instead an increment operator (`operator++()`) `ooItr` implements the method `next()` that returns a `bool`. The method `next` increments the position of the iterator and tests for the container end at once.
- Two `ooItr`'s cannot be compared.
- An `ooItr` cannot be assigned to another one.

Despite these facts, we were able to write a wrapper `AssocIterator` around `ooItr` that provides an interface that almost conforms with the forward iterator interface of the C++ standard library, see e.g. [81]. The remaining restrictions of the class `AssocIterator` are:

- The assignment is restricted. If `i` and `j` are objects of type `AssocIterator` the assignment `i = j` is only defined if
  - `j` points to the beginning of a collection and `j` is not yet dereferenced, or
  - `j` points to the end of a collection.

  The implemention of an assignment operator that is valid for all iterator states is in principle possible but the operation would have a time behavior of $\mathcal{O}(k)$ where k denotes the current iterator position from the beginning.

Figure 4.2.4 Association-based Event Collection



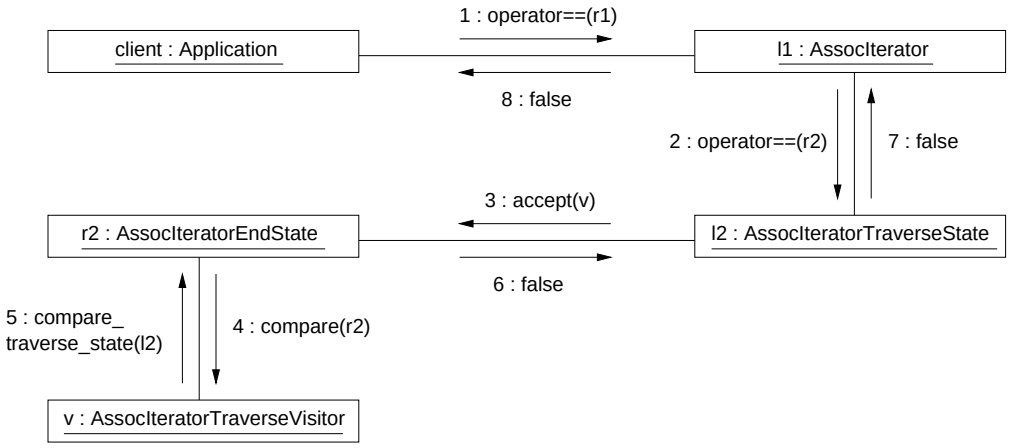
- The post-increment operator returns `void`. The correct behavior would be that the iterator returns a copy of itself before the increment operation. This restriction is a consequence of the lacking copy and assignment operator. The C++ standard defines the following operational semantics for the post-increment operator `r++` where `r` is from type `X` :

```
{ X tmp = r;
  ++r;
  return tmp; }
```

  Since this code snippet makes use of the copy constructor it cannot be used.

The event collection is relatively simple. It has a 1-N associaton to the events. The

Figure 4.2.5 Interaction diagram



main task of the collection class is to forward and map the container methods to the association.

Fig. 4.2.4 depicts the class diagram of the event collection `AssocBasedEC` and the accompanying iterator `AssocIterator`. We use the state pattern [30] to alter the behavior of `AssocIterator`. `AssocIterator` can be in of three states:

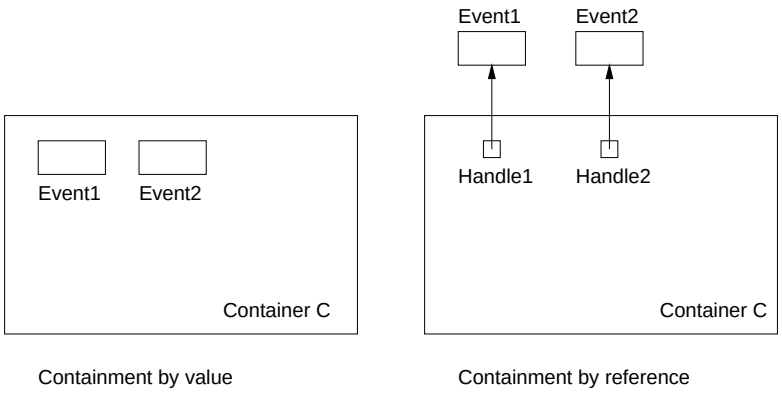
- `AssocIteratorBeginState`. The iterator that is returned from the method `begin()` in the class `AssocBasedEC` is in this state. An iterator in this state may be assigned to another one. The dereference or increment operations take the iterator to the `AssocIteratorTraverseState`.

- `AssocIteratorTraverseState`. This class keeps track of the number of elements visited. This number can be used to compare two iterators that are both in this state. If the iterator reaches the end of the collection it changes the state to `AssocEndState`.

- `AssocIteratorEndState`. The iterator that is returned from the method `end()` in the class `AssocBasedEC` is in this state. An iterator in this state may assigned to another one.

The comparison operators make use of the visitor pattern [30]. The visitor pattern uses a technique called *double dispatch*. Some languages support double-dispatch directly (CLOS, for example). In single-dispatch languages, two criterita determine which operation will fulfill a request: the name of the request and the type of receiver. "Double-dispatch" simply means the operation that gets executed depends on the kind of request and the types of *two* receiver. In our application the code that gets executed for a comparison depends on the kind of comparison (`==` or `!=`), the state of the iterator on the left side, and on the state of the iterator on the right side. Fig. 4.2.5 illustrates the message sequence triggered by a comparison of two iterators: `l1 == r1` (1). We assume that `l1` has the state `AssocIteratorTraverseState`

Figure 4.2.6 Containment by value vs. containment by reference



and `r2` has the state `AssocIteratorEndState`. `AssocIterator l1` forwards the comparison to its state object `AssocIteratorTraverseState l2` (2). `l2` passes the object `AssocIteratorBeginVisitor v` to the state object `AssocIteratorEndState r2` of `r1` (3). `r2` sends the message `compare` (4) to `v`. `v` answers with the message `compare_traverse_state(l2)` (5). `r2` knows now that the other iterator is in `AssocIteratorTraverseState` and the comparison for equality fails. The return value `false` is sent back to the initial client (6–8).

The last container class we discuss is based on Objectivity/DB containers. We use the word *container* in the general sense. A container is a data structure that holds other objects. Objectivity/DB provides a specific persistent data structure capable of holding objects, called Objectivity/DB container. If we address this specific container, we will always write the full name. The Objectivity/DB iterator `ooItr<T>` can also be used to iterate over the objects in an Objectivity/DB container. The implementation is almost a blue-copy of the association-based event collections. The semantic of object membership is different compared to the association-based event collection. An association-based collection contains the objects by reference. In contrast, an Objectivity/DB container contains its elements by value. The same object can only exist in one container. Containment by reference can be implemented using a class `Handle<T>` that is persistent and contains basically an one-directional 1-1 association to an object of type `T`. Let t denote an object of type `T`. Instead of storing t in the Objectivity/DB container the object handle with the association set to t is added to the container. Fig. 4.2.6 illustrates how events can be stored in an Objectivity/DB container either by value or by reference.

4.3 Detector Description

The ATL-SW-Detector_Description-Requirements-V1 document[4] distinguishes between a physical description of the detector and a logical description of the detector.

[4] http://www.cern.ch/Atlas/GROUPS/SOFTWARE/OO/deliverables/ATL-SW-Detector_Description-Requirements-V1/

The physical description covers the dimensions, shape and material composition of the different types of elements from which the detector is constructed. There is also information which corresponds to each element which is actually manufactured and assembled into a detector, for example the positioning of each element. Both active and passive detector elements must be described. The description of active elements should allow for the specification of deficiencies, e.g. dead channels or broken wires, as well as the inclusion of alignment corrections to the element positioning. Finally, there is also detector response characteristics, e.g. the drift velocity of straw tubes, and the energy normalization of calorimeters.

The logical description should provide three primary functions. The first is a simplified access to particular parts of a physical detector description. An example of this would be a hierarchical description where a detector system is described as containing a barrel and two end-caps, each of which is made up of a certain number of layers, etc. There would be a simple way for a client to use this description to navigate to the information of interest.

The second primary function of the logical description is to provide a means of detector element identification. This should allow for different sets of information which are correlated to specific detector elements to be correctly associated with each other. For example, detector element identifiers could be added to raw data in an event. Then a reconstruction program may build a detector model based on the description in the Detector Description Database and using the corresponding identifiers, be able to associate the raw data of each event with its constructed detector model.

The final function provided by the logical detector description is to support access to subsets of a physical detector description which may be qualified by special "attributes". For example, one may use the attribute "sensitive" to refer to active detector elements. Thus, specifying this attribute for a logical description would allow access to all active elements.

4.3.1 AMDB - The Atlas Muon Database

The AMDB database describes in detail the basic parameters of all muon detectors: the Monitored Drift Tube Chambers (MDT) and the Cathode Strip Chambers (CSC) which are the components of the precision chambers system, the Resistive Plate Chambers (RPC) and the Thin Gap Chambers (TGC) that form the trigger system, see Fig. 4.3.1.

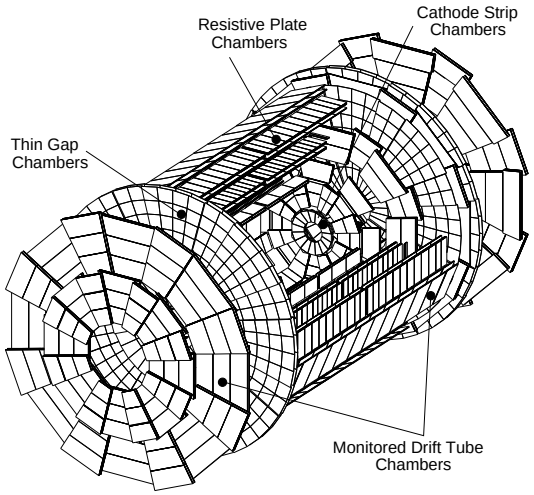
Originally all the data was stored as a textfile. We converted this textfile into an object-oriented database. The resulting class hierarchy is presented in Fig. 4.3.2. We describe each of these classes:

Station A station is a module of several chambers.

StationPosition The absolute position of the station. The type of relationship between Station and StationPosition is 1-N. The same station structure is used in several positions in the muon detector.

ChamberPosition Describes the relative position of a chamber inside the station. For each chamber in the station there is one ChamberPosition object.

Figure 4.3.1 Three-dimensional view of the muon spectrometer instrumentation indicating the areas covered by the four different chamber technologies [17]



Chamber Chamber is the base class for the different chamber technologies (CSC, MDT, RPC, TGC).

CSCChamber, MDTChamber, RPCChamber, TGCChamber, Spacer Describes the internal structures and the dimensions of the chambers. Even if two chambers have the same type (e.g. RPC) their dimensions or structure may vary. For each such structure an own object is used.

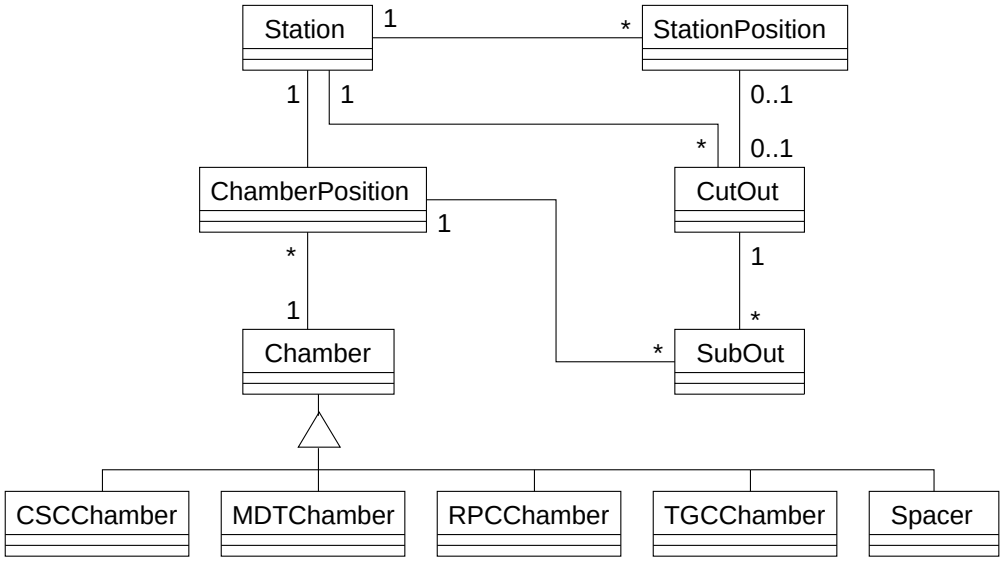
CutOut In some parts of the muon detector space has to be reserved for electronics and cabeling. The CutOuts describe these volumes. Each station can potentially have several CutOuts. The StationPosition specifies the CutOut (if there are any). Each CutOut is composed of several SubOuts.

SubOut Specifies the volumes that has to be taken away from a chamber. SubOut has a relationship to ChamberPosition that specifies the exact chamber where the volume has to be taken away.

4.4 Calibration/Alignment data

Calibration and alignment constants may be referred to but are not stored with the events. There is a plan to try out BaBar's conditions database (adapted by RD45).

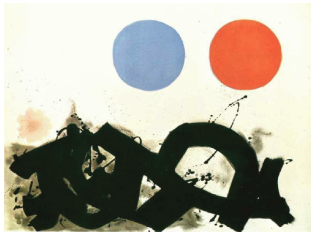
Figure 4.3.2 Class diagram of the AMDB



Chapter 5

The Clustering Algorithm HAMMING

Adolfe Gottlieb
Rolling, 1961
Oil on canvas
Private Collection
New York, USA



This chapter presents a novel reclustering algorithm for HEP data. We nicknamed it HAMMING because it uses the Hamming distance between two bit-vectors. In HEP data analysis the same event collections are repeatedly processed. The performance of the analysis is mainly bounded by the I/O. Hence the question arises in which order the objects should be stored to achieve the highest transfer rates. HAMMING exploits the fact that the event order inside the collections is irrelevant from the physicist's point of view. We will show that HAMMING reduces the number of disk seeks that are required to read the collections almost to the theoretical limit.

This chapter is organized as follows. Section 5.1 describes the general concept and the purpose of object clustering in ODBMSs. In Section 5.2 we present the background that gave the motivation for HAMMING's development. Section 5.3 gives an overview of general clustering algorithms and existing clustering algorithms in HEP. In Section 5.4 the algorithm is mathematically described and defined. Section 5.5 analyses HAMMING analytically and experimentally. For the case of complex compound objects we describe in Section 5.6 a further optimization possibility. In Section 5.7 we put the clustering algorithm in the context of the physical reorganization of a database. Section 5.8 summarizes this chapter and draws some conclusions from the work presented herein. The reclustering algorithm is also described in Ref. [74] and [75].

5.1 Introduction

An application that uses an ODBMS accesses many persistent objects during execution. An accessed object that is not yet in memory has to be fetched from secondary storage. Typically the fetch phase requires a disk search. If the ODBMS is implemented as a page server not only the requested object is brought into memory but the whole page that contains the requested object. The performance of the application will strongly depend on the number of pages that have to be read and on the number of disk seeks that are necessary to read the pages. The possibilities to improve the performance of an ODBMS application by given hardware are:

- *Reduce the number of objects that are accessed.* If the same objects are repeatedly requested a cache improves the performance. Most ODBMS have caches on the client side as well as on the server side. The ODBMS first checks if an requested object is already in the cache. If this is the case a time-expensive request to secondary storage can be prevented. If an application does not access the same objects more than once a client-side cache will not deliver performance improvements.

- *Reduce the number of pages that are accessed.* If objects that are highly likely to be used together are stored on the same page two or more object requests can be satisfied with one page request.

- *Reduce the time of disk accesses.* If logically related data is stored contiguously on disk, the data can be read sequentially without disk seeks.

Clustering describes the object placement on secondary and tertiary storage. A good clustering will decrease the number of pages that have to be accessed and the time of the disk seeks to read these pages. A good clustering will also increase the cache utilization. Cache space is usually restricted. Hence, if too many pages are needed, some of them must be stored onto disk. *Access pattern* describe the frequency of object requests and the object traversal paths. The *clustering problem* is then the problem of finding the optimal object placement for a given set of access patterns. The performance gain of clustering depends strongly on the diversity of the access patterns. Obviously, there is no way of clustering data that satisfies all conceivable access patterns. However, by understanding the dominant access patterns, it is possible to greatly improve both read and write performance. Access patterns can change with time. An initially well-clustered database can become badly-clustered. *Reclustering* describes a change of the object placement on external storage as a response to changed access patterns.

Ever since the "early days" of DBMSs, clustering has proven to be one of the most effective performance enhancement techniques. Figure 5.1.1-(b) shows a compound object that consists of four simple objects. The compound object is badly clustered. Four page accesses and four disk seeks are necessary to read in the whole compound object. Assuming an average access time of 10 ms per disk seek and 1 ms per page transfer this fetch phase lasts 44 ms. In contrast, Figure 5.1.1-(a) shows the same compound object this time well-clustered. Since the pages are contiguously stored only one disk seek and two page accesses are necessary to read in the compound object. This fetch phase lasts now only 12 ms.

Figure 5.1.1
(a) Badly-clustered compound object
(b) Well-clustered compound object

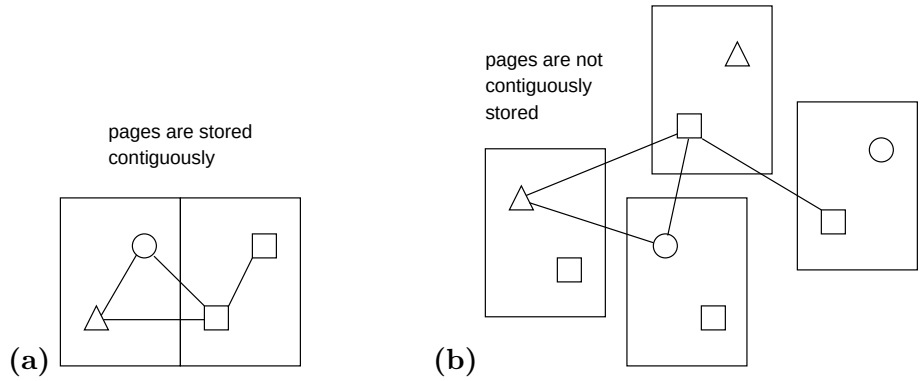
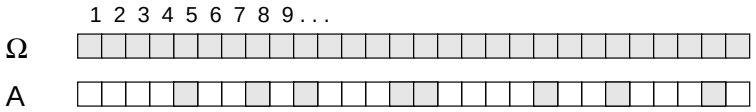


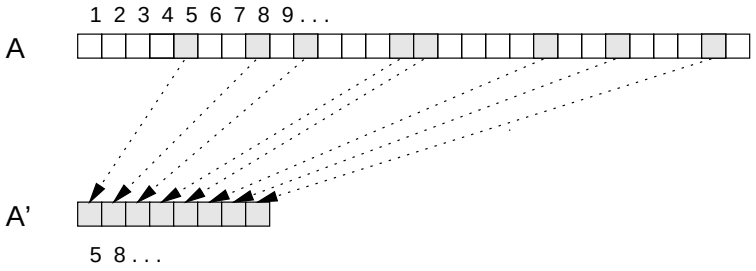
Figure 5.2.1 Sparse event collection



5.2 Motivation

Data analysis in HEP is described in Section 1.4. The physicist repeatedly processes the same data sample. Occasionally, he makes changes to this data sample. For each analysis, the physicist loops over an event collection that contains the OIDs of the events in the data sample. If the data is neither reclustered nor duplicated we have the situation depicted in Figure 5.2.1. Ω denotes the set of all events. A denotes an event collection. Typically the size of an event collection is much smaller than the number of events in the database. This leads to a highly selective read access pattern for the collection A

Figure 5.2.2 A sparse event collection is contiguously copied to a new place



and therefore to bad performance. The traditional solution is to copy the events that are referred to by the event collection A to a new location. We obtain the situation depicted in Figure 5.2.2. This approach may be applicable for small collections but for large collections it is clearly a waste of storage. Furthermore copies of the event or components of the event complicate the data integrity of the database. Typically the event collection contains only a certain object group of the event (e.g. AOD or ESD). The possibility that the same event has several copies of a certain object group creates additional constraints for the event model and the navigation inside the event. For example a bidirectional 1-1 association between the AOD and ESD would have been to be replaced by two 1-N associations.

We overcome these problems by proposing a new reclustering algorithm for HEP event data on secondary storage. The algorithm is based on the Hamming-distance between two bit-vectors, therefore we nicknamed it HAMMING. Using the algorithm, data is clustered according to multiple access patterns. HAMMING exploits the freedom that the event order inside a collection can be changed without affecting the analysis task. This leads to a high scalability. HAMMING maintains the performance for a high number of access patterns.

HAMMING might also be applicable in other domains, e.g.:

- analysis of huge data amounts where queries will be refined iteratively;
- frequently processing of object collections, where the order of objects in a collection is irrelevant;
- compact organization of materialized views.

5.3 Overview of Existing Clustering Algorithms

5.3.1 General Clustering Algorithms

In the last few years much effort has been spent investigating clustering algorithms for ODBMSs, see [83] for a mathematical treatment and [32] for a unifying description of well-known heuristic clustering algorithms. Clustering algorithms can be classified into *sequence-based* techniques and *partition-based* clustering algorithms.

Sequence-based clustering algorithms transform the object net into a *clustering sequence* which is then stored on pages such that all objects on one page form a subsequence of the clustering sequence. Sequence-based clustering works in two steps:

1. All objects to be clustered are sorted applying a method *PreSort*, resulting in the input sequence for the next step.
2. The object net is traversed using the method *Traversal*. *Traversal* is initialized by the first non-visited object of the input sequence and then traverses all objects reachable from this one – this is repeated until all objects have been visited.

The application of different methods for *PreSort* and different methods for *Traversal* results in different clustering algorithms. Ref. [33] gives an overview of typically used combinations.

Figure 5.3.1
(a) Access pattern without read-ahead optimisation [43]
(b) Access pattern with read-ahead optimisation [43]

Partition-based clustering is closely related to the graph partitioning problem. The *object graph* is constructed considering objects as vertices and the associations between the objects as directed edges. From the object graph and the expected access patterns a *clustering graph* is derived. The vertices and the edges of the clustering graph are labeled with weights: vertex weights represent object sizes and edge weights represents the application's access behavior (higher edge weights denote a higher traverse frequency). The clustering problem is to find a partitioning of the clustering graph such that the size of each partition, i.e. the total size of its objects, is less or equal the page size and the total weight of those edges of the clustering graph that cross page boundaries is minimized.

Partition-based and sequence-based clustering were designed for general-purpose application where in a transaction say tens or hundreds of objects are accessed. The general nature of these clustering algorithms becomes a severe disadvantage when they are applied to event data and event collections. These general clustering algorithms are not applicable to the event data, because:

- These clustering algorithms work on the object level. A clustering algorithm for events and event collections has to deal with say 10^9 events. The partitioning of a graph of 10^9 objects exceeds the capabilities of current computer systems. Therefore they scale badly for the order of magnitudes found in HEP.

- They do not exploit the freedom that allows to reorder the events inside an event collections. A specific clustering algorithm could exploit the freedom to find a better performing object placement.

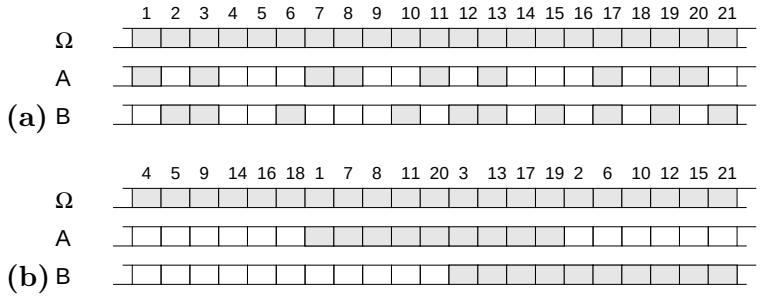
- High transfer rates can only be achieved if pages that are accessed sequentially are also stored sequentially. Of course, this cannot be achieved for all pages and all access patterns but it should be a property of most page accesses. In contrast, partition-based clustering does not describe how the pages should be arranged.

5.3.2 Existing Clustering Algorithms for HEP Data

In Ref. [45] an alternative approach for automatic reclustering of objects in very large databases for HEP is presented. The prototype described therein performs two types of reclustering, called *on-the-fly reclustering* and *batch reclustering*.

- *On-the-fly reclustering.* If the events in a collection are highly fragmented they are copied contiguously to a new location. Figure 5.2.2 illustrates this process.

Figure 5.3.2 Two collections fulfill always the CRP.
(a) Initial object sequence.
(b) Object sequence fulfills CRP.



In contrast to earlier HEP storage systems on-the-fly reclustering is invoked automatically, whenever the performance/price factor exceeds a certain threshold.

- *Batch reclustering.* The set of collections is decomposed into atomic regions, see Figure 5.4.1 and Figure 5.4.2 and the atomic regions are clustered together. Batch-reclustering also removes all duplicates created by on-the-fly reclustering.

If an object has duplicates an *access engine* takes transparently care which copy of the object the user will get. The atomic regions formed by batch reclustering and the collections formed by on-the-fly reclustering form a pool of regions (i.e. chunks of objects). If a job requires a new collection an *optimiser* tries to serve the job from this pool. The optimiser calculates the set of regions that has a minimal sum of their sizes and that covers the new collection, i.e. the new collection is a subset of the union of the set of regions. The optimiser performs this task by solving an instance of the *set covering problem*. In contrast to the clustering algorithm HAMMING that will be introduced in Section 5.4 the iteration order of the collections is not changed. If we assume that a collection is composed of two regions the reading order of the objects might show a pattern as depicted in Figure 5.3.1-(a). This pattern could form a severe performance degradation if both regions are on the same disk. After each object access the disk head would have to move to the alternate region. This performance degradation is prevented by a *read-ahead optimisation*. If an region is accessed a bunch of objects is read in and cached for future utilization. This leads to the efficient read pattern of Figure 5.3.1-(b).

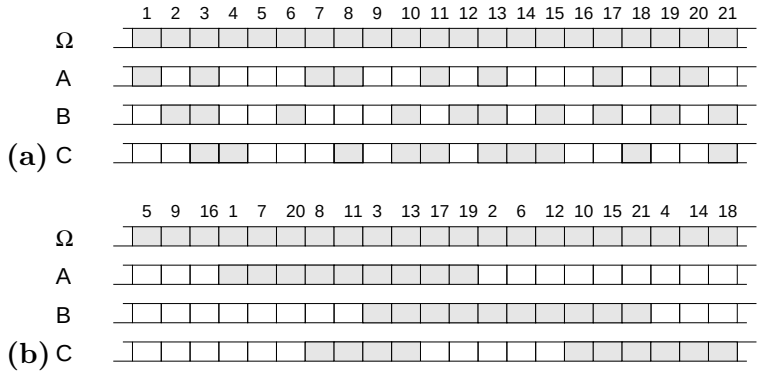
5.3.3 Other Clustering algorithms

Another example of a specific clustering algorithm for scientific databases used in the area of climate modeling can be found in [12]. Ref. [46] describes techniques to use disk space more efficiently by clustering the hot data that is brought from tape to disk.

5.3.4 The Consecutive Retrieval Property

Ghosh [35] introduced in a seminal paper the Consecutive Retrieval Property (CRP). He gives conditions under which a set of records $\{R\}$ can be arranged in such a way

Figure 5.3.3 In general more than two collections do not fulfill CRP.
(a) Initial object sequence.
(b) Object sequence does not fulfil CRP.



that for each query in $\{Q\}$ the relevant records of $\{R\}$ are stored consecutively on a storage medium. If there exists one such organization between $\{Q\}$ and $\{R\}$ then the query set $\{Q\}$ is defined to have the Consecutive Retrieval Property with regard to the record set $\{R\}$. There were a large number of papers published on the subject, e.g. [36]. Unfortunately if the number of queries increases, it is less probable that $\{Q\}$ fulfills the CRP with regard to $\{R\}$. If $card(\{Q\}) = 2$ CRP is always fulfilled. Figure 5.3.2-(a) shows an initial object sequence and two event collections. Two collections have always the CRP. Figure 5.3.2-(b) illustrates how the object sequence can be changed such that the collections are consecutively stored. For $card(\{Q\}) = 3$ there exist already examples that to not fulfill CRP, e.g.:

$$\begin{aligned} \{R\} &= \{1,2,3,4\}, \\ Q_1 &= \{1,2\}, \\ Q_2 &= \{2,3\}, \\ Q_3 &= \{2,4\}. \end{aligned}$$

Non of the 24 permutations of the set $\{1,2,3,4\}$ fulfills the CRP. Figure 5.3.3-(a) shows three event collections. There is no object sequence such that all three collections are stored consecutively. The optimal case is already reached when two collections are stored consecutively, see Figure 5.3.3-(b). The approach taken in this thesis differs from CRP-related works for the case in which a set of queries (collections) can not fulfil the CRP. Ghosh suggests to partition the query set in such a way that each partition fulfils the CRP property, resulting in data replication. We consider the case that, because of disk space constraints, only one copy of the record (object) is allowed and try to minimize the reading time for the set of queries (collections).

5.4 The Clustering Algorithm HAMMING

This section gives a mathematical treatment of the proposed clustering algorithm. We assume that each data item is stored as a single object. For the case where the data item consists of several objects, which might not be accessed in the same analysis task, Section 5.6 describes a further possible optimization. The following notation is used throughout this chapter.

Def. 5.4.1 *Let Ω denote the set of all objects (called the universe) and let $(A_i)_{i\in I_k}$ be a family of (possibly overlapping) subsets of Ω. I_k denotes the first k natural numbers: $I_k = \{1, 2, \ldots, k\}$. We use n to denote the cardinality of Ω: $n = card(\Omega)$.*

We assume that each set A_i is stored in a collection that contains the constituent objects by reference. Hence, an object that is a member of several collections exists only once and is shared between these collections. Let t_i be the time that is needed to read a collection A_i. The goal of the clustering algorithm is to find an object placement that minimizes the sum $\sum_i t_i$. Later on we will also treat the case for which the retrieval of the collections is not equally likely.

The clustering algorithm we propose works in two steps.

1. Decompose the set Ω into chunks, called atomic regions of the sets A_i. The atomic regions form a partition of Ω.

2. Transform the atomic regions obtained in step 1 into a linear sequence. This is done by assigning bit-vectors to the atomic regions and finding the shortest path with regard to the Hamming distance in this set of bit-vectors.

The objects within an atomic region are clustered together. The order of the atomic regions on the disks is determined by the sequence obtained in step 2. The result of the algorithm is a new physical order of the elements in Ω. To benefit from this new order, one has to make sure that the collections are read in this order. Preferably, this should be done transparently to the user.

5.4.1 Determining the Atomic Regions

The set of objects is first decomposed into chunks. Each chunk has the property that either all objects in this chunk are in a collection or none of them. Consider Figure 5.4.1. The three sets A, B and C divide the plane into eight atomic regions $a, b, \ldots h$, depicted in Figure 5.4.2. We denote the complement of a set A by $A^c = A \setminus \Omega$. Mathematically, the atomic regions are determined by the following set relations:

$$\begin{array}{rcccccc} a & = & A & \cap & B & \cap & C \\ b & = & A & \cap & B^c & \cap & C \\ & & & \vdots & & & \\ g & = & A & \cap & B^c & \cap & C^c \\ h & = & A^c & \cap & B^c & \cap & C^c. \end{array}$$

The general case is given by the following definition.

Figure 5.4.1 Three intersecting subsets A, B, C of the universe Ω.

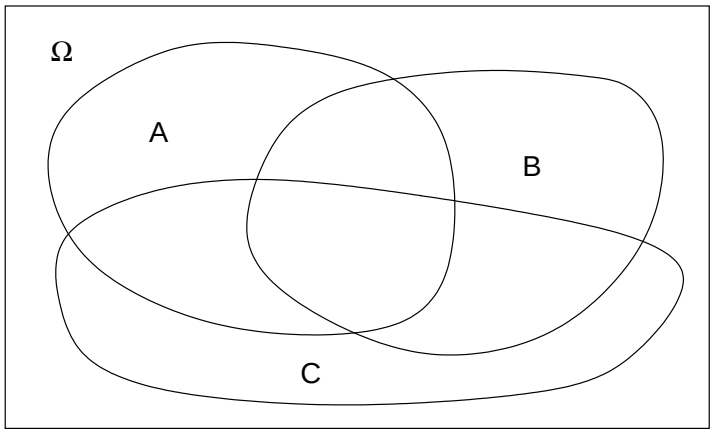
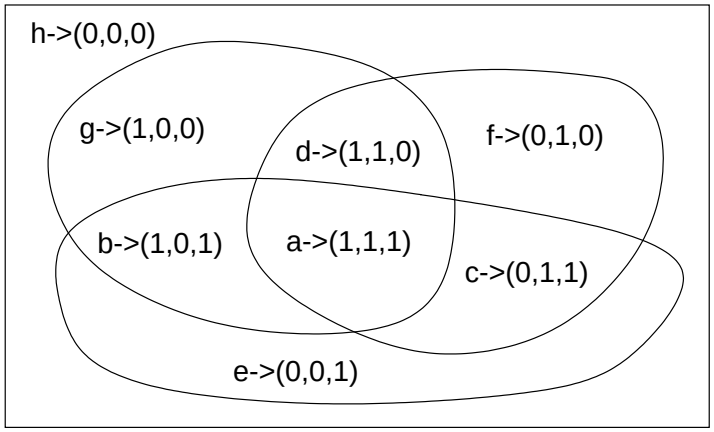


Figure 5.4.2 The sets A, B, C split up into their atomic regions $a, \ldots, h$.



Def. 5.4.2 (atomic region) *Let Ω denote a set and $(A_i)_{i \in I_k}$, a family of subsets of Ω. We call the elements of the set $\mathcal{R}$, given by*

$$\mathcal{R} = \{ \bigcap_{i \in I_k} B_i : B_i \in \{A_i, \Omega \setminus A_i\} \}$$

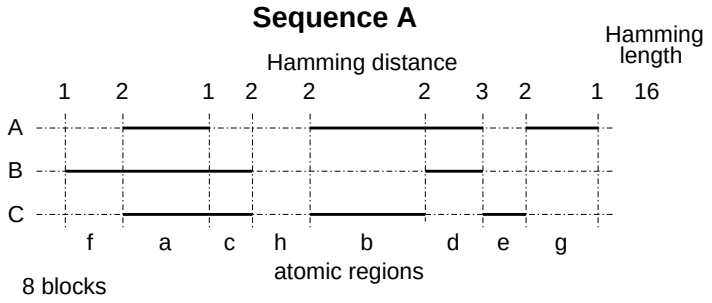
the atomic regions of Ω with regard to $(A_i)_{i \in I_k}$.

$\mathcal{R}$ forms a partition of Ω. Furthermore, for each set $A_i, i \in I_k$, there exists a set of atomic regions such that A_i is their union. There is also another way to construct the atomic regions. Let $\{0,1\}^k$ denote the set of all bit-vectors of length k.

Def. 5.4.3 *The mapping $\varphi : \Omega \to \{0,1\}^k$, which maps an element a of Ω to a bit-vector of length k, is defined by:*

$$(\varphi(a))_i = \begin{cases} 1 & \text{if } a \in A_i \\ 0 & \text{if } a \notin A_i. \end{cases}$$

Figure 5.4.3 The atomic regions of Figure 5.4.2 ordered in a random manner.



Lemma 5.4.4 *φ gives rise to an equivalence relation $\sim$ in Ω by setting $a \sim b$ if and only if $\varphi(a) = \varphi(b)$. $\mathcal{R}$ is obtained as the set of all equivalence classes of $\sim$, also called the quotient set of Ω relative to the equivalence relation $\sim$.*

Def. 5.4.5 (characteristic bit-vector) *The natural map $\nu : \mathcal{R} \to \{0,1\}^k$ of the equivalence relation $\sim$ is defined by $\nu([a]) = \varphi(a)$, where $[a]$ denotes an equivalence class of $\sim$. Let R be an atomic region. The bit-vector $\nu(R)$ is called the characteristic bit-vector of the atomic region.*

A standard algebra book like Ref. [47] explains the relationship between a map and an equivalence relation in more detail. The mapping ν is illustrated in Figure 5.4.2. If the collections are implemented as ordered data-structures and the order of the object references in each collection is the same, the atomic regions can be calculated efficiently by traversing all collections in parallel.

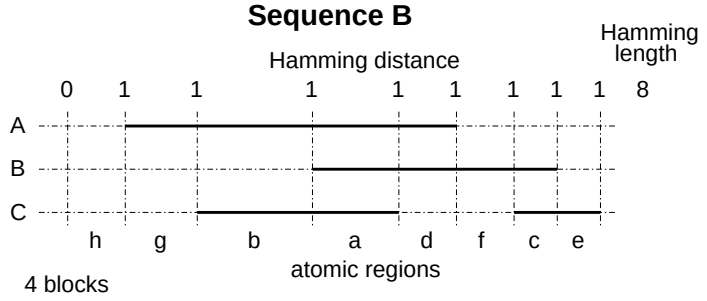
We shall consider next the question: Given k sets, what is the maximum number of atomic regions that can be obtained ? The mapping ν is injective, hence the number of atomic regions is less than or equal to 2^k. If $n = \text{card}(\Omega) \geq 2^k$ the subsets $A_i, i \in I_k$ can be chosen in such a way, that the number of atomic regions is also equal to 2^k, leading to 2^k as an upper bound.

5.4.2 Transforming the Atomic Regions into a linear Sequence

An analysis task reading one of the collections will either access all objects in an atomic region or none of them. The objects in an atomic region are therefore stored together. Thus the issue is how to order the atomic regions on disk. The problem is mapped to a graph in which each atomic region is a node and the metric chosen is the Hamming distance of the characteristic bit-vectors. A heuristic can then be used to find an approximation to the shortest tour.

Figure 5.4.3 shows a particular sequence of atomic regions. Some atomic regions that are contained in the set A are stored contiguously; between others there are gaps. We call a maximal interval of contiguously stored atomic regions that are all contained in the set A a block of A with regard to the order of the atomic regions. The blocks in Figure 5.4.3 are marked as bold lines. The formal definition follows.

Figure 5.4.4 The atomic regions of Figure 5.4.2 ordered by a TSP heuristics.



Def. 5.4.6 (block) *Let $(R_1, R_2, \ldots, R_r)$ be a sequence (i.e. permutation) of the atomic regions. A subsequence $(R_p, R_{p+1}, \ldots, R_q)$ is called a block of the set $A_j, j \in I_k$, with regard to the sequence $(R_1, R_2, \ldots, R_r)$ if:*
(i) all $R_i, p \leq i \leq q$, are subsets of A_j;
(ii) the subsequence is maximal: ($p = 1$ or $R_{p-1} \notin A_j$) and ($q = r$ or $R_{q+1} \notin A_j$).

To read the first object in a block the disk drive has to perform a seek. All other objects in the block can be read sequentially. The number of disk seeks required to read the collections (independently) is therefore equal to the total number of blocks. The cost model that is applied to minimize the reading time is a simple one. Since the transfer time of a block is invariant with regard to the ordering of the blocks, transfer time requirements do not have to be taken into account. The difference in the cost between two different sequences of the atomic regions is therefore due to the different number of disk seeks and seek times. To simplify the matter further, the assumption is made that all disk seeks have the same cost.

We will make use of the Hamming distance between two bit-vectors. Let x and y be bit-vectors of length k. The Hamming distance between x and y, denoted by $d(x, y)$, is the number of digits in which x and y differ:

$$d(x, y) = \text{card}(\{i : x_i \neq y_i\}).$$

The Hamming distance forms a metric over a set of bit-vectors. We extend the Hamming distance to a sequence of bit-vectors in a natural way.

Def. 5.4.7 (Hamming length) *For a path $p = (x^{(1)}, \ldots, x^{(j)})$, consisting of j bit-vectors of length k, the Hamming length is defined as*

$$l(x^{(1)}, \ldots, x^{(j)}) = d(\mathbf{0}, x^{(1)}) + \sum_{i=1}^{j-1} d(x^{(i)}, x^{(i+1)}) + d(x^{(j)}, \mathbf{0}). \tag{5.1}$$

$\mathbf{0}$ *denotes the zero bit-vector* $(0, \ldots, 0)$.

The total number of blocks is related to the Hamming length in a very simple way.

Proposition 5.4.8 *Let $(R_1, \ldots, R_r)$ be a sequence of the atomic regions, where every atomic region is contained exactly once. Setting $x^{(i)} = \nu(R_i)$ leads to the sequence*

$(x^{(1)}, \ldots, x^{(r)})$ of characteristic bit-vectors. The total number of blocks of the sets $A_i, i \in I_k$, with regard to this sequence of atomic regions is given by

$$\#blocks = l(x^{(1)}, \ldots, x^{(r)})/2.$$

Proof: The problem is first reduced to the one-collection case. Pick an arbitrary collection A_i. The sequence $(x_i^{(1)}, \ldots, x_i^{(r)})$ represents the membership of the atomic regions to the set A_i. To give an example, for the sequence in Figure 5.4.3 and the set B one would obtain the sequence $(1, 1, 1, 0, 0, 1, 0, 0)$. The intervals of 1s form the blocks for this set. The Hamming distance $d(x, y)$ between two bits x, y is equivalent to the Exclusive-Or operation $x \oplus y$ $(0 \oplus 0 = 0, 0 \oplus 1 = 1, 1 \oplus 0 = 1, 1 \oplus 1 = 0)$. Each crossing from an interval of 0s to an interval of 1s and vice versa adds a 1 to the Hamming length. Inside an interval the Hamming distance of neighbor elements is 0. Since a block is characterized by a 0-1 crossing at the beginning and a 1-0 crossing at the end, every block adds 2 to the Hamming length. The **0**-bit-vectors used in Def. 5.4.7 allows for the case in which the sequence starts or ends with a 1-interval. For the sequence above, the Hamming length is 4, corresponding to the 2 blocks of set B. Since the Hamming length is linear in each component the proposition follows. □

Figure 5.4.3 and Figure 5.4.4 illustrate this relation for the example given in Figure 5.4.1. Figure 5.4.3 shows a random order of the atomic regions resulting in 8 blocks. In contrast, Figure 5.4.4 shows an optimal sequence with the minimum number of 4 blocks.

5.4.3 The Hamming Salesman Problem

To minimize the number of blocks, a shortest path in a set of bit-vectors with regard to the Hamming distance, for which every element is visited exactly once, has to be found. This is an instance of the traveling salesman problem (TSP). Chapter 7 is devoted to the Traveling Salesman Problem. Here we introduce only the most important concepts. The traveling salesman problem is one of the most widely studied combinatorial problems, see e.g. [69]. Let $G = (V, E)$ be a graph and let $C = (c_{ij})$ be a distance (cost) matrix associated with the edges E. The traveling salesman problem consists of determining a minimum distance circuit passing through each vertex in V once and only once. It is well known that this problem is NP-complete. If the vertices are bit-vectors and the distance between two of these vectors is their Hamming distance, the problem is known as the Hamming salesman problem. It is also NP-complete [23].

If the collections are not equally likely to be read, a weighted Hamming distance can be used. Let x, y be bit-vectors of length k and let $w_1, \ldots, w_k$ be real numbers in the interval $[0, 1]$ satisfying $\sum_i w_i = 1$. The weighted Hamming distance $d_{\mathbf{w}}$ is defined as

$$d_{\mathbf{w}}(x, y) = \sum_{i=1}^{k} w_i(x_i \oplus y_i)$$

where $\oplus$ denotes again the Exclusive-Or operation. The use of a weighted Hamming distance still leads to a TSP.

There are numerous heuristics that find an approximate solution to the TSP. One of the most simple and fastest heuristics is the nearest neighbor algorithm, e.g. [69]:

1. Choose one bit-vector as the beginning of the tour.

2. As long as there are bit-vectors not yet at the tour, do the following. Find the bit-vector not yet on the tour that is closest to the bit-vector last added and insert it at the end of the tour.

An upper bound for the optimal Hamming length is obtained by using the Gray encoding. Let B be the set of the 2^k bit-vectors of length k. The Gray code with Hamming distance 1 that maps the first 2^k integers to the 2^k bit-vectors of length k supplies a path exhausting the set B. The Hamming length of the path is 2^k. Now consider a set C which is a subset of B. Choose the same path, omitting the vectors which are not in C. Due to the metric properties of the Hamming distance the path length will also be equal or less than 2^k. The number of blocks of the optimal sequence is therefore less than or equal to 2^{k-1}. Note that this upper bound is only meaningful if the cardinality of C is not much less than 2^k.

5.4.4 Illustration of the Clustering Algorithm HAMMING

Figure 5.4.5 illustrate the clustering algorithm. The pictures in this figure show different object sequences (permutations) of a universe of $n = 400$ elements and 6 randomly chosen collections, each having 120 elements. These numbers are not typical and are chosen only for illustration purposes. Each row corresponds to one subset and each column corresponds to an object. The points in a column correspond to the membership of the object with regard to the 6 subsets. Figure 5.4.5-(a) shows the initial configuration of the sets on disk. In Figure 5.4.5-(b) the objects contained in an atomic region are stored together, but the regions themselves are arranged in a random order. The other two figures show different sequences of the atomic regions. The sequence in Figure 5.4.5-(c) was produced using the Gray code with Hamming distance 1, Figure 5.4.5-(d) shows the sequence resulting from the nearest neighbor algorithm. In this special case the Gray code performs better than the nearest neighbor algorithm because the number of regions is close to the maximum number, which is 2^6. Figure 5.4.5-(c) exposes also a potential danger that one has to be aware of when implementing a TSP algorithm: the algorithm should be fair to all the collections involved. For most heuristics this can easily be achieved by careful implementation, as was done for the nearest neighbor algorithm used in Figure 5.4.5-(c).

5.5 Qualitative and Quantitative Analysis

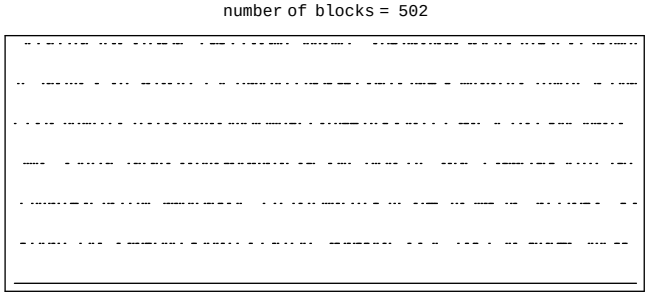
First we derive some properties of uniform distributed sets. Then we study the Hamming salesman problem that arises from transient sets used in a simulation. Finally, measurements from a prototype that implements HAMMING using an ODBMS are presented.

5.5.1 The Number of Atomic Regions of Uniform Distributed Sets

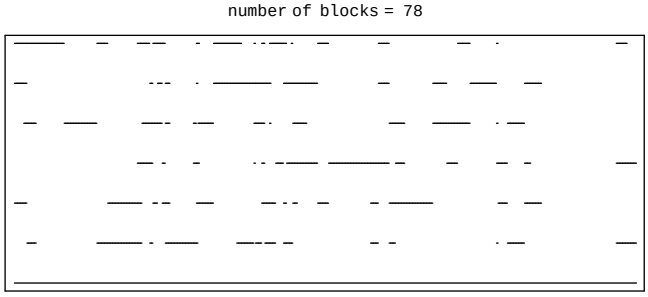
Basic probability theory is used to derive some characteristics of uniform distributed sets. Let Ω denote the universe and let $(A_i)_{i \in I_k}$, be a family of subsets of Ω. For all $a \in \Omega$ and all $i \in I_k$ let $s \in [0,1]$ be the probability that $a \in A_i$. The matter of interest is the probability distribution of the cardinality of an atomic region $R \in \mathcal{R}$ corresponding

Figure 5.4.5 Different permutations of the the same universe and the same collections: $n = 400$, $k = 6$
(a) Initial object sequence.
(b) Objects inside an atomic region are clustered together.
(c) Atomic regions are sorted by the Gray code.
(d) Atomic regions are sorted by the nearest neighbor algorithm.

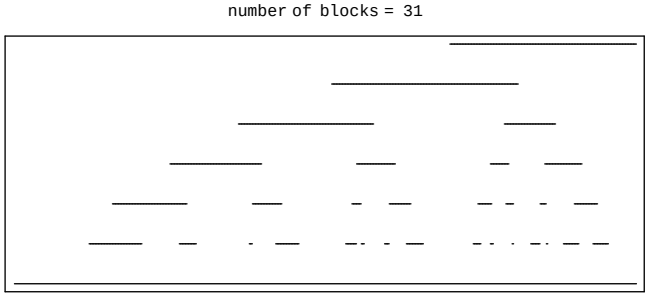
(a)



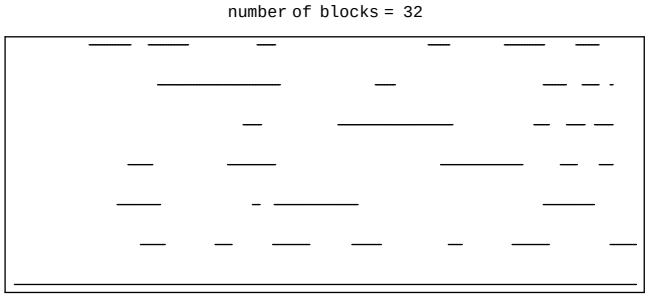
(b)



(c)



(d)



to the characteristic bit-vector $x \in \{0,1\}^k$. x is mapped to an atomic region $R \in \mathcal{R} \cup \{\emptyset\}$ by setting $R(x) = \{a \in \Omega : \varphi(a) = x\}$. This mapping is essentially the inverse to the mapping ν given in Def. 5.4.5. The weight of a bit-vector x, denoted by $w(x)$, is the number of ones which it contains. Since the sets are not correlated, the probability that $\varphi(a) = x$ is $s^{w(x)}(1-s)^{k-w(x)}$ for each element $a \in \Omega$. This leads to a binomial distribution.

Lemma 5.5.1 *Let R be an atomic region and x its characteristic bit-vector. The probability that R has the cardinality t is given by*

$$p(card(R(x)) = t) = \binom{n}{t}[s^{w(x)}(1-s)^{k-w(x)}]^t[1 - s^{w(x)}(1-s)^{k-w(x)}]^{n-t}. \tag{5.2}$$

Corollary 5.5.2 *The probability that the atomic region R is not empty, is given by*

$$p(card(R(x)) > 0) = 1 - p(card(R(x)) = 0) = 1 - [1 - s^{w(x)}(1-s)^{k-w(x)}]^n. \tag{5.3}$$

This corollary leads directly to the average number of atomic regions.

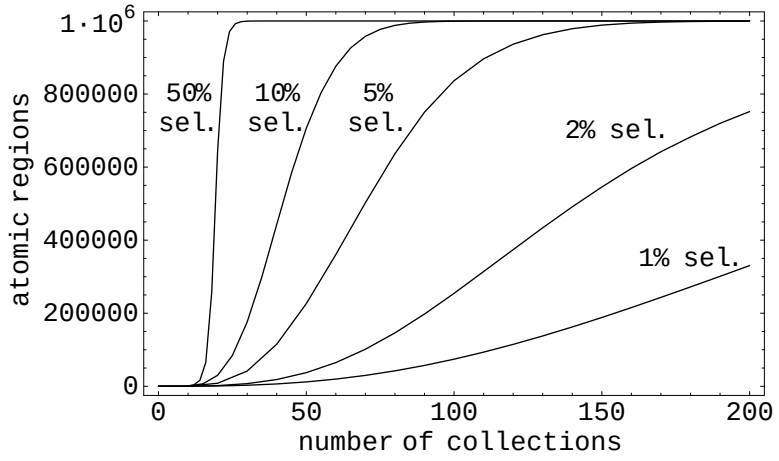
Proposition 5.5.3 *Let Ω be a universe with cardinality n. Let $(A_i)_{i \in I_k}$ be a family of subsets. For all $A_i, i \in I_k$, and all $a \in \Omega$ let s be the probability that a is an element of A_i. The average number of atomic regions is then given by*

$$\overline{r} = \sum_{x \in \{0,1\}^k} p(card(R(x)) > 0) = \sum_{w=0}^{k} \binom{k}{w} \{1 - [1 - s^w(1-s)^{k-w}]^n\}. \tag{5.4}$$

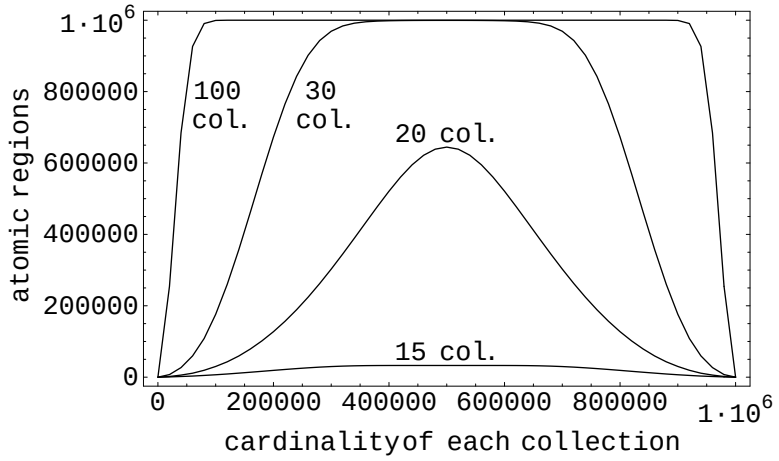
The behavior of $\overline{r}$ is shown in Figure 5.5.1-(a) and Figure 5.5.1-(b) for a universe containing 1'000'000 elements. Figure 5.5.1-(a) depicts $\overline{r}$ vs. the number of collections. The plots are parameterized by the selectivity of the collections. Selectivity s and cardinality m of a collection are related by $s = m/n$, where n denotes the number of objects in the universe. Figure 5.5.1-(b) shows $\overline{r}$ vs. the cardinality of each collection. The plots are parameterized by the number of collections used. Similar plots are depicted in Figure 5.5.2-(a) and Figure 5.5.1-(b) for a universe containing 10^9 elements. Figure 5.5.3 and Figure 5.5.4 show enlarged details of Figure 5.5.1-(a) and Figure 5.5.2-(b). This figures will later on used to estimate the scalability limits of HAMMING. Where the number of atomic regions is 1% of the number of elements in the universe a dashed line is drawn.

We consider again an universe Ω with n elements and k subsets of Ω, each with a selectivity s. The minimum number of atomic regions is one. In this case either all collections are empty or all are equal to Ω. The maximum number of atomic regions is given by $min(n, 2^k)$. We use Monte-Carlo simulation to investigate the statistical distribution of the number of regions. Figure 5.5.5 depicts a histogram showing the statistical distribution for a universe of 20'000 elements and 20 collections each with a selectivity of 10%. The histogram is filled with 11'736 data points and the bin size is 10. Formula (5.5.3) delivers for the average number of regions the value 3661 which is in excellent agreement with the peak location of the histogram. The mean of the

Figure 5.5.1 Number of atomic regions. $n = 10^6$
(a) Number of atomic regions vs. number of collections parameterized by the selectivity of each collection.
(b) Number of atomic regions vs. cardinality of the collections parameterized by the number of collections.



(a)

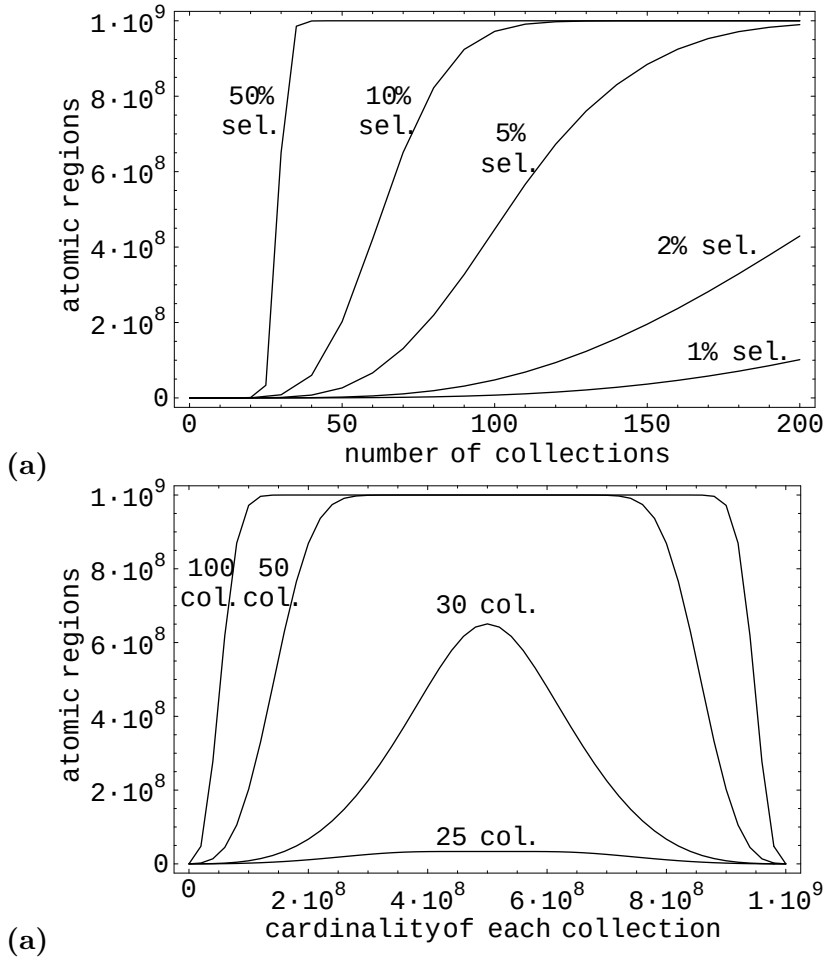


(b)

experimental data is $\overline{r} = 3661$, the standard deviation is $s = 40.6$ and the coefficient of variation is given by $CV = s/r = 1.1\%$. The small standard deviation leads to the conjecture that the number of regions of real HEP data is close to the mean of the simulated data.

To estimate the performance of HAMMING, the reduction in the Hamming length, which can be achieved by a heuristic for the TSP, has to be understood. This will be experimentally investigated in the remainder of this section.

Figure 5.5.2 Number of atomic regions. $n = 10^9$
(a) Number of atomic regions vs. number of collections parameterized by the selectivity of each collection.
(b) Number of atomic regions vs. cardinality of the collections parameterized by the number of collections.



5.5.2 Simulation Results

We implemented HAMMING using transient sets. The next subsection will report about experiments carried out on persistent data. Here we report about the reduction of the number of blocks that can be achieved with a TSP heuristic. Figure 5.5.6 shows the two measurement series that were made. The simulation used a universe with 500'000 elements. In the first series the selectivity of the collections was 2%, in the second it was 10%. The experiments were performed with a different number of collections. Since a selectivity is used to create the collections, the collections are uniformly distributed and are not statistically correlated. The nearest neighbor algorithm was used as a TSP

Figure 5.5.3 Enlarged detail of Figure 5.5.1-(a). $n = 10^6$

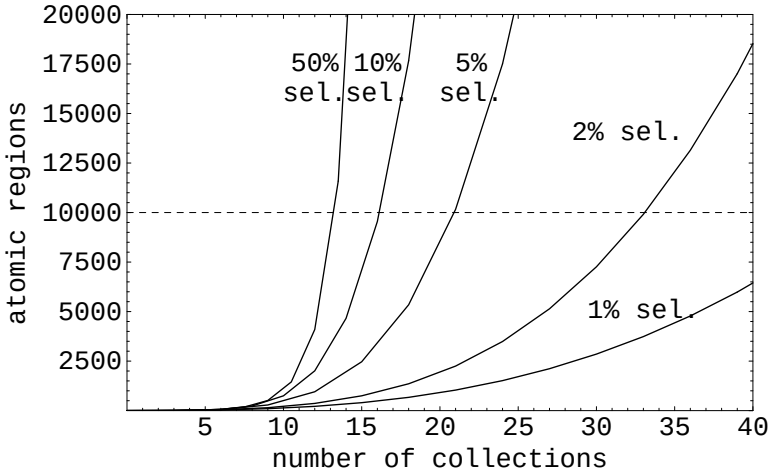
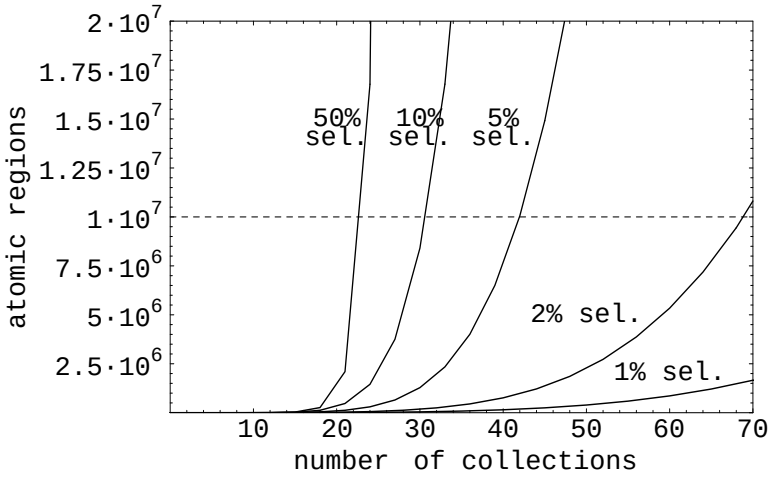


Figure 5.5.4 Enlarged detail of Figure 5.5.2-(b). $n = 10^9$



heuristic. Figure 5.5.6 shows the average number of blocks per collection obtained in these experiments. Figure 5.5.7 depicts the ratio of the Hamming length (or equivalently the number of blocks) between the initial sequence of regions and the final one. The initial sequence refers to a randomly shuffled sequence of the atomic regions.

Knowing the average number of blocks per collection, one can estimate the time it takes to read a collection. We assume that the objects are stored on a disk, which is characterized by transfer rate, average seek time and average latency time. Again we make the simplification that all disk seeks have the same cost.

Figure 5.5.5 Histogram showing the statistical distribution of the number of atomic regions. $n = 20000$, $k = 20$, $s = 2\%$.

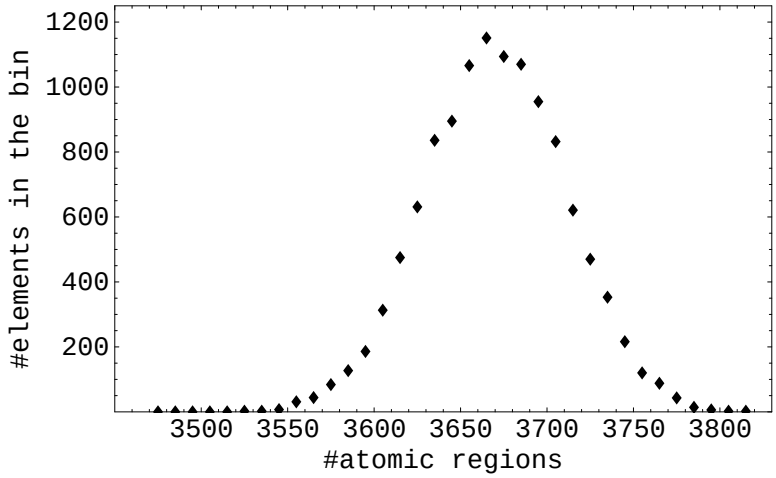
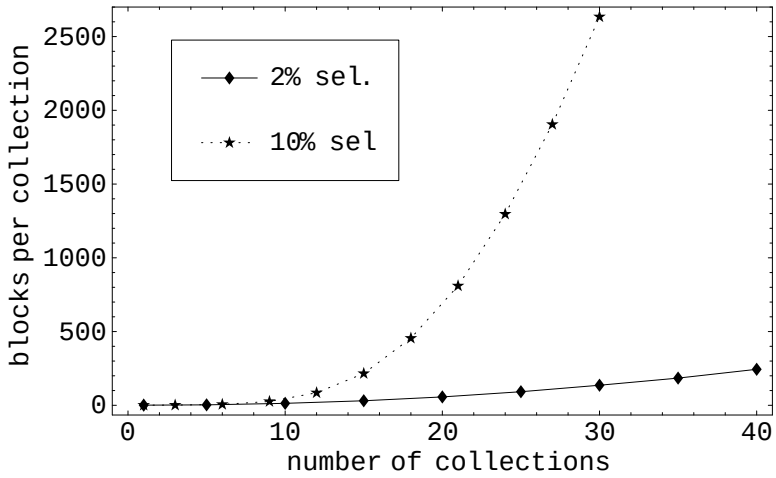


Figure 5.5.6 Average number of blocks per collection vs. number of collections after the TSP heuristic.



Lemma 5.5.4 *The average reading time per collection is given by*

$$t_{read} = \frac{numOfObjects \times objectSize}{transferRate} + bpc \times (seekTime + latencyTime), \qquad (5.5)$$

where bpc denotes the average number of blocks per collection.

CPU requirements are not taken into account. We assume that the reading is disk-bound. Figure 5.5.8 depicts this estimation for the number of blocks we obtained. The following parameters for the disk and for the objects were used: Object Size = 8 KB,

Figure 5.5.7 Ratio between the Hamming length before and after the TSP heuristic.

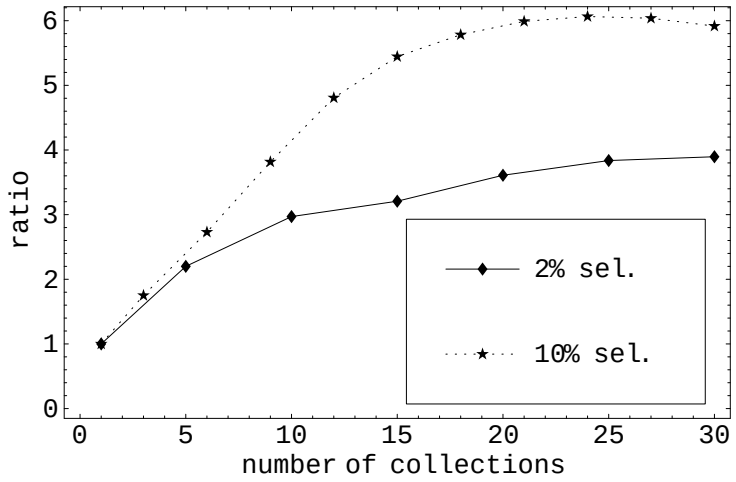
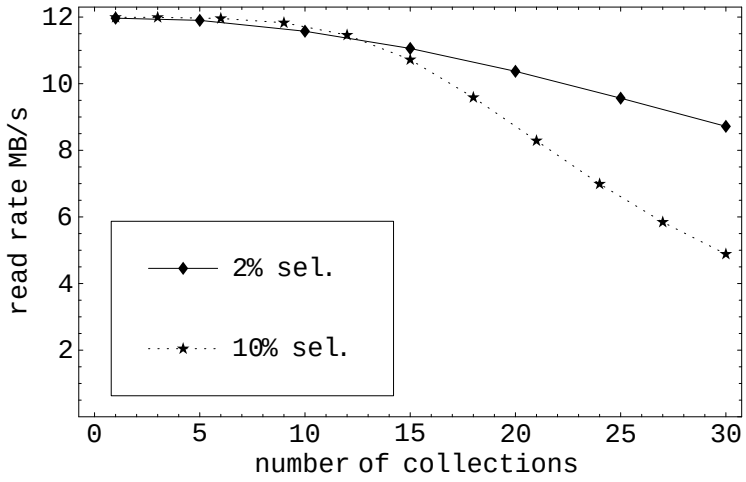


Figure 5.5.8 Estimated read rate vs. number of collections.



Objects = 500'000, Disk Transfer Rate = 12 MB/s, Disk Average Seek = 13 msec, Disk Average Latency = 5 msec.

5.5.3 Experimental Results

We implemented the clustering algorithm HAMMING using the commercial ODBMS Objectivity/DB [64]. A Seagate Elite Disk was used, which has the same parameters as given above, except for the transfer rate. Due to Zone Bit Recording (ZBR) the transfer rate covers the range 10.8 to 15.5 MB/s. In the first experiment the database was populated with 500'000 objects, each with a size of 8 KB. We measured the read rate after reclustering for different numbers of collections, each collection having a selectivity

Figure 5.5.9 Read rate measured vs. number of collections, selectivity = 10 %, object size = 8 KB.

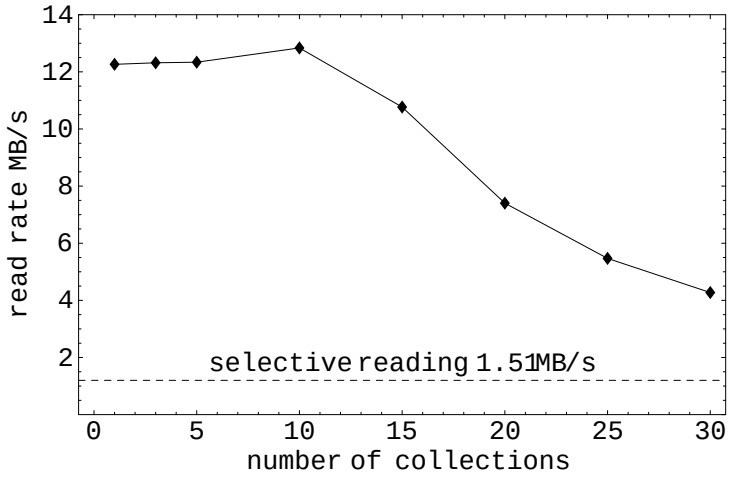
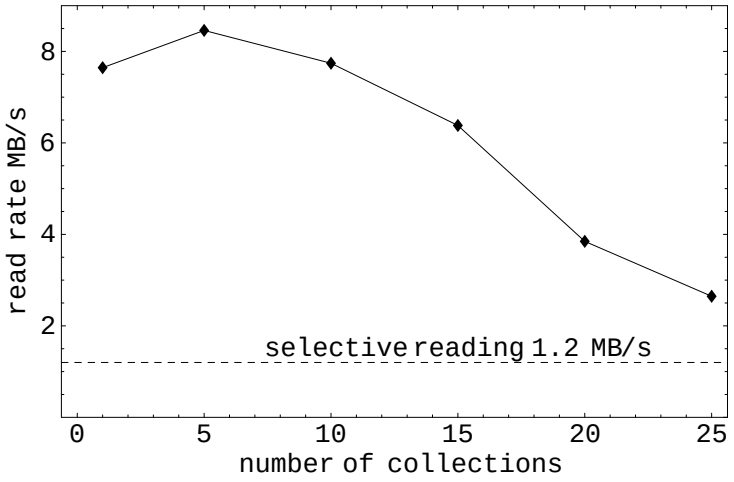


Figure 5.5.10 Read rate measured vs. number of collections, selectivity = 10 %, object size = 1 KB.



of 10%. The page size of the database was 8 KB. Figure 5.5.9 shows the results. The read rate for the unclustered case does not depend on the number of collections and is drawn in Figure 5.5.9 as a dashed line. The same experiment was also carried out with 1'000'000 objects, each having an object size of 1 KB. Figure 5.5.10 shows the results for a collection selectivity of 10 % and Figure 5.5.11 shows the results for a collection selectivity of 2 %. The measured results depicted in Figure 5.5.9 are in accordance with the simulated results in Figure 5.5.8, taking into account the broad transfer range due to ZBR. For the first experiment, using 8 KB objects, it took an average of 2.5 hours to

Figure 5.5.11 Read rate measured vs. number of collections, selectivity = 2 %, object size = 1 KB.

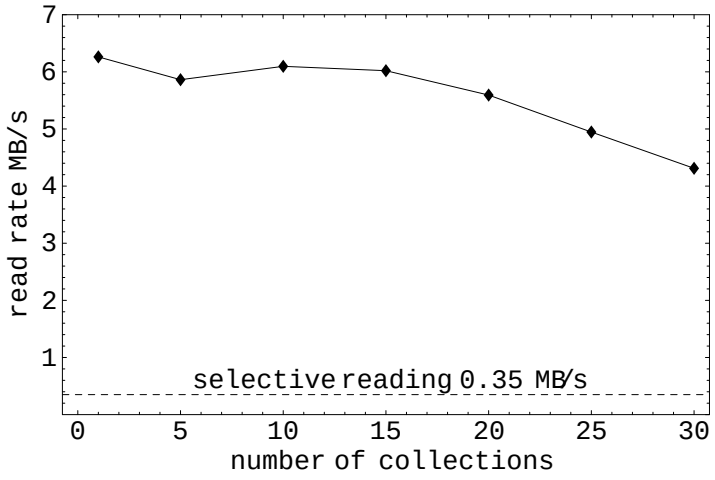
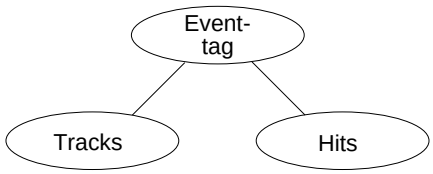


Figure 5.6.1 A simplified event object model.



recluster the database (4 GB). The reclustering was done sequentially. We describe a typical measurement in more detail. For the case of 15 collections, the reading time for one collection was reduced from 247 sec before reclustering to 36 sec after reclustering, resulting in a speedup factor of 6.9. If every collection is accessed at least 3 times, the total time gain is larger than the time to recluster the objects. The reduction factor is even better for smaller object sizes and less selectivity.

5.6 Declustering of Objects and Prefetching

Complications which arise, when compound objects are used, can be solved by a read-ahead (prefetching) mechanism. The kind of data considered in this paper has two dimensions: the number of attributes and the number of instances (compound objects). In HEP, mentioned in the introduction, these are the number of events and the attributes of each event. An event itself is a very complex entity and will therefore split into numerous objects. A physicist performing analysis is in general only interested in some of these objects. For the sake of illustration a very simplified object model of an event is introduced, see Figure 5.6.1. The Event-tag object contains summarized information and links to the other objects. The Hits object represents the raw data, the outcome of

Figure 5.6.2 Two possibilities to cluster the components of an event.

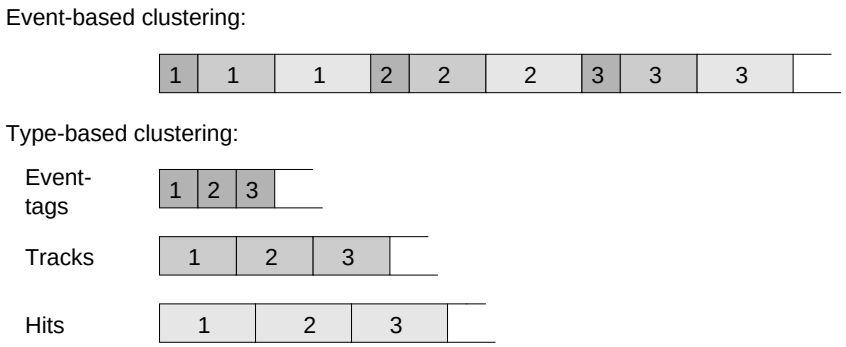
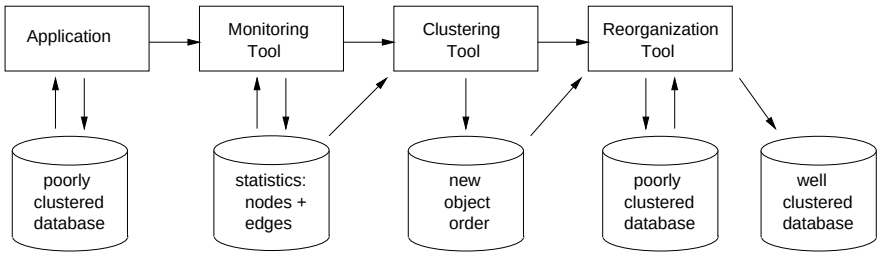


Figure 5.7.1 Overview of the restructuring process [34]



the detector. The Tracks object contains the reconstructed particle trajectories.

There are two possible scenarios concerning how the components of an event should be stored on disk (see Fig 5.6.2). Event-based clustering stores the Event-tag, the Hits and the Tracks together. For a physicist who wants to process all these components in his analysis job, this is the best solution. On the other hand, a physicist who is only interested in tracks would experience a performance degradation because the data of interest is not stored consecutively. Type-based clustering declusters the events. Event-tags, Tracks and Hits are stored as separate streams. At first sight, it seems that some performance degradations will have to be accepted. A physicist who only wants to process tracks now obtains the maximal performance. A physicist's job that accesses all components of the event experiences performance degradation because of the disk seeks between the readings of the components. In contrast to the first case, a solution is now possible. If the data of a stream is read ahead into a buffer and the stream is only changed afterwards, nearly sequential throughput is achievable (see [43]). Figure 5.3.1 illustrates the unoptimized and the optimized case. For up to about 10 streams, almost no performance degradation is reported.

5.7 Monitoring and Reorganization

The clustering framework involves three different services, see Figure 5.7.1 and [34], [59]:

- *Monitoring*: Information about the object accesses in the database have to be gathered, that serve as input for the clustering algorithm. Furthermore the logged information may also serve to estimate when the benefits of reclustering outweighs the costs of physical reorganization. In general object databases two kinds of information have to be gathered during the monitoring phase [34].
  - *reference counts*: the number of times an object is accessed, and
  - *link counts*: the number of times two object identifiers id_i and id_j occur consecutively within the trace.
- *Computation of a new object order*: A clustering algorithm takes monitored access patterns as input and calculates a new order of objects in the database optimized for these access patterns.
- *Physical reorganization*: In this phase the objects are physically moved to their new location. If physical object identifiers are used, the association between the objects have also to be updated.

The monitoring of the access patterns of data analysis in HEP is relatively easy. Instead of reference counting individual objects it is sufficient to count the references to the individual event collections. Some policy has to be chosen regarding how iterations over collections should be treated that do not iterate over the whole collection range. As an example solution one might chose for each iteration a real number in the range $[0, 1]$ that expresses the range over which the iteration was done. The input to the clustering algorithm will not only consist of the logged access patterns, but also of user hints. E.g. if a user wants to start soon a new analysis using different data samples the system should also provide a possibility to supply the new data sample as a hint.

The input to the clustering algorithm HAMMING is either a set of collections if the unweighted Hamming distance is used or a mapping of the collections to the real numbers, whereby the real numbers expresses the expected access frequency in the future. The outcome of the clustering algorithm is a new object order. This order could for example expressed as a list of logical event IDs.

The physical reorganization actually moves the objects. If the database uses logically OIDs the reorganization process is finished after the object shifting. In contrast, if the database uses physical identifiers one-directional associations and indexes have to be updated. This can be done as follows. The new physical identifier of an object is entered into the *object placement map*. It consists of a mapping of old physical identifiers to new physical identifiers. If the object map is completed, a second run over the database allows to correct the associations between the objects. Also existing indexes have to be updated in the second run.

5.8 Summary

We have described a clustering algorithm for HEP data and analysis that reduces the number of disk seeks almost to the theoretical minimum. The assumptions that were made are:

- no data duplication, taking disk space constraints into account;

Table 5.8.1 Scalability of the reclustering algorithm HAMMING

card. of the	selectivity of the collections				
universe	1%	2%	5%	10%	50%
10^6	46	33	21	16	13
10^9	110	68	42	30	22

- no caching, because the data volumes are too large;
- no joint-query evaluation, queries are processed independently to obtain short response times;
- only hot objects are read.

A simple model for the response time of a query was applied, treating all disk seek times as constant. Since the experimentally obtained block sizes are relatively large, most disk seeks that move the disk drive's head to the collection's next block will be inter-cylinder seeks, this assumption is to some degree justified. Page alignment of objects that are smaller than the page size was not taken into account. If a collection's block spans at least 10 pages, the effects of page alignment should be negligible. Since the reclustering algorithm works on the collection level instead of the object level, the size of the resulting TSP problem is order of magnitudes less than the number of objects in the database. If 10^9 stored objects lead to say 10^6 nodes in the TSP graph, the TSP problem can already, with today's hardware, be quickly approximated to a good quality.

Table 5.8.1 gives an estimation of the expected scalability of HAMMING. The estimation is based on the number of atomic regions given in Figure 5.5.3 and in Figure 5.5.4, and on the experimental results given in Figure 5.5.10 and in Figure 5.5.11. The numbers in Table 5.8.1 specify the number of collections that can simultaneously reclustered without too much performance loss. For the given number of collections the number of regions is 1% of the number of objects in the universe, see Figure 5.5.3 and in Figure 5.5.4. For these number of regions the obtained transfer rate is still approximately 75% of the maximum, see Figure 5.5.10 and in Figure 5.5.11.

We mention also a drawback of semantic reclustering. A partial retrieval that reads from the beginning of a collection but stops somewhere in the middle can not be treated as a random sample. This means that the complete computation must be performed whereas, with random event ordering, it might be aborted part way through if it were discovered to be uninteresting.

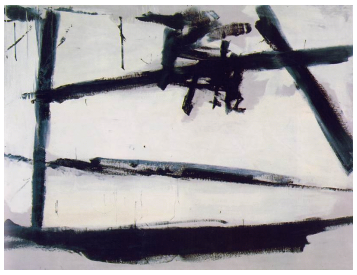
Monitoring the access frequencies of collections is relatively easy and imposes no serious overhead on the database performance. Therefore reclustering can always be scheduled when the benefit outweighs the reorganization costs. The physical reorganization of multi-100 TBs of data offers both a challenge and a new potential for optimization compared to smaller database sizes.

In this paper we assumed that there is only a single copy of each event. Depending on the number of collections, their cardinality and their distribution a single copy may not be sufficient to yield a reasonable performance for all users. Several scenarios are possible

to reduce the number of collections that need to be reclustered. It is common practice in HEP that small collections are copied to a local system. Furthermore, analysis is done by several groups, each investigating a different physical interaction. One plausible scenario would be that every group has its own copy of data. Reclustering is then carried out on the group level. Even if data is to some extend duplicated, reclustering still remains a very important topic. It allows to use the disk space much more efficiently and it reduces the amount of data to be duplicated. Further investigation of data duplication is the subject of future research.

Chapter 6

Description of a Reclustering Prototype



Franz Kline
Painting Number 2, 1954
Oil on canvas, 204.3 x 271.6 cm
The Museum of Modern Art, New York

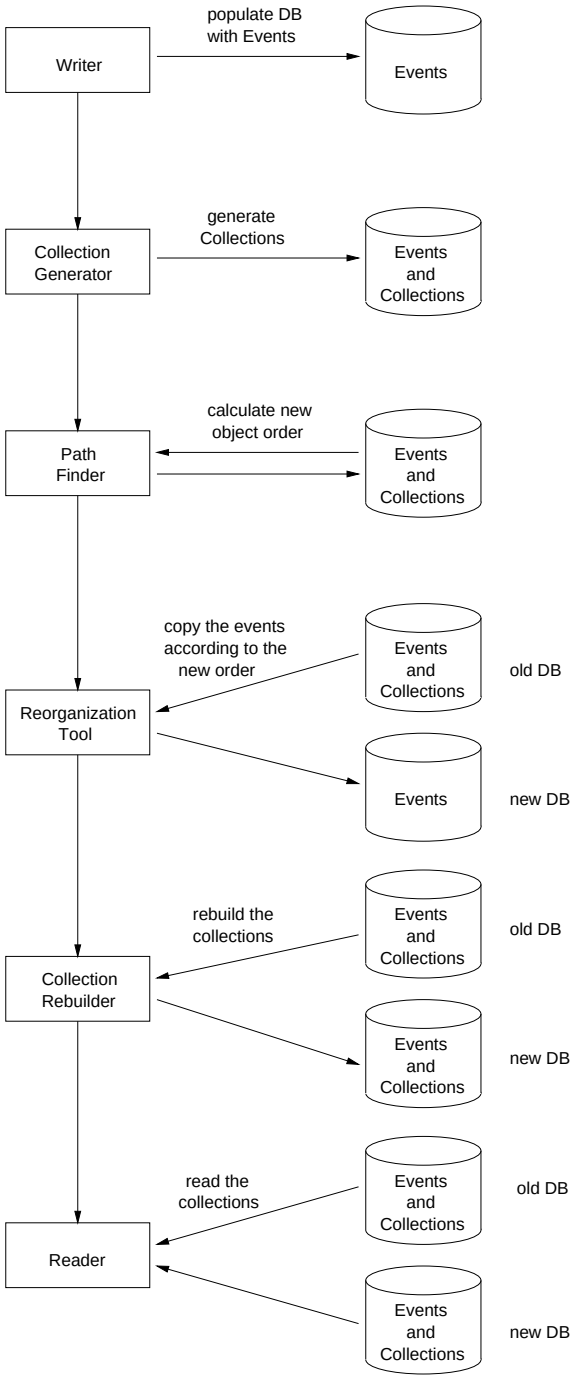
This section describes a prototype implementation of the clustering algorithm HAMMING that was presented in the last chapter. The prototype was implemented in C++ and used the commercial ODBMS Objectivity/DB. Section 6.1 gives an overview of the prototype. Section 6.2 treats some general implementation issues. The individual modules that form the prototype are described in Section 6.3.

6.1 Description

The prototype is built in a modular way as a toolkit . Care was taken that each module performs a well-defined task. The prototype consists of the modules or tools given in Table 6.1.1. Figure 6.1.1 shows an overview of the components constituting the kit and indicates the data flow between the different tools. The prototype has not only to provide services to restructure the database but also services to populate the database with events and read them back.

The tool *Writer* populates the database with a given number of events. It takes care of the creation of Objectivity/DB databases or containers that hold the events. The simulated events that are used in the prototype are simple objects. They consist of an fixed-size array and a logical OID. The size of these objects is specified at compile

Figure 6.1.1 Overview of the prototype



time. The *CollectionGenerator* tool is used to simulate the access patterns. In the current version a number of event collections with a certain selectivity is created. Both the number of collections and the selectivity is specified at the command line. These

Table 6.1.1 The components of the prototype

Executable	*Description*
Writer	generates dummy objects and writes the objects into the federated database.
CollectionGenerator	generates the collections
PathFinder	calculates the new object order, it uses the Clustering algorithm HAMMING
ReorganizationTool	copies the new objects, corresponding to the new object order
CollectionBuilder	rebuilds the collections
Reader	iterates over the collections and reads the objects in the database
ClusterMeter	measures, how good the objects in the database are clustered

event collections constitute the access patters. The event collections are also persistent objects and are stored in the database. Hence there is no need for a monitoring tool. The tool *PathFinder* implements the reclustering algorithm HAMMING. It calculates a new optimized object order. The new object order is stored in the database. Making the new object order persistent has certain advantages. First, it supports increased modularity. If the new object order would only exist in a transient version the tasks performed by *PathFinder* and *ReorganizationTool* could not have been split up into two modules. Second, it increases the scalability of the prototype. Due the fact that the used data structures in the algorithm are persistent, the data structures are not limited by the memory size. The tool *ReorganizationTool* creates a new database and copies the event objects into this database according to the new optimized order. In a production mode either the objects would be moved or the original database would be erased after this step. Since the prototype was built to measure the speedup of the reclustering algorithm the original database is in the prototype conserved. Due to the fact that the event objects do not have any relationships a second iteration over the database to adjust them is not necessary. However, the event collections have to be rebuild. This is done by the *CollectionRebuilder* tool. For each event collection a new one is created, that contains references to the event objects in the new database. Let C denote one of the original event collections. Let C' denote the newly created event collection which is based on C. Let $\sim$ be the following relation between event objects: $e \sim e'$ if and only if e' is a copy of e. The relationship between the original and the newly created collection can then be expressed as follows: $e \in C$ if and only if $e' \in C'$. The collection C and C' contain therefore equal (but not identical) events. However the order of the events inside a collection is not preserved. The event order inside a event collection is irrelevant from a physicist's point of view. This fact is exploited by the reclustering algorithm to obtain further speedups. Finally the tool *Reader* can be used to read the events contained in an event collection. Either one the original event collections can be read or one of the restructured event collections that accesses the reclustered database. The tool *Reader*

can be used to measure the performance gain after reorganizing the databases. The tool *ClusterMeter* analyses the event collections. It calculates the Hamming length of the collections, see Section 5. This quantity can be used to determine how well the database is clustered. Based on this quantity a decision can be made whether it is worthwhile to reorganize the database or not. Furthermore it allows a comparison independent of the collection reading times between the original and the reclustered database.

6.2 Implementation

As mentioned above the prototype was implemented in C++. Exhaustive use of the container classes in the C++ standard library was made, see e.g. [61] or [81].

6.2.1 Bit-vectors

Since the bit-vector is the crucial data structure the reclustering algorithm is utilizing, we describe the alternatives and the final choice in more detail. The C++ Standard Library offers two specialized data structures to handle bit-vectors.

The specialization `vector<bool>` is provided as a compact `vector` of `bool`. A `bool` variable is addressable, so it takes up at least one byte. However, there are techniques that allow to implement `vector<bool>` so that each element takes up only one bit [81]. This data structure has the advantage that the size of of it can be specified at run time. The lack of logical operators (e.g. NOT, AND, etc.) is a disadvantage.

The second data structure provided by the C++ Standard Library is the class `bitset`. It is a templated class that is parametrized by an integer. The integer specifies the size (number of bits) of the `bitset`. The size has to be specified at compile time. Furthermore `bitset` is not resizable at run time. Individual bits can be addressed using the subscript operator like the `vector` class. The positive features of the class are the bit manipulation operations. It offers all the standard operations: AND, OR, XOR as well as shift and flip operations.

The latter fact was the reason to use the class `bitset`, despite the fact that it is not dynamically resizable. Due to the fixed size the array dimensions have to specified at compile time. The following code snippet is taken from a header file.

```
#define BITSET_N 1000000     // maximum size of universe
#define BITSET_K 30          // maximum number of collections
```

It is not good style to use macro processor directives for constants but since the header file is parsed by the DDL preprocessor and the preprocessor has to know the `bitset`-dimension this was the simplest solution.

A further alternative would have been the development of a data structure that is dynamically resizable and provides the standard bit manipulation operations as well. Time constraints prevented this alternative.

The exhaustive usage of templates leads easily to unwieldly names. Therefore the following `typedef`s are introduced which can be regarded as a shortcut for the full name.

```
typedef vector< bitset<BITSET_N> >          VectorBitset_N;
typedef vector< bitset<BITSET_K> >          VectorBitset_K;
typedef list< bitset<BITSET_K> >            ListBitset_K;
```

```
typedef set<int, less<int> >                          IntSet;
typedef map<bitset<BITSET_K>, IntSet, bitset_less>  RegMap;
```

6.2.2 Object Collections

We used an segmented VArray (`raArray`) provided by Objectivity/DB as a base to build the persistent object collections. The class `Vec` contains an `raArray` and provides a subset of the C++ Standard Library `vector` interface. The class `Vec` is templatized. A persistent array that contains objects of the class `FooBar` by reference takes the form `Vec< ooRef <FooBar> >`.

6.3 Description of the individual modules

6.3.1 Writer

`Writer` is a relatively simple module. All it does is to write a given number of objects `DummyObject` in the object store and to create a persistent vector that contains the object references of all the objects. The only complications are due to several restrictions given by the maximum size of a container and the maximum file size. A container is capable of storing a maximum of 65536 pages. If the default pagesize of 8 KB is chosen, this results in a maximum container size of 511 MB. The second restriction is the maximum filesize of the operating system. The Solaris operating system, where the most tests were performed, has a maximum file size of 2 GB.

To deal with these restrictions in a transparent way the creation of containers and databases were done by the class `ContAndDBFactory`. The clustering hint that is specified is the container in which the event objects shall be stored. The clustering hint is provided by the class `FactoryClusteringHint` that makes use of the class `ContAndDBFactory`

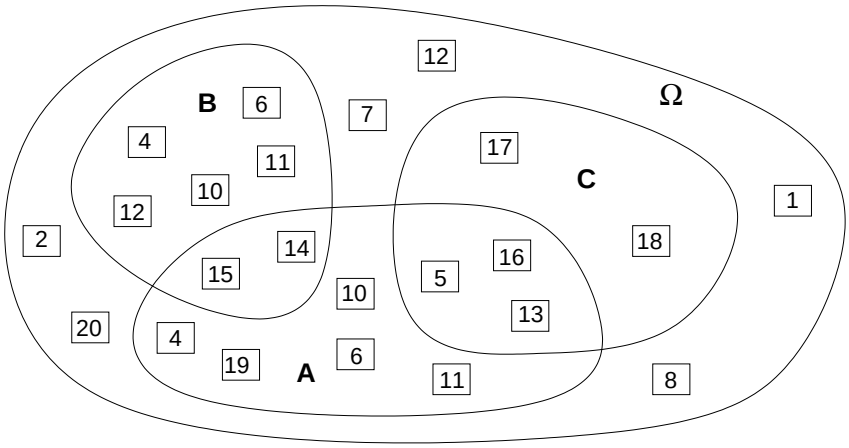
```
//...
ContAndDBFactory factory("db_old_", "cont_old_", session);
FactoryClusteringHint obj_cluster(factory);
//...
// creation of the objects inside a loop
for (i = 0; i < n; ++i) {
    cur_object = new ( obj_cluster.hint() ) DummyObject(i);
    universe->push_back(cur_object);
}
```

If the method `hint()` is called on the object `obj_cluster`, it checkes first to see if either the container or the database size exceeds the upper bound. If this is the case `FactoryClusteringHint` sends a message to an object of the class `ContAndDBFactory` to create either a new container or database. The object `universe` is an object of type `Vec` that holds all OIDs of the created objects of type `DummyObject`.

6.3.2 CollectionBuilder

This module creates a number of collections with a given selectivity. The number of collections as well as the selectivity is specified at the command line. The tool *CollectionBuilder* iterates over the `universe` and creates for every collection a vector of

Figure 6.3.1 Three sets, see also Figure 6.3.2



type `Vec`, where OIDs of the *universe* are inserted with the specified selectivity. These collections are persistent and are stored in the database.

6.3.3 PathFinder

The tool *PathFinder* implements the reclustering algorithm HAMMING that is described in chapter 5. Figure 6.3.1 and Figure 6.3.2 illustrate the implementation of the algorithm. Figure 6.3.1 shows a Venn diagram of the three collections A, B, and C. The squares represent events. The number inside the squares denote the OID of the object. To simplify explanation we assume there are n objects and k collections in the database.

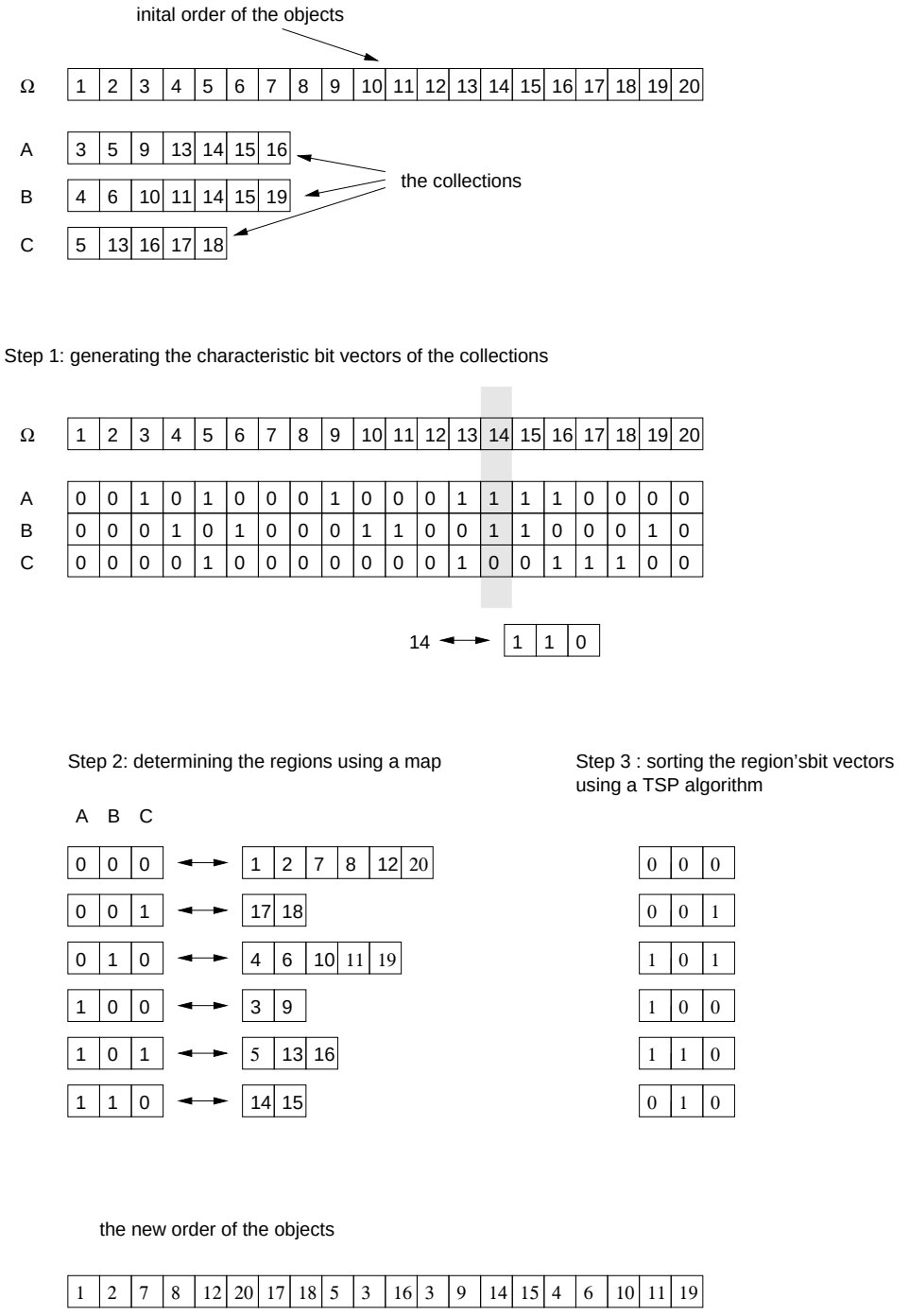
First, the algorithm reads the collections and generate for the j-th collection the characteristic bit-vector $\mathbf{y}_j$ of length n. This bit-vector $\mathbf{y}_j$ expresses the *characteristic function* associated with the collection. Let Ω denote a set and let X be a subset of Ω. The characteristic function χ_X of the set X maps the set Ω into the set $\{0, 1\}$. It is defined by:

$$\chi_X(a) = \begin{cases} 1 & \text{if } a \in X \\ 0 & \text{if } a \notin X. \end{cases}$$

For the sets in 6.3.1 this is performed in Step 1 in Figure 6.3.2. The following simplified code snippet shows the generation of the characteristic bit-vector for one collection.

```
Vec< ooRef<DummyObject> > u;            // universe
Vec< ooRef<DummyObject> > c;            // collection
// ... set u and c
bitset<BITSET_N> b;        // char. bit-vector, all bits are initially 0
int j = 0;                 // index used for c;
for (int i = 0; i != u->size(); ++i)
  if ( u->at(i) == c->at(j) ) {
    b.set(i);
    if ( ++j == c->size() ) break;
  }
```

Figure 6.3.2 The working of the algorithm



The precondition that has to hold is that the OIDs (i.e. the objects of type `ooRef<DummyObject>`) in the `universe` and `collection` are sorted in the same order.

We assign to each object (or OID) a bit-vector $\mathbf{x}_i, 1 \leq i \leq n$ of length k where k

denotes the number of collections. The j-th bit expresses whether the object is member of the j-th collection or not. The bit-vector $\mathbf{x}_i$ forms a representation of the function φ, see Def. 5.4.3. The set of characteristic bit-vectors $\mathbf{y}_j$ of the collections can be viewed as a matrix, where each of the collections' bit-vectors form a row, see again Figure 6.3.1. Let a be the i-th object. The characteristic bit-vector $\varphi(a) = \mathbf{x}_i$ of an object a is derived by reading the row in the matrix that corresponds to a (or to the OID of a). The following matrices equation expresses the relationship between the bit-vectors.

$$(\mathbf{x}_1\mathbf{x}_2\ldots\mathbf{x}_n) = \begin{pmatrix} \mathbf{y}_1^T \\ \vdots \\ \mathbf{y}_k^T \end{pmatrix} \tag{6.1}$$

Step 2 in Figure 6.3.1 determines the atomic regions. An atomic region is formed by grouping all OIDs together which have the same characteristic bit-vector, see Lemma 5.4.4. The prototype uses a `map` to form the atomic regions. A `map` is the C++ Standard Library implementation of an associative data structure. A `map` is a sequence of (key, value) pairs that provides for fast retrieval based on the key. The key value of the `map` is the characteristic bit-vector of an object. The value is used to group the OIDs together. The prototype uses a `set` but other containers could have been used as well (e.g. a `vector` or `list`).

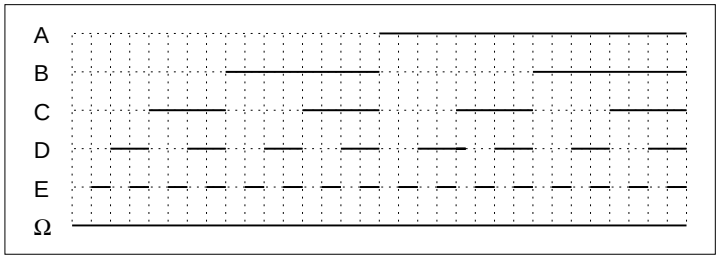
Since the C++ Standard Library requires an ordering relation for the keys, we supplied the predicate (a function object whose return type is `bool`) `bitset_less` that implements the lexicographic order. The predicate `bitset_less` takes two arguments and returns `true`, if the second argument is greater than the first one with regard to the lexicographic order and `false` otherwise.

```
class bitset_less :
  public binary_function< bitset<BITSET_K>, bitset<BITSET_K>, bool>
{
 public:
  bool operator() (bitset<BITSET_K> lhs, bitset<BITSET_K> rhs) {
    for (int i = 0; i < BITSET_K; ++i) {
      if(lhs[i] < rhs[i])  return true;
      if(lhs[i] > rhs[i])  return false;
    }
    return false; // lhs == rhs
  }
};
```

The following method determines the regions. It assumes that `vbs` contains the bit-vectors of the collections. The last argument `n` is the number of objects in the universe. The regions are entered into `reg_map`.

```
// calculate the regions
void
ClusterAlgBitset::regions(RegMap& reg_map,
                          VectorBitset_N& vbs, int n)
{
  reg_map.clear();
  for(int i = 0; i < n; i++) {                // loop through univere
    bitset<BITSET_K> elem_in_sets;
```

Figure 6.3.3 The collections $A, \ldots, E$ are lexicographically ordered.




```
    for(int j = 0; j < vbs.size(); j++)       // loop through subsets
      elem_in_sets[j] = bool(vbs[j][i]);
    reg_map[elem_in_sets].insert(i);
  }
}
```

There is an additional minor detail. Instead of the OIDs the index of an OID in the `Vec universe` is stored in the `map`. An Objectivity/DB OID occupies 8 byte. On the other hand an `int` occupies normally only four bytes.

Knowing the regions they can be sorted by a heuristic for the TSP problem, Step 3 in Figure 6.3.1. First the region's bit-vectors are extracted from `reg_map`. These bit-vectors are the keys of the `reg_map`. Due to the order relation these bit-vectors are lexicographically ordered. Figure 6.3.3 illustrates this ordering. The nearest neighbor algorithm is sensitive with regard to the initial tour. If the bit-vectors arranged in the lexicographically order are taken as the starting tour, the nearest neighbor algorithm favors the collection that map to higher-significant bits, e.g. collection A is favored compared to collection E in Figure 6.3.3. To overcome this problem the bit-vectors are first randomly shuffled (C++ Standard Library function `random_shuffle`). The randomly shuffled sequence is used as the starting point for the nearest neighbor algorithm. The `nearest_neighbor_path` method copies first the region's `bitset`s to a list. The first bit-vector added to the new path is either the zero bit-vector, or if it is not in the region's `bitset`s, then the bit-vector nearest to the zero bit-vector is chosen. As long as there are bit-vectors not visited yet, the following is done. The bit-vector closest to the bit-vector last added is first inserted at the end of the new path and then erased from the list. The implementation is shown in Program 6.3.1, method `nearest_neighbor_path`

The method `nearest_neighborpath` uses the helper function `nearest_neighbor`. The method `nearest_neighbor` takes two arguments: a list of `bitset`s and an individual `bitset`. The function returns an iterator that points to the `bitset` in the list that is the nearest one with regard to the Hamming distance to the `bitset` that is passed as second argument. This method is also depicted in Program 6.3.1.

The Hamming distance between two `bitset`s is easily performed by carrying out an EXCLUSIVE-OR first and then counting the number of one's in the resulting `bitset`, see Program 6.3.1

For transient simulations the Gray code was used as a quality control of the nearest neighbor algorithm. The bit-vectors of the regions were sorted using the Gray code and the Hamming length was compared with the nearest neighbor algorithm. We used a predicate to realize this ordering.

Program 6.3.1 The nearest neighbor algorithm

```
void ClusterAlgBitset::nearest_neighbor_path (VectorBitset_K& res,
                                              const VectorBitset_K& vbs)
{
  res.clear();
  ListBitset_K lbs(vbs.begin(), vbs.end());
  bitset<BITSET_K> zero;
  ListBitset_K::iterator it;
  it = find(lbs.begin(), lbs.end(), zero);  // if zero vector is in set
  if (it != lbs.end()) {
    res.push_back(*it);
    lbs.erase(it);
  }
  bitset<BITSET_K> cur;                     // cur is zero, anyway
  while (lbs.size() > 0) {
    it = nearest_neighbor(lbs, cur);
    res.push_back(*it);
    lbs.erase(it);
    cur = *it;
  }
}

ListBitset_K::iterator
ClusterAlgBitset::nearest_neighbor(ListBitset_K& lbs,
                                   bitset<BITSET_K>& cur)
{
  ListBitset_K::iterator it, nearest;
  int min = BITSET_K;
  for(it = lbs.begin(); it!= lbs.end(); ++it) {
    if (hamming_distance(cur, *it) < min) {
      min = hamming_distance(cur, *it);
      nearest = it;
      if (min == 1)
        return nearest;
    }
  }
  return nearest;
}

int ClusterAlgBitset::hamming_distance(const bitset<BITSET_K>& x,
                                       const bitset<BITSET_K>& y)
{
  bitset<BITSET_K> res = x ^ y;
  return res.count();
}
```

```
class gray_code_less :
public binary_function<bitset<BITSET_K>, bitset<BITSET_K>, bool>
{
 public:
```

```
  bool operator()(bitset<BITSET_K> lhs, bitset<BITSET_K> rhs) {
    int ones = 0;
    for (int i = 0; i < BITSET_K; ++i) {
      if(lhs[i] < rhs[i]) return !(ones % 2);
      if(lhs[i] > rhs[i]) return  (ones % 2);
      ones += lhs[i];
    }
    return false; // lhs == rhs
  }
};
```

Having a predicate that realizes a specific order, a set of bit-vectors is easily sorted by inserting them into a `set` container that uses this predicate as sorting criteria. The function `gray_code_path` accepts an unsorted vector of `bitset`s as argument. The function returns a sorted vector of `bitset`s.

```
vector<bitset<BITSET_K> >
gray_code_path(vector<bitset<BITSET_K> > vbs) {

  // copy vector to set
  set<bitset<BITSET_K>, gray_code_less>  s;
  vector<bitset<BITSET_K> >::iterator it;
  for (it = vbs.begin(); it != vbs.end(); ++it) {
    s.insert(*it);
  }

  // copy set to vector
  vector<bitset<BITSET_K> > res;
  copy(s.begin(), s.end(), back_inserter(res));
  return res;
}
```

We return to the nearest neighbor algorithm. The result of the nearest neighbor algorithm is an order of the atomic regions ad therefore also an order for the objects contained in the atomic regions. A vector of type `PathVector` containing the new order of objects is stored in the database.

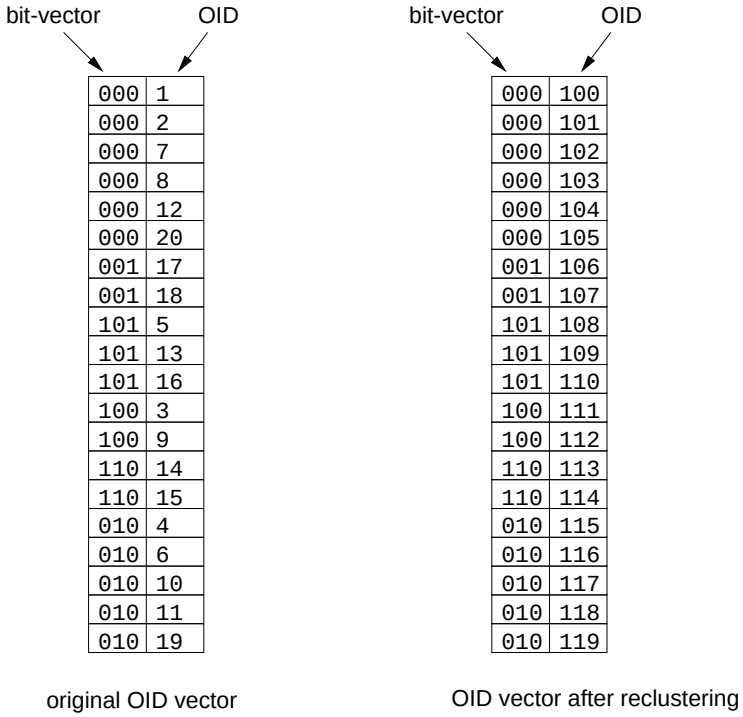
```
typedef Vec < pair< bitset < BITSET_K >, ooRef(DummyObject) > > PathVector;
```

The first value in the `pair` contains the characteristic bit-vector of a `DummyObject` with regard to the membership in the collections. The second value is the OID. The left-hand side of Figure 6.3.4 shows this vector for the example given in Figure 6.3.2 before rearranging the regions.

6.3.4 ReorganizationTool

The module `ReorganizationTool` reorganizes the federated database. As mentioned above, for the case of the prototype, the objects are not moved, but copied. The module iterates over the vector of type `PathVector` that contains the new order of objects (Figure 6.3.4) and copies the current object to a new database. It also creates a new vector of type `PathVector`. This vector maps the OIDs of the new copied object to the characteristic bit-vector of an object with regard to the membership in the collections, see the right-hand side of Figure 6.3.4. The example illustrated in this figure assumes that the OIDs of copied objects starts by 100.

Figure 6.3.4 The mapping between characteristic bit-vectors and OIDs



6.3.5 CollectionBuilder

After the objects are copied, new collections have to be created that point to the copied objects. Since the `PathVector` (Figure 6.3.4) contains the necessary information it can be used to rebuild the collections. The module iterates over the `PathVector` and for each 1 encountered in one of the bit-vectors it adds the object reference in the corresponding collection.

6.3.6 Reader

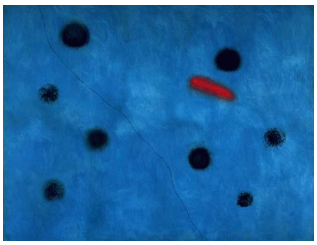
The module `Reader` reads the objects contained in a collection. The collection is specified as an argument at the command line. The implementation is straight-forward and is therefore not described.

6.3.7 ClusterMeter

The module `ClusterMeter` measures the Hamming length of the collections. It does this by reading the `universe` vector and the collections. It does not read the objects referenced by the collections. Since only the collections are read, which only occupy a small space in the whole database, the execution time of the module is fast. The Hamming length delivers a measure for the quality of the clustering. It can be used to decide whether the benefits of a reorganization of the database outweighs the reorganization costs.

Chapter 7

The Traveling Salesman Problem



Joan Miró, Bleu I, 1962
Oil on Canvas, 270 X 355 cm
Collection: Centre Georges Pompidou

"The classes of problems which are respectively known and not known to have good algorithms are of great theoretical interest. ... I conjecture that there is no good algorithm for the traveling salesman problem. My reasons are the same as for any mathematical conjecture: (1) It is a legitime mathematical possibility, and (2) I do not know."

Jack Edmonds, 1966

The HAMMING algorithm maps the clustering problem to the travel salesman problem. This chapter defines the TSP and gives a gentle overview of standard heuristics to find approximate solutions for very large problem instances. For further introduction to the TSP see e.g. [54].

The most prominent member of the rich set of combinatorial optimizaton problems is undoubtly the traveling salesman problem (TSP), the task of finding a route through a given set of cities with shortest possible length. It is one of the few mathematical problems that frequently appears in the popular scientific press or even in newspapers. It has a long history, dating back to the 19th century.

7.1 Definition

Let $G = (V, E)$ be a graph where V is a set of n vertices, E is a set of arcs or edges, and let $C = (c_{ij})$ be a *distance* (or *cost*) matrix associated with E. The TSP consists of determining a minimum distance circuit passing through each vertex once and only one. Such a circiut is known as a *tour* or *Hamiltonian circuit* (or *cycle*). In several applications, C can also be interpreted as a cost or travel time matrix. It will be useful to distiniguish between the cases where C (or the problem) is *symmetrical*, i.e. when $c_{ij} = c_{ji}$ for all $i, j \in V$, and the case where it is *asymmetrical*. Also, C is said to satisfy the *triangle inequality* if and only if $c_{ij} + c_{jk} \geq c_{ik}$ for all $i, j, k \in V$. This occurs in *Euclidian* problems, i.e. when V is a set of points in $\mathcal{R}^2$ and c_{ij} is the straight-line distance between i and j.

7.1.1 The Hamming salesman problem

If the vertices are bit-vectors and the distance used between two of these bit-vectors is their Hamming distance the problem is known as *Hamming salesman problem*. It is also NP-complete [23]. The Hamming salesman problem is encountered in data compression [24], [25], and in the design and testing of computer hardware [14]. Since the Hamming distance forms a metric the Hamming salesman problem is symmetric and it fulfills the triangle inequality.

7.2 Complexity, Exact and Heuristic Solutions

It is well-known that the TSP is NP-complete [31]. NP stands for non-deterministic polynominal and refers to an abstract computer model used to classify the complexity of algorithms, see e.g. [18]. A problem is called NP-complete if every problem in NP is polynomially reducible to it. Assuming the widely believed conjectured that P $\neq$ NP any algorithm for optimal tours must have a worst-case running time that grows faster than any polynomial. The largest TSP instance up to now where an exact solution was found contains 13'509 nodes [2]. The total solution time (i.e. the sum of the running times on parallel machines) for this problem was about 10 years. Since the clustering algorithm HAMMING and the order of magnitudes of HEP data lead to TSP instances say up to 1'000'000 nodes we concentrate in the sequel on heuristics that will find in general only approximative solutions.

Broadly speaking, TSP heuristic can be classified into *tour construction algorithms* which involve gradually building a solution by adding a new vertex at each step, and tour improvement algorithms which improve upon a feasible solution by performing various exchanges. The best methods are *composite algorithms* combining these two features.

When evaluating the empirical performance of heuristics, the optimal solution is in general not available. In consequence the difference of an heuristics to the optimal solution is not known. Instead the difference to an lower bound gained by another heuristic is used. Every heuristic for the TSP delivers an upper bound for the tour length, denoted by c_U. A heuristic for the lower bound results in a lower bound c_L. The heuristic tour is said to have a *quality* $c_U/c_L - 1$. Heuristics used to gain a lower bound are for example the Held-Karp algorithm, see e.g. [69].

7.3 Tour Construction Heuristics

TSP heuristics can be classified into *tour construction heuristics* which involve gradually building a solution by adding a new vertex at each step, and *tour improvement heuristics*, which improve upon a feasible solution by performing various exchanges. We give a short description of the *nearest neighbor algorithm* and a short overview of *insertion algorithms*. For a more detailed treatment see [53] or [69].

7.3.1 The nearest neighbor algorithm

This heuristic for constructing a traveling salesman tour is near at hand. The salesman starts at some city and then visits the city nearests to the starting city. From here he visits the nearest city that was not visited so far, etc., until all cities are visited, and the salesman returns to the start.

1. Consider an arbitrary vertex as a starting point.
2. Determine the closest vertex to the last vertex considered and include it in the tour. If any vertex has not yet been considered yet, repeat step 2.
3. Link the last vertex of the tour to the first one.

The algorithm runs in time $\mathcal{O}(n^2)$. The quality of the solutions can be expected to be in range of 15% t 25% above optimality [69].

7.3.2 Insertion algorithms

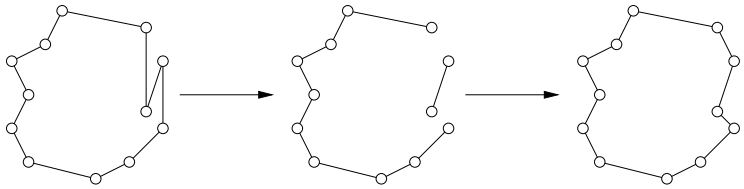
1. Construct a first tour consisting of two vertices.
2. Consider in turn all vertices not yet in the tour. Insert in the tour a vertex chosen with respect to a given criterion, for example:
   – the vertex yielding the least distance increment;
   – the vertex closest to the current tour;
   – the vertex forming the largest angle with two consecutive vertices of the tour, etc.

Examples of insertion algorithms are the nearest insertion algorithm and the cheapest insertion algorithm. The nearest insertion algorithm inserts the node that has the shortest distance to a tour node. The cheapest insertion algorithm inserts the node that causes the lowest increase in the length of a tour. Other important tour construction algorithms are the greedy algorithm, the Clarke-Wright algorithm, and the Christofides algorithm, see e.g. [49]. Ref. [69] contains a detailed description and analysis of the various insertion algorithms.

7.4 Tour Improvement Heuristics

For a more comprehensive overview see [53], [69].

Figure 7.4.1 A 2-opt-move



7.4.1 The r-opt algorithm

1. Consider an initial tour.

2. Remove r arcs form the tour and tentatively reconnect the r remaining chains in all possible ways. If any reconnections yields a shorter tour, consider this tour as a new initial solution and repeat step 2. Stop when no improvement can be obtained.

In general, r is taken as 2 or 3. A 2-opt move is shown in Figure 7.4.1. We describe the 2-opt algorithm in more detail.

1. Let T be the current tour.

2. For every node p in T: examine all 2-opt moves involving the edge between p and its successor in the tour. If it is possible to decrease the length of the tour this way, then choose the best such 2-opt move and update T.

3. If no improving move could be found then stop.

Checking whether an improving 2-opt move exists takes time $\mathcal{O}(n^2)$ because all pairs of tour edges have to be considered.

7.4.2 The Lin-Kernighan Algorithmus

Till around 1990 the *Lin-Kernighan algorithm* (LK) [55] was the leading heuristic for the TSP. It is a complex algorithm incoporating 2-opt and 3-opt moves. An exact description of the algorithm is outside the scope of this work. There exist also numerous variations of this algorithm that offer some advantages, see e.g. [49]. In the last decade the Lin-Kernighan algorithm was merged with ideas coming from genetic algorithms resulting in the *iterated Lin-Kernighan algorithm* (IKL). If the LK algorithm terminates with tour T the tour will be changed to the tour T' and the LK algorithm will be started again. If the LK algorithm finds a tour T'' which is shorter then T', T'' can play the role of T and a further iteration can started. The change from T to T' is in general an *uphill move* to pull the tour out of a local minimum. It seems that nowadays the iterated LK heuristic is the algorithm of choice for large TSPs where high-quality approximations are wanted.

In the last years free and open source implementations of the iterated LK heuristics appeared. LK[1] is written by D. Neto [63] and Concorde[2] is written by D. Applegate et al. Despite the fact that they are especially tuned for euclidian TSPs, they can with some modifications be used to solve the Hamming salesman problem

We present some example runs for very large TSPs instances found in the literature. In Ref. [49] the following run is described. The Lin-Kernighan algorithm was used for a random euclidean TSP instance with one million cities. The computer system in question is an SGI Challenge containing sixteen 150 Mhz MIPS R4400 processors. Only one of these processors was used because the program was running sequentially. The quality that was achieved was 2% after 2650 seconds running time. Ref. [71] and [70] present results for a TSP from the Guide Star Catalog consisting of 18'837'227 cities. There the iterated LK algorithm was used. After two weeks running time on 10 computers of type IBM RS/6000 model 590 and 550 using on the idle times on this system, a tour was found that had a quality of 0.91%.

7.4.3 Comparison of two TSP Heuristics

Ref. [49] reports an average quality of 25% for the nearest neighbor algorithm running on random Euclidean instances. In contrast the iterated Lin-Kernighan algorithm generates tours with qualities better than 1% in comparable run-times. The draw-back of this algorithm is the high complexity, but in the last years free and open source implementations appeared in the academic sector.

Data about the quality of this algorithms used for the Hamming salesman problem was not availabe. Therefore we compared for a small number of instances the tour length of the nearest neighbor algorithm with the tour length of the iterated Lin-Kernighan algorithm. We used D. Neto's open source code implementation LK. Since this implementation was not written for the Hamming salesman problems, some slight extensions in the code were necessary. The results are given in Table 7.4.1. We used bit-vectors that resulted from the atomic regions, see Chapter 5. A universe with 10'000 elements was used. The cardinality of each subset was in each case 1'000. The number of subsets (k) was varied from 10 to 30. The row *initial* denotes the Hamming length of the initial sequence, r gives the number of regions, *clustered* denotes the Hamming length for the case that the elements inside a region are clustered together, but the regions themselves are sorted in a random manner. *NN* gives the tour length obtained by the nearest neighbor algorithm, *LK* the tour length of the Lin-Kernighan algorithm and *IKL* denotes the tour length of the iterated Lin-Kernighan algorithm. The iterated Lin-Kernighan algorithm used 20 iterations. The last row states how much longer the nearest neighbor tour is than the iterated Lin-Kernighan tour. The difference between the outcome of the nearest neighbor algorithm and the iterated Lin-Kernighan algorithm was in all three cases smaller than 20%. This justifies the use of the nearest neighbor algorithm in a prototype. On the other hand, we would like to stress the point, that for production an algorithm with a better quality should be used.

[1] http://www.cs.toronto.edu/~neto/research/lk/
[2] http://www.caam.rice.edu/~keck/concorde.html

Table 7.4.1 Comparison between the nearest neighbor (NN), the Lin-Kernighan (LK), and the iterated Lin-Kernighan (IKL) algorithm. The universe contains 10'000 elements. The columns "initial", "NN", "LK", "IKL" depict the Hamming length.

k	initial	r	clustered	NN	LK	IKL (i=20)	%
10	18482	288	1228	354	300	300	18.0%
20	35818	2352	13952	3330	2854	2850	16.8%
30	53796	5711	39852	10878	9570	9544	14.0%

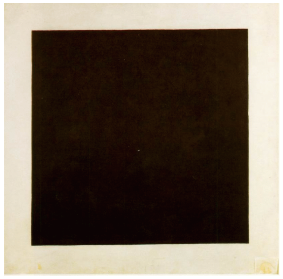
7.5 Conclusions

The prototype described in Chapter 6 used the nearest neighbor algorithm. Hamming salesman problem instances with up to 200'000 nodes were solved. The run time was in the range of up to 2 hours, but in general it was less. For a couple of instances a slightly modified freely available implementation of the iterated Lin-Kernighan algorithm was used. The differences of the tour lengths were in the range 14-18%. Both implementations did not make use of the metric properties of the Hamming distance. We conclude that the risks in applying a heuristic will be less in 2005. We assume a minimum region size of at least 500 events and a number of events on disk of 10^9 (one year's production).

The expected number of nodes (atomic regions) will be not bigger than say 2'000'000. This number already assumes that the whole federation is reclustered. If the reorganization process is limited to a single physics channel, or the data analysed by a group, the number is less.

Chapter 8

Future Work



Kasimir Malevich, Black Square
[1913] 1923-29
Oil on canvas, 106.2 x 106.5 cm
State Russian Museum, St. Petersburg

The following gives an outlook how future developments will influence the aspects considered within this thesis.

8.1 Object-Oriented Database Management Systems

The market for ODBMS products is growing much slower than was predicted. ODBMS products still form a niche market. In contrast object-relational databases have a healthy growing. Coming applications might use object-relational databases instead of ODBMSs not because the former are superior but because they are in a stronger market position. The question that nobody can answer today is whether ODBMSs will be still around in 10 or 20 years. LHC starts at 2005 and will have a run time of 15 or 20 years. For the case that the commercial market cannot provide an appropriate ODBMS product RD45 is currently investigating a fall-back solution. The development of an own ODBMS prototype will show how much effort is necessary to develop and maintain an ODBMS by the HEP community.

8.2 Persistent ATLAS Data Structures

The development of the persistent ATLAS data structures and the physical event model is still in an early phase. The ATLAS computing review recommends a database-application binding based on an transient event model.

8.3 The Clustering Algorithm HAMMING

We presented in Chapter 5 a novel and simple reclustering algorithm for event data. HAMMING calculates the optimal object placement for the event database. It does not treat duplicates of events. The contributions of the reclustering algorithm HAMMING to the general clustering problem in HEP are:

- It calculates a nearly optimal object placement.
- Due to a sound theoretical base, it can be shown that the object placement is nearly optimal.

Therefore HAMMING defines a benchmark for any other reclustering algorithms.

Assuming a given set of access patterns we can distinguish three different storage problems for HEP data:

- *Clustering problem* What is the optimal object placement with regard to the access patterns ? No duplicates allowed.
- *CRP-related problem* What is the optimal object placement with regard to the access patterns if copies are allowed ? CRP-theory searches for the lower bound of storage space so that each collection is continuously stored. The storage space is not a priori bounded.
- *Bounded storage space problem.* We assume the aggregate size of the objects that are referred to in the access patterns is s_{data}. The available disk space is s_{disk} whereby $s_{data} < s_{disk}$ holds. The bounded storage problem is concerned with: Which objects should be copied ? What is the best object placement of the objects (original and duplicates) ?

The CRP-problem is practically irrelevant because of the high storage space requirements. We found a clean and simple solution to the clustering problem. The bounded storage problem seems to be much harder to solve since the choice of objects which become duplicated forms an additional degree of freedom.

Several factors may influence the development of the storage system:

- Detailed access patterns are not known. The existing estimations are very rudimentary. There are different reasons for the lack of detailed access patterns:
  - Earlier experiments used other storage paradigms (FORTRAN software, ZEBRA banks).
  - Monitoring was/is only done on tape level.

  - The data volumes in the coming LHC experiments are order of magnitudes higher than the data volumes in the earlier or current experiments.
  - Access patterns depend very much on the physics channel, i.e. the decays that are investigated.

- Parallelization of analysis jobs and joint query evaluation will have an impact on the physical organization of the database and on the clustering strategy.
- It is not exactly clear how large the effort for the physical reorganization of a PB database is and how far the restructuring process could be optimized.
- Depending on the access patterns it may be necessary to duplicate event data to obtain the necessary performance. The possibility of event duplicates leads to new optimization problems:
  - Which objects should be copied ?
  - How should the objects be clustered ?
  - If a job accesses an object that exists in several copies, which copy should the job access ?

  The disk system is emerging as a bottleneck factor in data-intensive applications. Therefore clustering on object-oriented databases will play an even more important rôle in the future.

Appendix A

Glossary

AOD Analysis Object Data. The AOD is an object group of the ATLAS event model. See Section 4.1.

AMDB ATLAS Muon Database. See Section 4.3.1.

AFS Distributed file system. AFS is widely used at CERN.

ATLAS One of the LHC experiments. See Section 1.3.

atomic region A number of (possibly overlapping) sets can be decomposed into non-overlapping atomic regions. See Section 5.4.

CERN European Laboratory for Particle Physics. Home of LHC and ATLAS.

clustering Describes the object placement on secondary or tertiary storage.

container An object that holds other objects.

CRP Consecutive Retrieval Property. See Section 5.3.

C++ Hybrid programming language. Object-oriented extension of C.

DBMS Database Management System. General-purpose software to create, maintain, query and manipulate databases.

ESD Event Summary Data. The ESD is an object group of the ATLAS event model. See Section 4.1.

event Denotes both the physical collision of particles as well as the software object that stores the data.

event collection A container that holds (parts of) event objects.

FORTRAN Formula Translator. One of the earliest programming languages. Still widely used in HEP.

Hamming distance The number of bits in which two bit-vectors differ.

HEP High-Energy Physics. See Section 1.1.

HPSS High-Performance Storage System.

ILK Iterated Lin-Kernighan algorithm. A TSP heuristic. See Section 7.4.2.

iterator An object that is used to traverse a collection or a container.

Java Platform-independent object-oriented programming language and architecture.

LHC Large Hadron Collider. See Section 1.2.

LK Lin-Kernighan algorithm. A TSP heuristic. See Section 7.4.2.

MSS Mass Storage System.

NP Non-deterministic polynomial. A complexity class. If a problem is in NP but not in P it is widely assumed that it cannot be solved in polynomial time on a deterministic machine (e.g. common computer).

Objectivity/DB A commercial ODBMS.

ODBMS Object-oriented Database Management System. See Section 2.2.

ORDBMS Object-Relational Database Management System. See Section 2.2.

ODMG Object Database Management Group. See Section 2.2.

ODL Object Definition Language. See Section 2.2.

OID Object Identifier. See Section 2.4.

OML Object Manipulation Language. See Section 2.2.

OMG Object Management Group.

P Polynomial. Class of problems that can be solved in polynomial time.

RDBMS Relational Database Management System. DBMS based on the relational data model (i.e. tables).

RD45 A Research and Development group at CERN. Investigates object persistence for HEP data.

STL Standard Template Library. Now part of the C++ Standard Library.

TSP Travel Salesman Problem. See Chapter 7

VArray Variable-size persistent-capable array provided by Objectivity/DB. See Section 4.2.

ZEBRA Storage management system for FORTRAN software.

References

[1] S. Abiteboul and O. M. Duschka. Complexity of Answering Queries Using Materialized Views. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, volume 17, 1998. Available from http://logic.stanford.edu/people/duschka/publications.html.

[2] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. On the solution of traveling salesman problems. *Doc. Math. J. DMV Extra Volume ICM III*, 1998.

[3] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, MA, second edition, 1998.

[4] M. Atkinson, F. Bancilhon, D. DeWitt, D. Maier, K. Dittrich, and S Zdonik. The object-oriented database system manifesto. In *First Int. Conf. on Deductive and Object-Oriented Databases , Kyoto*, December 1989. Also published in/as: provided at SIGMOD May.1990.

[5] ATLAS Collaboration. ATLAS Computing Technical Proposal. Technical Report CERN/LHCC 96-43, CERN, 1996. Available from http://atlasinfo.cern.ch/Atlas/GROUPS/notes.html.

[6] J. Barton and L. Nackman. *Scientific and Engineering C++*. Addison-Wesley, 1994.

[7] Günther Blaschek. *Object-Oriented Programming with Prototypes*. Springer-Verlag, New York, N.Y., 1994.

[8] G. Booch. *Object-Oriented Analysis and Design.* Benjamin Cummings, 1994 (2nd ed).

[9] R. G. G. Cattell. *Object Data Management*. Addison-Wesley, Reading, MA, 1994.

[10] R. G. G. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, Los Altos (CA), USA, 1997.

[11] A. Chaudhri and M. Loomis. *Object Databases in Practice*. Prentice-Hall, Upper Saddle River, 1998.

[12] L. T. Chen, R. Drach, M. Keating, S. Louis, D. Rotem, and A. Shoshani. Efficient Organization and Access of Multi-Dimensional Datasets on Tertiary Storage Systems. *Information Systems Journal*, 1995. Available from http://gizmo.lbl.gov/optimass.html.

[13] Peter M. Chen, Edward L. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID : High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[14] G. Cohen, S. Litsyn, and G. Zémor. On the traveling salesman problem in binary hamming spaces. *IEEE Trans. on Information Theory*, 42(4):1274–1276, 1996.

[15] J. Coldewey. An Access Layer for Object Databases. In Mary E. S. Loomis and Akmal B. Chaudhri, editors, *Object Databases in Practice*, pages 21–32. Prentice-Hall, 1998.

[16] ATLAS Collaboration. ATLAS Technical Proposal for a General-Purpose pp Experiment at the Large Hadron Collider at CERN . Technical Report CERN/LHCC/94-43, CERN, 1994. Available from ftp://www.cern.ch/pub/Atlas/TP/NEW/HTML/tp9new/tp9.html.

[17] ATLAS Muon Collaboration. ATLAS Muon Spectrometer Technical Design Report. Technical Report CERN/LHCC 97-22, CERN, 1997. Available from ftp://www.cern.ch/pub/Atlas/TP/NEW/HTML/tp9new/tp9.html.

[18] C.Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[19] D. J. Dewitt, P. Futtersack, D. Maier, and Velez.F. A study of three alternative workstation-server architectures for object oriented database systems. In *VLDB Conference, Brisbane, Australia, pgs. 107-121*, 1990.

[20] K. R. Dittrich and R. A. Lorie. Version support for engineering database systems. *IEEE Transactions on Software Engineering*, 14(4):429–437, April 1988.

[21] A. Eickler, C. A. Gerlhof, and D. Kossmann. A performance evaluation of OID mapping techniques. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB '95: proceedings of the 21st International Conference on Very Large Data Bases, Zurich, Switzerland, Sept. 11–15, 1995*, pages 18–29, Los Altos, CA 94022, USA, 1995. Morgan Kaufmann Publishers.

[22] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems, 3th Ed.* Addison-Wesley, Reading, MA, 1999.

[23] J. Ernvall, J. Katajainen, and M. Penttonen. NP-Completeness of the Haming Salesman Problem. *BIT (Denmark)*, 25(1):289–92, 1985.

[24] J. Ernvall and O. Nevalainen. Compact storage schemes for formatted files by spanning trees. *BIT*, 19:463–475, 1979.

[25] J. Ernvall and O. Nevalainen. Estimating the length of minimal spanning trees in compression of files. *BIT*, 24:19–32, 1984.

[26] C. Caso et al. Review of particle physics. *European Physical Journal*, C3, 1998.

[27] D. Baden et al. Joint D0/CDF/CD Run II Data Management Needs Assessment. Technical Report CDF/DOC/COMP_UPG/PUBLIC/4100, D0 Note 3197, Fermi National Accelerator Laboratory, 1997. Available from http://fncduh.fnal.gov:8080/workinggroups/runII/Nag_2reports.html.

[28] W. D. Dagenhart et al. An Interim Status Report from the Objectivity/DB Working Group. Technical Report CDF/DOC/COMP_UPG/PUBLIC/4522, Fermi National Accelerator Laboratory, 1998. Available from http://tuhepa.phy.tufts.edu/CDF-OBJY/index.html.

[29] Jr. F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, Twentieth Anniversary Edition*. Reading, MA: Addison-Wesley, 1995.

[30] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[31] Michael R Garey and David S Johnson. *Computers and Intractibility, a Guide to the Theory of NP-Completeness*, chapter A2. W.H. Freeman and Co., San Francisco, 1979.

[32] C. Gerlhof, A. Kemper, C. Kilger, and G. Moerkotte. Partition-Based Clustering in Object Bases: From Theory to Practice. *Lecture Notes in Computer Science*, 730:301, 1993. Available from http://pi3.informatik.uni-mannheim.de/forms/publquery_de.html.

[33] C. Gerlhof, A. Kemper, and G. Moerkotte. Clustering in object bases. Technical report, Universität Karlsruhe, Fakultät für Informatik, 1992. Available from http://dodgers.fmi.uni-passau.de/publications/index-all.phtml#1992.

[34] C. A. Gerlhof and G. Moerkotte. On the cost of monitoring and reorganization of object bases for clustering. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(3):22–??, ???? 1996.

[35] S. P. Ghosh. File Organization — The Consecutive Retrieval Property. *Communications of the ACM*, 15(9):802–808, September 1972.

[36] S. P. Ghosh, Y. Kambayashi, and W.jr.(eds) Lipski. *Database File Organization: Theory and Application of the Consecutive Retrieval Property*. Academic Press (New York NY), 1983.

[37] D. Griffiths. *Introduction to Elementary Particles*. Wiley, 1987.

[38] E. Grochowski and R. F. Hoyt. Future Trends in Hard Disk Drives. *IEEE Transactions on Magnetics*, 32(3):1850–1854, 1996.

[39] The LHC Study Group. The Large Hadron Collider, Conceptual Design Report. Technical Report CERN/AC/95-05(LHC), CERN, 1995. Available from http://www.cern.ch/CERN/LHC/YellowBook95/LHC95/LHC95.html.

[40] F. Halzen and A. D. Martin. *QUARKS AND LEPTONS: An Introductory Course in Modern Particle Physics*. Wiley, 1984.

[41] A. Hanushevsky and M. Nowak. Pursuit of a scalable high performance multi-petabyte database. In *Proc. of 16th IEEE Symposium on Mass Storage Systems*, San Diego, USA, 1999. Available from http://wwwinfo.cern.ch/asd/rd45/reports.htm.

[42] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, California, second edition, 1996.

[43] K. Holtman. Clustering and Reclustering HEP Data in Object Databases. In *Proc. of CHEP'98*, Chikago, USA, 1998. Available from http://home.cern.ch/~kholtman/.

[44] K. Holtman. CPU requirements for 100 MB/s writing with Objectivity. Technical Report MONARC note n. 2/98 - version 1.1, CERN/CMS, Nov. 1998. Available from http://home.cern.ch/~kholtman/monarc/cpureqs.html.

[45] K. Holtman, P. Stok, and I. Willers. Automatic Reclustering of Objects in Very Large Databases for High Energy Physics. In *Proc. of IDEAS '98*, pages 132–140, Cardiff, UK, 1998. IEEE. Available from http://home.cern.ch/~kholtman/.

[46] K. Holtman, P. Stok, and I. Willers. A cache filtering optimisation for queries to massive datasets on tertiary storage. In *Proc. of DOLAP'99*, Kansas City, Missouri, USA, 1999. to be published.

[47] N. Jacobson. *Basic Algebra I*. W. H. Freeman and Company, San Francisco, 1985.

[48] R. Jain. *The art of computer systems performance analysis : techniques for experimental design, measurement, simulation, and modeling*. John Wiley and Sons, Inc., New York, 1991.

[49] D. S. Johnson and L. A. McGeoch. The Traveling Salesman Problem: A Case Study in Local Optimization. Draft of November 20, 1995. To appear as a chapter in the book *Local Search in Combinatorial Optimization*, E. H. L. Aarts and J. K. Lenstra (eds.), John Wiley and Sons, New York., 1995.

[50] D. Jordan. *C++ Object Databases*. Addison-Wesley, 1997.

[51] Randy H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Computing Surveys*, 22(4):375–408, December 1990.

[52] A. Kemper and G. Moerkotte. *Object-Oriented Database Management*. Prentice Hall, 1994.

[53] G. Laporte. The Traveling Salesman problem. *European Journal of Operational Research*, 59:231–247, 1992.

[54] E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan, editors. *The Traveling Salesman Problem*. John Wiley & Sons, 1985.

[55] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, 21:498–516, 1973.

[56] E. Lohrmann. *Einführung in die Elementarteilchenphysik*. Teubner, 1990.

[57] Mary E. S. Loomis. *Object Databases: The Essentials*. Addison-Wesley, Reading, Mass., 1995.

[58] D. M. Malon and E. N. May. Critical Database Technologies for High Energy Physics. In *Proceedings of the 23rd VLDB Conference*, Greece, 1997.

[59] W. J. McIver and R. King. Self-adaptive, on-line reclustering of complex object data. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(2):407–418, June 1994.

[60] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, N.Y., second edition, 1997.

[61] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, Reading (MA), USA, 1996.

[62] O. Nachtmann. *Phänomene und Konzepte der Elementarteilchenphysik*. Vieweg, 1986.

[63] D. Neto. *Efficient Cluster Compensation for Lin-Kernighan Heuristics*. PhD thesis, University of Toronto, 1999.

[64] Objectivity. Objectivity Technical Overview. Available from http://www.objectivity.com/.

[65] RD45 collaboration. Object Database features and HEP Data Management. Technical Report CERN/LHCC 97-8, CERN, 1997. Available from http://wwwinfo.cern.ch/asd/cernlib/rd45/reports.htm.

[66] RD45 collaboration. Object Databases and their Impact on storage-related Aspects of HEP Computing. Technical Report CERN/LHCC 97-7, CERN, 1997. Available from http://wwwinfo.cern.ch/asd/cernlib/rd45/reports.htm.

[67] RD45 collaboration. Using an Object Database and Mass Storage System for Physics Analysis. Technical Report CERN/LHCC 97-9, CERN, 1997. Available from http://wwwinfo.cern.ch/asd/cernlib/rd45/reports.htm.

[68] RD45 collaboration. Status Report of the RD45 Project. Technical Report CERN/LHCC 98-11, CERN, 1998. Available from http://wwwinfo.cern.ch/asd/cernlib/rd45/reports.htm.

[69] G. Reinelt. The traveling salesman: Computational solutions for TSP applications. *Lecture Notes in Computer Science*, 840:viii + 223, 1994.

[70] André Rohe. Parallel Lower and Upper Bounds for Large TSPs. *ZAMM*, 77, 1977. Available from http://www.ing.unlp.edu.ar/cetad/mos/TSPBIB_home.html.

[71] André Rohe. Parallele Heuristiken fuer sehr grosse Traveling Salesman Probleme. Master's thesis, University of Bonn, 1997. Available from http://www.ing.unlp.edu.ar/cetad/mos/TSPBIB_home.html.

[72] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, March 1994. Available from http://www.hpl.hp.com/personal/John_Wilkes/papers/index.html.

[73] RD Schaffer. Overview of ATLAS Database Activities. In *Proc. of CHEP'98*, Chicago, Illinois, USA, 1998. Available from http://www.hep.net/chep98/PDF/sessions.html.

[74] M. Schaller. Reclustering of HEP Data in Object-Oriented Databases. In *Proc. of AIHENP'99*, Crete, Greece, 1999. Available from ftp://ftp.physics.uch.gr/aihenp99/Schaller/.

[75] M. Schaller. Reclustering of High Energy Physics Data. In *Proc. of SSDBM'99*, Cleveland, Ohio, 1999.

[76] J. Shiers. Building a Multi-Petabyte Database: The RD45 Project at CERN. In Mary E. S. Loomis and Akmal B. Chaudhri, editors, *Object Databases in Practice*, pages 164–176. Prentice-Hall, 1998.

[77] J. Shiers. Massive-scale data management using standards-based solutions. In *Proc. of 16th IEEE Symposium on Mass Storage Systems*, San Diego, USA, 1999. Available from http://wwwinfo.cern.ch/asd/rd45/reports.htm.

[78] W. Stallings. *Computer Organization and Architecture*. Prentice Hall, London, 1996.

[79] M. Stonebraker and J. Hellerstein. *Readings in Database Systems*. Morgan Kaufmann, third edition, 1998.

[80] M. Stonebraker, L. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, P. Bernstein, and D. Beech. Third-generation data base system manifesto. *ACM SIGMOD Record 19, 3*, September 1990. Also published in/as: Memorandum UCB/ERL M90/28, Apr.1990. Also published in/as: DS4, Jul.1990, Windermere. Also published in/as: Workshop on OODB Standardization 1, May.1990, Atlantic City.

[81] B. Stroustrup. *The C++ Programming Language (3rd Edition)*. Addison-Wesley, Reading, Mass., 1997.

[82] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems - Design and Implementation*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, second edition, 1997. Includes CD-ROM.

[83] M. M. Tsangaris and J. F. Naughton. A Stochastic Approach for Clustering in Object Bases. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 20(2):12–21, June 1991.

[84] Jeffrey D. Ullman. *Principles of Database and Knowledge-Bade Systems. Volume I: Classical Database Systems.* Computer Science Press, 1988.

[85] Jeffrey D. Ullman. *Principles of Database and Knowledge-Bade Systems. Volume II: The New Technologies.* Computer Science Press, 1989.

Curriculum Vitae

Family name:	Schaller
First name:	Martin
Date of birth:	March 16, 1968
Place of birth:	Vienna, Austria

1974 – 1978	Elementary School
1978 – 1982	Secondary School
1982 – 1987	Higher Technical School, Electronics and Telecommunication
1987 – 1995	Technical University of Vienna, Technical Physics, Diploma thesis "Automaton Logic"
Oct. 1996 – May. 1997	Military Service
Nov. 1997 – Sep. 1999	PhD Student at CERN, affiliated with University Innsbruck, Dep. of Experimental Physics