INDEX DATA STRUCTURES IN OBJECT-ORIENTED DATABASES

The Kluwer International Series on ADVANCES IN DATABASE SYSTEMS

Series Editor

Ahmed K. Elmagarmid

Purdue University
West Lafayette, IN 47907

Other books in the Series:

DATABASE CONCURRENCY CONTROL: Methods, Performance, and Analysis
by Alexander Thomasian
ISBN: 0-7923-9741-X

TIME-CONSTRAINED TRANSACTION MANAGEMENT
Real-Time Constraints in Database Transaction Systems
by Nandit R. Soparkar, Henry F. Korth, Abraham Silberschatz
ISBN: 0-7923-9752-5

SEARCHING MULTIMEDIA DATABASES BY CONTENT
by Christos Faloutsos
ISBN: 0-7923-9777-0

REPLICATION TECHNIQUES IN DISTRIBUTED SYSTEMS
by Abdelsalam A. Helal, Abdelsalam A. Heddaya, Bharat B. Bhargava
ISBN: 0-7923-9800-9

VIDEO DATABASE SYSTEMS: Issues, Products, and Applications
by Ahmed K. Elmagarmid, Haitao Jiang, Abdelsalam A. Helal, Anupam Joshi, Magdy Ahmed
ISBN: 0-7923-9872-6

DATABASE ISSUES IN GEOGRAPHIC INFORMATION SYSTEMS
by Nabil R. Adam and Aryya Gangopadhyay
ISBN: 0-7923-9924-2

The Kluwer International Series on Advances in Database Systems addresses the following goals:

- To publish thorough and cohesive overviews of advanced topics in database systems.
- To publish works which are larger in scope than survey articles, and which will contain more detailed background information.
- To provide a single point coverage of advanced and timely topics.
- To provide a forum for a topic of study by many researchers that may not yet have reached a stage of maturity to warrant a comprehensive textbook.

INDEX DATA STRUCTURES IN OBJECT-ORIENTED DATABASES

by

Thomas A. MUECK
Martin L. POLASCHEK
Universität Wien
Vienna, Austria

SPRINGER SCIENCE+BUSINESS MEDIA, LLC

ISBN 978-1-4613-7849-5 ISBN 978-1-4615-6213-9 (eBook)
DOI 10.1007/978-1-4615-6213-9

Library of Congress Cataloging-in-Publication Data

A C.I.P. Catalogue record for this book is available
from the Library of Congress.

Originally published by Kluwer Academic Publishers, New York in 1997
Softcover reprint of the hardcover 1st edition 1997

Printed on acid-free paper.

CONTENTS

PREFACE

Object-oriented database management systems (OODBMS) are used to implement and maintain large object databases on persistent storage. Regardless whether the underlying database model follows the object-oriented, the relational or the object-relational paradigm, a key feature of any DBMS product is *content based access* to data sets. On the one hand this feature provides user-friendly query interfaces based on predicates to describe the desired data. On the other hand it poses challenging questions regarding DBMS design and implementation as well as the application development process on top of the DBMS.

The reason for the latter is that the actual query performance depends on a technically meaningful use of access support mechanisms. In particular, if chosen and applied properly, such a mechanism speeds up the execution of predicate based queries. In the object-oriented world, such queries may involve arbitrarily complex terms referring to inheritance hierarchies and aggregation paths. These features are attractive at the application level, however, they increase the complexity of appropriate access support mechanisms which are known to be technically non-trivial in the relational world.

In the field of databases and database management systems, such an access support mechanism for improved query performance relies on one or more underlying search data structures and is usually called *index*. Informally, the central idea behind this kind of data structure application is to find the identifiers of all objects fulfilling a given query predicate *without reading the objects from disk*. The practical benefit of indexing large persistent object sets is therefore a significant reduction in the number of disk I/O operations thus yielding a performance gain.

The purpose of this book is to provide technical information about current and future issues of search data structures used to index large object-oriented databases. The intended audience of this book includes all kinds of practitioners involved in OODBMS product selection, application dependent database performance tuning and application development on top of object databases as well as researchers and students interested in the technical issues of object-oriented

databases. The only prerequisite for understanding the material presented in this book is a working knowledge object-oriented modeling and programming concepts and a minimum knowledge of algebraic concepts like for example sets.

After the introduction two preparatory chapters present the underlying database model as outlined in the ODMG-93 [Cat96] proposal on the one hand and a chapter elaborating on the technical issues of search data structures and their use for indexing large data sets on the other hand. The three subsequent chapters deal with major indexing topics in object-oriented databases, in particular, type hierarchy indexing, aggregation path indexing, and speedup of collection operations. The presentation is concluded with a performance analysis example in the field of type hierarchy indexing.

A related issue not covered in this book is physical object clustering or, in other words, the mapping of object identifiers to physical storage addresses. Decoupling support for content based access from physical object management and, therefore, the indexing component from an OODBMS's persistent object store provides a high degree of flexibility for both application programmers and system developers. Therefore the issues in the context of object clustering form a separate research domain beyond the scope of this book. Details about the indexing components of particular OODBMS products have been omitted from this book for two reasons. At first, it is hardly possible to get detailed technical information on the indexing components from vendors and secondly, even if this kind of information could be obtained, it is quickly dated. So it seems to be more appropriate to describe the technical issues and solutions in this field and help the reader in this way to decide about products in presence of timely and hopefully detailed information.

Acknowledgments

We thank our colleagues at the Abteilung für Data Engineering, Universität Wien, for hours of fruitful discussions and in particular our former room mate Erich Schikuta for introducing us to the versatile field of search data structures in the early days.

Also, we would like to thank Professor Ahmed K. Elmagarmid for supporting this book project.

This book would not exist without the continuing encouragement by the people at Kluwer Academic Publishers, especially by Scott Delman and his staff. Special thanks for being patient.

1

INTRODUCTION

One of the most important features of database management systems is *associative* or, in other words, *content based* access to large data sets held on persistent storage media like magnetic disks.

In terms of the object-oriented model, an OODBMS provides access to stored objects not only by using object references[1] but also by using query predicates for particular object properties. The basic requirements to be met by any content based retrieval mechanism are well-known, for example from database management systems based on the relational model. However, due to the semantic richness and the expressive power of the object-oriented model, there is an even larger manifold of content based access patterns to be used in object-oriented query specifications.

Important examples (see also Figure 1.1 for a simple database schema) for object-oriented modeling features yielding characteristic access patterns in the context of OODBMS are:

- properties of a particular object type which are implicitly also part of all subtypes by using the mechanism of *type inheritance*, for example, the property income belongs not only to object type Person but also to types Student and Staff,
- links between different object types (also known as *relationships*) like worksIn relating type Staff to type Department by means of a so called *aggregation path* and

[1] Technically, by data items referring to stored objects using an address or a unique object identifier used to abstract from addresses

T. A. Mueck et al., *Index Data Structures in Object-Oriented Databases*

- instance variables corresponding to multi-valued properties, for example, the property hobbies of type Person used to store a *set* of hobbies per object of type Person.

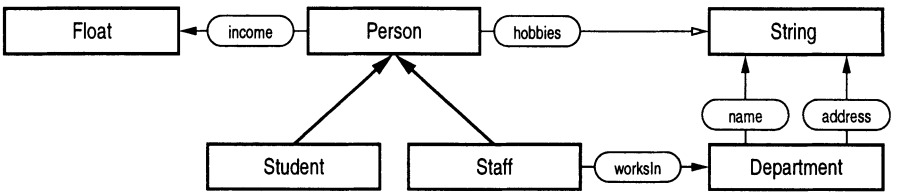


Figure 1.1 Simple object-oriented database schema

The implications of these features for the processing of the resulting query predicates are briefly discussed below.

1.1 OBJECT-ORIENTED DATABASES AND INDEXING

In each of the above mentioned cases, the performance characteristics of content based queries can be enhanced by an access support mechanism, or in other words, by an *index* which uses search data structures to locate all the qualified data.

What is the reason for this enhancement and the central idea behind access support mechanisms in object-oriented databases? To put it into a nutshell: when processing content based queries without any indexing support, an OODBMS has to fetch all objects belonging to the referenced types from mass storage in order to determine the objects fulfilling the predicates.

Informally, the advantage of an index is that the OODBMS is able to locate qualifying objects *without fetching all objects of the appropriate type* from disk therefore providing *associative* or *content based* access to object sets. However, the disadvantage is a certain amount of maintenance overhead incurred by each index, basically one or more search data structures which have to be updated whenever the corresponding object set is updated.

So far, there is not much difference to indexing in relational database systems. Now, what is so special with indexing in object database systems? Generally

speaking, the main differences stem from the extended modeling capabilities sketched above. In the context of this book we will focus on the *structural* aspects of this modeling approach and their impact on indexing, in particular:

- Object-oriented databases provide the concept of *inheritance*: a type can be defined as a subtype of an already existing type which becomes its supertype. In this case, the subtype inherits all properties of the supertype and can extend the set of properties by new ones. In our example above, Staff is defined as a subtype of Person (indicated by a thick, unlabeled arrow) and therefore inherits all properties of Person. Consequently, all objects of type Staff have the properties income and hobbies just like any object of type Person.

  Chapter 4 deals with indexing problems related to type inheritance indexing and presents possible solutions. While a few of these proposals use standard search data structures like B^+-trees ore multi-dimensional data structures, most of the approaches present new or at least significantly modified data structures. As a consequence, this chapter is strongly oriented towards the technical details of search data structures.

- A second important feature of object databases is the concept of *aggregation*. Properties of an object need not be of a simple type like Float or String (see Person.income or Department.name, for example) but can also be relationships between two types. An example for the latter case is the property worksIn which is defined for type Staff and therefore extends the inherited set of properties of this type (see above). The modeling concept *relationship* leads to so-called *aggregation paths* of arbitrary length like Staff.worksIn.address and the related concept of a *path expression* like aStaff.worksIn.address starting with a data item (variable) of type Staff and representing the value of Department.address of the department in which the referenced staff person works. In queries, aggregation paths are handled syntactically more or less the same way as properties (see examples below).

  The indexing problems related to indexing of aggregation paths are discussed in Chapter 5. The presented proposals generally use standard data structures like B^+-trees or hashing schemes. The difference between the various approaches is mainly, *how* these data structures are applied to the indexing task rather than *which* data structure is chosen. Consequently, the chapter focuses on path decomposition rather than the actual search data structures. Like in [BM91] and [KM94a] the implementation of decomposed paths with the help of appropriate data structures is considered straightforward and therefore not discussed in detail.

- In contrast to the relational model where an attribute always carries a single value, properties in object databases may be *multi-valued* (or *collection-valued*). Multi-valued properties can carry a number of values at the same time, for example, the property hobbies of type Person holds a (possibly empty) set of strings. Thus, a query can contain collection comparison operators (e.g., *is-subset-of*) for multi-valued properties in addition to the usual comparison operators defined for single-valued properties.

  Related problems and possible solutions are discussed in Chapter 6. Again, the presentation focuses on the application of standard data structures, in particular, so called *signature files* to the special requirements imposed by multi-valued properties.

With respect to the query language, these problems are more or less orthogonal to each other. Each query predicate may contain terms with collection comparison operators and may involve path expressions referring to an aggregation path probably inherited from supertypes in a type hierarchy.

Example query predicates and query results corresponding to the three cases described above are the query terms:

- aPerson.Income > 20000 which refers not only to Person, but also to subtypes Student and Staff. Without a type hierarchy index, all objects belonging to the type have to be fetched from mass storage (always abstracting from all buffer cache considerations) and checked to determine the objects qualified by the query.
- aStaff.worksIn.address = "Foggy Bottom 4711" which refers to all staff people working in a department with address "Foggy Bottom 4711". Again, without access support, all Staff objects have to be fetched. For each of these objects, the related Department object has to be fetched too. Based on these Department objects, all qualifying staff people, i.e., all staff members working in a department located at the given address can be determined.
- ("Hiking", "Football", "Aussie Football") >= aPerson.hobbies which refers to all persons who have no other hobbies than hiking, football or Aussie football. Clearly, without any access structure, all objects of type Person have to be retrieved and checked to compute the query result.

1.2 APPLICATION ASPECTS

The examples in the previous section show that query processing in large object sets can and, in general, will be slow without index support. So the problem is how to choose for a given database structure and possibly also for a given query profile known in advance for an application or for an interactive user appropriate access support mechanisms which, on the one hand, are favorable with respect to the (estimated) query profile and which, on the other hand, cause only a minor (or at least acceptable) system overhead with respect to update operations.

Usually, the work distribution between OODBMS and application developer (or database administrator) in the area of indexing is as follows. In the most convenient case, a monitoring component of the OODBMS determines which data items have to be indexed and passes this kind of information to the indexing component, i.e., another subsystem of the OODBMS.

The more likely case is a bit less convenient: the database administrator has to choose the properties and relationships between types to be indexed whereas the OODBMS is responsible for creating and maintaining the access support structures. If the indexing subsystem offers different implementations for indices (e.g., different data structures forming the technical backbone of the desired index), the administrator has to choose one of these implementations.

In the worst case, the indexing functionality as offered by the system is insufficient for a particular implementation or there is no indexing functionality at all. However, if the OODBMS provides access to its internal object handles (object identifiers) which usually do not vary over the lifetime of the object, the user (in this case, the application programmer) is free to implement her own indexing functionality on top of the actual object management system.

The latter option exists since any indexing subsystem in this context (either as a part of the OODBMS or implemented on top of it) will be used to retrieve *object identifiers* instead of objects. Using such a system architecture, content based search and retrieval is separated from the persistent object store which is responsible for object to disk mapping, or in standard database terminology for *object clustering*. Such a persistent store provides object identifiers as an abstraction from the technical peculiarities of physical object handling and relocation. Consequently, if the indexing component (either of the system or of an application) is able to determine the object identifiers of all objects qualifying for a particular query request in an efficient way, the persistent object store will do the rest, namely object retrieval based on reference, usually the

object identifier. A simple overall architecture for persistent object handling and content based access is shown in Figure 1.2.

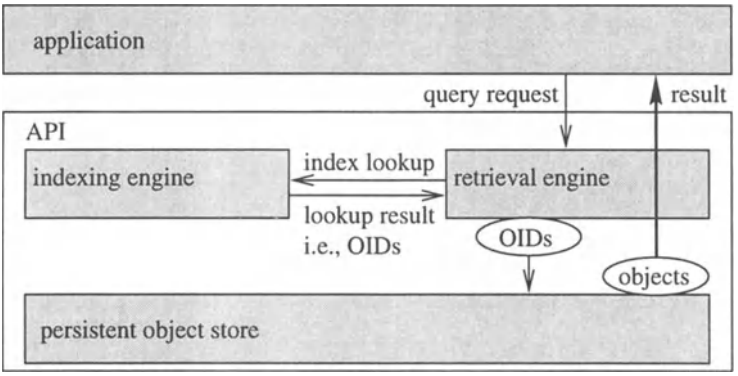


Figure 1.2 Simple architecture for persistent object systems

The book is structured as follows: in the next chapter, the necessary database model and its concrete notation is described together with an example which is used throughout the rest of the book. Chapter 3 contains a description of the tool box subsequently used to build a database index, in particular search data structures. A brief outline of search data structure principles is followed by an outline of prominent one-dimensional and multi-dimensional data structures.

As indicated above, Chapters 4, 5 and 6 elaborate on problems and solutions of type hierarchy indexing, aggregation path indexing and efficient processing of collection operators, respectively. In Chapter 7, an example for a comparative evaluation in the domain of type hierarchy indexing is outlined. Bibliographical references and an index conclude the presentation. Pointers to the bibliography at the end of the book can be found in the text as well as in small extra bibliographies at the end of Chapters 2 to 6.

2

DATABASE MODEL

In several technical domains like transaction handling, versioning and physical data management (to mention just a few), the enhanced modeling features of the object-oriented paradigm yield new technical requirements in addition to the traditional system requirements to be met by relational DBMS. As already mentioned in the introduction, also indexing is one of these domains. In this chapter, we provide a more detailed account of the relevant model constructs and of their practical implications with respect to indexing.

For this purpose, this chapter elaborates on the following issues:

- Relevant parts of an object-oriented data model are briefly described. The notation of the corresponding data description language adheres to the *Object Definition Language* or ODL as presented in the ODMG-93 proposal in the version of Release 1.2 (see [Cat96]). A formal description of either a full-fledged object-oriented model or a complete standard proposal like ODMG-93 is clearly beyond the scope of this book. Consequently, we focus on these parts of the proposal which influence the indexing task in some way.
- Based on the abovementioned data model description, some features of the data manipulation language and in particular, of the query language are discussed. Again, the presentation is ODMG-93 oriented, the *Object Query Language* or OQL is used as a notational framework for language parts relevant to the indexing task. The intention is to demonstrate for these language parts in how far the standard requirements as known from relational systems are extended.
- A final issue addressed in this chapter is the introduction of a small database schema as a running example used throughout the rest of the

T. A. Mueck et al., *Index Data Structures in Object-Oriented Databases*

book. It is taken from a university environment with students, courses, staff people and the like. As always, the actual example structure is a compromise between the necessary triviality of pure example schemes and the complexity of real life settings.

A closer look at object-oriented database literature, in the systems domain as well as in the application and modeling domain shows that almost any article or book on particular technical issues contains some kind of data model overview. The reason for this somewhat unsatisfying situation is the lack of any unified and, probably most important, *universally accepted* formal model comparable to the formal model in the realm of relational databases (see [Cod70]).

Although most of the proposed models incorporate the same core of modeling constructs, the technical details differ to some extent (detailed material on models can be found for example in [Heu92], [BDK92] and [KM94a]). The most serious problems for any book focusing on some technical issues in this domain are the obvious differences in terminology and notation. Therefore we also spend some space on a model description.

Despite the absence of *the* agreed upon formal model or standard, several researchers both, from industry and academia, tried to define the terms *object-oriented database (system)* as such. In one of the most influential publications on the principles of such systems which is also known as the *OODB-Manifesto*, see [ABD+89], the authors define a rule set for deciding whether or not a particular system should be considered an OODBMS. For this purpose they set up three categories for rules:

- **Golden Rules**: Mandatory requirements to be met by a database management system if it is called OODBMS. These requirements include traditional DBMS functionality like persistency, mass storage management, concurrency control and recovery for transactions and the delivery of an ad-hoc query interface on the one hand. On the other hand extended functionality from the object-oriented and object-relational paradigm like complex objects, object identity, abstract data types and encapsulation, extensibility (e.g., with respect to query operators), completeness, types or classes in connection with inheritance, operator overloading and dynamic binding is added.
- **Goodies**: Optional requirements which are considered valuable enhancements for any OODBMS which already fulfills the Golden Rules. Important examples for goodies are multiple inheritance, static typing and type

inferencing, means for version maintenance, long transactions (e.g., with a duration of several hours or even days) and also nested transactions.

- **Open Choices**: Basically all other technical issues not constrained by any rule described in the article. Examples are the database language paradigm (procedural, functional, etc.), the predefined data types, the technical details of type constructors and the handling of the metadata (e.g., the database structure information).

An example for an elaborated formal model in this context can be found in [Bee89] and [Bee90]. Although this kind of formalism is most appealing from a theoretical point of view, we use the less formalized but also less complex model contained in the ODMG-93 proposal as our reference model. Proposed by researchers mainly from industry, this model seems to be intuitive for software developers and other practitioners. Containing all technical details necessary for the presentation of indexing concepts it is the model of choice for this book.

2.1 OBJECT MODEL

Prior to a detailed semi-formal discussion of the object model, the following glossary contains informal descriptions of the central modeling concepts subsequently used.

- An *object* is a database entity which is uniquely identifiable by its *object identifier* (*OID*). The OID is assigned by the OODBMS upon object creation and does not not change over time.
- Each object has a set of *characteristics*. Each characteristic is either a *property* or an *operation*. The *state* of an object corresponds to a particular set of values such that each value belongs to one of its properties. On the other hand, the object *behavior* is the result of a set of operation executions. Properties are categorized into *attributes* and *relationships*. An attribute belongs to one object, whereas a relationship represents a link between an object and one or more other objects.
- All objects belonging to the same *type* share the same state space and behavior. This means that all objects of a type have the same properties and operations. Objects of a type are also called instances of this type.
- One type can *inherit* the characteristics of one or more other types (*supertypes*), may extend its heritage by additional characteristics and may

pass on its set of characteristics (inherited as well as own) to other types (*subtypes*). One of the main motivations for introducing the concept of *inheritance* is to isolate common characteristics of two or more types and to set up a common supertype for these types. One of the immediate consequences of this concept is that an object of a particular type can be viewed also as an object of any of its supertypes. The inheritance relation is usually represented by a so-called type inheritance graph which is directed and acyclic.

- The *extent* of a type is the set of its instances and the instances of all of its (direct and indirect) subtypes. Properties or sets of properties which uniquely identify objects within extents can be marked as *keys*.
- Objects can be grouped using *collections*. Collections can be sets, bags, lists, or arrays. In this sense a type extent is a special case of a collection.
- A type has an *interface* that defines its characteristics and one or more *implementations* which implement these characteristics by means of data structures and methods.

2.1.1 Types

In ODMG-93, a type defines structure and behavior of all objects (instances) of this type. Each object is instance of exactly one type, and each type has zero or more instances. A type which cannot have instances is called *abstract type* and is used to propagate characteristics to subtypes (see inheritance discussion below).

In ODL, a type definition is called *interface* and includes a type name and a list of characteristics definitions. Each type interface may have one or more *implementations*. The combination of a type's interface and one of its implementations is called a *class*. Each characteristic is either a property or an operation. Below the structure of the interface definition of type Staff (left hand side) is shown together with an appropriate class definition structure using the C++ binding.

```
interface Staff () {
  properties
  operations
};
```

```
class Staff {
  data structures (instance variables)
  methods (member functions)
};
```

o-type : $O \rightarrow (T \cup L)$
p-type : $P \rightarrow T$
p-domain : $P \rightarrow (T \cup L)$
p-atom : $P \rightarrow \{\text{true}, \text{false}\}$

Figure 2.1 Type handling functions

ODL allows the definition of an extent collecting all objects of that type. One or more properties may be declared keys similar to candidate keys in the relational model. The scope of a key is the corresponding type extent. In the following example the interface of a type Staff is defined. Modeling the employees of a university the three properties staffID, phones, and worksIn define the state space of the objects, in the sense that each object of type Staff has a unique staff identifier, zero or more phone numbers and a department to which it is assigned.

Each object of type Staff is member of the set staff, also referred to as the type extent of Staff. The attribute staffID is the only user-defined key for uniquely identifying objects within this extent.

Property worksIn is modeled as a relationship to Department. The implications of the inverse property definition Department::members are discussed below.

```
interface Staff (extent staff key staffID) {
  attribute Long staffID;
  attribute Set<Long> phones;
  relationship Department worksIn inverse Department::members;
  operations
};
```

In what follows, the relationship between types and their properties is outlined. Let T, L, P denote the set of object types, literal types, and properties of a database schema, respectively. The term *literal type* refers to immutable data types like Float or Date. At the instance level literals have a constant state and do not have an OID, they are identified by their value instead. For example, 17.4 denotes a literal of type Float. The set of all currently existing objects and literals of a database is denoted by O. A function o-type (see Figure 2.1) will be used to determine the type of an object or literal.

The graph which contains, for a given database structure, the object and literal types as nodes and the corresponding properties as edges is called property graph. The following functions (see Figure 2.1) are defined for prop-

erties: p-type gives for a property the object type which it is part of, e.g, p-type(phones) = Staff. Function p-domain defines for any property the corresponding object or literal type containing possible values for the property. For example, p-domain(phones) = Long, whereas p-domain(worksIn) = Department. In order to distinguish between single-valued and multi-valued properties a boolean function p-atom is used.

The edges of a property graph carry a property name as label. The label l is shown as an edge inscription close by the node which represents p-type(l). Distinct arrow types are used to reflect the cardinality of a property. In particular, black and white arrow heads for single-valued and multi-valued properties, respectively. For notational convenience inverse relationships are symbolized by bi-directional edges in the graphical representation of the property graph. These conventions are summed up in Definition 2.1.

Definition 2.1 (Property graph)
The directed graph $G_p(V_p, E_p)$ *called Property Graph is defined as follows:*
$V_p = T \cup L$ *and* $E_p = \{(t', label, card, t'')\} \subseteq T \times P \times \{S, M\} \times (T \cup L)$
$(t', p, \mathbf{S}, t'') \in E_p \iff$
$\quad \exists p \in P : \text{p-type}(p) = t' \wedge \text{p-domain}(p) = t'' \wedge \text{p-atom}(p)$ *and*
$(t', p, \mathbf{M}, t'') \in E_p \iff$
$\quad \exists p \in P : \text{p-type}(p) = t' \wedge \text{p-domain}(p) = t'' \wedge \neg\, \text{p-atom}(p)$

Extending our example by the interface definition of type Department, the inverse relationship to worksIn of type Staff can be noted.

```
interface Department () {
  attribute String name;
  relationship Set<Staff> members inverse Staff::worksIn;
  operations
};
```

Using the syntactical structure of the ODL relationship definition the usual one-to-one, one-to-many, and many-to-many relationship structures can be implemented. The relationship names are kept redundantly whereas the corresponding domains are not. Sets, bags, lists, and arrays are represented as multi-valued properties, there is no distinction between these constructs in the graph shown in Figure 2.2. The second part of the interface definition contains the definition of the operations of the type. As already mentioned the set of operations defines the behavior of an object of a particular type. Semantically correct operation invocation sequences yield semantically correct observable

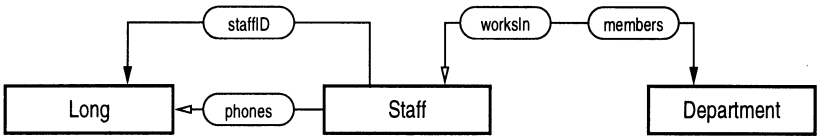


Figure 2.2 Property graph of Staff

behavior. The definition of an operation contains name, type, and mode of the formal parameters, the type of the returned value and optional exceptions to be raised during execution of the operation. Name and type of formal parameters are supplied in a C/C++ convention and preceded by a mode specification declaring each formal parameter as input, output, or input/output parameter. Exception handling via the raises mechanism are often used to catch error conditions. In the above Staff/Department example Staff could have an operation

```
Long mostLikelyPhone (in Date theDay, in Time theTime) raises
(notReachable)
```

that returns of a staff member the best phone number for a given day and time. If for the given day/time combination the employee is unreachable the exception notReachable is raised. Department could have an operation

```
Set<Course> offeredCourses (in String major) raises (invalidMajor)
Set<Course> offeredCourses (in String major, out Long unitSum) raises
(invalidMajor)
```

returning for a given major the courses offered by department members. There are two versions of the operation offeredCourses that are distinguishable by the parameters they expect. In the second version an output parameter is used to pass the total aggregated number of course units to the calling procedure. Based on the parameters supplied with an actual operation call the correct version is activated. This mechanism is called *overloading.*

2.1.2 Type inheritance

In any object-oriented system a type inherits structure and behavior from zero or more supertypes. Different semantic flavors of the inheritance mechanism can be found, for example, in [Heu92] and are not within the scope of this book. Informally, any type in a so-called inheritance hierarchy inherits all properties and operations from its supertypes and possibly extends its heritage by additional characteristics. Considering the instance level, any instance of a subtype

possesses all characteristics of any of its supertypes. As a consequence an instance of type t can be used like an instance of any of t's supertypes. With regard to terminology the graph representing the subtype/supertype relationship is called inheritance graph. In the general case the inheritance graph is a directed acyclic graph. Systems allowing more than one supertype per type provide multiple inheritance whereas systems limiting the number of supertypes per type by one provide only single inheritance. In the latter case the structure of the inheritance graph is constraint to a set of trees.

More precisely, if we consider the more general case of multiple inheritance, we denote the direct subtype/supertype relationship between any two types of the type set by sub, i.e., $t \,\mathrm{sub}\, t'$ means that t is a direct subtype of t'. The following Definition 2.2 clarifies the meaning of sub, of the derived hierarchy relationship $\leq$, and of the term type inheritance hierarchy. Informally, the inheritance relationship sub defines only direct inheritance. However, the type inheritance mechanism works in a transitive fashion since any type inherits structure and behavior from its parents, grandparents, and so on. Consequently, some kind of extended inheritance relationship $\leq$ is necessary in several occasions. $t' \leq t''$ means that t' inherits all characteristics of t'' because there is a direct or indirect inheritance relationship between these two types. This relation together with the type set is called type hierarchy.

Definition 2.2 (Type inheritance)
Let T and sub *denote a type set and the corresponding direct inheritance relationship, respectively. In what follows, the reflexive and transitive closure of* sub *will be denoted by $\leq$.*

The relation $\leq$ is defined as: $\forall t \in T : t \leq t$ (reflexivity) and $\forall t, t', t'' \in T : t \,\mathrm{sub}\, t' \wedge t' \leq t'' \Rightarrow t \leq t''$ (transitivity). The pair $(T, \leq)$ is called type inheritance hierarchy.

Returning to the ODL definition we observe that the type inheritance relation sub is explicitly stated as part of the interface definition. Extending the prior ODL example by two subtypes of Staff could produce a type interface like

```
interface Academic : Staff () {
  attribute Set<String> degrees;
  other characteristics
};
interface Admin : Staff () {
  characteristics
```

```
};
interface Department () {
  attribute String name;
  relationship Set<Staff> members inverse Staff::worksIn;
  relationship Academic head;
  operations
};
```

According to the inheritance concept Academic inherits the characteristics of Staff and augments these characteristics by more specific items like degrees. The corresponding inheritance relation is sub ={(Academic, Staff), (Admin, Staff)}.

Type hierarchies are usually visualized as directed graphs, more specifically, a single inheritance type hierarchy corresponds to a set of trees, whereas the multiple inheritance case produces directed acyclic graphs.

Definition 2.3 (Type inheritance graph)
The directed acyclic graph $G_t(V_t, E_t)$ *with* $V_t = T$, $E_t = \{(t', t'')\} \subseteq T \times T$ *and* $(t', t'') \in E_t \iff t' \,\mathrm{sub}\, t''$ *is called type inheritance graph of* $(T, \leq)$.

Figure 2.3 shows the type inheritance graph. This kind of inheritance mech-

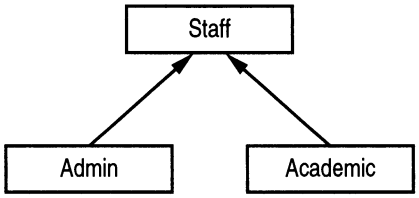


Figure 2.3 Type inheritance graph

anism implies that each instance of a particular type t can be used like an instance of any supertype of t. In other words o' can be substituted for o'' if o-type(o') $\leq$ o-type(o''). Consequently, if one would like to retrieve all Staff members fulfilling a particular predicate, the scope of the query is not only the set of all instances of type Staff, but also the sets of instances of Admin and Academic. This leads to the concept of a type's *extent*. Informally, the extent of type t is the set of of all instances of type t together with all instances of all direct and indirect subtypes of t.

Definition 2.4 (Type extent)
The extent $\mathrm{Ext} : T \to O$ *of* $t \in T$ *is defined as* $\mathrm{Ext}(t) = \{o \in O : \text{o-type}(o) \leq t\}$

Based on the above definition some conclusions can be drawn. Each instance of a particular type is an element of the corresponding extent:

$$\text{o-type}(o) = t \Rightarrow o \in \text{Ext}(t)$$

A particular object is in the extent of type t if and only if the objects type is direct or indirect subtype of t:

$$\text{o-type}(o) \leq t \iff o \in \text{Ext}(t)$$

The extent of type t' is a (possibly improper) subset of the extent of type t'' if and only if t' is a direct or indirect subtype of t'':

$$\text{Ext}(t') \subseteq \text{Ext}(t'') \iff t' \leq t''$$

The last conclusion leads to the observation that the inheritance relationship $\leq$ at the type level produces a subset relationship at the extent level. Recalling the previously defined concepts of the type hierarchy $(T, \leq)$ and the corresponding type hierarchy graph, the set of types defining a particular $\text{Ext}(t)$ corresponds exactly to the subhierarchy rooted at t.

Definition 2.5 (Subhierarchy)
Let $(T, \leq)$ denote a type hierarchy and let $t \in T$. The subhierarchy of T rooted at t is denoted as $(T_{\leq t}, \leq_t)$ with $T_{\leq t} = \{t' \in T | t' \leq t\}$ and $\leq_t = \{(t', t'') \in \leq | t'' \in T_{\leq t}\}$.

Like in case of a type inheritance hierarchy, any subhierarchy can be visualized by the subgraph of G_t which is induced by $T_{\leq t}$. Since in ODMG-93 type hierarchy queries always refer to type extents (see following section) the type subhierarchy is the basic retrieval pattern in type hierarchies.

Combining the property graph and the type inheritance graph results in the *schema graph.*

Definition 2.6 (Schema graph)
The directed graph $G_s(V_s, E_s)$ with $V_s = T \cup L$ and $E_s = E_t \cup E_p$ is called schema graph of $(T, L, P, \leq)$.

Figure 2.4 shows the schema graph for our database example. Considering a schema graph together with an arbitrary object database language usually leads to the term *path.* In graph theoretical terms a path in the schema graph

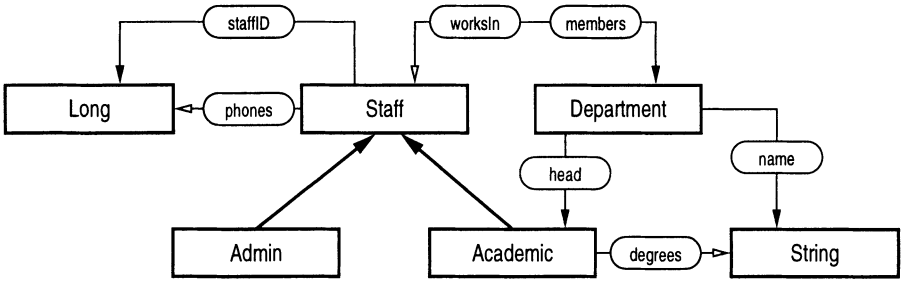


Figure 2.4 Schema graph

is a sequence $v_1, e_1, \cdots, v_n, e_n, v_{n+1}$ such that $\forall i \in [1, n+1] : v_i \in V_s$ and $\forall i \in [1, n] : e_i \in E_s \wedge (e_i = (v_i, v_{i+1}) \in E_t \vee e_i = (v_i, label, card, v_{i+1}) \in E_p)$.

For practical reasons, the notation of paths in OODBMS is shortened. Consider, for example, the path (Department, (Department, head, S, Academic), Academic, (Academic, Staff), Staff, (Staff, phones, M, Long), Long). In a first step, edges of the type inheritance graph together with their respective end nodes are eliminated. For our example path this results in (Department, (Department, head, S, Academic), Academic, (Staff, phones, M, Long), Long). Semantically, this is still a correct path, since Academic is a subtype of Staff. However, the shortened path is syntactically incorrect since it is not a path in the schema graph. To overcome this syntactical inconvenience, we derive the so-called *path graph* from the schema graph. Informally, the path graph is devoid of explicit inheritance information and contains all semantically legal path components. For this purpose inherited properties are represented by additional edges emerging from the respective subtypes. All edges stemming from the type inheritance graph are omitted.

Definition 2.7 (Path graph)
The directed graph $G_{path}(V_{path}, E_{path})$ *with* $V_{path} = T \cup L$,
$E_{path} = \{(t', label, card, t'')\} \subseteq T \times P \times \{S, M\} \times (T \cup L)$ *and*
$(t', label, card, t'') \in E_{path} \iff (t', label, card, t'') \in E_p \vee$
$\quad \exists t''' \in T : (t''', label, card, t'') \in E_p \wedge t' \leq t'''$
is called path graph.

Reconsidering our database example in the context of the path graph definition yields the situation depicted in Figure 2.5. Now, replacing the syntactically offending edge (Staff, phones, M, Long) in the shortened path (Department,

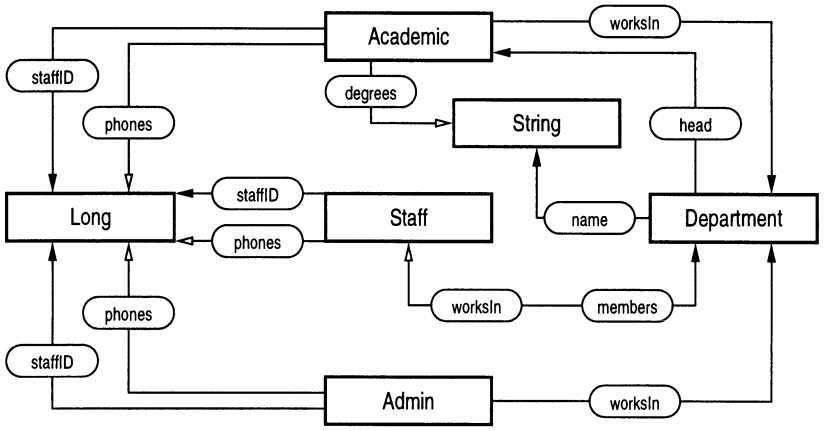


Figure 2.5 Path graph

(Department, head, S, Academic), Academic, (Staff, phones, M, Long), Long) by (Academic, phones, M, Long) yields the syntactically correct[1] path in the path graph (Department, (Department, head, S, Academic), Academic, (Academic, phones, M, Long), Long).

In a second step, all node identifiers in the path except the first one are removed, because each edge uniquely identifies its end node. Therefore, inner nodes in the path are derivable from their preceding edges. This shortening step leads to (Department, (Department, head, S, Academic), (Academic, phones, M, Long).

In a third and final step, the remaining edges are replaced by their labels. This does not produce any ambiguities because it is not possible to have identically named properties in the interface definition of any type. Consequently, a combination of type name followed by property name is unique within a path graph. This allows a significant shortening of the path notation. In our running example it results in (Department, head, phones).

2.1.3 Objects

All considerations above deal with the structural aspects of a database. A set of objects forming the corresponding information base can be represented by another directed graph called *object graph.* The object graph contains all

[1]It should be noted that in neither case there is a semantic problem with the shortened path.

objects of a particular database as nodes and all object properties (attributes as well as relationships) as edges.

For example, if property worksIn of object ω_{30} has ω_{20} (i.e., the OID of the Electrical Engineering Department) as its value, then there is an edge from node ω_{30} to node ω_{20}. On the other hand, if the property staffID of object ω_{32} has 9001 as its value an edge from ω_{32} to the literal 9001 represents that fact (recalling that a literal is identified by its value). Figure 2.7 shows a possible object graph as instance of the database schema graph of Figure 2.4. Any retrieval operation navigates through this graph. The contents of the edge set is usually constraint by structural information contained in the schema graph.

To provide a correct definition of the relationship between schema graph and object graph we have to unfold the schema graph in a similar way like the expansion of the schema graph to the path graph of the previous subsection. The difference between the resulting *unfolded schema graph* and the path graph as used above can be described as follows: in both graphs subtypes inherit *outgoing* edges from their supertypes. However, in the unfolded schema graph, subtypes also inherit *incoming* edges. Consequently, the path graph is a subgraph of the unfolded schema graph and the definition of the unfolded schema graph given below corresponds to a slightly augmented definition of the path graph.

Definition 2.8 (Unfolded schema graph)
The directed graph $G_u(V_u, E_u)$ *with* $V_u = T \cup L$,
$E_u = \{(t', label, card, t'')\} \subseteq T \times P \times \{S, M\} \times (T \cup L)$ *and*
$(t', label, card, t'') \in E_u \iff$
$\quad (t', label, card, t'') \in E_p \vee$
$\quad \exists t''' \in T : (t''', label, card, t'') \in E_p \wedge t' \leq t''' \vee$
$\quad \exists t''' \in T : (t', label, card, t''') \in E_p \wedge t'' \leq t'''$
is called unfolded schema graph.

The unfolded schema graph of our database example is shown in Figure 2.6. In this figure the graph contains edges labeled members from Department to Admin and Academic which are inherited from the initial incoming arc from Department to Staff. Based on the definition of the unfolded schema graph we are able to define legal object graphs for particular database structures. Basically, each edge $(o', label, o'')$ in the object graph must correspond to an edge $(t', label, card, t'')$ with the same label in the unfolded schema graph in such a way that o' and o'' are of type t' and t'', respectively. Furthermore, if the edge in the unfolded schema graph is marked as single-valued the property

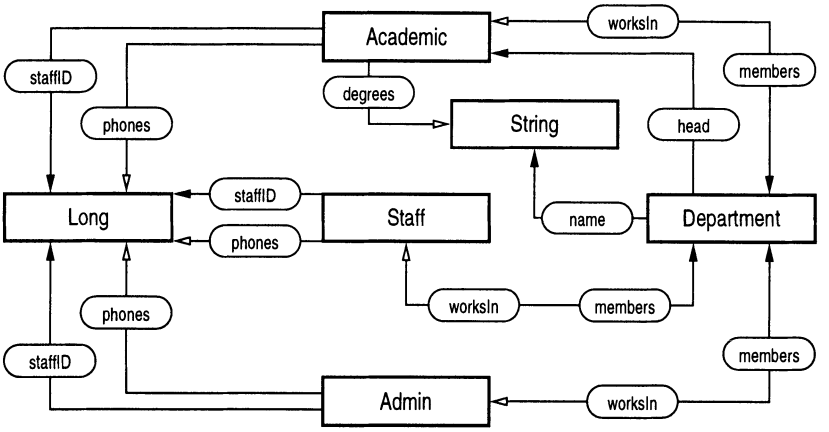


Figure 2.6 Unfolded schema graph

of o' denoted by label *label* has to have the value o''. On the contrary, if the edge in the unfolded schema graph is multi-valued o'' has to be an element of the property's value set. This is summed up in the following definition.

Definition 2.9 (Object graph)
The directed graph $G_o(V_o, E_o)$ *with* $V_o = O$,
$E_o = \{(o', label, o'')\} \subseteq O \times P \times O$ *and*
$((o', label, o'') \in E_o \iff$
$\quad (\text{o-type}(o'), label, S, \text{o-type}(o'')) \in E_u \wedge o'' = o'.label) \vee$
$\quad (\text{o-type}(o'), label, M, \text{o-type}(o'')) \in E_u \wedge o'' \in o'.label)$
is called object graph.

2.1.4 Database example

In this section we introduce a small database schema which will be used as an example throughout the rest of the book. It is not intended as a database *design* example but should be understood as a base for query examples and as a vehicle to illustrate the challenges and problems of indexing in object databases.

The database manages information about people at a university, i.e., students and university staff. All types have a type extent, which by convention has the same name as the corresponding type with a lower case initial letter. The type Person defines all properties common to both students and university staff, i.e.,

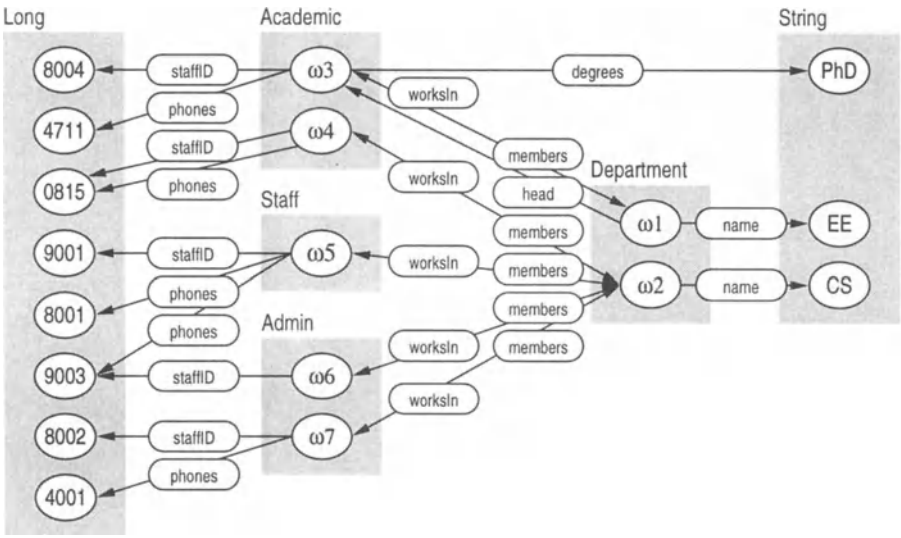


Figure 2.7 Object graph

name, income, social security number (which is the key of the extent), date of birth, address, weight, and a set of hobbies. Students are additionally identifiable by their enrollment number. They have a major and take a set of courses. Staff members are identifiable by a unique staff ID, work in a department and have a set of phone numbers. Academic staff members give courses and have a set of academic degrees. They may be either faculty members (assistant professors, associate professors, full professors) or people who give lectures at the university and work for some other employer (lecturers). Teaching assistants are staff members who at the same time are still students. They assist a number of courses, each course has zero or more assistants.

The ODL definitions specifying the structural part of this schema:

```
interface Person (extent person key ssnr) {
  attribute String name;
  attribute Float income;
  attribute Long ssnr;
  attribute Date dateOfBirth;
  attribute String address;
  attribute Float weight;
  attribute Set<String> hobbies;
```

```
};
interface Student : Person (extent student key enrollmentNr) {
  attribute Long enrollmentNr;
  attribute String major;
  relationship Set<Course> courses inverse Course::participants;
};
interface Staff : Person (extent staff key staffID) {
  attribute Long staffID;
  relationship Department worksIn inverse Department::members;
  attribute Set<Long> phones;
};
interface TeachingAssistant : Student, Staff (extent teachingAssistant) {
  relationship Set<Course> assists inverse Course::assistants;
};
interface Admin : Staff (extent admin) { };
interface Academic : Staff (extent academic) {
  attribute Set<String> degrees;
  relationship Set<Course> instructs inverse Course::instructor;
};
interface Lecturer : Academic (extent lecturer) {
  attribute String employer;
};
interface FacultyMember : Academic (extent facultyMember) { };
interface AssistantProfessor : FacultyMember (extent assistantProfessor) { };
interface AssociateProfessor : FacultyMember (extent associateProfessor) {
};
interface FullProfessor : FacultyMember (extent fullProfessor) { };
interface Department (extent department key iD) {
  attribute String name;
  attribute String address;
  attribute Integer iD;
  relationship Academic head;
  relationship Set<Staff> members inverse Staff::worksIn;
};
interface Course (extent course key (year, semester, name)) {
  attribute String name;
  attribute Short year;
  attribute Character semester;
  attribute Short credits;
  relationship Academic instructor inverse Academic::instructs;
  relationship Set<Student> participants inverse Student::courses;
};
```

2.2 QUERY LANGUAGE ISSUES

An object query language like for example OQL as part of the ODMG-93 proposal has to provide *navigational* access using the object graph as basic retrieval data structure as well as *predicate based* access to object collections.

The former access type is in the tradition of object-oriented programming languages (pointer based access) whereas the latter stems from the relational database model and its underlying logic-based theory. The pros and cons of both worlds have been extensively discussed: theoretical soundness providing valuable results for query optimization and parallelization is the dominating positive aspect of relational languages like SQL, whereas computational completeness due to a seamless integration in object-oriented programming languages is a big advantage of the object-oriented database model. Looking at the negative aspects one could consider the situation perfectly symmetrical: the well-known impedance mismatch between set-oriented relational query language and traditional, i.e., procedural application programming languages yielding quite artificial mediator concepts like cursor arrays is usually considered a major drawback of the relational database model. On the contrary, the obvious lack of theoretical foundation on the object-oriented side (at least when compared to the maturity level reached by relational theory) yields significant problems with respect to query optimization and parallelization, again compared to the outstanding results obtained during twenty years of research in relational DBMS. A natural consequence of this situation is a strong research direction both in industry and academia towards object-relational systems combining the best of both worlds (see [Sto96]).

Focusing on the theoretical baselines of various database query languages a top level categorization can be given as follows (according to [Heu92]):

- Algebra based query languages: Several query languages have been proposed which are direct extensions of the original relational (query) algebra (see, for example, [Ull88]) used as formal base line for relational systems. Examples for such languages are the query languages belonging to the ENCORE model [SZ89], to the EXTREM model [Heu92] and to the COCOON model [SLT91], respectively and a proposal by Straube and Özsu [SÖ90]. The major advantage of the language category is its theoretical soundness due to a solid mathematical background. On the contrary, the main disadvantage is (similar to the relational case) a lack of industrial products implementing such approaches.
- SQL-based languages: Another more or less natural development path has been followed by proposing SQL extensions. Examples are ONTOS Object

SQL, Iris OSQL [FAC+89], RELOOP [CDLR89, CDLR90], an approach in [Kim90] and XSQL [KKS90]. Also the ODMG-93 Object Query Language (OQL) as used throughout the book is related to SQL (see below). OPAL, the query language of the GemStone system [SM91, BMO+89, MS90] is based on the nested relation model rather than traditional SQL.

- Rule based languages: Also a number of rule based languages for the relational model have been extended to match the needs of object-oriented systems (see [Heu92] for details).

In the following we do not discuss requirements and technical aspects of object query languages as such, but rather focus on aspects relevant in the context of indexing. Concerning the actual query language we mainly rely on OQL as another part of the ODMG-93 proposal. The use of OQL as primary demonstration vehicle is more or less predetermined by the use of ODL as data definition language in the previous section. However, the following example should demonstrate that the choice of a particular query language does not make too much difference with respect to indexing. For example, a query retrieving all employees named Hurka working in the Electrical Engineering department could look like

- in OQL

```
select x from staff x
where x.name = "Hurka" and x.worksIn.name = "EE"
```

- in OPAL (staff being the set of all Staff objects)

```
staff select:
  { aStaff | aStaff.name = 'Hurka' & aStaff.worksIn.name = 'EE'}
```

2.2.1 Query language features

Like in various other object query languages also in OQL the fundamental query structure resembles at least syntactically the well-known SQL select from where statement, e.g.,

```
select s.phones from staff s
where s.staffID = 8004
```

The query retrieves all phone numbers of the staff person with staff identification 8004. In fact, one of the goals of ODMG-93 release 1.2 OQL was strict

upward compatibility to SQL, therefore several syntactic alternatives are also possible. For example, the identifier s representing some kind of object/tuple variable ranging over collection staff could have been omitted. However, a closer look at the advanced features of OQL queries reveals significant extensions to standard SQL.

A first semantic difference between SQL and OQL queries is that the latter refer to collections. This is a generalization to the relational approach in so far as collections can not be only insertion defined (like relations in the relational model) but also predicate defined in the sense that all objects rendering a particular predicate to true are member of a corresponding collection. One important example of a predicate defined collection is the extent of a particular type. As already mentioned above, the extent of a type t includes all instances of t and all instances of all its subtypes. Then the predicate defining the collection representing the extent of type t could be formulated as o-type$(o) \leq t$.

Putting the pieces together we observe that in the above query staff is not the type (denoted by Staff) but the type extent. This implies that a query seemingly containing a single type like Staff in fact refers to the subhierarchy routed at this type. It should be noted, however, that the notation xyz for the type extent of Xyz is only a naming convention, in other words, the extent of type Staff could also be named zwergNase or whatsoever. Consequently, in OQL queries against subhierarchies are more or less the default case, queries against a single type (i.e., without its subtypes) are only possible, if there is a user-maintained collection containing exactly the instances of this type. On the contrary, there are also object query languages in which queries refer to types instead of collections. In this case there is usually a special notation to differentiate between the set of instances of a type and the extent of a type.

A second enhancement of OQL is a feature allowing the construction of property name sequences of arbitrary length. These sequences are somewhat related to paths in the schema graph as defined above. In particular, the dot operator as used in SQL to denote a particular attribute of a tuple variable is used to set up so-called *path expressions.*

Consider the following simple OQL example used to retrieve for staff member 8004 the name of the department she is working in:

```
select s.worksIn.name from staff s
where s.staffID = 8004
```

A simple case of a path expression as also used in SQL is s.staffID. The path expression starts with an object/tuple variable followed by a property of the

corresponding type. However, if this property has an object type as domain, like in s.worksIn, the path expression can be extended by another property belonging to this type, and so on. In the above example s.worksIn is of type Department, so it can be extended by an arbitrary property of Department like, for example, name.

At the instance level such a path expression is instantiated by matching paths of the object graph. From any object which can be bound to the object variable representing the begin of the path expression there emerge zero or more paths: if the second entry in the path expression, i.e, the first property, is single-valued, there is at most one matching edge in the object graph. If this property is multi-valued there might be several edges corresponding to this property. The same holds for all subsequent properties. In the above example only one object can be bound to variable s (because staffID is a key in staff), further worksIn and name are both single-valued. As a result, the subgraph of the object graph corresponding to the path expression s.worksIn.name in conjunction with the selection predicate is trivial: s.staffID = 8004 qualifies ω_3, so ω_3.worksIn.name evaluates to $\omega_3.\omega_1$.EE. Since by convention the path expression evaluates to the end nodes of the corresponding subgraph of the object graph, the result is EE.

With respect to terminology, the path expression s.worksIn.name is often referred to as implicit join, because the data belonging to Staff are joined with the data belonging to Department. In the above example this is not a big deal, as there is exactly one Staff instance with staffID 8004 who works in exactly one department, which in turn has exactly one name. The query complexity increases if there are multi-valued properties along the path. For example, in order to find the names of departments that have members with phone number 9003 the path Department.members.phones has to be traversed.

In OQL paths containing multi-valued properties have to be broken up into smaller chunks such that each path chunk contains only one multi-valued property which has to be the last part of the path chunk. Consequently, the path is subdivided into d.members with d referring to instances of Department and m.phones with m referring to the elements of d.members. The actual selection predicate is defined over m.phones. Thus, the resulting query could be formulated as

```
select d.name from department d, d.members m
where 9003 in m.phones
```

In addition to implicit joins yielded by path expressions there are also explicit joins in OQL. For example, to select the names of all persons who have the

same address as a department in which they do not work, one could specify the following query:

```
select p.name from person p, department d
where p.address = d.address and p.worksIn != d
```

Although the notation of the explicit join is straightforward and done in SQL style, another term in the query significantly differs from usual SQL structures: in the inequality term p.worksIn != d the value of object variable d is compared to the value of property worksIn of the object denoted by object variable p. Both expressions are of type Department, thus denoting distinguishable objects. The equality of two such expressions holds if and only if both expressions refer to the same object. At the technical level, the respective OIDs are compared and two such expressions refer to the same object if and only if their evaluation yields identical OIDs. Relational systems could provide comparable functionality by means of tuple identifiers at the technical level. However, at the model and language interface level there is no such concept like tuple identity. Although several extensions of the relational model like, for example, RM/T contain direct predecessors of OIDs named *surrogates*, this kind of concept never made its way into a major industry product prior to recent object-relational systems.

A closely related problem is the necessary differentiation of object identity and object equality. In OQL x = y with x and y denoting object variables of the same type evaluates to true if and only if x and y refer to the same object. On the other hand, *x = *y evaluates to true, if and only if the referenced objects have the same values, even if they are not the same object [Cat96].

Collection data types like sets, bags, lists, and arrays are directly supported in OQL by appropriate collection operations. In the following example the names of all students are retrieved who participate in all courses of a particular professor, say Hurka.

```
select s.name from student s, facultyMember p
where p.name = "Hurka" and s.courses >= p.instructs
```

For bags and sets OQL defines the usual operators like union (union), intersection (intersect), and difference (except) and the comparison operators subset (<=), proper subset (<), superset (>=), and proper superset (>) as well as membership test operator (in) already used in a previous example.

OQL also provides means to make use of derived properties which can be implemented by operations. An operation with an appropriate return type can be

used like any property of that type. For example in a path expression any element can be either a stored data item or an operation. If the operation does not have parameters the two alternatives are even syntactically indistinguishable. In the query below retrieving the names of all professors with students younger than 18 the element age in the path expression p.instructs.participants.age is either a stored data item (which for obvious reasons would not be a reasonable solution) or a derived property implemented by an operation which accesses the property dateOfBirth.

```
select p.name from fullProfessor p, p.instructs c, c.participants s
where s.age < 18
```

Operations that expect parameters can be called the same way, the actual parameters are supplied in parentheses. The following query retrieves the names of all students that take all courses the computer science department offers for their respective major.

```
select s.name from student s, department d
where d.name = "CS" and s.courses >= d.offeredCourses(s.major)
```

2.3 BIBLIOGRAPHY

The *OODB Manifesto* can be found in [ABD+89]. Works on object-oriented database models and query languages include [Bee89, Bee90], [SZ89], [Heu92], [SLT91], [SÖ90], [FAC+89], [CDLR89, CDLR90], [KKS90], and [SM91, MS90].

A lot of material on OODB in general can be found, for example, in [KM94a], [BM91], [Kim90], [KL89], [Kim95], [BDK92], and [Heu92].

There are also several contributions from standardization bodies or related groups. The probably most important contribution is the SQL3 proposal by the ANSI X3H2 committee (see [Mel94a, Mel94b]) representing an extension of the current SQL standard. The current version of the proposal by the Object Database Management Group (ODMG) is Release 1.2 (see [Cat96]) thus superseding [Cat94a, BF94] and Version 1.1 in [Cat94b]. A comparison between SQL3 and an elder version of ODMG-93 can be found in [MM94]. Additional remarks on the pre-1.2 proposals can be found in [Kim94].

3

DATA STRUCTURES AND INDEXING

3.1 BASICS

In this introductory section an informal outline of the major problems, solution concepts and evaluation criteria is given. Additionally, a number of terminological issues are discussed. The subsequent sections of this chapter contain refinement material on the main points addressed in this overview section.

3.1.1 Problem statement

A closer look at object-oriented concepts like object types, instance variables, aggregation paths and the like shows a general need for efficient access to large sets containing persistent[1] records of a fixed format. In the context of our previous example, an OODBMS has to maintain data sets with object specific attributes (i.e., instance variables) like for example name and income for object type Person as well as record sets used to support the access to aggregation paths like Staff.worksIn.name. Any mechanism, or in other words, *index* providing fast access to such large and persistent data sets is based on one or more search data structures.

It should be recalled that the purpose of search data structures in this context is the fast retrieval of object *identifiers* rather than objects. For example, an index search returns the identifiers of all objects of type Person with an income greater or equal 100000 or with a home address in Vienna, Virginia, but usually *not the objects*. Using such an indexing framework, the content based search and retrieval task can be decoupled from physical object handling. Object

[1]The term *persistent* can be read as *to be kept on mass storage* in this context.

T. A. Mueck et al., *Index Data Structures in Object-Oriented Databases*

identifiers are used as handles for a low-level object storage engine which is responsible for the translation of these identifiers to physical addresses.

Considering a *persistent data set* in this particular application framework, such a data set contains one or more attributes holding object identifiers and zero or more attributes holding atomic values like integers or strings. Recalling the previous examples and assuming a special attribute domain Oid used to denote the set of all object identfiers currently in use, a data set RS1(p:Oid, name:String, income:Long) could be used for fast access to the identifiers of objects with a particular name or income like aPerson.name = "Kwapil" or aPerson.income < 1000, whereas another record set RS2(aStaff:Oid, aDepartment:Oid) could be helpful in answering queries about the relationship between Staff and Department thus containing path expressions like aStaffMember.worksIn.address = "Falls Church 11".

Summing up, the problem is to design and implement data structures which provide fast associative access to large persistent object sets.

3.1.2 Terminology

Like in any other computer science discipline, also in the field of search data structures and index organization some specific terminology has to be considered. In what follows, we provide a short glossary containing informal descriptions of some of the important terms in this context, both, from data structure technology, i.e., the tool box, and from indexing, i.e., the application of the tool box, for our purpose.

As usual, *domain* refers to a standard predefined data type as known from any commercial DBMS, e.g., Integer or String. Stored data items are described by an attribute specification consisting of an *attribute name* and a corresponding *attribute domain*, e.g., (name, String). A *record specification* contains a record type name and a list of attribute specifications, for example, (Person, (name, String), (ssnr, Long), (income Long)).

A *record* of a particular type is a list of values substitued for the list of attribute specifications, e.g.[2], (Mayer, 4711, 32000), or in other words, one instance of the underlying record specification. A *data set* is a number of records which are all instances of the same record specification. In this sense, {(Mayer, 4711,

[2]If the context is not ambigous, all strings are written without quotation marks in the sequel.

32000), (Vrbala, 5001, 35000)} is a data set[3] for the record specification given above.

A *query request* to be processed for a particular data set is a list of interval specifications for the attributes of the underlying record specification. The *execution* of a query request yields a subset of the data set such that each record in the subset has values within the interval specifications of the corresponding query request.

In this context, each interval specification may be *proper*, i.e, lower boundary less than the upper boundary, or *improper*, i.e., lower boundary equal to upper boundary. Interval specifications containing the minimum and maximum values of the corresponding domain as lower and upper boundary, respectively, can be used to qualify all stored values of this domain. Such interval specifications are called *open* in the sequel. There are several terms used in connection with proper, improper and/or open query interval specifications.

For example, a query request containing only proper interval specifications is called a (multi-dimensional) *range query* request. A query request containing only improper interval specifications is called a (multi-dimensional) *exact match query* request. A frequently appearing special case of all other (mostly unnamed) mixtures, represented by a set of improper interval specifications together with some open intervall specifications, is called *partial match* query request. A search data structure used to store records of a particular type is called *one-dimensional* structure, if it supports the processing of proper or improper interval specifications for *one fixed attribute* of the underlying record specification, say for attribute ssnr in record specification Person. In this example, a one-dimensional search structure for this attribute would store particular ssnr values to provide fast access in case of interval specifications referring to this attribute. Technical details are to be found in the next two sections.

On the contrary, if a search data structure can be used for the efficient processing of several interval specifications at once, in other words, if it supports several *searchable attributes* in a record set, again fixed for the life time of the data structure, it is said to be a *multi-dimensional* search data structure.

The special case of a search data structure defined for a fixed *sequence* of attributes (sometimes called *compound* structure) is, from a technical point of view, a one-dimensional structure for a hypothetical derived attribute in such a way that the values of this attribute result from the value concatenation of the

[3]The term *set* implies that there are no two records with exactly the same values in all attributes.

Data set
(Mayer, 4711, 32000), (Vrbala, 5001, 35000)

Query requests		
type	*specification* Person.name Person.ssnr Person.income	*result*
range query 1	[Anton, Zeppelin] [3000, 5000] [25000, 40000]	(Mayer, 4711, 32000)
range query 2	[min, max] [min, max] [10000, 40000]	(Mayer, 4711, 32000) (Vrbala, 5001, 35000)
exact match 1	[Mayer, Mayer] [4711, 4711] [40000, 40000]	$\emptyset$
exact match 2	[Mayer, Mayer] [4711, 4711] [32000, 32000]	(Mayer, 4711, 32000)
partial match 1	[Mayer, Mayer] [4711, 4711] [min, max]	(Mayer, 4711, 32000)
partial match 2	[min, max] [0815, 0815] [min, max]	$\emptyset$

Figure 3.1 Example data set and query requests

participating attributes at the record level. For example, a search data structure on name||ssnr for record specification (Person, (name, String), (ssnr, Long), (income Long)) and data set (Mayer, 4711, 32000), (Vrbala, 5001, 35000) basically is a data structure containing the data values Mayer4711 and Vrbala5001 and resembles therefore a one-dimensional structure on a derived attribute.

Search data structures represent the tool box which can be used to construct an access mechanism. In particular, an index is the implementation of a physical access path to a persistent data set by means of one or more search data structures. The relationship between a data set, an index and the underlying data structures is shown in Figure 3.2. On the application level, it is often

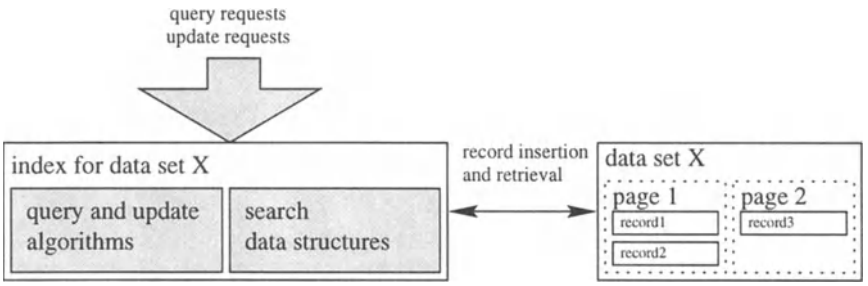


Figure 3.2 Access path implementation using search data structures

distinguished between *primary* and *secondary* indices. In the former case, any query execution based on an exact match query request returns at most one qualifying record, whereas in the latter case, the number of qualifying records is unbound. The term *primary* index is somewhat related to the term *primary key* at the database modeling level. However, more intuitive alternatives for *primary* and *secondary* in this context are *unique* and *non-unique*, respectively.

Another application oriented distinction (mostly in the one-dimensional case) is whether or not the value sequence during the record insertion process is monotonically increasing (or decreasing) or perturbed. In this case, e.g., indexing of a time stamp attribute or of a monotonically increasing student enrollment number, the index is called *sequential key* index, *non-sequential key* index in all other cases.

With respect to the support of different query types, the above mentioned distinction with respect to dimensionality can be found to some extent also on the structure application (indexing) level. A single-key (composite-key) index providing a physical access path to one fixed attribute of the data set is

implemented by means of a one-dimensional data structure. A multi-key index supporting fast access to several attributes can be constructed either by one multi-dimensional search data structure or by a combination of several one-dimensional structures. Theoretically, also a mixture is possible in the latter case.

Any data set is stored in one or more storage chunks (often called *pages*) of identical size. Usually, such a storage chunk is one disk block or a set of consecutive disk blocks. Each chunk has its unique identifier called *address* and is able to hold a number of records depending on chunk size and record size. Also the supporting data structures need a certain amount of persistent storage. Looking at the overall set of storage chunks used for record storage and indexing purposes, the former are called the *value structure*, whereas the latter are called the *access structure*. The idea is to distinguish between space consumed by the actual data on the one hand and by the supporting data structures on the other hand.

Upon record insertion, the place for (i.e., the address of) the new record can be determined either randomly (for example, after searching for a vacant slot in any storage chunk) or by one of the supporting data structures in the index.

If there is one of the supporting data structures used to determine the address of a new record, this data structure is called *placing* or *clustering*. Often, there is no distinction made between search data structure and resulting index, in these cases, the index itself is said to be placing or clustering. If one supporting data structure is placing, all others, if there are additional search data structures in the index are necessarily *non-placing* or *non-clustering*. In that case, the data structure does not store the actual record, but only indirection records, or, in other words, the values of the indexed attribute(s) together with references to the records in the data set. Figure 3.3 and 3.4 show the principle of placing and non-placing data structures.

In the example shown in the figures, the data set Person with attributes name, ssnr and income can be accessed with the help of two search data structures. DS1 is assumed to be one-dimensional, defined for ssnr and placing. DS2 is assumed also one-dimensional, defined for income and non-placing. The above example shows that in a technical sense all of the data structures in an index are placing. In fact they have to, otherwise they would not be able to find anything. The only difference concerns the kind of record to be placed: in case of a placing data structure in the above sense, the records of the data set are placed in one of the storage chunks, whereas in case of a non-placing data structure, indirection records containing one or more attribute values and a reference to the storage chunk of the record are placed.

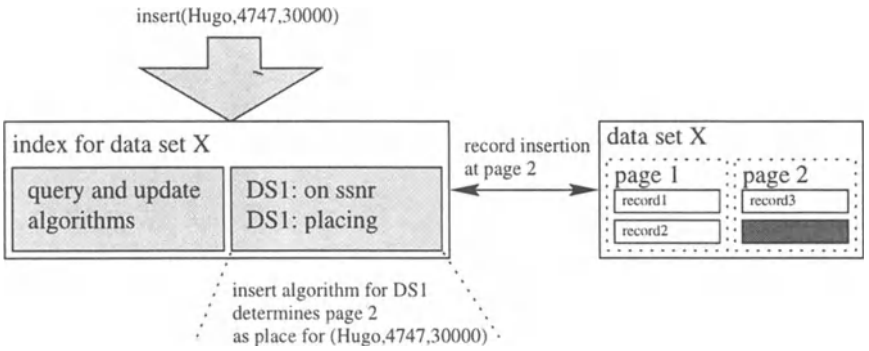


Figure 3.3 Placing data structure in an index

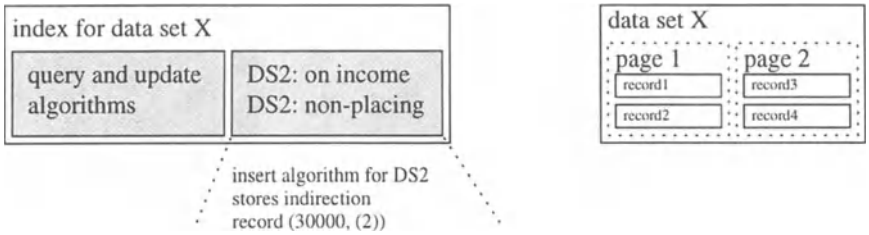


Figure 3.4 Non-placing data structure in an index

A final point closely related to the "placing" versus "non-placing" issue deals with the number of references to storage chunks maintained per search structure in an index. A non-placing data structure in the above sense has to store *one reference per stored record* by means of an indirection record, since the actual place of a new record is not controlled by the data structure. Instead, the structure has to accept and store whatever address is determined by something else (typically, by a placing data structure). Such a data structure containing one reference per stored record is called a *dense* structure. A placing data structure determines algorithmically the record's storage address, so it has to contain only the addresses of all storage chunks currently in use. The insert algorithm (and subsequently the retrieval algorithm) chooses the appropriate address based on the relevant attribute values. Consequently, such a structure containing only *one reference per storage chunk* currently in use is called a *sparse* structure. For the same data set and the same attribute size (e.g., 4 byte for an integer attribute), a sparse structure will be considerably smaller than a dense one, since each reference will at least consume 4 additional bytes.

3.1.3 Informal outline of the basic approaches

So far, an index and its incorporated data structures have been viewed as a black box which is able to store and retrieve records. At this point, we will take a first (again absolutely informal) look inside this black box.

The only prerequisite for this look is some knowledge about the meaning of the term storage chunk. Although such chunks could reside on arbitrary storage media, we assume in the sequel that at the technical level any OODBMS has to keep record sets on *mass storage devices*, usually magnetic disks, which provide persistent and high-capacity storage. Keeping in mind the concept of block oriented transfer to and from such devices, a large persistent data set is stored in disk blocks. Consequently, the term *storage chunk* is to be read as a *constant number of consecutive disk blocks seen as an atomic transfer volume* by the indexing subsystem. This roughly corresponds to the term *page* as used in most text books on database management systems.

Sequential organization

Even if no search data structure at all is maintained for a data set, at least the information which storage chunks belong to this particular data set has to be stored. The result is the well-known sequential storage organization: new

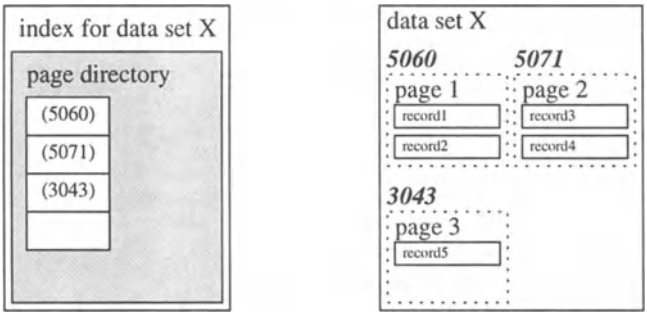


Figure 3.5 Sequential organization

records are placed sequentially in the most recently allocated storage chunk. If this chunk is filled up, a new chunk is allocated and the address of this chunk is placed in the directory. This page directory can be considered a rather simplistic search data structure where the only data ordering criterion is the record insertion sequence.

The result shown in Figure 3.5 (page x denoted by x and references to page x denoted by (x), respectively) resembles the concept of a "sequential file" with an operating system maintained storage chunk (page, block, whatsoever) directory[4]. The same situation is shown in Figure 3.6 using the more familiar graphical *pointer notation* instead of page numbers. In what follows the graphical notation is always used to avoid confusion between integers representing attribute values and integers representing page numbers. Calling this a *sequen-*

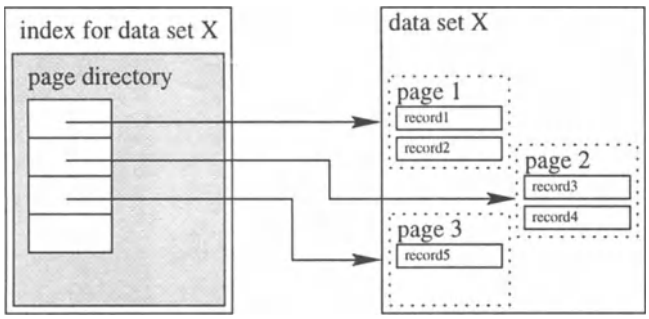


Figure 3.6 Sequential organization (alternative notation)

tial organization of the data set, the obvious simplicity of this approach (which is positive with respect to the record insertion algorithm outlined above) is often outweighed by a serious drawback, namely the usually unacceptable query resource consumption in case of large record sets.

The reason for this drawback is that in almost all database related scenarios, the intention is not to fetch records which are at a particular place in the initial insertion sequence (which is usually of little interest for the database application), but to fetch records fulfilling one or more search criteria given in query specifications as outlined above.

Assuming a sequential organization, during query execution all storage chunks belonging to the queried data set have to be physically retrieved by a mass storage controller, transferred to a main memory location and subsequently processed, i.e., searched for qualifying records. In absence of any additional data structure functionality, the operating system's mass storage subsystem is forced to handle probably gigabytes of data when passing the contents of all disk blocks to the main memory subsystem. In the extreme but practically not unlikely case that none of these records qualifies for the query request, the

[4]Interestingly, in UNIX systems the corresponding system data structure is usually called "index node" or just *inode* for short

complete effort is actually useless. Even worse, also a considerable amount of CPU time is spent just to process and subsequently discard all the unwanted data. The turn-around times of all other tasks will also increase, since the retrieving process uses hardware resources which represent the most important bottlenecks of conventional von Neumann architectures, i.e., disk I/O channels and the system bus.

Following from this, the restriction of the query execution process to the smallest possible number of data blocks is a crucial performance issue for any state-of-the-art DBMS. Technically speaking, a restriction of the retrieval process to a smaller number of data blocks is achieved by using more elaborated search structures than storage chunk directories. To put it in a nutshell: a mass storage oriented search structure is a mechanism used to exclude storage chunks (and therefore disk blocks) from being fetched.

From a different point of view, a search structure allows to locate all disk blocks which *possibly* contain records qualifying for a particular query request. This is already achieved by the simplistic directory outlined above: it contains *all* chunk addresses and therefore also all addresses possibly containing qualifying records. However, ruling out some chunks, or, in other words, knowing about some chunks that they *cannot contain any qualifying record* requires additional effort with respect to search structure technology (see following discussion). An important point to notice is that ruling out some chunks is the best thing we can hope for, because the search mechanism can neither guarantee that the selected chunks contain qualifying records (maybe there are actually no qualifying records stored at all), nor that these chunks do not contain records not qualifying for the query. The only possible guarantee is that there are no chunks not selected by the search mechanism which contain qualifying records. In this sense, a correct search structure implementation has to select the minimal set of chunks containing all stored records qualifying for a particular query.

Although this is surely not achieved by a sequential organization, the reverse side of the coin should not be ignored: maintaining a search data structure to accomplish the above goal is also a resource consuming task, so one has to check whether or not using a search data structure is a viable alternative to a simplistic sequential organization. The major factor in this context is of course the size of the underlying data set: for small data sets, the advantage of an elaborated search data structure could easily be outweighed by its maintenance cost. On the contrary, using a sequential organization for a large persistent data set containing megabytes or even gigabytes of data yields unacceptable retrieval performance.

If one has decided to use a more elaborated data structure than the chunk directory of the sequential organization, there are only two principled approaches: implementations based on the storage of interval information (*subspace mapping* approaches) and implementations based on direct value transformation (*point mapping* or *hashing* approaches).

In what follows, these two approaches are described. In both cases, the overall data space, i.e., the Cartesian product of the attribute domains has to be mapped to the linear address space provided by the mass storage subsystem. The difference between the two approaches is simple to understand: in the first case, the data space is partitioned into smaller subspaces which are in turn mapped to the mass storage address space, whereas in the latter case, the points in the data space are mapped directly, i.e., without any intermediate data space partitioning.

Subspace mapping

Subspace mapping approaches are based on a decomposition of the overall data space into a set of subspaces. The decomposition information is contained in one part of the search data structure (boundaries). In the second part, the subspace mapping is stored thus yielding for each subspace the corresponding storage chunk which contains all stored records which are geometrically enclosed by the subspace.

There are a number of different flavors concerning the partitioning strategy. The most important design alternatives are:

- overlapping subspaces versus non-overlapping subspaces and
- partitioning based on the actual data values versus partitioning based on subspace structure (thus neglecting the actual data)

The basic idea is shown in Figure 3.7. In this figure, non-overlapping subspaces of a 2-dimensional data space are used. For each new record, in a first step the smallest enclosing subspace is determined. In a second step, the mass storage address of the corresponding storage chunk is retrieved. Thirdly, the storage chunk is fetched from disk and the record is inserted. If an overflow in the storage chunk happens (not enough space for the additional record), the respective subspace is split into two smaller subspaces, another storage chunk is allocated for the data set and the records in the overfull chunk are distributed according to their geometrical position.

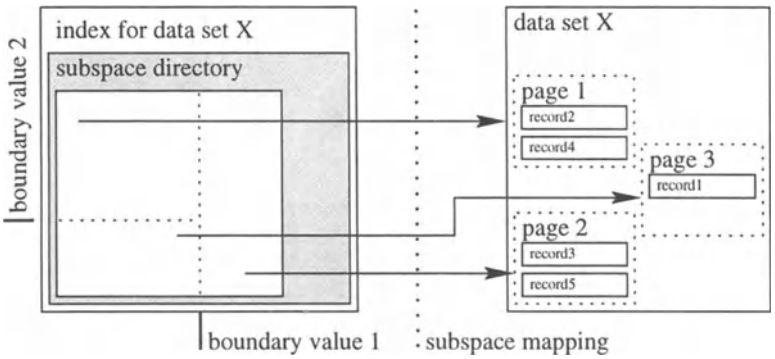


Figure 3.7 Initial setting

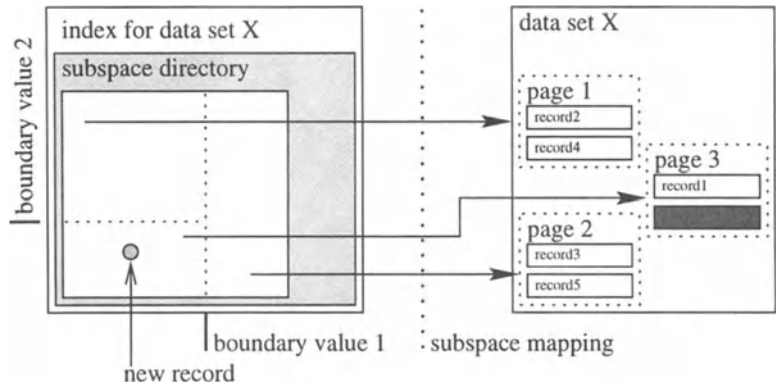


Figure 3.8 Record insertion

During a query execution, all subspaces which possibly contain qualifying data have to be determined (first step). Subsequently, the corresponding chunk addresses have to be retrieved from the directory (second step) and finally the chunks themselves are fetched (third step).

The positive aspect of this kind of data space maintenance schema is its direct support for orthogonal range queries as shown above (if the query area is covered by only a few subspaces, the number of chunks to be fetched is also small). The problem is the overhead (storage space and insertion algorithm) caused by the subspace partitioning.

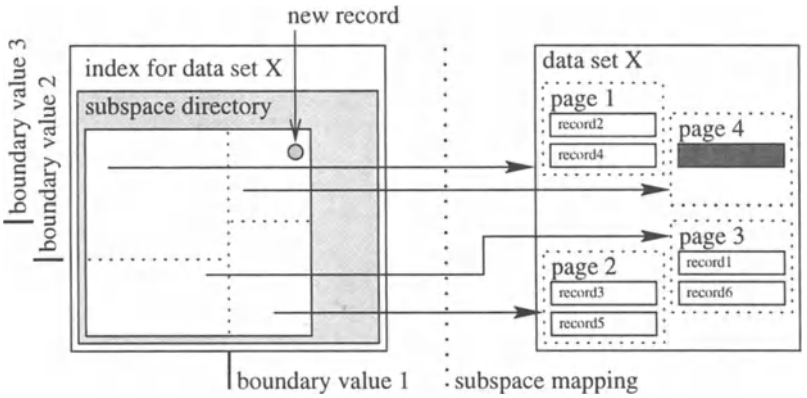


Figure 3.9 Storage chunk overflow and subspace split

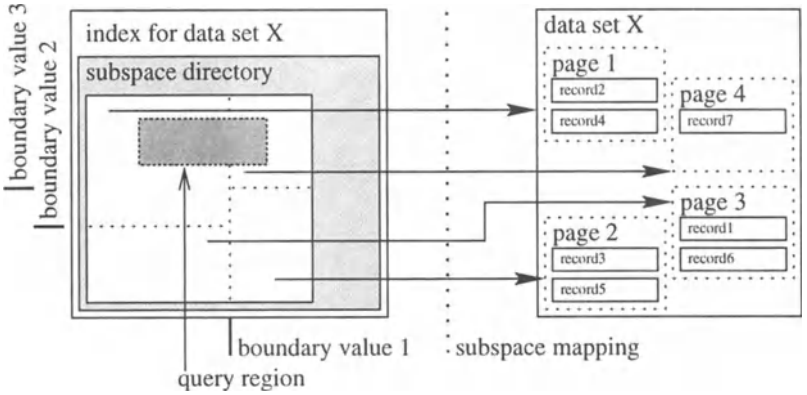


Figure 3.10 Query

Point mapping

Point mapping schemes use a direct mapping of data space elements (points) to storage chunks. The mapping is not bijective, since a large number of data points (possible records) have to be mapped to a comparatively small number of storage chunks. The mapping itself usually consists of two steps: in a first step (often called *hashing*), one or more data values of a record (to be inserted or retrieved) are used to determine the *record signature*, this signature is used in step two to determine the address of the corresponding storage chunk.

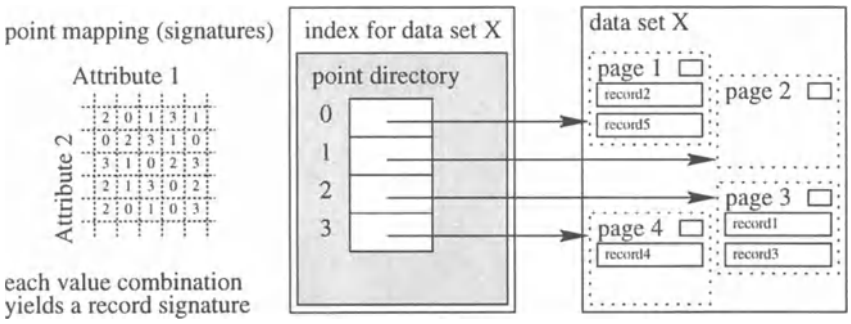


Figure 3.11 Initial setting

Inserting a new record includes the following steps: In a first step, the record signature is computed using a simple hash function. The result of the hash function is used as pointer into the chunk address directory in a second step, e.g., if the record signature is x, the $(x+1)$-th entry in the directory is selected. In the third step, the storage chunk is retrieved and the record is inserted. Again, there can be an overflow in the target chunk which triggers some kind of overflow handling procedure (for example, the organization of linked lists for overflow handling).

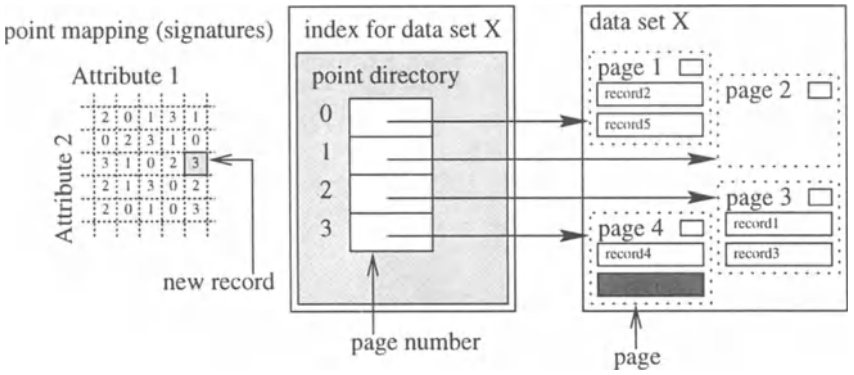


Figure 3.12 Record insertion

Executing a query request involves the following steps: at first, for each point in the query area, the corresponding record signature has to be computed; secondly, the chunk address found in the directory for the resulting signature has to be stored in some kind of *address hit set* containing the addresses retrieved

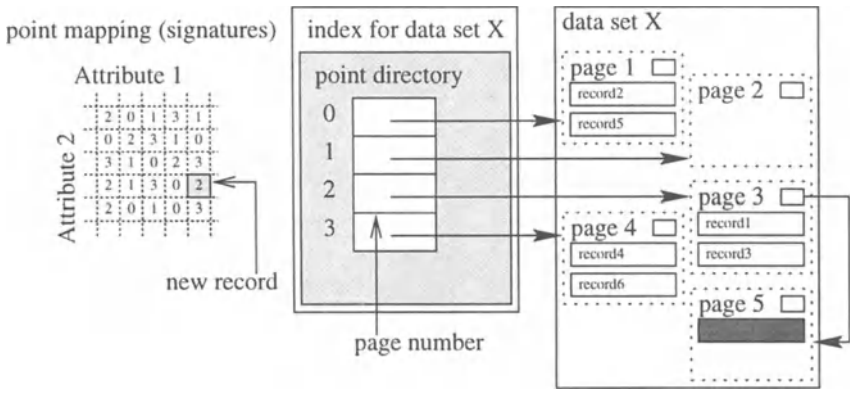


Figure 3.13 Storage chunk overflow handling

so far from the directory; if the two steps above have been done for all points in the query area, the actual storage chunks have to be fetched and processed.

Like in case of subspace mapping, this kind of query processing schema again allows immediate conclusions with respect to the pros and cons of point mapping approaches, however, the resulting situation is somewhat antagonistic to subspace mapping. Orthogonal range query requests with large query areas cause significant problems: not only that the signatures of a large number of data points (possible records) have to be computed, an even worse problem is the potentially large number of chunk addresses in the resulting address hit set (recalling that the record signatures intentionally produce a random distribution over the directory entries). In the worst case, all chunks have to be fetched and the previous signature computation task has been pure overhead. On the contrary, small query areas and exact match queries can be processed in an efficient manner avoiding the overhead caused by the maintenance of partitioning information in subspace mapping schemes.

Application example

At the end of this subsection, a simple example specification is used to compare the basics of sequential organizations, subspace mapping approaches and point mapping approaches. Additionally, a possible combination of a subspace mapping structure together with a point mapping structure is presented. For the same data set, the former is used in a placing way, whereas the latter in a non-placing way. The record specification is the same as above containing

name, ssnr and income as attributes. The example data set is given below in order of insertion:

name	ssnr	income
Mayer	4711	32000
Vrbala	5001	35000
Hugo	4747	30000
Hans	6101	32020
Otto	6102	32020
Kwapil	5100	5025
Poldi	5021	1050
Karwotni	5200	42055
Hias	5403	23050

In the following, for each design alternative a snapshot with the nine records given above is shown. For this snapshot, one exact match and one a range query are executed. After the query examples, the structures are updated through the insertion of additional records.

#	*type*	*specification*	*result*
1	exact match	[Mayer, Mayer] [4711, 4711] [32000, 32000]	(Mayer, 4711, 32000)
2	range	[Anton, Zeppelin] [3000, 5000] [25000, 40000]	(Mayer, 4711, 32000) (Hugo, 4747, 30000)
3	insert	(Anna, 5701, 5075)	n.a.
4	insert	(Berta, 6712, 45000)	n.a.

Alternative 1: sequential organization

Figure 3.14 shows a snapshot of a sequential organization after the insertion of the nine records specified above, also in the sequence depicted above.

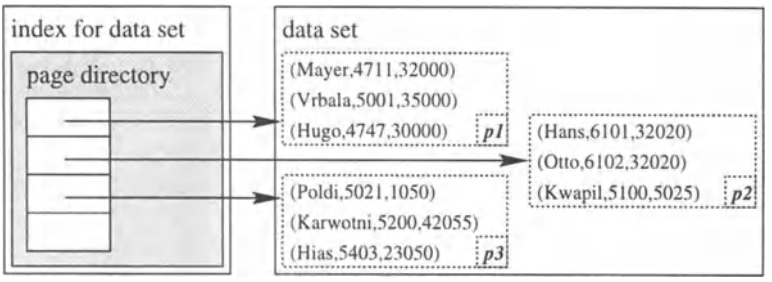


Figure 3.14 Sequential organization, 9 records inserted.

Query execution:

request	*lookup result*	*qualifying records*	*read operations*
1	5060	(Mayer, 4711, 32000)	
	5071	none	
	3043	none	3
2	5060	(Mayer, 4711, 32000)	
		(Hugo, 4747, 30000)	
	5071	none	
	3043	none	3

Record insertion: Ignoring the possibility of space reuse in case of record deletion (e.g., maintaining some kind of *delete flag* to mark slots which have been occupied by meanwhile deleted records), any insert request will cause a fetch of the most recently allocated storage chunk. In the above example, the first insert request hits a chunk without a vacant slot, so another chunk will be allocated, e.g., chunk *p4* as depicted in Figure 3.15. The second insert request can be handled without further allocation. The final data set configuration is shown in Figure 3.15.

Alternative 2: subspace mapping, one-dimensional, attribute income

Figure 3.16 shows the situation after the insertion of the nine records in a subspace mapping structure defined for attribute income.

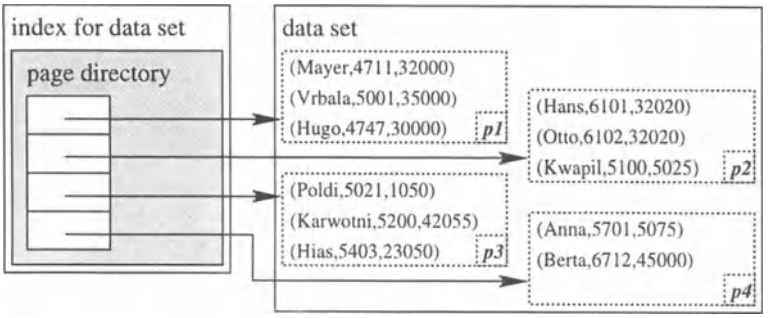


Figure 3.15 Storage chunk allocation

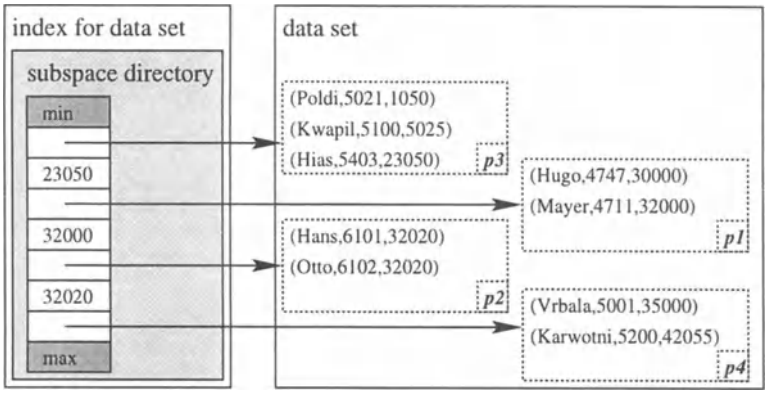


Figure 3.16 Subspace mapping, 9 records inserted.

Query execution:

request	*lookup result*	*qualifying records*	*read operations*
1	]23050,32000] → *p1*	(Mayer, 4711, 32000)	1
2	]23050,32000] → *p1*	(Mayer, 4711, 32000) (Hugo, 4747, 30000)	
	]32000,32020] → *p2*	none	
	]32020,max] → *p4*	none	3

Record insertion: The directory lookup for the record (Anna, 5701, 5075) to be inserted yields subspace]min, 23050] for data value 5075. Consequently,

chunk *p3* is fetched from disk. The insertion of the new record causes an overflow and therefore a split of the storage chunk. In the context of this split, a new boundary value has to be chosen, for example, 5025. After allocating a new storage chunk for the value structure, say *p5*, the records in the overfull chunk can be redistributed and the new boundary value can be inserted into the access structure (for the moment, the access structure is assumed to have infinite capacity). Figure 3.17 shows the new storage chunk in the value structure and the new boundary value in the access structure. The insertion of record (Berta, 6712, 45000) causes a lookup for subspace]32020,max] and a subsequent read operation for chunk *p4*. This chunk has vacant space, so the record is inserted. The final configuration of the access structure and of the value structure is shown in Figure 3.17.

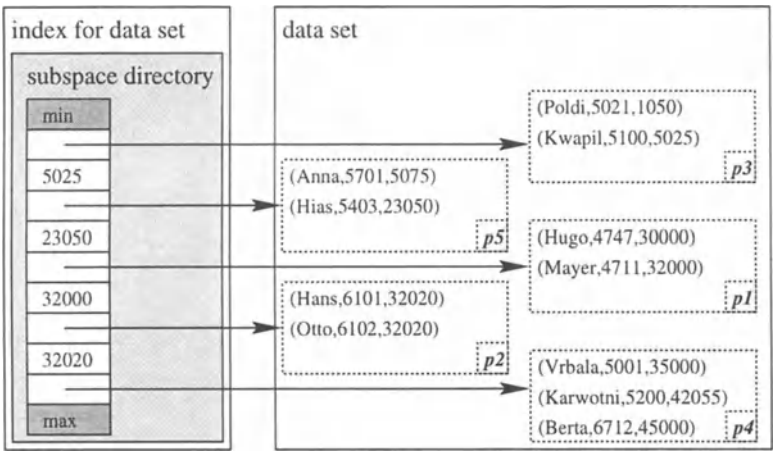


Figure 3.17 Record insertion and storage chunk allocation

Alternative 3: point mapping, one-dimensional, attribute ssnr

Figure 3.18 shows a point mapping structure using (ssnr mod 4) as the record signature, i.e., ssnr is the searchable attribute. The access structure has a similar structure like in a sequential organization, however, the search and update semantics are different (see point mapping discussion).

Query execution: The range query request is processed as a sequential scan through the point directory, i.e., without using the record signatures. Otherwise all signatures from 3000 to 5000 would be computed and 2001 directory lookup operations would end up with all (i.e., 4) chunk addresses relevant for the query.

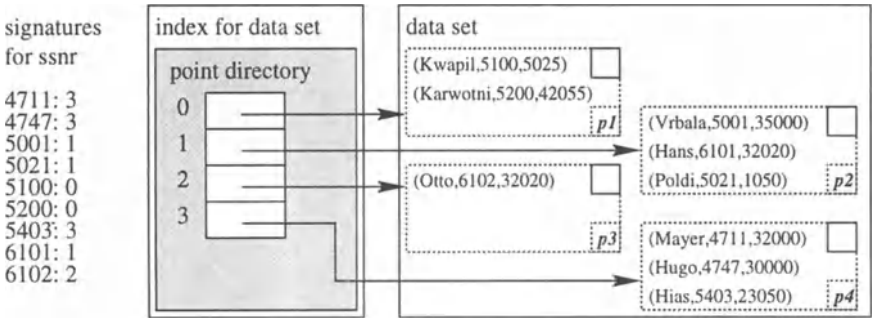


Figure 3.18 Point mapping, 9 records inserted.

request	*lookup result*	*qualifying records*	*read operations*
1	$(4711 \bmod 4 = 3) \rightarrow p4$	(Mayer, 4711, 32000)	1
2	*p1*	none	
	p2	none	
	p3	none	
	p4	(Mayer, 4711, 32000)	
		(Hugo, 4747, 30000)	4

Record insertion: The record signature for the first record is $(5701 \bmod 4) = 1$. Consequently, the address of *p2* is retrieved from the directory and the corresponding chunk of the value structure is fetched from disk. Since there is an overflow at *p2*, an overflow chunk is allocated and inserted in the overflow structure for this address. In this example, a simple linked list organization is chosen for overflow handling (see 3.19). The second insertion request yields a signature of $(6712 \bmod 4) = 0$ and therefore the address of *p1* as insertion place. After reading the chunk, a vacant slot is found and the record is inserted. The situation after both insert operations is shown in Figure 3.19.

Alternative 4: subspace mapping for income placing, point mapping for ssnr non-placing

In this case the access structure contains two data structures. A subspace mapping structure placing the records of the data set in the storage chunks of the value structure and a point mapping structure with indirection records containing chunk addresses together with the data values of ssnr. The configuration is shown in Figure 3.20. In indexing terminology, the former data structure is

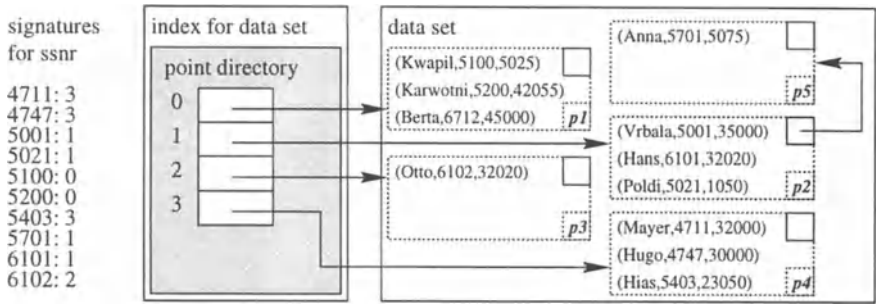


Figure 3.19 Record insertion and chunk allocation

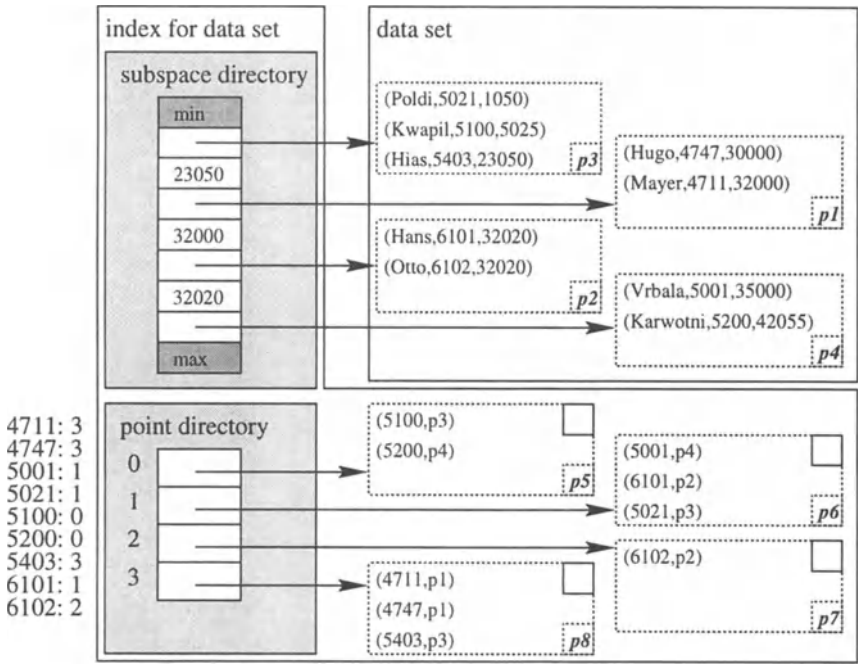


Figure 3.20 Combined data structures, income placing

sparse (one address entry per chunk), whereas the latter is dense (one address entry per record). Although any of the two data structures could be based on subspace mapping or point mapping, the choice in the above example is quite reasonable. Range query specifications for attribute income are likely, thus a subspace mapping structure has been chosen. On the contrary, range query

specifications for attribute ssnr are rather meaningless (considering ssnr some kind of running number without particular semantics), consequently a point mapping structure seems appropriate.

Alternative 5: subspace mapping for income non-placing, point mapping for ssnr placing

In this schema, the access structure again contains two data structures. However, the point mapping structure is used for record placement and the subspace mapping structure contains indirection records. The result is shown in Figure 3.21 with sparse point mapping and dense subspace mapping.

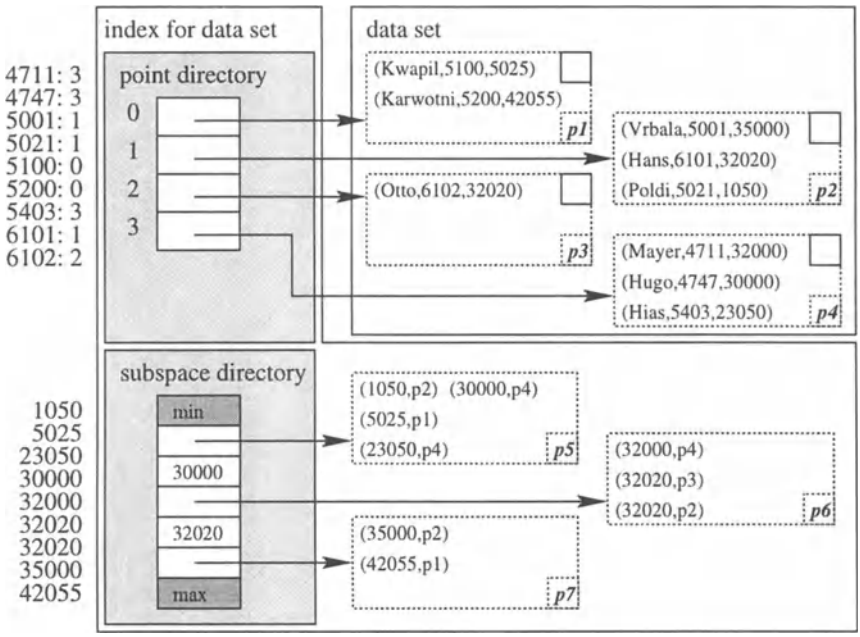


Figure 3.21 Combined data structures, ssnr placing

3.1.4 Evaluation criteria

Looking at the above examples, a challenging question arises. Since there are in general several possibilities to implement an indexing mechanism, we would like to set up quality indicators which could be used to determine whether or not one index implementation is better than another.

The difficult part can be outlined as follows: an index should minimize the number of disk transfer operations. An evaluation with respect to a particular query template, e.g., retrieve all persons with an income less than a particular value X, on the one hand, and with respect to a particular raw data distribution, e.g., an assumption that the stored values for attribute Income are uniformly distributed between, say 10000 and 5000000, on the other hand, is possible (see [Yao77]) although not totally straightforward.

For example, an index implementation A is considered superior to an implementation B, if the expected number of chunks fetched assuming implementation A is smaller than assuming implementation B, always for the given query template and the given raw data distribution. Unfortunately, the complexity of such evaluations increases drastically, if several query templates have to be considered simultaneously, e.g., range queries mixed with exact match queries, or, if the underlying raw data distribution does not follow one of the distribution types well covered by probability theory[5].

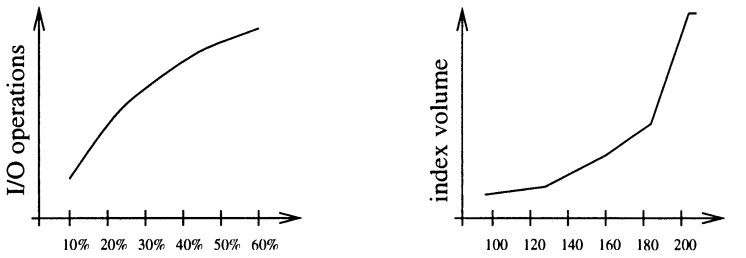
Additional considerations deal with the interaction between query profiles and update operations, and with questions concerning the storage space consumption resulting from particular index implementations like multiway trees or the like. Probability theory has been used extensively in order to achieve general results for various raw data, update request and query request distributions. However, due to the variety of practical application profiles, a dominating index implementation has not been found so far and numerous proposals for index implementations exist. Basically, each of these proposals aims at a particular set of query-update profiles.

Following from the above considerations, the practical evaluation criteria for such mechanisms are retrieval performance on the one hand and storage space overhead on the other hand. The former is measured in number of disk accesses necessary to answer a query, whereas the latter is measured in the number of disk blocks (pages, storage chunks) used to store the underlying search data structure(s).

To obtain comparable results, all measures are usually normalized: the number of disk accesses for the execution of a particular query is related either to the number of disk accesses necessary to fetch the raw data volume qualifying for this particular query (corresponding to the theoretical minimum number of I/O operations) or to the raw data volume, or to query volume in terms of fractions

[5]In practice, any assumptions about raw data distributions or query profiles are hard to state. It goes without saying that any use of ad-hoc query interfaces in addition to canned transactions increases these problems.

of the corresponding attribute domain (see for example Figure 3.22 left hand side for a hypothetical index structure). The number of disk blocks used to store the index data itself is usually related to the raw data volume, i.e., the number of disk blocks containing the persistent record set (see for example Figure 3.22 right hand side, again for a hypothetical index).



(a) I/O cost vs. query size (b) index volume vs. data volume

Figure 3.22 Evaluation patterns with respect to I/O cost and index size

Since all these measures usually depend on raw data distribution, clustering and correlation, one has to state a number of model assumptions for practical evaluations like uniform or non-uniform data distribution, a certain degree of attribute correlation (e.g., attributes age and income will have a high correlation coefficient in most countries) and the like. Additionally, one may distinguish between best case, worst case and average case evaluations. Readers interested in analytical evaluation techniques find an evaluation example for type hierarchy indices in Chapter 7.

3.2 A SYSTEMATIC APPROACH

The informal description of concepts and basic approaches as given in the previous section is refined in a semi-formal style in this section. In particular, we distinguish between the technical storage layer as described in the first subsection and the conceptual data space and query model described in the second subsection.

3.2.1 Physical storage model

Throughout this section, a *data set* is assumed to contain *records* of identical attribute structure given by the data set's *attribute set* denoted by *attr* =

$\{Attr_1, \ldots, Attr_n\}$. Each attribute $Attr_i$ has a fixed value set or *domain* denoted by dom_i. The minimum and maximum values of dom_i are denoted by min_i and max_i, respectively. A record (or tuple) $t = (v_1, \ldots, v_n)$ stored in such a data set is a list of n values such that $v_i \in dom_i \forall i \in [1, n]$. In what follows, the set of all records currently stored in a data set is denoted by $recs = \{t_1, \ldots\}$.

As usual, *addresses* are used to refer to storage areas (chunks). The set of all addresses currently in use for a particular data set is denoted by $addr = \{a_1, \ldots\}$. Assuming that all storage areas are of the same size, the constant b is used to denote the maximum number of tuples per storage area. In mass storage oriented search structures based on block transfer devices like magnetic disks, the value of b is given by $\left\lceil \frac{\text{size}(chunk)}{\text{size}(record)} \right\rceil$. This is the most important difference to main memory oriented search structures, in which b can be chosen at will by the structure implementer. However, since index structures for large persistent object sets have to be implemented on mass storage b is always assumed to be fixed in the sequel.

Usually, in a storage area denoted by an address, one or more elements of *recs* will be stored, the function a-contents : $addr \rightarrow \mathcal{P}(recs)$ is used to denote the set of records stored at a particular address. On the contrary, each record of the data set will be stored at exactly one address (in other words, we omit all redundancy issues in our presentation), so a function r-address : $recs \rightarrow addr$ returns for any element of the data set its current address (i.e., the address of the storage area the record resides in).

For example, the structure of a data set for object type Person could be defined as $Attr_1$ = dateOfBirth, $Attr_2$ = income, $Attr_3$ = Old, $Attr_4$ = weight with dom_1 = Date, dom_2 = Float, dom_3 = Long, dom_4 = Float. Records in the data set could be $recs = \{t_1, t_2, t_3\}$. Further on assuming $addr = \{a_1, a_2\}$, i.e., two storage areas in use for the data set, t_1 and t_3 could be stored at address a_2 and t_2 could be stored at a_1. The connection between *recs* and *addr* is given by a-contents$(a_1) = \{t_2\}$, a-contents$(a_2) = \{t_1, t_3\}$ and the corresponding function r-address.

If an indexing technique is used, a (possibly improper) subset of the attribute set is chosen as the set of *searchable* attributes, i.e., as the set of attributes to be referred to in query requests. All other attributes (if there are any such) are called *descriptive* attributes. The answer to the question whether or not an attribute of a particular data set should be searchable or descriptive is always application specific. Without loss of generality, attributes $Attr_1, \ldots, Attr_m$ are assumed to be searchable and attributes $Attr_{m+1}, \ldots, Attr_n$ are assumed descriptive. If $m = n$, all attributes have to be part of one or more search data

structures (see example in previous section), if $m = 0$, no search data structure will be maintained and the result is a sequential organization of the data set.

If in the running example the application designer states that queries will frequently refer to attributes dateOfBirth and income, these two attributes have to be searchable ("indexed"), whereas attributes Old and weight remain descriptive; thus $m = 2$.

Implementing all attributes as searchable attributes by default is usually not a good idea since any searchable attribute causes some kind of overhead with respect to storage space consumption and/or CPU resource consumption. It should also be noted that descriptive attributes may occur in query specifications as well. On most occasions, an end user launching a query request will not even know if an attribute is implemented as searchable or descriptive. The only difference is that query requests containing descriptive attributes in the search condition will suffer from worse query execution performance (due to sequential scans of storage areas and condition evaluation for each stored record).

In our example, assuming dateOfBirth and income searchable and Old and weight descriptive, a query

```
select p from person p where p.income > 10000
```

will hopefully take advantage of the underlying search structure mechanism whereas a query (which is structurally equivalent at identical at the query language level) like

```
select p from person p where p.weight > 100
```

hits a descriptive attribute and therefore causes a possibly time-consuming sequential scan through the data set (in our example through the collection person). However, as already indicated above, if the target collection is small, the overhead incurred by the search structure could even outweigh its theoretical advantage over the sequential organization.

3.2.2 Logical data space and query model

Abstracting from the currently stored records of a data set, one could consider in a first step the overall combined domain structure of the searchable attributes (as spanned by their m attribute domains) a geometrical space. In this sense, the Cartesian product of the m attribute domains $DS_s = dom_1 \times \cdots \times dom_m =$

$[min_1 \ldots max_1] \times \cdots \times [min_m \ldots max_m]$ is called the *searchable data space* in contrast to the overall data space $DS = dom_1 \times \cdots \times dom_n = [min_1 \ldots max_1] \times \cdots \times [min_n \ldots max_n]$. An element p of a data space is called *point* of the particular data space.

There is a close relationship between the points in the searchable data space and the records in the data set: for each record $t = (v_1, \ldots, v_m, v_{m+1}, \ldots, v_n) \in recs$, there is a corresponding $p = (p_1, \ldots p_m)$ in the searchable data space. It should be noted however, that the points in a searchable data space are unique (data spaces are point sets), whereas the corresponding projection of the data set on the searchable attributes is, in general, a bag. If, for example, dateOfBirth would be the only searchable attribute in our example data set, the probability that more than one tuple with a given attribute value is stored in the data set is close to 1 for reasonably large data sets.

Data space partitionings as well as query specifications are based on subspaces of the data space. In this presentation, we focus on partitioning and query subspaces resembling hyperrectangles, therefore a subspace $[lb_1 \ldots ub_1] \times \cdots \times [lb_m \ldots ub_m]$ of DS_s or $[lb_1 \ldots ub_1] \times \cdots \times [lb_n \ldots ub_n]$ of DS is called *orthogonal subspace* in the sequel. The set of all orthogonal subspaces of DS and DS_s are denoted by DS^* and DS_s^*, respectively. Since the machine representations of all attribute domains are finite, also DS^* and DS_s^* can be considered finite sets. A subset $DSP = \{oss_1, \ldots\}$ of DS_s^* is called a *partitioning* of the searchable data space, if $\bigcup oss_i = DS_s$. As already demonstrated above, subspace mapping approaches always maintain some kind of data space partitioning, either subintervals in the one-dimensional case (see example above) or m-dimensional hyperrectangles in the multi-dimensional case.

A *query specification qss* is an orthogonal subspace of $DS = [lb_1 \ldots ub_1] \times \cdots \times [lb_n \ldots ub_n]$ used to qualify all elements of *recs* which are geometrically within *qss*. A *query execution* is a computation of the function q-result yielding for a query subspace all qualifying records:

$$\begin{aligned}&\text{q-result} : DS^* \rightarrow \mathcal{P}(recs) \text{ such that}\\ &\text{q-result}(qss) = recs^{qss} \text{ with}\\ &recs^{qss} = \{t(v_1, \ldots v_n)\} \subseteq recs \wedge \forall 1 \leq i \leq n : lb_i^{(qss)} \leq v_i \leq ub_i^{(qss)}\end{aligned}$$

In what follows, the actual computation of q-result in point mapping approaches is compared to the respective computation in subspace mapping approaches.

Storage and retrieval in point mapping approaches

Conceptually, a single point p of the searchable data space is mapped to a single address a (c.f., hashing and related approaches). However, if more than b records referring to the same point are stored in the data set, or if more than b different records are mapped to the same address, some kind of overflow organization has to be used. For the moment, we abstract from the technical details of appropriate overflow management and use a mapping of points to **sets** of addresses, in particular, a function p-map which yields for a given point p in the searchable data space a set of addresses in such a way that these addresses (usually exactly one address) contain all stored records corresponding to $p(p_1, \ldots p_m)$.

Let rec^p denote $\{t(v_1, ..v_n) \in rec | \forall 1 \leq i \leq m : v_i = p_i)\}$, i.e., all stored records referring to point $p(p_1, \ldots p_m)$.

$$\begin{aligned} &\text{p-map} : DS_s \rightarrow \mathcal{P}(addr) \text{ such that} \\ &\text{p-map}(p) = addr^p \text{ with} \\ &addr^p \subseteq addr \wedge (\forall t \in rec^p : \text{r-address}(t) \in addr^p) \end{aligned}$$

The function p-map (often called *hash* function) is the core of any point mapping search data structure. It can be either computed on demand or stored or implemented as a combination of these two alternatives.

For any new record $t(v_1, \ldots v_n)$, a vacant slot in one of the storage chunks referred to by p-map$((v_1, \ldots v_m))$ is allocated. If there is no such slot, either p-map is modified, or p-map$((v_1, \ldots v_m))$ is extended by another newly allocated storage chunk.

For a query specification $qss = [lb_1 \ldots ub_1] \times \cdots \times [lb_n \ldots ub_n]$ the result is produced as follows:

foreach $p \in \{p(p_1, \ldots p_m) | [lb_1 \leq p_1 \leq ub_1] \times \cdots \times [lb_m \leq p_m \leq ub_m]\}$ **do**
    $hitset \leftarrow hitset \cup$ p-map(p_i)
end

foreach $a_i \in hitset$ **do**
    foreach $t_i(v_1, \ldots v_n) \in$ a-contents(a_i) **do**
        if $lb_i \leq v_i \leq ub_i \forall i \in [1, n]$ **then**
            q-result$(qss) \leftarrow$ q-result$(qss) \cup \{t_i(v_1, \ldots v_n)\}$
        end
    end
end

Looking at the first part of the retrieval algorithm (hit set construction) the query effort is directly proportional to the number of points in $[lb_1 \ldots ub_1] \times \cdots \times [lb_n \ldots ub_m]$. Consequently, point mapping favors exact match queries and range queries with small volume.

In any case, the client software using the indexing components (typically the query optimizer of a DBMS) has to decide whether or not the index should be used for a particular query request: if a subset of attributes 1 to m is restricted in the query request (small intervals for some or all attributes in the searchable data space), the use of the index will be favorable, since a number of storage chunks are not qualified and therefore not fetched from disk. However, if some or all of the attributes in the searchable data space are restricted using large query intervals or even not restricted at all (using $[min_i, max_i]$ for some or all searchable attributes) and a subset of attributes $m+1$ to n is targeted in the query request instead of the searchable attributes, the use of the index could not only yield no advantage but even a severe performance penalty: in the worst case, all chunks are hit during index processing thus saving no disk I/O operations at all in the second phase and the index processing itself (also consuming a moderate amount of machine resources) has to be considered pure overhead.

Storage and retrieval in subspace mapping approaches

The idea in this context is to map each subspace of a particular partitioning *DSP* of the searchable data space to one storage chunk. Theoretically, overflow management is not an issue for subspace mapping approaches because in case of a storage chunk overflow, i.e., more than b tuples mapped to the same address, another boundary value is chosen and the overfull subspace is further partitioned into two smaller subspaces in such a way that none of these subspaces is overfull.

However, from a more practical point of view, also the ugly case of more than b identical tuples has to be considered (more precisely, more than b tuples with identical attribute values in the indexed attributes). In that case, further partitioning of the overfull subspace is impossible, since there exists no new boundary value. The result is the same as for point mapping approaches: overflow management, typically linked lists of storage chunks belonging to the same subspace. For a given partitioning, a function s-map yields for *each subspace* of the partitioning a set of storage chunk addresses.

For a subspace $oss = [lb_1 \ldots ub_1] \times \cdots \times [lb_m \ldots ub_m] \in DSP$, rec^{oss} denotes the record set $\{t(v_1, \ldots v_n) | t \in rec \wedge (\forall 1 \leq i \leq m : lb_i \leq v_i \leq ub_i)\}$, i.e., all stored

records referring to points in the subspace.

$$\begin{aligned}&\text{s-map} : DSP \rightarrow \mathcal{P}(addr) \text{ in such a way that}\\&\text{s-map}(oss) = addr^{oss} \text{ with}\\&addr^{oss} \subseteq addr \wedge (\forall t \in rec^{oss} : \text{r-address}(t) \in addr^{oss})\end{aligned}$$

Similar to the hash function in point mapping approaches, the function s-map is, together with the partitioning information of the searchable data space, the essence of any subspace mapping structure. Usually s-map is not computed on demand (although such schemes are possible) but rather stored in a directory. The partitioning information is also stored in the directory, in one way or the other, either explicit (boundary values) or implicit in the subspace identifiers (in these cases, subspace identifiers are constructed algorithmically from the corresponding subspace boundaries, see DYOP file [OO85] and the BANG file [Fre87]).

For any new record $t(v_1, \ldots v_n)$, the matching subspace $oss \in DSP$ is determined. In particular, if the partitioning is pairwise disjoint ($oss_i \cap oss_j = \emptyset$ for arbitrary $i \neq j$), $oss^{(t)}$ simply is the subspace having point $p(v_1, \ldots v_m)$ as element. In case of intersecting subspaces, things are not that simple. If for any two intersecting subspaces $oss_i, oss_j \in DSP$ holds: $oss_i \subset oss_j \vee oss_j \subset oss_i$, then $oss^{(t)}$ is the smallest subspace having point $p(v_1, \ldots v_m)$. If there are arbitrary intersections, an additional rule has to be used to determine $oss^{(t)}$. The former two partitioning strategies are practically relevant, whereas the latter is usually omitted due to the additional overhead necessary to uniquely determine the relevant subspace for a tuple.

In a next step, s-map($oss^{(t)}$) yields the relevant address(es) and the record is put in a vacant slot in the corresponding storage chunk. If there is no vacant slot, the partitioning is modified in one of the two ways described below:

- $oss^{(t)}$ is split solely based on its boundary values (e.g., binary splitting) and the data distribution over the two parts is checked afterwards. In that case, the resulting distribution can turn out to be insufficient (in the worst case: all the data in one part and none in the other) and the process has to be repeated with the overfull until the data distribution reaches some predefined threshold (e.g., one third to two thirds or better) or the split fails.
- $oss^{(t)}$ is split based on the data in a-contents(s-map($oss^{(t)}$)) and on its boundary values (e.g., median splitting). There are usually less redistribution problems in this case, however, for example in the worst case already

mentioned above, i.e., identical tuples with respect to attributes 1 to m, the split may also fail due to a lack of new boundary values.

After the subspace split attempt, a new storage chunk has to be allocated. If the split has been successful, a new subspace $oss^{(t)'}$ is inserted into the partitioning, s-map is modified accordingly, and the data in a-contents(s-map($oss^{(t)}$)) is redistributed over $oss^{(t)}$ and $oss^{(t)'}$. If the split fails, the partitioning remains unchanged and s-map($oss^{(t)}$) is extended by the address of the newly allocated chunk (overflow handling without subspace split).

For a query specification $qss = [lb_1 \dots ub_1] \times \cdots \times [lb_n \dots ub_n]$ the result is produced as follows:

```
foreach oss_i ∈ DSP do
    if oss_i ∩ qss ≠ ∅ then
        hitset ← hitset ∪ s-map(oss_i)
    end
end

foreach a_i ∈ hitset do
    foreach t_i(v_1, ... v_n) ∈ a-contents(a_i) do
        if lb_i ≤ v_i ≤ ub_i ∀i ∈ [1, n] then
            q-result(qss) ← q-result(qss) ∪ {t_i(v_1, ... v_n)}
        end
    end
end
```

Interestingly, the only difference to point mapping is the way the set of relevant chunk addresses (hit set) is produced. The foreach loop does not iterate over all elements (points) of *qss* yielding an unacceptable query performance pattern for large query volumes, but only over the significantly smaller set of orthogonal subspaces in the partitioning of the searchable data space. If using a number of concrete subspace mapping structures, the situation is even better, since they allow to terminate the iteration when encountering the first subspace with empty intersection. So we conclude that subspace mapping proposals favor range queries in the sense that the additional overhead caused by data space partitioning is compensated by a significantly smaller number of disk I/O operations (compared to range queries in point mapping structures).

The overall problem of how to decide whether or not to use an index for a specific query request *at all* (see discussion point mapping) is more or less the same for subspace mapping and point mapping.

Storage and retrieval in sequential organizations

Inserting new records into a sequentially organized data set is trivial. If currently storage chunks $a_1 \ldots a_k$ are in use for the data set, the first vacant slot in a_k is used to place the record. If a_k is full, a new chunk is allocated and registered as chunk a_{k+1}. In this case, the new record is the first record on chunk a_{k+1}.

For a query specification $qss = [lb_1 \ldots ub_1] \times \cdots \times [lb_n \ldots ub_n]$ the result is produced as follows:

```
foreach a_i ∈ addr do
    foreach t_i(v_1, ... v_n) ∈ a-contents(a_i) do
        if lb_i ≤ v_i ≤ ub_i ∀i ∈ [1, n] then
            q-result(qss) ← q-result(qss) ∪ {t_i(v_1, ... v_n)}
        end
    end
end
```

We conclude that the overall success of indexing mechanisms is based on two decisions: what kind of search data structure should be used for an estimated query profile (likelihood of exact match queries and of range queries) on the one hand, and whether or not the resulting index should be used at all for a particular query request, on the other hand.

The latter is not totally straightforward at a first glance (one could believe that using a fancy index structure always yields better results than a dumb sequential organization), but comparing the retrieval patterns in case of point mapping, subspace mapping and sequential approaches, the reason becomes clear: the phase in which the qualified storage chunks are fetched from disk is shortened by an indexing mechanism, if and only if the set of qualified addresses (called *hit set* above) is smaller than *addr*. Since any optimizer has to estimate for a particular query request *in advance* whether or not the hit set will be considerably smaller than *addr* thus compensating for index traversal overhead, this poses quite a challenge for system developers (unfortunately, these considerations are far beyond the scope of this book which focuses on the technical prerequisites, i.e., search data structures and indexing).

In the next two sections, we will introduce some important examples of subspace mapping and point mapping structures. The intention is to show the technical details of some design proposals while referring to the principles as outlined above. The sections deal with one-dimensional and multi-dimensional search structures (subspace mapping as well as point mapping), respectively.

This choice for the top-level structuring is somewhat arbitrary, a similar presentation could be done using subspace mapping and point mapping as top-level categories.

3.3 ONE-DIMENSIONAL SEARCH DATA STRUCTURES

As already mentioned above, one-dimensional search data structures are defined and maintained for a fixed single attribute or for a fixed concatenation of attributes. In general, they support the fast execution of query requests involving the indexed attribute. Depending on the technical details of a structure, exact match, partial match and range query requests are more or less favorable cases of query requests (see Section 2).

In this category, multiway trees and dynamic hash schemes are the most prominent and widely used data structures. The former are based on subspace mapping, whereas the latter use point mapping.

Since the early seventies, a large number of researchers contributed to proposals focusing on balanced multiway search trees. In what follows, we describe the so called B^+-tree representing the widely used successor structure of an early proposal by Bayer and McCreight (see [BM72]).

There is also a large number of *dynamic hashing* proposals. In this context, a point mapping scheme is called *dynamic*, if the size of *addr*, i.e., the number of storage chunks used, can vary over the life time of the search data structure. For mass storage oriented search structures, this is a mandatory requirement to avoid the large amount of reserved (and most of the time unused) storage space yielded by any static hashing scheme (e.g., fixed size hash tables). From the large set of major contributions, we give an outline of an early and conceptually simple approach called *extensible hashing* by Fagin, Nievergelt, Pippenger and Strong in [FNPS79].

3.3.1 B^+-Trees

In a B^+-tree, the partitioning DSP of the searchable data space DS_s (i.e., in this case, of the domain of the indexed attribute) together with the function s-map is stored in a balanced multiway tree. One tree node corresponds to one

storage chunk. Inner nodes represent the access structure. They implement the partitioning by stored boundary values and storage chunk references to successor nodes (see figures below). Informally, each reference enclosed by two boundary values b_1 and b_2 points to a subtree which provides access to all stored values in the interval $[b_1, b_2[$.

In case of a non-placing (i.e., dense) B$^+$-tree, the leaf nodes contain indirection records with the stored values of the indexed attribute in the data set and references to the appropriate storage chunks in the data set (c.f. Figure 3.4 in Section 3.1.2), e.g., if a record (Vrbala, 5001, 35000) is stored in chunk 4711 and a non-placing B$^+$-tree is used for attribute income, one of the tree's leaf nodes will contain an indirection record (35000, 4711). It should be recalled that the term *non-placing* refers to the records of the data set which means that a non-placing B$^+$-tree is non-placing only with respect to the data records but of course *placing with regard to the indirection records* used to store the physical access information. As already mentioned, *any* search structure has to place access information in some way, otherwise it would be unable to locate a piece of stored information in an efficient way.

However, in case of a placing (i.e., sparse) B$^+$-tree, there are two possible points of view with respect to its leaf nodes. On the one hand, the storage chunks used to hold the records of the data set can be considered the leaf nodes of the placing B$^+$-tree, in this case the access structure as well as the value structure are considered parts of the B$^+$-tree (see Figure 3.24). This is the usual view on a placing B$^+$-tree. On the other hand, the placing B$^+$-tree can be seen as

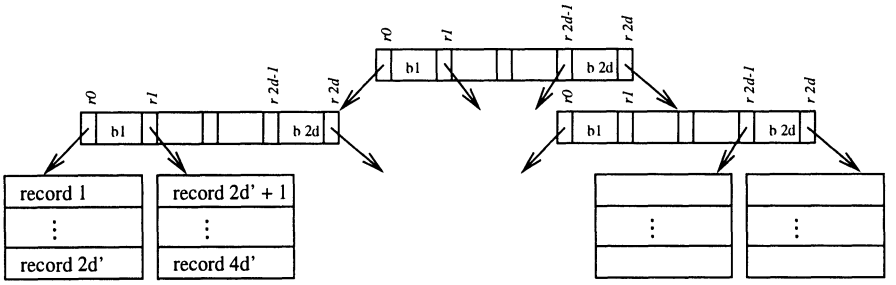


Figure 3.23 Overall structure of a B$^+$-tree.

"add-on" to the actual data set consisting of several data chunks. Seen this way, the leaf nodes do not contain the placed records but one reference per storage chunk and the reference is enclosed by two boundary values describing the domain interval covered by the storage chunk. In this case, the B$^+$-tree is

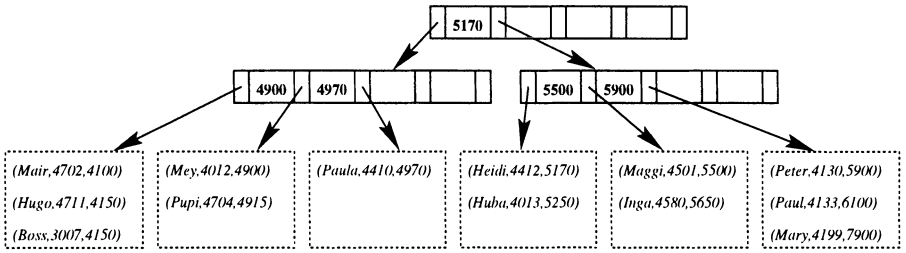


Figure 3.24 Placing B$^+$-tree on income.

(part of) the access structure of the index and the value structure is completely separated.

The structural shape of a B$^+$-tree is somewhat restricted in order to guarantee worst case path length and node occupancy. In other words, B$^+$-trees are not arbitrary multiway trees, but are characterized by the following properties. A multiway search tree with the following properties is called B$^+$-tree of order d, d' (overall structure shown in Figure 3.23):

- structural properties
  - each leaf node has the same path length from the root node
  - the root node is either the only node in the tree or has at least two successor nodes
  - each inner node has at least $d + 1$ and at most $2d + 1$ successors
- content properties
  - each inner node contains n sorted boundary values ($d \leq n \leq 2d$)
  - each inner node contains $n + 1$ references to successors
  - each leaf node contains at least d' and at most $2d'$ entries (data records or indirection records)
- ordering properties
  - in an inner node containing
    a maximal sequence $(r_0, b_1, r_1, b_2, r_2, ...b_{2d}, r_{2d})$ and
    a minimal sequence $(r_0, b_1, r_1, b_2, r_2, ...b_d, r_d)$,
    all r_i refer to references and all b_i refer to boundary values;
  - r_0 refers to the subtree with values less than b_1

- r_i refers to the subtree containing values greater than or equal to b_i and less than b_{i+1} with $1 \leq i \leq 2d - 1$
- r_{2d} refers to the subtree with values greater than or equal to b_{2d}

Figure 3.25 shows a B$^+$-tree with $d = 2$ and $d' = 2$ referring to attribute income of the record specification used in Section 3.1.2. The data structure is non-placing and therefore contains indirection records in the leaf nodes which are linked to speed up query processing (see discussion below). If there are more than $b = 2d'$ records with the same value in the indexed attribute, overflow chunks are inserted into the resulting linked list of leaf nodes. A query inter-

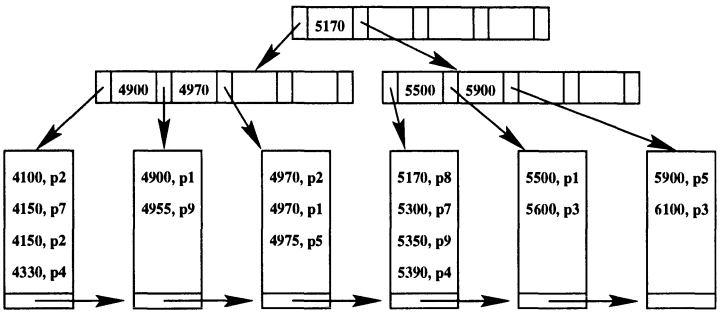


Figure 3.25 Non-placing B$^+$-tree on income

val referring to a single value of the indexed attribute (exact match query) is processed as a single top-down traversal from the root to the appropriate leaf node. For example, an exact match query [4970, 4970] for income yields the traversal marked by e in Figure 3.26. Any proper query interval, i.e., range, is processed in the following way: in a first phase, a top- down traversal aiming at the interval lower bound is done (same procedure like for an exact match query); in a second phase, the leaf nodes are fetched directly using the links between sibling leaves. The leaf node traversal terminates when the first data value outside the query interval is encountered. Again, referring to an example, e.g., income restricted to [4950, 5400], the corresponding query execution is shown in Figure 3.26 using r tags.

Insert operations start with a top down traversal to locate the appropriate leaf node. If there is an overflow in the leaf node, an attempt to split the node is made. In particular, a storage chunk is acquired, the median of all entries is taken and used as a new boundary value which has to be inserted in the corresponding parent node. This situation is shown in Figure 3.27. In case of a parent node overflow, the splitting scheme is repeated, i.e., a split might be

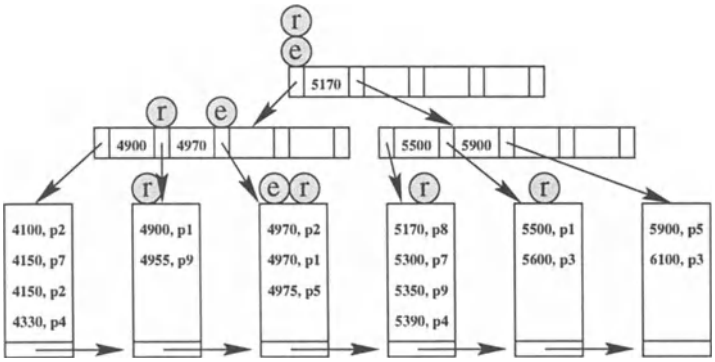


Figure 3.26 Traversals in a B^+-tree

propagated bottom up through the index. In the worst case, the root is split and the height of the B^+-tree increases. If the split operation fails due to a lack of possible boundary values (i.e., too many identical data values), the newly allocated chunk is inserted into the leaf node chain without being referenced by an inner node.

Delete operations may be handled either by delete flags in the data chunks or by merge operations in case of chunk underflow (inner node or leave node). In the former case the implementation is technically straightforward, but requires periodical reorganization to keep the tree in proper shape. If not only merge operations but also *inner node split* operations are deferred until the next periodical reorganization run, the resulting data structure schema resembles closely the traditional ISAM approach. If both kinds of structure dynamics are implemented, i.e., node split after node overflow and node merge after node underflow, the implementation is more complex but guarantees all B^+-tree properties after each insert or delete operation.

3.3.2 Extensible hashing

As proposed by Fagin, Nievergelt, Pippenger and Strong in [FNPS79] an extensible hash structure contains an array A containing chunk references (often called directory) together with the actual storage chunks, which play the same role as the leaf nodes of a B^+-tree. In particular, if the extensible hash structure is placing, the storage chunks contain the records of the data set, if it is non-placing, indirection records are stored. In the former case, the same

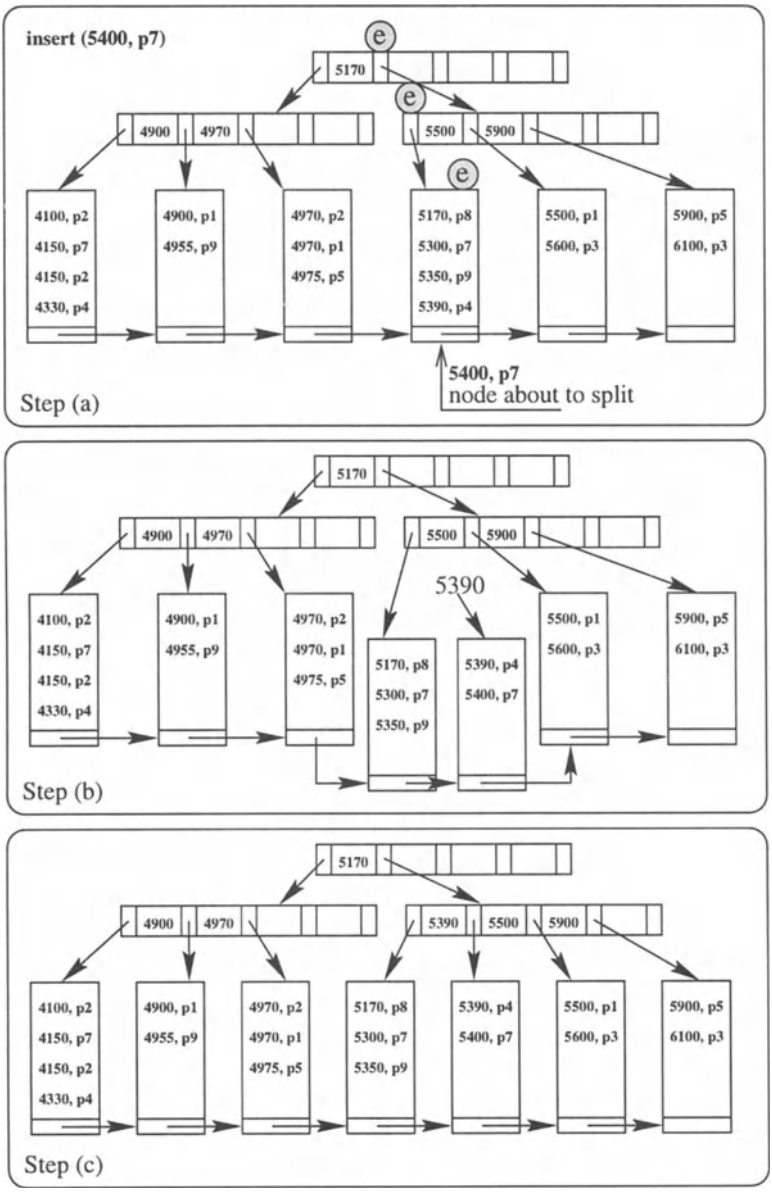


Figure 3.27 Node splits in a B$^+$-tree.

considerations with regard to data set integration or separation as for B$^+$-trees apply.

The size of A has to equal an integer power of 2. If a particular array size is 2^d, d is called *global depth*. It may vary due to insert and delete operations, i.e., an extensible hash table may be doubled or cut in half in size if necessary with respect to update operations. Extensible hashing does not use any overflow chaining as long as there are for any data value no more than b records with that value (same problem as outlined in the B$^+$-tree discussion above). If there is an overflow in any storage chunk and no chance to (re)use an existing directory entry, a directory entry is made available by doubling the directory (see insert discussion below).

In the framework outlined in Section 3.2, the array A is the data structure used to store the function p-map. In particular, p-map(y) for a data value y in the searchable data space is $A[h(y)]$. The function $h(y)$ can be more or less elaborate. In the simplest possible version $h(y) := y \bmod 2^d$, i.e., the d least significant bits of y.

In more sophisticated implementations $h(y) := u(y) \bmod 2^d$ with u denoting a function which hopefully yields for arbitrary raw data distributions in the indexed attribute uniformly distributed $u(y)$ values.[6] The overall structure is shown in Figure 3.28, the variable t stored separately in each storage chunk is called *local depth* and used during record insertion (see below). Any storage chunk of an extensible hash structure is referenced 2^{d-t} times by the directory. Accessing a particular data value y is trivial, only the chunk referenced by

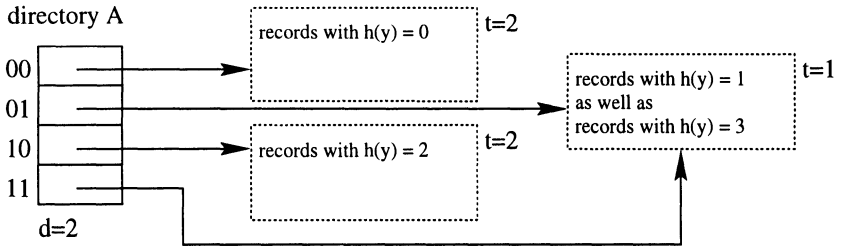


Figure 3.28 Layout of an extensible hash structure with $d = 2$.

$A[h(y)]$ has to be fetched from mass storage. However, as already indicated in

[6]There are also proposals using the d most significant bits of y which is only feasible if a reasonable function u is found. Without any distribution function u, i.e., using the d most significant bits of the raw data values could yield problems if there are few records with high data values and many records with small data values. For the latter, the d most significant bits will be all zero even for relatively large d.

the previous section, range query processing is feasible only for small interval specifications.

Figure 3.29 below shows an example of an extensible hash structure (non-placing, i.e., storage chunks filled with indirection records) for attribute ssnr and the same data set as in Section 3.1.2 with $b = 3$ and $d = 2$. Like in Figure 3.28, A[1] and A[3] refer to the same storage chunk, whereas A[0] and A[2] contain different addresses. In the example, the simplest possible implementation for p-map(y) is used, i.e., p-map(y) := $A[y \bmod 2^d]$. A lookup for ssnr = 5001 yields $A[1]$, the referenced storage chunk contains the indirection record (5001, px) indicating that the actual record is to be found in storage chunk px of the data set. The insert algorithm relies on both, the global depth d and the local depth

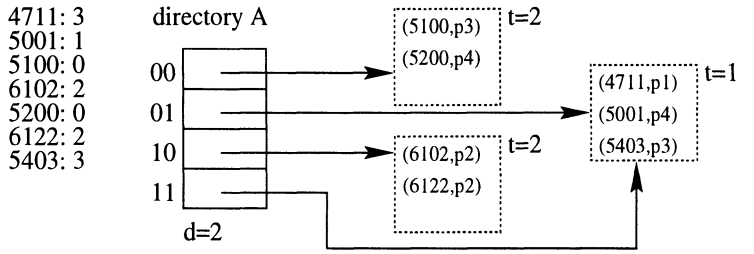


Figure 3.29 Extensible hashing used for ssnr (non-placing), $b = 3$ and $d = 2$.

t which is a characteristic property of each storage chunk used. Again, any insert operation starts with a location phase similar to a single value search, i.e., the storage chunk referenced by p-map(y) := $A[y \bmod 2^d]$ is fetched for record insertion.

In case of a data block overflow, a new chunk is allocated and the values are distributed over both chunks. In contrast to a B^+-tree node split, the distribution cannot be chosen according to the median element of the values in question. Instead of taking the median element as a new separator, the bit $t + 1$ of the stored values is used to discriminate between the two blocks. An extension of the example in Figure 3.29 can be used to illustrate the procedure.

Inserting an additional record with data value 5005 produces an overflow of the chunk referenced by p-map(5005) := $A[5005 \bmod 4]$ is fetched for record insertion. However, the overflow can be handled without directory expansion by using one of the existing directory entries and the bit $t+1$ (i.e., bit 2) for discrimination. The situation before and after the split is shown in Figure 3.30(a) and (b). A necessary (but unfortunately not sufficient) condition for this kind of overflow resolution is that the overfull chunk is referenced by more than one

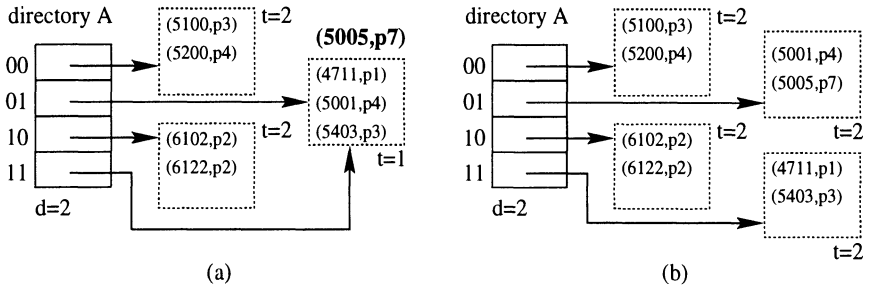


Figure 3.30 Before (a) and after (b) chunk split.

directory entry. This can be determined locally, since $t < d$ indicates exactly that case. $t < d$ is not sufficient for overflow resolution, since in pathological cases the distribution of the data values using additional bits may end up with all records in one chunk and none in the other. This happens if all bits used for additional discrimination (starting with the bit $t + 1$ and finishing with bit d) are equal for all records to be redistributed.

In some cases, an overflow causes a directory doubling. In particular if a split affects a chunk with $t = d$, i.e., a chunk for which no spare reference in the directory exists, the directory has to be doubled and d has to be incremented. In this case, the new directory entries are initialized guaranteeing a $t < d$ situation in the splitting chunk. Figure 3.31 shows the situation before insertion (a), after directory expansion (b) and finally after record insertion (c). With respect to record deletion, the trade-off mentioned above for B^+-trees is quite the same in this context. Delete flags in the storage chunks are easy to implement and decrease overall chunk occupancy, whereas physical deletion of records improves occupancy while requiring elaborated chunk merge and directory shrink algorithms.

3.3.3 Comparing B^+-trees and extensible hash structures

Comparing B^+-trees and extensible hash structures with respect to query execution performance reveals the typical situation for subspace mapping and point mapping structures described in Section 3.2. Any reasonable decision which structure to use has to be an application dependent decision. In particular, considering improper interval specifications, a search in a B^+-tree requires $O(\log N)$ chunk read operations assumed that N index nodes (i.e., inner nodes)

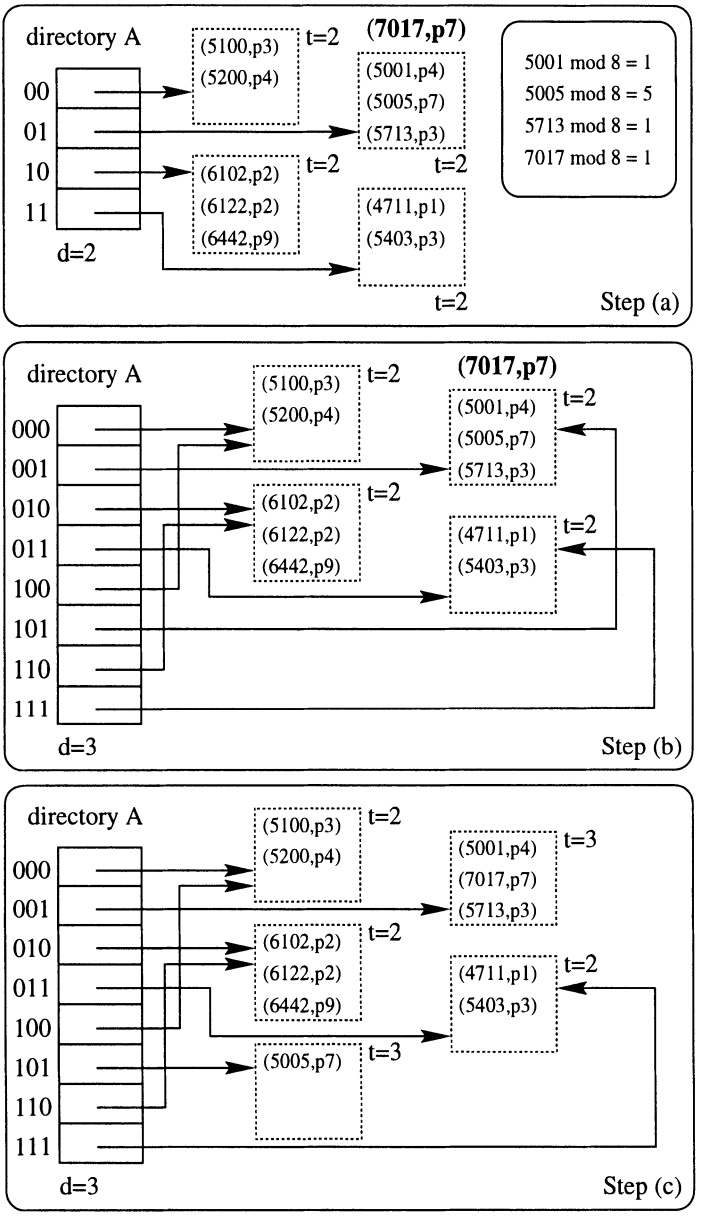


Figure 3.31 Before directory expansion (a), after (b) directory expansion, insert completed (c)

are in use. This is not so bad as it might seem at a first glance. Due to the high fan-out of a B$^+$-tree node, i.e. the large number of successor references in any inner node, the tree grows rather slowly. For example, even a low fan-out of 100 provides access to 1000000 chunks a 3-level B$^+$-tree.

However, the same search process in an extensible hash structure always takes $O(1)$, or in other words, a constant number of chunk retrievals. In reality 2 read operations are necessary, since the directory array often becomes too large to be held in main memory. We conclude in the realm of the previous section that extensible hash structures are an attractive choice as far as improper interval specifications are involved.

On the contrary, B$^+$-trees are clearly superior in case of proper interval specifications. Let *act* denote the number of values contained in $[lb \ldots ub]$ which are actually stored in a search data structure. In a B$^+$-tree, the corresponding query execution will cause $O(\log_{fanout} N + act)$ block retrievals, since the first block has to be found in the context of a top down traversal and all other blocks can be accessed through the leaf links (see above). In an extensible hash table, either a sequential search has to take place or $ub - lb$ data block addresses have to be determined and the resulting set of data blocks has to be retrieved. This will give B$^+$-trees a clear lead for almost all proper interval specifications with the exception of small interval sizes.

3.4 MULTI-DIMENSIONAL SEARCH DATA STRUCTURES

In contrast to one-dimensional structures, so called *multi-dimensional* search data structures provide fast content based access to several attributes at once. Again the structure of the searchable data space, or in other words, the searchable subset of the attribute set is fixed over the lifetime of the data structure.

Focusing on access support for arbitrary query specifications in the sense of Section 3.1.2, i.e., specifications with a mix of proper and improper query intervals in the searchable data space, yields additional requirements for search data structures. An important issue in this context is *physical clustering*. Informally, the idea is to map points (corresponding to records to be stored) to a linear scale, i.e., the disk block address space, in such a way that points with a small geometrical distance should also have a small distance to this linear scale.

During range query execution, the number of disk I/O operations will be small if the physical clustering of the search data structure is good. A simple example in the one-dimensional case is a B^+-tree yielding an optimal storage pattern with respect to physical clustering. However, mapping a one-dimensional data space on a linear scale is somewhat straightforward. Gaining similar results for the multi-dimensional case is a significantly more complex problem, since the physical clustering should be fair in the sense that no attribute should be favored above another.

A simple example for an unfair clustering policy is a compound B^+-tree index resulting from the concatenation of the $Attr_1, \ldots, Attr_m$ searchable attributes of a record specification. Such a compound index is sufficient for exact match queries, i.e. queries containing only improper interval specifications, but produces poor results for range queries.

These unsatisfying results are easy to understand, if one figures out the resulting physical clustering: $Attr_1$ determines the mapping to the storage chunks of the value structure therefore discriminating all other searchable attributes $Attr_2, \ldots, Attr_m$. Even worse, all storage chunks of the access structure would be fetched in case of interval specifications referring to $Attr_2, \ldots, Attr_m$. Hopefully, the query optimizer would decide to neglect the data structure in such a case and launch a sequential scan through the data set.

Using m one-dimensional structures like B^+-trees is not a good idea either. The positive aspect is that such a scheme supports arbitrary range queries in a simple way: if there are m B^+-trees, one has

- to determine for each of the m one-dimensional structures the set of relevant chunk addresses, then
- to determine the intersection of the resulting address sets and finally
- to fetch the qualified storage chunks.

However, there are two major drawbacks. At first, this schema also ends up with an unfair physical clustering: one data structure will be placing, all others non-placing. So small sets of storage chunk addresses will be fetched for the placing attribute and comparatively large sets will be fetched for all other attributes (assuming query intervals of roughly the same size). Secondly, all data values are stored two times, one time in the record itself and a second time in an indirection record of one of the supporting search data structures. Together with the space requirements for successor pointers in the structures,

the storage space consumption of such a solution has to be considered hardly acceptable for higher-dimensional data spaces.

The latter leads to a different problem category, namely storage space considerations. A multi-dimensional data structure which is attractive with respect to fair physical clustering and which provides fast exact match and range query processing has to fulfill the same requirements with respect to storage space like any one-dimensional structure. In all cases, there are two main concerns, namely *index size* and *chunk occupation ratio.* Intuitively, the size of a multi-dimensional structure depends on

- the organization of the data structure (more or less overhead for storing p-map, or s-map) together with the subspace boundaries
- the raw data distribution, since correlated raw data might expand directory structures beyond reasonable limits in terms of raw data volume (see grid file example below)
- the record insertion sequence, like for example insertion in random order in contrast to insertion in sorted order.

An important requirement in this context is a guaranteed worst case behavior with respect to storage space consumption. This has been probably the most important argument in favor of B^+-tree implementations as far as the design of commercial products has been concerned. Most people agreed that sets of B^+-trees are not a very good choice for the construction of a multikey index, but at least upper bounds for the index size could be provided. However, this section shows that the abovementioned argument may not hold any longer, i.e., it is argued that there is no reason left not to use recent multi-dimensional data structures for commercial DBMS products. Consequently, we do not recommend the use of traditional one-dimensional structures to build multikey index structures and turn our attention to state-of-the-art multi-dimensional structures.

The intention is to give an impression of the development process behind multi-dimensional data structures. In particular, we give a brief outline of the early k-d tree approach, of the original grid file and an improved version called the balanced and nested grid (BANG) file, and a more recent structure called hB-tree. Numerous important contributions are omitted since a comprehensive review of the field would probably fill several hundred pages (see Bibliography).

3.4.1 K-d trees

Informally, a k-d tree (short form of *k-dimensional tree*) as proposed by Bentley in [Ben75, Ben79] is a generalized binary search tree. The idea can be described as follows: in a standard, i.e., one-dimensional binary search tree, each node contains an element of *recs* and two references to (possibly empty) subtrees. The actual value of the searchable attribute $Attr_i$ is used to navigate in the tree. For example, during the execution of an exact match query based on an improper interval specification $lb_i = ub_i$ for $Attr_i$, a query execution branches to the left if lb_i is less than the key value of the current node, otherwise a branch to the right occurs.

In a k-d binary search tree the k searchable attributes (k corresponds to m in the notation of Section 3.2) of a record specification are used *alternatively* as discriminators during record insertion and query execution. One node contains one record and two successor pointers, there is no distinction between access structure and value structure.

In particular, at the first level of the k-d tree (i.e., the root level), the value of $Attr_1$ is used as discriminator value. At the second level, the values of $Attr_2$ are used and so on. In general, a k-d tree consists of more than $k+1$ levels, so the construction principle is cyclically repeated. On level $k+1$, the discriminator values are the data values in $Attr_1$ and so on. Figure 3.32 shows a 3-d tree resulting from the insertion of the records (7,a,12.5), (2,k,5.5), (4,x,10.0), (5,c,12.0), (4,p,4.0), (6,q,0.0) in that order. In the geometrical interpretation

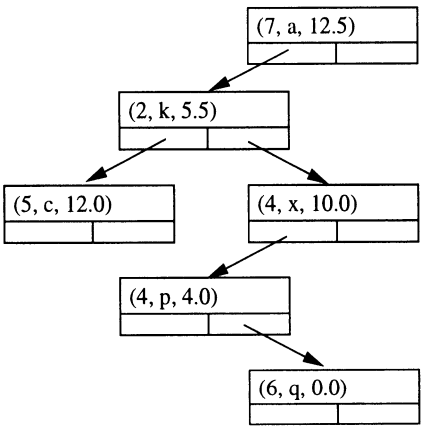


Figure 3.32 3-d tree

of the k-d tree approach, the data value for $Attr_1$ in the root node divides the

k-dimensional searchable data space into two subspaces. Each node at level 2 divides one of these subspaces into two additional subspaces according to the stored element of $Attr_2$. Figure 3.33 shows the resulting data space partitioning pattern in case of a 2-d tree. Insert and search procedures are similar to

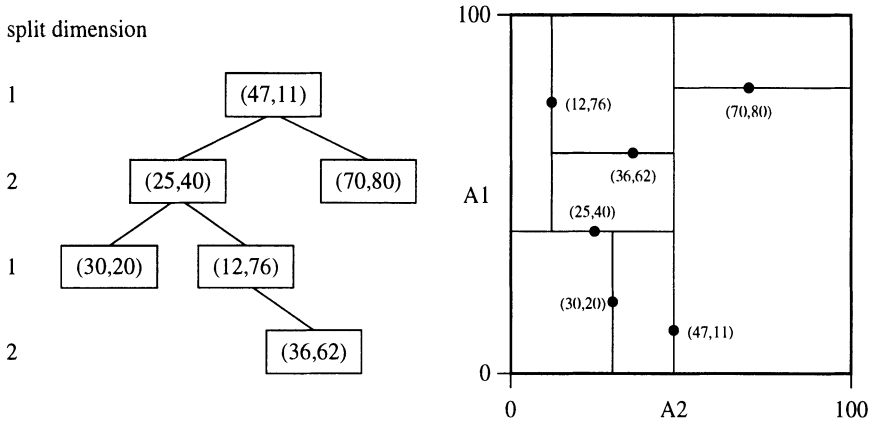


Figure 3.33 Data space partitioning of a 2-d tree

binary search trees and conceptually simple. Node deletion requires significant reorganization effort.

Exact match queries follow one path in the k-d tree until the first NULL reference is found. For example, searching for (4,x,10.0) in the k-d tree depicted in Figure 3.32 yields the following traversal:

(7,a,12.5), 4≤7, (2,k,5.5), x>k, (4,x,10.0), 10.0≤10.0, (4,p,4.0), 4≤4, NULL

The second part of the traversal, i.e., the part after the retrieval of (4,x,10.0) is necessary, since more than one record with the relevant values could be stored in the tree.

Range queries in general have to traverse more than one path in the tree, since both successor pointers of one node could refer to relevant subspaces of DSP. For example, considering a range query ($[0, 20], [c, o], [0.0, 100.0]$) and the node containing $(2, k, 5.5)$, both subtrees of this node have to be traversed because $[c, o]$ contains elements $\leq k$ as well as elements $> k$.

In case of insertion, the tree is traversed following the traversal pattern described above, until a vacant place (actually a NULL reference) is found. At that place, the new record is inserted.[7]

There are two major drawbacks with respect to multikey index maintenance. At first, keeping k-d trees balanced requires considerable reorganization effort. Secondly, using a k-d tree in a mass storage oriented environment would require an additional maintenance level, since several k-d nodes are likely to fit into one mass storage chunk. Despite these drawbacks Bentley's early proposal still has important impacts on recent developments. For example, the hB-tree discussed below can be considered a direct successor structure focusing on the necessary mass storage mapping not contained in the k-d proposal.

3.4.2 Scale based grid file

In the early eighties a research group headed by Nievergelt worked on a multidimensional search data structure which is neither directly related to multiway trees nor to traditional multi-dimensional point mapping. The result, commonly know as the *grid file*, is described in [NHS84]. In the meanwhile, the original grid file approach is often called *scale based* grid file to distinguish it from some of the descendants.

The original grid file uses m value lists (called *scales*, one per searchable attribute, and a directory data structure, basically an m-dimensional array. The situation is depicted in Figure 3.34. One directory entry corresponds to one m-dimensional subspace (called *grid region*) of the initial searchable data space. Grid regions are identified by numbers in Figure 3.34. The boundary values for each grid region are stored in the scales, which could be implemented either as static arrays or as linked lists. In this proposal, the m scales and the m-dimensional array represent the access structure containing DSP and s-map. As usual, the storage chunks containing the data records (placing application) or the indirection records (non-placing application) represent the value structure.

Each directory entry contains a chunk reference (address) referring to stored records. Following the schema outlined in Section 3.2, a stored record is referenced by a grid region, if the projection of this record to the searchable data space yields an m-dimensional point within the grid region (more precisely, within the subspace corresponding to the grid region). However, it is important to notice that in general several directory entries point to the same part of

[7] Finding a reasonable delete algorithm is more of a challenge

the value structure (e.g., grid regions 2, 5 and 8 in Figure 3.34). The subspace defined by the union of all grid regions mapped to the same storage chunk is called *block region*. Block regions are marked by capital letters in Figure 3.34. For the sake of algorithmic simplicity, all block regions have to be convex in the proposal by Nievergelt et al.

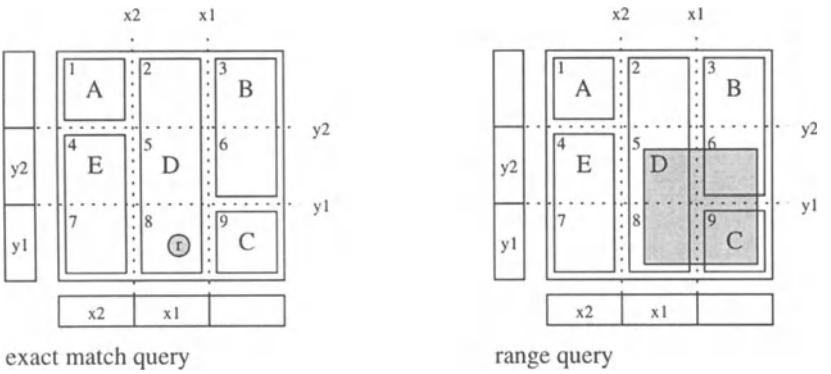


Figure 3.34 Scale based grid file

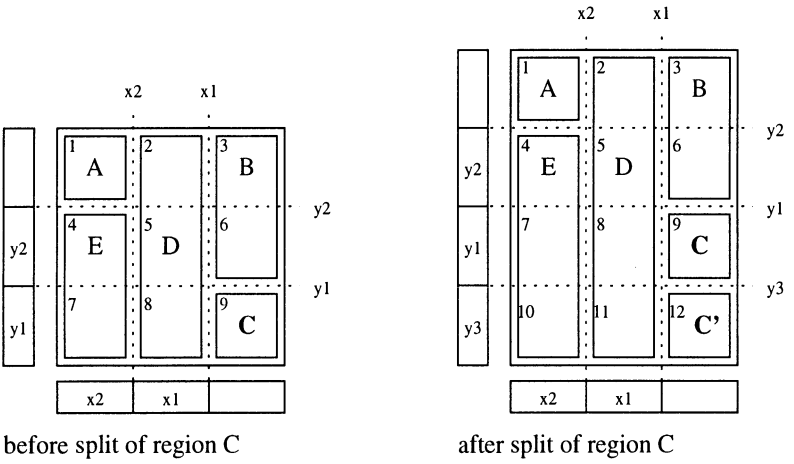


Figure 3.35 Scale based grid file after split of C

Query processing is straightforward: for an exact match query request, the m scales are used to determine the corresponding grid region g. Subsequently, s-map(g) is retrieved from the directory and the storage chunk can be read.

For range queries, the procedure is the same except that a set of grid regions is qualified and therefore several storage chunks are read during execution.

For example, a query request qualifying record r (see Figure 3.34 left hand side) hits via scale entries $x1, y1$ grid region 8 and therefore block region D. Storage chunk s-map(8) = D is fetched. Looking at the range query request in Figure 3.34 (right hand side) the grid regions 5, 6, 8, and 9 are touched. After applying s-map the hit set contains the block regions B, C, and D, so the corresponding storage chunks are fetched.

Chunks of the value structure might be split upon record insertions. In a straightforward implementation, a cyclic splitting scheme similar to the k-d tree proposal is used, or in other words, dom_1 is used for the first split, dom_m for split number m and again dom_1 for split number $m + 1$. In this case, a new scale entry has to be calculated using a particular splitting strategy. A standard strategy is to use the median of the stored data values in the splitting domain thus leaving approximately one half of the records in the splitting block and putting the rest in the newly allocated block.

A new scale entry in general implies several grid region splits, or in other words, not only the grid region referring to the overfull storage chunk is split. For example, a split of the chunk corresponding to block region C into C (old) and C' (new) in the example grid file directory affects grid region 9. However, no matter which dimension (domain) is used, a total of 3 grid regions is divided by the new scale entry (see Figure 3.35).

A more elaborated split algorithm can be used to partly overcome this problem. For example, splitting block region A in the example might be handled without additional scale entry: the block region contains more than one grid region, i.e., there is at least one stored boundary value (in one of the scales) geometrically within the block region. There are two prerequisites for the use of the existing boundary value:

- either the splitting algorithm allows a deviation from the cyclic domain splitting pattern, or the existing boundary value belongs by chance to the domain which is the next one to split, and
- the existing boundary value yields a data distribution over the two newly emerging block regions which is better or equal to some threshold (it is unlikely that the existing boundary value equals the median value of the data stored in the splitting dimension of the overfull storage chunk).

The former prerequisite can be easily met by a less stringent insert algorithm which checks for existing boundaries *prior to* choosing cyclically a new domain to split. However, whether or not the latter prerequisite is met depends on the actual raw data.

The scale based grid file is a mature multi-dimensional data structure for several practical applications. However, even an elaborated splitting procedure as described above produces undesirable effects in case of non-uniform data distributions. Considering for example highly correlated data (e.g., height and weight for records of type Person) the directory structure might grow exponentially. In this case, the strong clustering of the raw data will cause a large number of splits because hitting a few grid regions with any new record insertion is likely and hitting the rest (i.e., a comparatively large number of grid regions) is unlikely. The problem is that any split of one of the few often hit grid regions also splits a large number of sparsely populated or even completely uninhabited grid regions. Figure 3.36 illustrates the problem. This type of misbehavior is

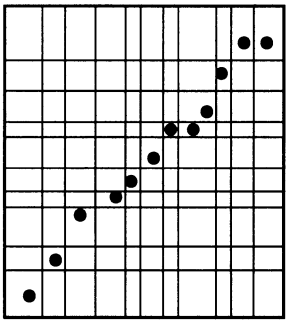


Figure 3.36 Scale based grid file and correlated data

clearly the main disadvantage of the original grid file proposal. A number of follow-up approaches tackle exactly this problem.

3.4.3 Balanced and nested grid file

Freeston proposed in [Fre87, Fre89] a binary splitting (see below) grid file with a significantly different grid region organization. The main intention in the so called balanced and nested grid (BANG) file approach is to avoid an exponential directory expansion in case of non-uniform or correlated raw data as depicted in Figure 3.36. The term *binary splitting* refers to a somewhat restricted splitting scheme adopted from the DYOP (for *DYnamic and Oorder Preserving data*

partitioning) proposal by Ozkarahan and Ouksel presented in [OO85]. The idea behind the DYOP splitting scheme is as follows: in the grid file the goal is to produce a reasonable data distribution by choosing a clever boundary value, typically a value close to the median of the stored data values in the respective split dimension. However, in a DYOP file any split has to divide the relevant interval, i.e., the length of the hyperrectangle in the splitting dimension into two subintervals of equal size without regard to the resulting record distribution.

This splitting policy allows to omit scales, since the boundary information can be derived from two integer values directly stored in the directory. Technically, these two integer values are used for region naming. On the other hand, binary splitting has to be combined with a sophisticated region organization in order to avoid a poor disk block occupation ratio. In general, the binary splitting schema is applied *iteratively* to produce a reasonable distribution.

In this context, the BANG approach uses *nested* block regions maintained with the help of a balanced multiway tree containing directory information (see Figure 3.38 right hand side). Nesting means that in contrast to the scale based grid file, two block regions might intersect. However, if two block regions intersect, one has to completely enclose the other. In other words, two block regions A and B might be either disjoint, or A encloses B or B encloses A. If A encloses B, the storage chunk referred to by A contains all k-dimensional points enclosed by the region boundaries which are *not enclosed by the region boundaries of* B as depicted in Figure 3.37. Consequently, a chunk address corresponding to A refers to a chunk containing all records within the geometric contents of the block region minus the geometric contents of all enclosed block regions. A search operation has always to proceed until the smallest enclosing

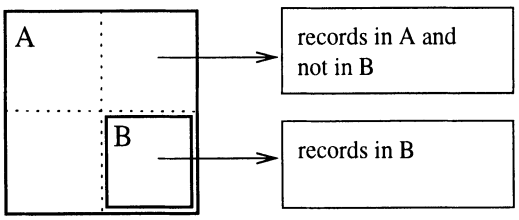


Figure 3.37 Nested regions and storage schema

block region has been found. In particular, searching for record y in the BANG file depicted in Figure 3.38 will cause a fetch of the storage chunk referenced by (the remainder of) region A, since $A \setminus (B \cup D \cup E)$ is the smallest block region containing y. Record x is contained in the space of region A as well but

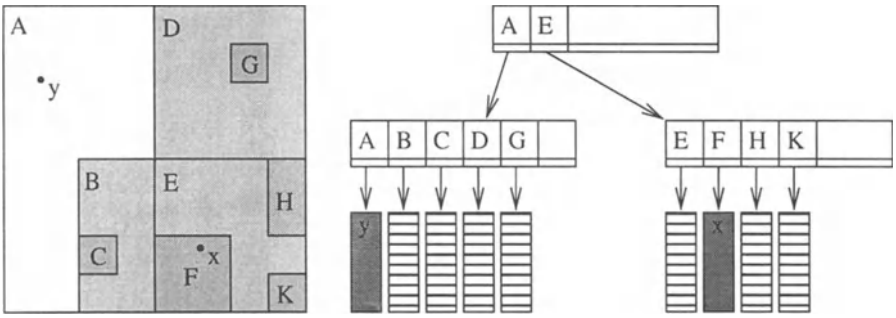


Figure 3.38 Partitioning and directory implementation

the space of region F is the smallest enclosing space. Consequently, searching for x yields the address stored with region F.

The BANG file is robust and especially well-behaved in case of non-uniform or correlated raw data. As in most cases, this reasonable behavior is accompanied by considerable algorithmic complexity. In particular, gaining some locality of search, i.e., fetching only a small fraction of all index chunks in case of small range queries, is possible only if rather sophisticated insert algorithms are employed (this topic is discussed using the term *spanning problem* in [Fre89]).

3.4.4 hB-tree

One of the more recent multidimensional data structure proposals by Lomet and Salzberg focuses on the worst case behavior of the search data structure with respect to the raw data distribution. In particular, in [LS90] a novel data structure called the *hB-tree* is introduced. This data structure is characterized, among other worst case bounds, by reasonable worst case index chunk and data chunk occupation ratios (see also [ELS95].

The term *hB* is an abbreviation for *holey brick* which refers to the splitting policy: quite similar to the BANG file, any split in an hB-tree causes the exclusion of a subspace from the overflowing block region. However, in contrast to the BANG file, the hB-tree does not adhere to a strict binary splitting scheme. In the geometrical interpretation of a block region as hyperrectangle or brick, this means to cut off a smaller brick from the overflowing brick thus leaving a holey brick (see Figure 3.39, right hand side). This observation gives the data structure its name. Brick boundary information is stored in a balanced

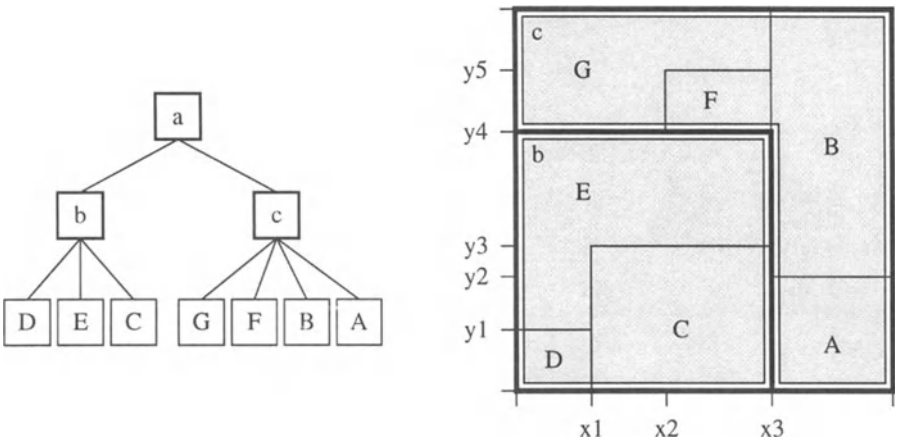


Figure 3.39 Nested regions and storage schema

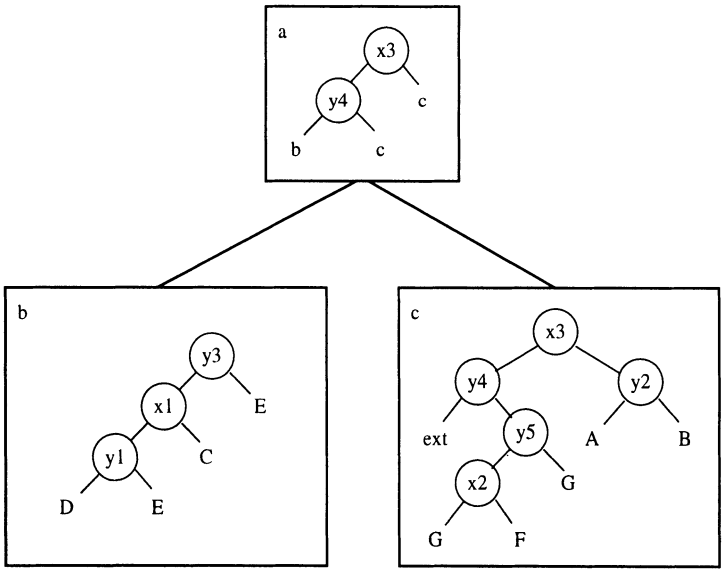


Figure 3.40 Full k-d tree representation of the hB-tree

multiway tree resembling a B^+-tree (see Figure 3.39, left hand side). Again, the directory tree organization might be considered similar to the organization used in the BANG approach.

Several positive aspects of the proposal are outlined in [LS90]: at first, it is proven that no arbitrary raw data distribution yields data partitioning ratios worse than 2:1 in case of overflows. In other words, there is always a sub-brick in an overflowing data chunk which contains at least one third of the total number of records stored in the splitting data chunk. Consequently, a worst case data chunk occupation ratio of 0.33 can be guaranteed. A similar argumentation is used to derive the same worst case figure for the index chunks.

Secondly, the index chunks are internally organized as k-d trees, i.e., the sub-brick boundary information contained in an index block is maintained as k-dimensional binary search tree (see Section 3.4.1). In Figure 3.40 the overall data space (region a) is partitioned into subregions b and c. The boundaries of these two regions are stored in the top-level index node a. It should be noted that also non-convex regions (like region c in the example) are possible. The k-d tree nodes corresponding to regions b and c are also shown in Figure 3.40. An index chunk overflow results in a partitioning of the corresponding stored k-d tree and a subsequent index chunk split. Again, each of the two resulting index chunks contains at least one third of the initial k-d tree nodes regardless of the raw data distribution. Consequently, the worst case index block occupation ratio is 0.33.

3.5 BIBLIOGRAPHY

Bayer and McCreight described in their seminal paper [BM72] the direct ancestor to all currently used B^+-trees. The original data structure proposal dating from 1972 is often called the B-tree (terminological issues see above). B^+-trees and dynamic hashing are covered in a textbook by Salzberg (see [Sal94]) also containing material on a hybrid approach, namely bounded disorder files proposed by Litwin and Lomet (see [LL86] and [LL87]).

Several researches use bijective mappings in order to produce either a single key value or a disk block address from the k values of a record. The idea behind the former approach is to use elaborated standard index structures for the resulting single key (often called "record signature"). The most important examples are bit interleaving techniques which go beyond simple value concatenation (simple concatenation yields unfair clustering, see above) and employ so called "shuffling functions".

Orenstein and Merrett proposed shuffling functions for bit interleavings or so called z-orderings (see [OM84]). Using the binary representations of the k values

to be mapped as a starting point at first an alternating cyclic bit interleaving is proposed, i.e., the "trivial shuffle" taking the first bit of the value in dimension 1 followed by the first bit of the value in dimension 2 up to the first bit of dimension k followed by the second bit of dimension 1 and so on.

The term z-ordering refers to the graphical representation of the trivial shuffle in a 2-dimensional data space. Additionally, any interleaving sequence which does not change the internal bit sequence in any of the data values is possible. The authors mention several application areas for z-ordering, e.g., binary trees, AVL trees or B^+-trees.

Faloutsos proposes in [Fal86] an interesting modification to point mapping schemes. In particular, the use of Gray code representations instead of standard binary code representations is investigated. The author shows that Gray codes yield a better physical clustering of the records in the storage chunks. Since 1984, bit interleaving techniques are widely used as a standard tool for the design of multikey index data structures. For example, the BANG file approach as outlined in Section 3.4.3 relies on the trivial shuffle used in [OM84].

4

TYPE HIERARCHY INDEXING

4.1 PROBLEM DESCRIPTION

A major challenge for search data structures is *type hierarchy indexing.* In contrast to the situation in relational databases, any query in an OODB may refer either to one type or, implicitly, to a subhierarchy and therefore to a set of types. In this chapter, the specific requirements, the technical issues, and some selected data structure solutions in the context of type hierarchy queries are discussed.

Figure 4.1 shows a part of the type hierarchy of our running example. As some type indexing approaches support multiple inheritance hierarchies and others do not, we use a multiple inheritance pattern (TeachingAssistant being subtype of Student as well as of Staff) when appropriate. The optional inheritance relation is depicted by a dotted line in the figure. In the following example queries, the central issue in this context is the implicit reference to subhierarchies:

Q_1 select x from person x
where x.income < 10000

Q_2 select x from academic x
where x.income > 15000 and x.income <= 31000

Q_3 select x from assistantProfessor x
where x.income < 45000

Q_4 select x from person x
where x.income = 30000

Q_5 select x from facultyMember x
where x.income = 35000

T. A. Mueck et al., *Index Data Structures in Object-Oriented Databases*

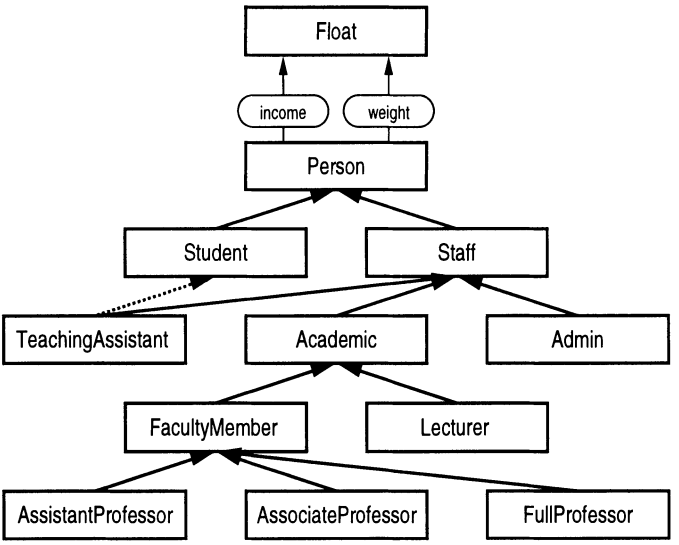


Figure 4.1 Part of the schema graph of the running example

Q_6 select x from associateProfessor x
where x.income = 40000

Q_7 select x from person x
where x.type = Person and x.income > 5000 and x.income <= 25000

With regard to the queried attribute income, queries Q_1–Q_3 are range queries and queries Q_4–Q_6 are exact match queries. Queries Q_1 and Q_4 are directed to the extent of the root type Person and therefore qualify objects of all types in the hierarchy. Queries Q_2 and Q_5 qualify types of a subhierarchy, queries Q_3 and Q_6 qualify only single (leaf) types. Query Q_7 is a range query that qualifies objects of the single (non-leaf) type Person without qualifying objects of Person's subtypes.

In what follows, a small object collection is used as an example data set for indexing purposes. Figure 4.2 contains a tabular representation of parts of this data set (type name and attributes) which is a possible extension for the type hierarchy given in Figure 4.1. This representation resembles the user view on the database. The physical storage organization of a particular implementation may differ from this view. The evaluation of the example queries for this dataset produces the following sets of object identifiers as query results: Q_1: $\{\omega_7, \omega_8, \omega_9, \omega_{10}\}$, Q_2: $\{\omega_{16}, \omega_{17}, \omega_{18}, \omega_{30}\}$, Q_3: $\{\omega_{16}, \omega_{17}, \omega_{18}\}$, Q_4: $\{\omega_{12}, \omega_{13}, \omega_{14}, \omega_{16}, \omega_{17}, \omega_{30}\}$, Q_5: $\{\omega_{31}\}$, Q_6: $\{\omega_{19}\}$, Q_7: $\{\omega_8, \omega_{10}\}$.

OID	*type*	income	weight
ω_8	Person	9000	130
ω_9	Person	4000	80
ω_{10}	Person	8000	85
ω_{12}	Student	30000	75
ω_{13}	Student	30000	71
ω_{14}	Student	30000	72.5
ω_6	Admin	20000	51.5
ω_7	Admin	8000	52.3
ω_{16}	AssistantProfessor	30000	72.3
ω_{17}	AssistantProfessor	30000	81.5
ω_{18}	AssistantProfessor	29000	88
ω_{19}	AssociateProfessor	40000	41.1
ω_{30}	AssociateProfessor	30000	75
ω_{31}	AssociateProfessor	35000	69

Figure 4.2 Some instances for the database schema in Figure 4.1

Despite the conceptual importance of type hierarchies in OODB, the corresponding indexing problem as outlined above is actually a special case of the so called multiple set indexing problem. In particular, the set of types $T = \{t_1, \ldots, t_{|T|}\}$ induces a set of type extents $\mathrm{EXT}(T) = \{\mathrm{Ext}(t_1), \ldots, \mathrm{Ext}(t_{|T|})\}$. From this point of view, the actual type hierarchy indexing problem is to implement fast access to a partially ordered set of sets $(\mathrm{EXT}(T), \subseteq)$ with order relation $\subseteq$ denoting set inclusion. However, omitting the partial order $\subseteq$ yielded by the type hierarchy, the remaining set of sets could be indexed as well. Consequently, an additional evaluation criterion for type hierarchy indices is their applicability to the more general problem of multiple set indexing. More precisely, an index data structure for type hierarchy indexing may or may not support multiple set indexing. In the former case, an additional question is whether or not the data structure provides special means for the handling of the special case, i.e., type hierarchy indexing. In the following we provide for each proposal a short summary also referring to these questions as well as to the question of multiple inheritance support.

Focusing on the data structure implementation of a type hierarchy index, there are two design approaches: key grouping versus type grouping approaches (see also [KM94b]):

- Data structures based on key grouping maintain a top-level data structure organization for key values. All object identifiers with the same key value

(e.g., all OIDs of Person objects with income = 10000) may or may not be further organized with respect to their types.

- Type grouping structures also maintain an asymmetric data organization, however, in this case the first-level order criterion is the object type (e.g., all objects of type Person are stored in one search data structure) and second-level search structures support key value access.

This design decision has a major influence on the resulting I/O performance in case of exact match and range queries, independently of the actual data structure implementation. In general, a type hierarchy index with key grouping supports exact match queries better than range queries, whereas for type grouping structures the reverse is true. Figure 4.3 depicts the relationship be-

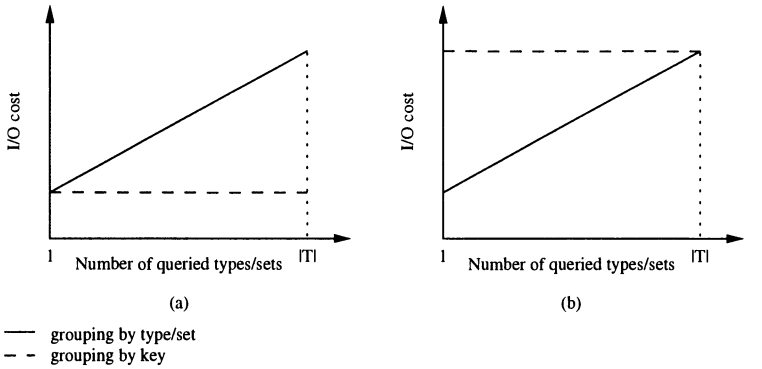


Figure 4.3 Efficiency of type grouping and key grouping with respect to (a) exact match queries and (b) range queries [KM94b]

tween key grouping and type grouping on the one hand and exact match and range queries on the other hand (resulting performance measured in number of I/O operations).

Using key grouping, the number of qualified types does not influence the I/O cost of a particular query, since the number of I/O operations is determined only by the range specified for the indexed attribute. On the contrary, in case of type grouping, the number of qualified types is the main performance factor. For each qualified type, usually a type-specific data structure has to be scanned for matching key values. The relation between key grouping and type grouping performance is interesting:

- Considering a particular exact match query, the I/O cost for any key grouping structure will be constant and smaller or equal than the I/O cost for any type grouping approach (equality case: number of qualifying types = 1).
- On the contrary, for a particular range query, the I/O cost for any key grouping structure will still be constant, however, larger or equal than the I/O cost for any type grouping approach (equality case: number of qualifying types = $|T|$)

Subsections 4.2 and 4.3 contain for both approaches (type grouping and key grouping) the initial proposals followed by recent and therefore more advanced approaches.

4.2 TYPE GROUPING

The straightforward type grouping approach of maintaining one data structure for each indexed type called single class index [KKD89] is outlined in Subsection 4.2.1, followed by a description of the more recent H-tree [LOL92].

4.2.1 Initial approach – SC-Index

An early technique used in the ORION system is christened *single class index* in [KKD89]. Using single class indexing, the index creation for an attribute of type t requires the construction of one search structure for each type in t's subhierarchy. Although in [KKD89] B^+-trees are the search data structure of choice, this indexing framework is not restricted to a particular search data structure. Executing a query specification involving the indexed hierarchy requires a traversal of all these search data structures, which are referred to as SC-index components. This point should be clarified by means of our example schema. In order to construct a single class index for Person and its subtypes based on attribute income, eleven search structures have to be maintained, i.e., one data structure for each type in the inheritance subgraph rooted at Person.

Assuming a B^+-tree implementation with leaf node chaining, one of these B^+-trees could look like the multiway tree depicted in Figure 4.4. The inner nodes contain interval boundary values like in any B^+-tree, the leaf node organization proposed in [KKD89] can handle variable length keys by means of the key length

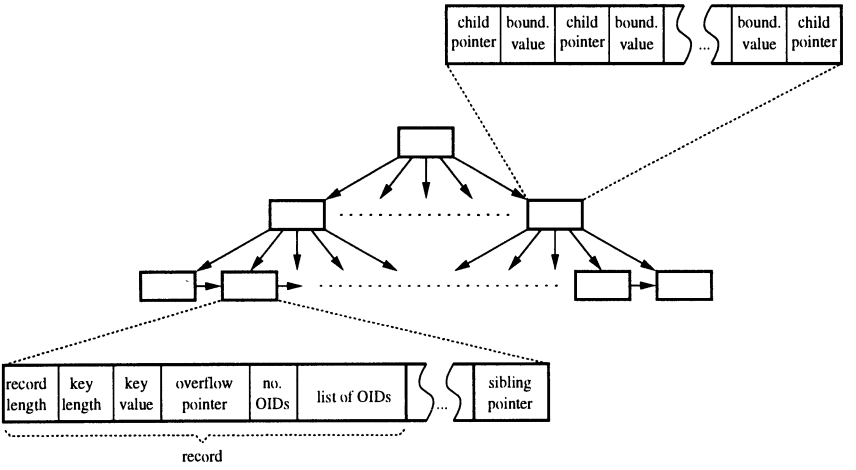


Figure 4.4 B$^+$-tree implementation of an SC-index component

field, which is omitted for fixed length keys. Without further consideration of the key length field, one leaf node record corresponds to one key attribute value (e.g., 30000 for attribute income in Figure 4.5) and contains a list of OIDs of objects with matching attribute values. The overflow page pointer is used for records which are larger than a leaf node. Referring to our example database the single class index B$^+$-tree for AssistantProfessor could look like Figure 4.5.

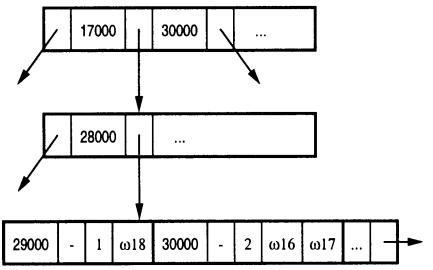


Figure 4.5 SC-index component for AssistantProfessor

Considering the B$^+$-tree structure in Figure 4.4 and query Q_2, Figure 4.6 shows the resulting traversal. Fetched nodes are shaded and numbered according to the retrieval sequence. A closer look at our seven example queries yields the following retrieval patterns.

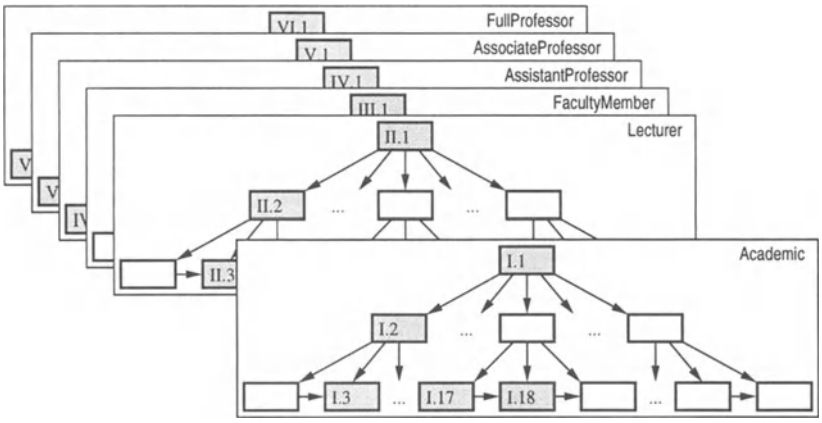


Figure 4.6 Traversal for query Q_2

Q_1 For each qualified type (11) the corresponding B^+-tree is traversed down to the leftmost leaf node and the leaf node chain followed up to the first value larger than 10000.

Q_2 For each qualified type (6) the corresponding B^+-tree is traversed in order to locate the values larger than 15000 and the leaf node chain followed up to the first value larger than 31000.

Q_3 The tree for AssistantProfessor is traversed down to the leftmost leaf node and the leaf node chain followed up to the first value larger than or equal to 45000.

Q_4 For each qualified type (11) the corresponding B^+-tree is traversed down to the leaf node containing 20000 (if stored).

Q_5 For each qualified type (4) the corresponding B^+-tree is traversed down to the leaf node containing value 35000 (if stored).

Q_6 The B^+-tree for AssociateProfessor is traversed down to the leaf node containing value 40000 (if stored).

Q_7 The B^+-tree for Person is traversed down to the leaf node containing value 5000 and the leaf node chain is followed up to the first leaf node containing a value larger than 25000.

Summary

Pros and cons Queries implicitly referring to large subhierarchies yield a traversal of a large number of B^+-trees. Although several trees are traversed, the positive aspect is that all retrieved OIDs qualify with regard to object type. In other words, using an SC-tree we do not fetch OIDs that have to be subsequently discarded due to non-matching type. A typical favorable case is an exact match or a range query over one type like Q_6 or Q_3. Exact match queries over more than one type like Q_4 or Q_5 are not favorable.

Supported hierarchy features The SC-index is applicable in multiple inheritance environments and can handle abstract types.

Multiple sets Although not suggested by the authors it can be used for indexing multiple sets as well. There are no provisions to take advantage of the relationship of the indexed sets in case of type hierarchy indexing.

4.2.2 An advanced approach – H-tree

A H-tree [LOL92, LLOH91] consists of a set of nested B^+-trees. These B^+-trees are referred to as H-tree components in the sequel. The main difference between the SC-index and the H-tree is that the former maintains a set of isolated type specific B^+-trees, whereas the latter maintains a nesting structure for these B^+-trees. This nesting reflects the structure of the indexed type hierarchy. In particular, the H-tree component of the indexed type is *nested* (see below) with the H-trees of the immediate subtypes of the indexed type, the H-trees of these types are nested with the H-trees of their respective subtypes and so forth. Thus an H-tree index for an attribute in an inheritance subgraph is a hierarchy of H-trees nested according to the supertype-subtype relation. The references used to establish the nesting are called nested tree pointers and are used for shortcuts during query execution.

The motivation behind index nesting is to avoid *full* scans of *each* H-tree component when a number of types in the indexed hierarchy is queried. In particular, when scanning the trees of a type and its subtypes, we need to perform a full search in the H-tree component of the supertype (called outer component) and only partial searches in the H-trees of the subtypes (called inner component). This restriction of the traversal process is the major advantage of the H-tree compared to the SC-index. In [LOL92] a number of correctness and efficiency conditions for the nesting structure is given. In particular, a correct nesting of an outer H-tree component H_{super} with an inner H-tree component H_{sub} is characterized by two conditions:

- Let n_r be the value range of node n defined by the boundary values of its parent node. If a node n in H_{sub} is referenced by a node N in H_{super}, each value of n_r must be also an element of N_r, except when N is the root node of H_{super}.
- All leaf nodes of H_{sub} must be reachable using nested tree pointers in H_{super}. In the terminology of [LOL92] this means that all leave nodes in H_{sub} must be covered by H_{super}.

Figure 4.7 shows the organization of a H-tree component. H-tree leaf nodes closely resemble single class index leaf nodes. In the figure, each leaf node record holds one value of the indexed attribute, a counter and a list of object identifiers of objects holding this value. Inner nodes contain boundary values and child pointers to successor nodes like in traditional B^+-trees. The link

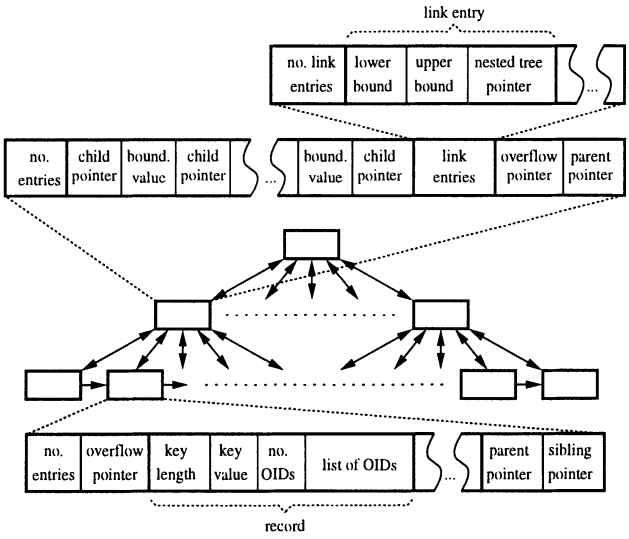


Figure 4.7 H-tree structure

entries implement the nesting feature. Each link entry consists of a pointer (nested tree pointer) to a subtree of the H-tree component of a subtype and the boundary values of this nested subtree. As the number of link entries entries per node is not restricted, overflow pages may be necessary to store additional pointers. In several cases during query processing the traversal has to access the predecessor of a particular node. This kind of access is supported by parent pointer entries in the H-tree nodes.

Figure 4.8 shows a few nodes of two components of a H-tree index created for attribute income of type Person and its subtypes. The figure shows one link

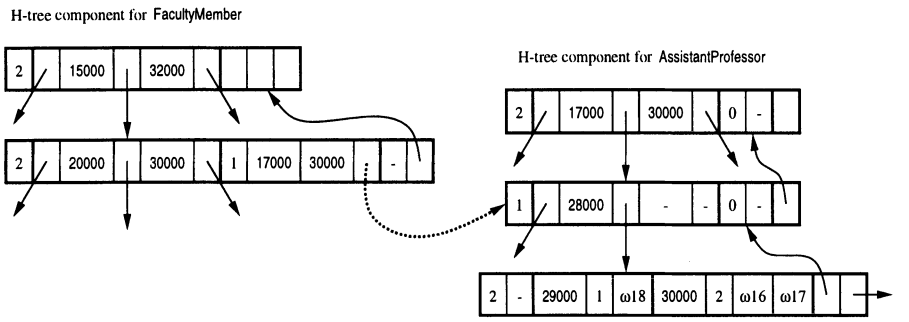


Figure 4.8 H-tree components for FacultyMember and AssistantProfessor

entry with a lower bound of 17000 and an upper bound of 30000, the nested tree pointer refers to a node of the H-tree component for type AssistantProfessor. Figure 4.9 shows the H-tree traversal for query Q_2.

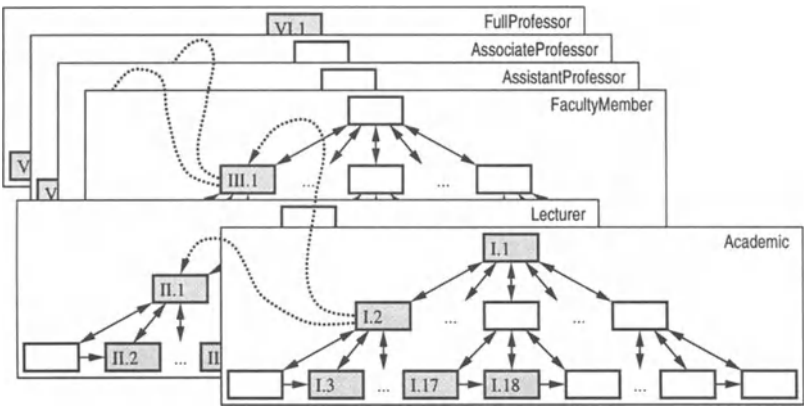


Figure 4.9 Traversal for query Q_2

Q_1 The H-tree component for Person is traversed down to the leftmost leaf node. The leaf node chain is followed up to the first value larger than 10000. As with the SC-index the components of all qualified types (11) have to be traversed. Matching nested tree pointers encountered during the traversal of the Person component are used to access the inner nodes

of the components for Student and Staff. In several cases the higher level nodes of the inner trees need not be traversed. While traversing the H-tree component for Staff the same cross-referencing scheme is used for its subtypes.

Q_2 The same procedure as for query Q_2 is applied with the exception that the search starts at the root of the H-tree component for Academic.

Q_3 The H-tree component for Student is traversed down to the leftmost leaf node and the leaf node chain followed up to the first value larger than or equal to 45000.

Q_4 Eleven components have to be traversed (as with query Q_1) down to the leaf node containing value 30000, starting with the component for Person. Using the nested tree pointers may help to exclude higher level nodes of inner components from the traversal.

Q_5 Four components have to be traversed down to the leaf node containing value 35000, starting with the component for FacultyMember. Using the nested tree pointers may help to exclude higher level nodes of inner components from the traversal.

Q_6 The H-tree component for AssociateProfessor is traversed down to the leaf node containing value 40000 (if stored).

Q_7 The H-tree component for Person is traversed down to the leaf node containing value 5000 and the leaf node chain followed up to the first leaf node containing a value greater than 25000.

Summary

Pros and cons Compared to the SC-index, the positive aspect of the H-tree approach is the exclusion of a certain number of inner tree nodes from the tree traversal during query processing. The problems are a decreased node fanout due to the space requirements of the link entries on the one hand and complex query and update algorithms on the other hand.

Supported hierarchy features Although not mentioned in the original publications, dealing with multiple inheritance is possible. However, it requires some modifications to the proposed algorithms, for example some kind of marking of already visited nodes in the query algorithms. Support for abstract types requires some modifications of the algorithms as well.

Multiple Sets The nesting structure of the H-tree is designed for type hierarchies and is not suitable for the general case of indexing multiple sets. When omitting the nesting structure access support for multiple sets is possible, however, the resulting index structure is a single class index with decreased node fanout.

4.3 KEY GROUPING

The first approach in this context is called class hierarchy index [KKD89]. The outline of this index structure is followed by a description of the more recent class division method [RK95]. A common property of both approaches is the use of B^+-trees.

4.3.1 Initial approach – CH-Index

In contrast to the single class index approach, a *class hierarchy index* [KKD89] maintains only one search structure for all objects all types of the indexed hierarchy. The class hierarchy index permits efficient single-scan access to the instances of all types of the indexed hierarchy. It outperforms single class indices in several cases, i.e., when queries aim at the indexed type and all its subtypes or at least at a major subset of the indexed hierarchy. However, if only a few types of the indexed hierarchy are hit by a query, the single class index performs better. In this case, only a few (small) indices have to be scanned, in particular one for each type in the query scope.

Similar to the SC-index B^+-trees are used as underlying search data structure, the format of non-leaf nodes is the same for both proposals (see Figure 4.10). A leaf node record of a class hierarchy index consists of record length, key length (for variable length keys), key value, overflow page pointer, and the list of OIDs of objects holding the key value in the indexed attribute. The OIDs in the list are grouped by type. The type directory contains the offset for each type having objects in the list of OIDs thus supporting intra-node lookup. This organization is motivated by the fact that the B^+-tree contains OIDs of instances of *all* types in the inheritance subgraph. If the scope of a query is a subtype of the indexed type, only the relevant portions of the leaf nodes need to be scanned.

Consequently, a class hierarchy index for attribute Person.income of our example database consists of one B^+-tree for all instances of all types of the inheritance

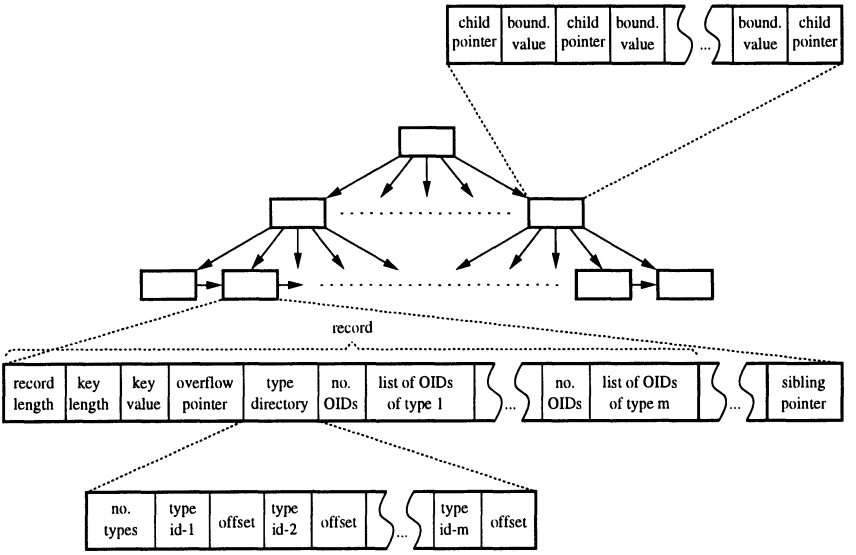


Figure 4.10 CH-tree structure

hierarchy rooted at type Person. Figure 4.11 shows parts of the B^+-tree and one leaf node of this index with entries for the instances given in the example database. The length of attribute income is assumed to be four bytes, counters and offsets need two bytes, type identifiers and page pointers four bytes, and object identifiers eight bytes. The key length component is not used because a fixed-length attribute is indexed. In order to answer a query only one B^+-tree

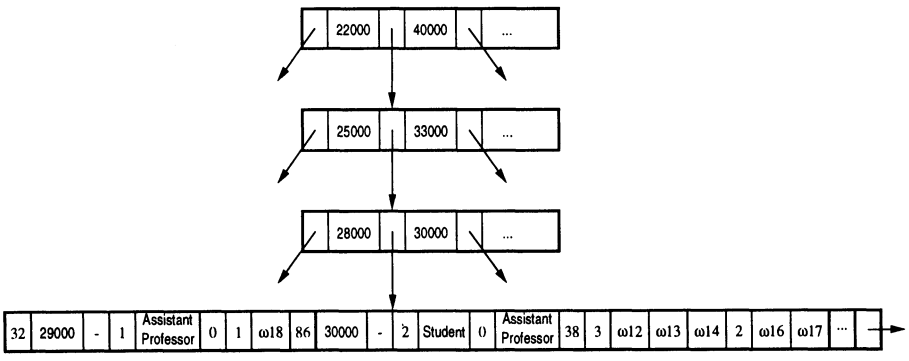


Figure 4.11 Example leaf node in class hierarchy index

has to be scanned. If a query refers to a large number of types, only a small fraction of the OIDs found during the index traversal have to be discarded and vice versa. Considering the index traversal depicted in Figure 4.12, the same number of B^+-tree nodes has to be fetched for a range query execution

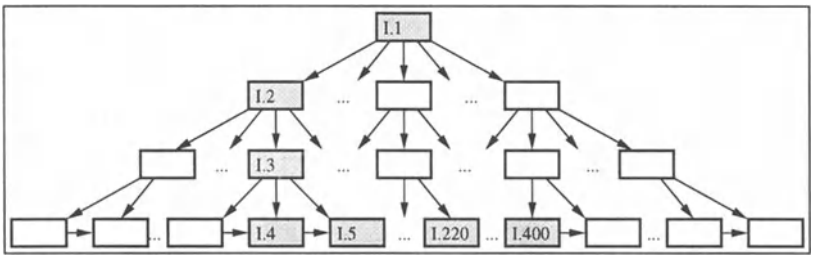


Figure 4.12 Traversal for query Q_2

regardless of the number of relevant types. There are two extreme cases: if all types are qualified by the query no OIDs will be out of query scope, whereas if a query refers only to one type probably a large number of OIDs are fetched in vain. This relationship between the number of qualified types and the I/O efficiency is further outlined by a closer look at our example queries.

Q_1 The CH-tree is traversed down to the leftmost leaf node and the leaf node chain followed up to the first value larger than 10000. In this case, all OIDs encountered during the traversal are qualified by the query with respect to their type.

Q_2 The CH-tree is traversed to the leaf node containing the value 15000 and the leaf node chain is followed to the first node containing a value larger than 31000. Although the type directory in the leaf nodes allows to ignore OIDs of non-qualified types, a waste of I/O bandwidth is inevitable, because the nodes read contain data not qualified by the query.

Q_3 The execution yields the same situation as for query Q_2. A further performance degradation due to an even worse ratio between data fetched and data qualified by the query will appear.

Q_4 The CH-tree is traversed down to the leaf node containing value 30000 (if stored). Ignoring overflow pages in our discussion, a single leaf node access allows to answer the query.

Q_5 The CH-tree is traversed down to the leaf node containing value 35000 (if stored). Again, a single leaf node access is sufficient to answer the query.

The type directory provides direct access to the relevant portions of the leaf node (i.e., the portions containing the OIDs of qualified types).

Q_6 The CH-tree is traversed down to the leaf node containing value 40000 (if stored). Again, a single leaf node access is sufficient to answer the query. Only AssociateProfessor objects are qualified by the query. The type directory provides fast access to the list of OIDs of this type.

Q_7 Same procedure as for query Q_3 yielding the same performance problems.

Summary

Pros and cons Using this search structure, the complete B^+-tree has to be scanned in *all* cases, even if most of the entries in the leaf nodes have to be discarded in a subsequent processing step. In case of a query over the full type hierarchy the CH-index is an attractive proposal for both, range queries and exact match queries. If the query refers to a subhierarchy or to single types, exact match queries still perform well, whereas range query performance degrades drastically.

Supported hierarchy features As the structure of the type hierarchy is not reflected by the CH-index, there is no need for modifications to support multiple inheritance. The same is true for abstract types.

Multiple Sets Although not taken into account in the original publications CH-indices are suitable for multiple set indexing as well.

4.3.2 An advanced approach – Class Division

The class division method (CD-index) introduced by Ramaswamy and Kanellakis in [RK95] is based on previous results in [KRVV93] and proposed as an extension to the CH-index. The central idea behind class division is to find a compromise between

- storing (indexing) for each type its instance set (i.e., the set of all instances of this type) and
- storing (indexing) for each type its extent (i.e., the set of all instances of this type and of all its direct or indirect subtypes).

begin basic-class-division $(T, \leq)$
  sort $(T, \leq)$ pre-order giving $t_1, t_2, \ldots, t_{|T|}$
  let $T_1^1, T_2^1, \ldots, T_{|T|}^1$ be the 1^{st}-level family of sets of types
    with $T_i^1 = \{t_i\}$ for $1 \leq i \leq |T|$
  for i from 1 to $\lceil \log_2 |T| \rceil + 1$ **do**
    create a family of sets of types $T_1^i, T_2^i, \ldots$ at i^{th} level
    merge T_1^i with T_2^i to get T_1^{i+1}, T_3^i with T_4^i to get T_2^{i+1}, etc.
    if there are odd number of sets in family, let $T_{l_i}^i$, the last set at round i
      be $T_{l_{i+1}}^{i+1}$, the last collection at round $i+1$
  end
  family of set of types at all levels forms $\boldsymbol{G}$
  the levels give this family a binary tree structure
end basic-class-division

Figure 4.13 Basic class division algorithm [RK95]

begin prune-space $(T, \leq)$
  construct $\boldsymbol{G}$ by basic-class-division
  for each type t in T, cover the set of t's subtypes in $(T, \leq)$ with maximal
    subtrees in $\boldsymbol{G}$'s binary tree structure such that the cover is exact
  remove any sets in $\boldsymbol{g}$ not used in covers
  let $\boldsymbol{G}$ be the resulting family of sets of types
end prune-space

Figure 4.14 Prune space heuristic [RK95]

In the SC-index each instance set is stored in a distinct data structure, whereas in the CH-index the union of all instance sets is stored in one data structure. Bearing in mind that query specifications refer to type extents we observe that in either case the relevant type extents have to be reconstructed during data structure traversal. With respect to query processing, storing type extents explicitly yields enhanced query performance on one hand, and significant storage space overhead on the other hand. The class division method elaborates on these two conflicting aspects. In particular, it maintains for an indexed type hierarchy a specific family of type sets $\boldsymbol{G} \subset \mathcal{P}(T)$ in such a way that one search data structure is used per type set. $\boldsymbol{G}$ is constructed by *class division*, more specifically by a simple hierarchy regrouping algorithm called *basic-class-division* with a subsequent heuristic improvement step (*prune-space*) as one alternative and by a more elaborated heuristic called *rake-contract* as an additional alternative. The results of both alternatives are evaluated with respect to storage space overhead and query performance. The relative merit of each alternative is described by two parameters r and q. These two parameters are upper bounds for the storage redundancy and the extent construction effort

begin rake-contract $(T, \leq)$
construct $\boldsymbol{G}$ by basic-class-division and space pruning
build a family of sets of types $\boldsymbol{J}$ by using rake and contract on $(T, \leq)$
fully replicate on raking leaves
use basic-class-division with space pruning for contracting paths
alternate rake and contract steps until the hierarchy is fully processed
if $\boldsymbol{J}$ improves replication or query factor replace $\boldsymbol{G}$ by $\boldsymbol{J}$
let $\boldsymbol{G}$ be the resulting family of sets of types
end rake-contract

Figure 4.15 Rake-contract heuristic [RK95]

during query execution, respectively. More precisely, the optimization problem is to find for given r and q a family of type sets $\boldsymbol{G}$ in such a way that

- no type is contained in more than r type sets in $\boldsymbol{G}$, in other words $\forall t \in T : |\{g \in \boldsymbol{G} : t \in g\}| \leq r$
- for each type t there is a subset of $\boldsymbol{G}$ containing at most q type sets in such a way that the union of these type sets yields exactly $T_{\leq t}$ (the subhierarchy rooted at t), in other words $\forall t \in T : (\exists \boldsymbol{G'} \subseteq \boldsymbol{G} : \bigcup_{\forall g' \in \boldsymbol{G'}} g' = T_{\leq t} \ \wedge \ |\boldsymbol{G'}| \leq q)$

At the object level the replication factor r defines the maximum number of data structures in which a particular OID is stored. The corresponding query factor q equals the maximum number of data structures to be traversed in order to retrieve one type extent. In [RK95] it is shown that for arbitrary single inheritance type hierarchies at least one $\boldsymbol{G}$ exists with $r = \lceil \log_2 |T| + 1 \rceil$ and $q = 2 \cdot \lceil \log_2 |T| \rceil$. With respect to the sets of stored OIDs $\Omega(B)$ in a data structure B and the function Inst : $T \rightarrow \mathcal{P}(O)$ denoting the set of instances of a particular type, the relationship between SC-index, H-tree, CH-index and CD-index can be outlined as follows:

- SC-index and H-tree: for each type t_i there is one B$^+$-tree B_i such that $\Omega(B_i) = \text{Inst}(t_i)$.
- CH-index: there is a single B$^+$-tree B such that $\Omega(B) = \bigcup_{t_i \in T} \text{Inst}(t_i)$.
- CD-index: for each type set $g_i \in \boldsymbol{G}$ there is one B$^+$-tree B_i such that $\Omega(B_i) = \bigcup_{\forall t_j \in g_i} \text{Inst}(t_j)$. A particular extent $\text{Ext}(t_i)$ is given either by a single $\Omega(.)$ or by a union of at most q sets $\Omega(.)$

The relationship between $\boldsymbol{G}$, q and r on the one hand and the two hierarchy decomposition alternatives (basic-class-division with prune-space heuristic vs. rake-contract heuristic) on the other hand is illustrated by the example shown in Figures 4.16a–4.16c (taken from [RK95]). In particular, Figure 4.16a shows

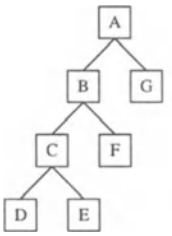


Figure 4.16a Type hierarchy

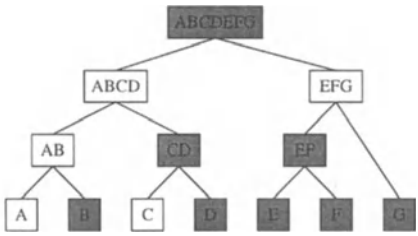


Figure 4.16b Result of basic class division with space pruning

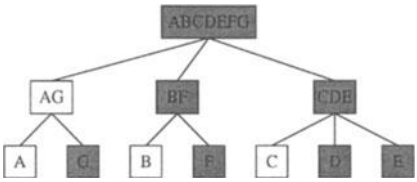


Figure 4.16c Result of rake-contract heuristic

the initial type hierarchy used as input for the class division algorithms. The result of basic-class-division and the subsequent prune-space step is shown in Figure 4.16b. Grey shaded nodes represent elements of $\boldsymbol{G}$, which means that for each of these nodes one B^+-tree is maintained. Considering the eight search data structures resulting from this decomposition we observe that each type occurs in at most three type sets (D, E, and F occur in three type sets, all other types in less than three). For each extent at most three search data structures have to be consulted (Ext(A): ABCDEFG, Ext(B): B + CD + EF, Ext(C): CD + E, extents of leaf nodes in the hierarchy always need only one component), therefore the decomposition is characterized by $r = 3$ and $q = 3$. A closer look at the result of the rake-contract heuristic (given in Figure 4.16c) shows an unaltered redundancy factor $r = 3$ but an improved query factor $q = 2$ (Ext(A): ABCDEFG, Ext(B): BF + CDE, Ext(C): CDE, leaf nodes as above). With respect to query execution a few observations can be made regarding the actual use of the resulting search data structures:

- One type set always corresponds to T. Reconsidering the previously discussed index proposals we observe that the CD-index is a direct extension of the CH-index in so far that the one of the components of the CD-index is always a CH-index. Taking into account the above mentioned merits of the CH-index this largest component of the CD-index will be used for exact match queries referring to more than one type and for range queries referring to all types in the hierarchy.
- Each leaf node of the type hierarchy always corresponds to one element of $\boldsymbol{G}$, i.e., a singleton type set. These CD-index components implemented for leaf node types are used for all queries (range as well as exact match) referring explicitly to leaf node types. In the framework of the previously discussed approaches these components closely resemble SC-index components for the respective types.
- For all query patterns not covered by the CD-index components mentioned in the previous items, in particular, for range queries referring to more than one but less than $|T|$ types, more than one CD-index component has to be traversed. However, at most q components are necessary to process any type extent.

Returning to the type hierarchy of our running example (excluding the unsupported case of multiple inheritance) we evaluate both decomposition alternatives. Basic-class-division with subsequent space pruning yields the situation depicted in Figure 4.17. An implementation based on these results contains fourteen index components, for the complete CD-index the redundancy factor $r = 4$ and the query factor $q = 3$. Again, the rake-contract heuristic yields an improvement, the result depicted in Figure 4.18 contains only eleven index components, redundancy factor is decreased to $r = 3$. With respect to the example queries Figure 4.19 shows the component traversals for query Q_2. Considering the extent of type Academic components 3 and 4 have to be consulted since the union of the corresponding type sets contains exactly the types of the subhierarchy rooted at Academic. During query execution there is no waste of I/O bandwidth, because all OIDs encountered during the tree traversals are part of the query result. At this point the well-known tradeoff between query performance and storage space overhead becomes obvious again. Query execution in a CD-index is further illustrated below.

Q_1 Component 1 is used to answer this query just like in the CH-tree (which is equivalent to component 1).

Q_2 See discussion above.

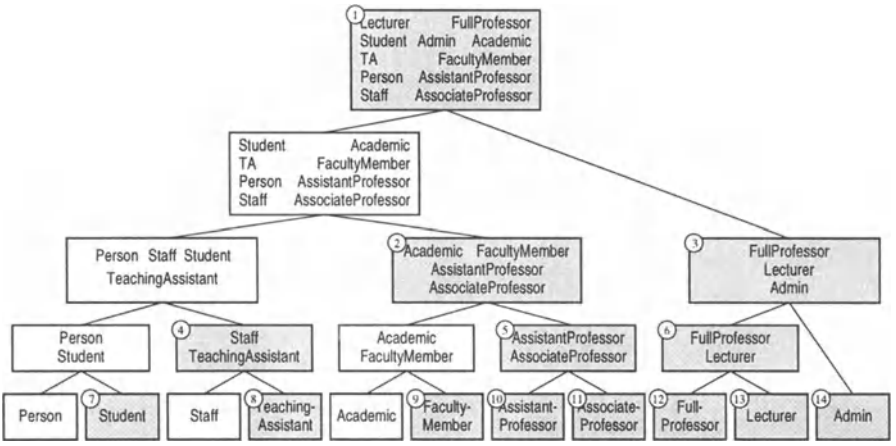


Figure 4.17 Decomposition of the example hierarchy after basic class division and subsequent space pruning

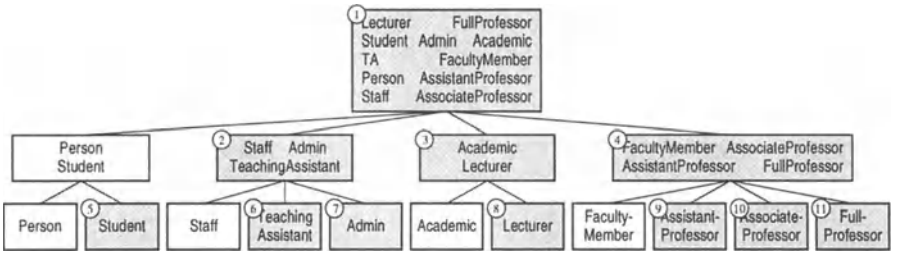


Figure 4.18 Decomposition of the example hierarchy after rake-contract

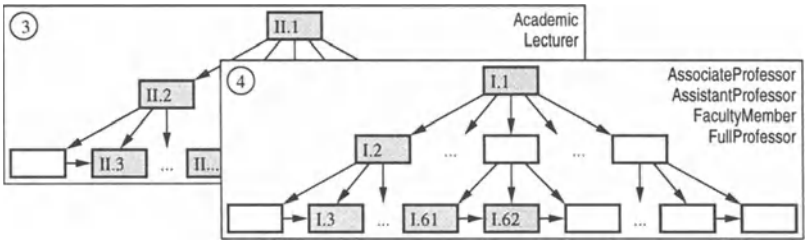


Figure 4.19 Traversal for query Q_2

Q_3 Component 9 (resembling a traditional SC-index component) is traversed to answer this query. A traversal of the chained leaf nodes is done.

Q_4 A single leaf node of component 1 needs to be fetched (not taking into account possible overflow pages).

Q_5 A single leaf node of component 4 needs to be fetched (again not taking into account possible overflow pages).

Q_6 A single leaf node of component 10 needs to be fetched.

Q_7 Component 1 (the CH-index equivalent) has to be traversed to answer this query. As this component contains OIDs of all indexed types the ratio between data fetched and data qualified by the query is unfavorable (as with the CH-index). This is in strong contrast to all other queries above, where only OIDs qualified by the query (with respect to type) were read during component traversals.

Summary

Pros and cons The CD-index is a recent proposal combining the advantages of both the CH-index and the SC-index. All types of queries referring to extents are efficiently supported. Queries referring to instance sets of inner nodes of the type hierarchy (and therefore not referring to extents) yield less efficient executions. This problem is also addressed in [RK95], the authors propose to include additional components to handle these cases. Storage space overhead cannot be neglected even for small redundancy factors. Additional parameters influencing the storage space overhead are the cardinality of the type hierarchy $|T|$ as well as the extent sizes of the indexed types. In case of frequent updates the synchronization overhead caused by the redundant data sets remains an open issue.

Supported hierarchy features Support for multiple inheritance is mentioned in [RK95], however, the authors do not elaborate on the necessary modifications. The CD-index is able to handle abstract types, but the heuristics do not include specific provisions for this case. As a result, obsolete index components may occur. If, for example, Academic is an abstract type, component 3 in Figure 4.18 contains the same OIDs as component 8 (the same is true for components 4 and 8 in Figure 4.17 if Staff is an abstract type).

Multiple sets Since the CD-index as presented in [RK95] relies on the partial order defined by set inclusion of the type extents, the adaption of this proposal to the more general case is not totally straightforward.

4.4 MULTIKEY TYPE INDEX

Like the CD-index, the multikey type index aims as a compromise between key grouping and type grouping approaches. In particular, the CD-index tries to avoid the inevitable deficiencies of pure key grouping and type grouping approaches by means of redundant storage. The multikey type index, however, interprets the type membership of objects as additional attribute of these objects. Thus it achieves symmetrical indexing of both attribute values and object types. An additional property of the multikey type index considered attractive in various application domains is its ability to support indexing of more than one attribute with a single search data structure.

4.4.1 Outline

The central idea is to incorporate the type hierarchy structure of a given database scheme into a standard multikey index implementation in such a way that the hierarchy is mapped to one of the multikey index domains. This special domain is called *type domain* in the sequel. The result is a *multikey type index* with $k + 1$ keys corresponding to the k indexed object properties and to one additional property representing type membership. The principle is shown in Figures 4.20a (4.20b). It contains one domain (two domains) for attribute

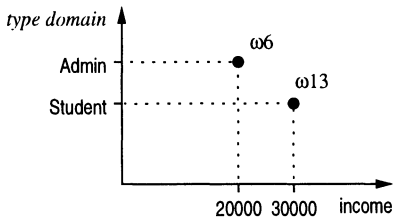


Figure 4.20a 2-dimensional data space of an MT-index for one indexed property (income)

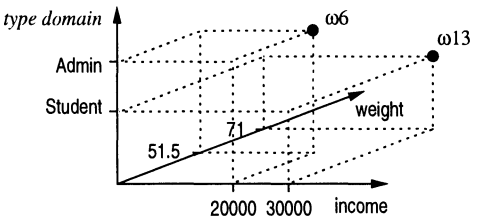


Figure 4.20b 3-dimensional data space of an MT-index for two indexed properties (income and weight)

income (attributes income and weight) and the type domain. Two objects are represented by dots in the resulting three-dimensional space. Any data structure implementation adhering to this kind of multidimensional pattern yields a balance of key grouping and type grouping as well as straightforward access support for several attributes at once. For example, queries like

Q_8 select x from facultyMember x

where x.income < 10000 and x.weight > 120

can be answered in this framework consulting one search data structure for both indexed attributes and the type domain. Such a query retrieves instances of FacultyMember as well as instances of its subtypes fulfilling the two range predicates on income and weight. Consequently, the example query contains in addition to the range predicates on the two attributes a third (implicit) predicate on the object type, i.e, o-type(x) $\leq$ FacultyMember, where $\leq$ denotes the partial order relation of the type hierarchy. Using an MT-index, such an implicit predicate can be mapped to a range predicate over the type domain.

4.4.2 Type hierarchy mapping

The rationale for the following algorithm is that the query performance of an MT-index is largely determined by the choice of the actual type hierarchy linearization. Figures 4.21 and 4.22 show the differences in the size of the actual query volumes based on different linearizations.

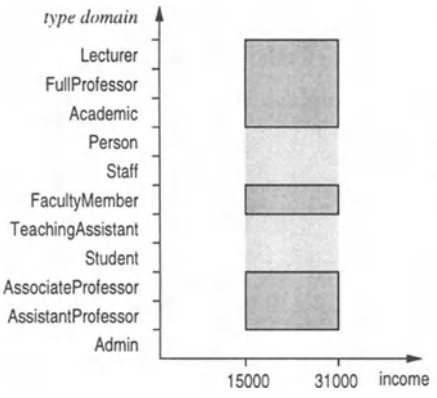


Figure 4.21 Query volume for Q_2 under suboptimal linearization

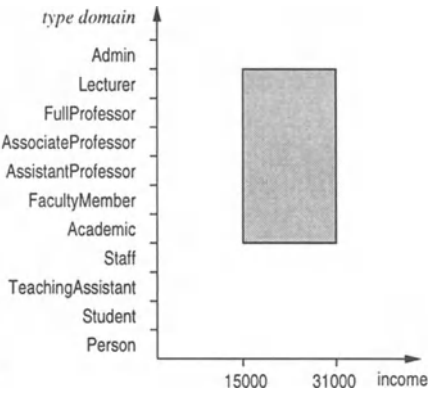


Figure 4.22 Query volume for Q_2 under optimal linearization

Assuming an arbitrary linearization, the query range in the type domain may also contain types not qualifying for the query request (see Figure 4.21). Since the resource consumption of a range query is positively correlated with the size of the respective range, we aim at minimal ranges for all extents (see Figure 4.22 for the extent of FacultyMember), i.e., at a linearization in such a way that exactly one interval contains all types which are part of one subhierarchy.

In geometrical terms, an optimal linearization yields for each possible type in a query a subspace not containing any object identifiers not belonging to the query result (see Figure 4.23 containing the queries given in Section 4.1). Consequently, a type domain setup (linearization) resulting in minimal query subspace volumes for *all possible query requests* is called *optimal.* More specif-

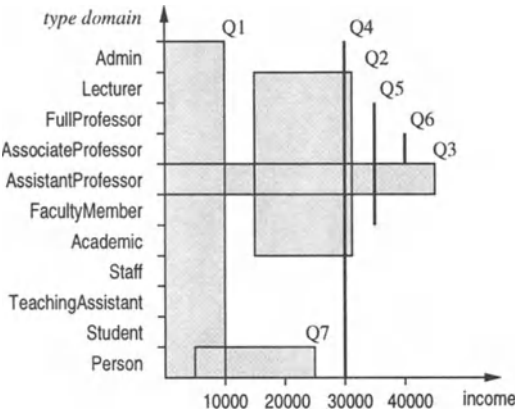


Figure 4.23 Regions of example queries

ically, an ordering $\sqsubseteq$ is optimal for $(T, \leq)$, if $\sqsubseteq$ is a total ordering (see (4.1) in definition below), and for each subhierarchy of $(T, \leq)$ (with $T_{\leq t}$ denoting the subhierarchy rooted at t), there is a closed interval $[u, v]$ in $(T, \sqsubseteq)$, containing the same elements (i.e., types) as $T_{\leq t}$ (see (4.2) in definition below).

Definition 4.1 (Optimal linearization)
Let $(T, \leq)$ be a type hierarchy and $T_{\leq t}$ be the subhierarchy rooted at t (i.e., the interval $]\infty, t]$ in $(T, \leq)$). An ordering $\sqsubseteq$ is called optimal linearization *for $(T, \leq)$, if*

$$\forall t, u \in T : \; t \sqsubseteq u \; \vee \; u \sqsubseteq t \quad \textit{and} \tag{4.1}$$

$$\forall t \in T : \; \exists u, v \in T_{\leq t} \; \textit{in such a way that} \; [u, v]_{(T, \sqsubseteq)} = T_{\leq t} \tag{4.2}$$

Obviously there are type hierarchies without optimal linearization. Figure 4.24 shows the smallest hierarchy without linearization. Although stating a necessary and sufficient condition for the existence of an optimal linearization of a type hierarchy is not totally trivial, a closer look at the above definition yields at least one necessary and one sufficient condition. In the conditions, super(t) denotes the set of direct supertypes of t:

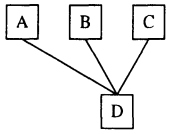


Figure 4.24 Hierarchy without optimal linearization

- An optimal linearization exists if each type has at most one supertype, i.e., in case of single inheritance. $\forall t \in T : |\,\text{super}(t)| \leq 1$ is sufficient.
- An optimal linearization does not exist if any type has more than two supertypes. $\forall t \in T : |\,\text{super}(t)| \leq 2$ is necessary.

In the case of single inheritance the computation of the optimal linearization is straightforward. For example, a standard depth-first traversal of the hierarchy will do (see [RK95]).

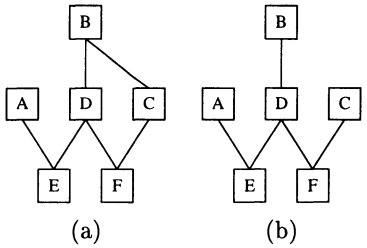


Figure 4.25 Type hierarchies (a) with and (b) without optimal linearization

In the multiple inheritance case Figure 4.25 illustrates that the second existence condition is only necessary. For both hierarchies depicted in this figure, the condition holds. However, a closer look at the two type hierarchies reveals that hierarchy (a) has an optimal linearization whereas hierarchy (b) has none. Informally, this result can be obtained by the isolation of all non-trivial subhierarchies, in particular {AE}, {CF}, {DEF}, {BCDEF} for (a) and {AE}, {CF}, {DEF}, {BDEF} for (b). The goal is a 'flattening' of the hierarchy such that the set of type identifiers forms a string and each subhierarchy is represented by a substring of this string. Drawing the corresponding set diagrams for the two hierarchies (see Figure 4.26) we observe that, in the first case, the diagram can be flattened in this way whereas in the second case this is not possible, since one of the subhierarchies cannot be represented by a substring (in Figure 4.26 this is {BDEF}).

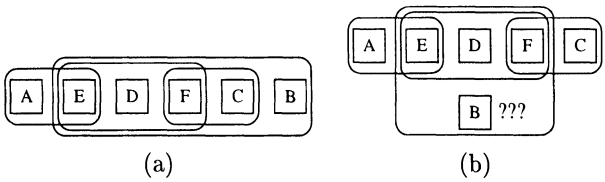


Figure 4.26 Set diagrams for type hierarchies of Figure 4.25

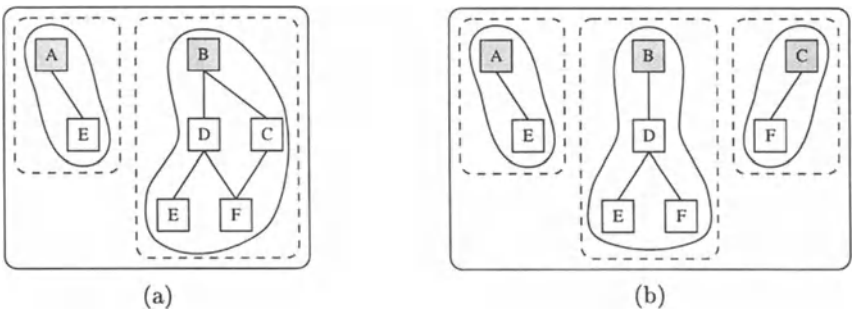
In the following section, we present an algorithm which is able to find all optimal linearizations for a given hierarchy $(T, \leq)$.

4.4.3 The mapping algorithm

Prior to an in-depth presentation of the mapping algorithm we use the example hierarchies of the previous section (see Figure 4.25) for an informal presentation of the linearization task.

Let $(S, \leq)$ denote the hierarchy to be processed.

1. The (unmarked) maximal elements of $(S, \leq)$ together with the subhierarchies rooted at these elements are determined and marked. For the running example the results for the two hierarchies are given as:

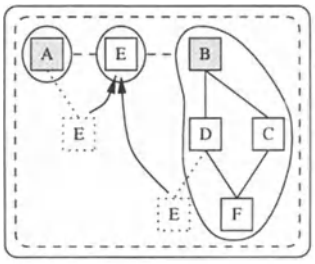


In what follows some notational conventions for the necessary data structures (abstract representations) hold:

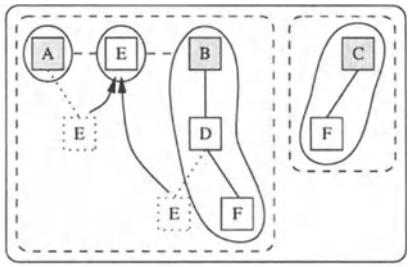
- squares represent single types
- marked types are shaded
- solid shapes represent sets

- dashed shapes symbolize lists

2. Non-disjoint subhierarchies are concatenated (operator $\circ$, see example Figure 4.27, exact definition below). If there are non-disjoint subhierarchies which cannot be concatenated (i.e., the intersecting parts are not located on either end of their respective lists) no linearization exists.

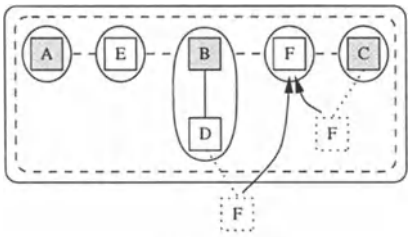


(a)



(b) Step 1

With respect to hierarchy (a) there is only one concatenation step (intersection contains E). The processing of hierarchy (b) involves two concatenation operations, one for an intersection containing E, the other one for an intersection containing F.

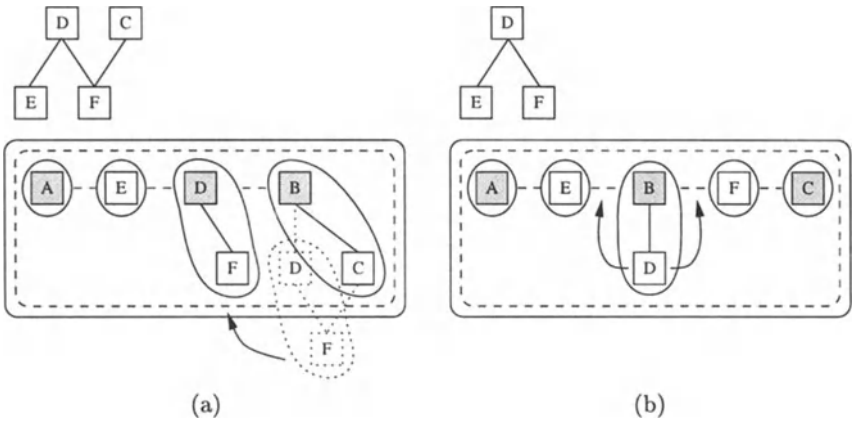


(b) Step 2

3. The lists resulting from the previous step are refined (operator $*$, see example Figure 4.28, exact definition below). More specifically, for each subhierarchy rooted at an unmarked maximum it is checked whether or not the subhierarchy has a nonempty intersection with more than one list element. In this case a refinement attempt is made. If there is any such subhierarchy without a possible refinement no linearization exists.

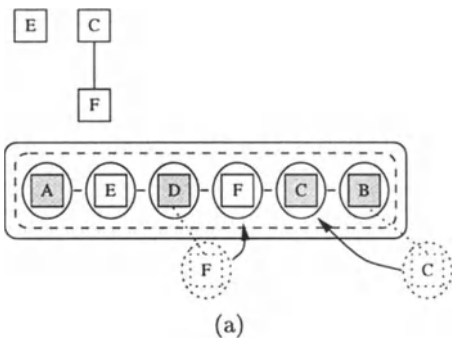
   Considering hierarchy (a) there are two candidates (subhierarchies) for refinement: {CF} and {DEF}. However, {CF} has a nonempty intersection with only one list element, i.e. {BCDF}. Consequently, no refinement is

done. {DEF} has nonempty intersections with both {BCDF} and {E}, the result of the refinement is given below.



In case of hierarchy (b) the only refinement candidate is {DEF}. This subhierarchy has a nonempty intersection with consecutive list elements. However, the refinement fails, because the inner list element {BD} is not a subset of {DEF}. At this point the linearization algorithm terminates for hierarchy (b). There is no optimal linearization for this hierarchy.

The next iteration for hierarchy (a) yields the subhierarchies {E} and {CF} as candidates. {CF} is a relevant candidate having nonempty intersections with {DF} and {BC}. We end up with a configuration like:



4. Using the $\circ$ and $*$ operators steps 1–3 produce lists of sets. The same processing scheme is applied recursively to each element of these lists. The results of the recursive descents are collected in the overall result set.

The final recursive calls for the list elements do not yield any modifications in the context of our running example. The final result is (A-E-D-F-C-B) and (B-C-F-D-E-A).

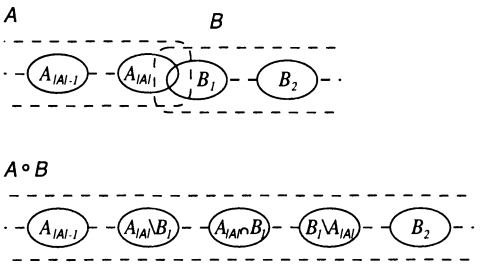


Figure 4.27 Concatenation operation (informally)

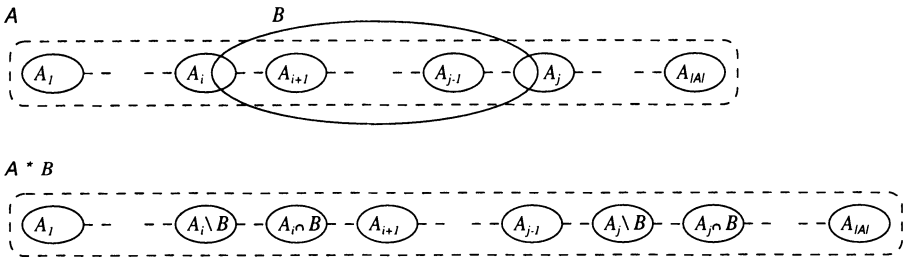


Figure 4.28 Refinement operation (informally)

The main part of the proposed algorithm is a recursive function *order*. Another integral part of this algorithm is a *structured set* $\boldsymbol{S}'$ constructed during the traversal of $(T, \leq)$. Elements of $\boldsymbol{S}'$ are either atoms (i.e., type identifiers) or *structured lists*. Elements of structured lists are again structured sets. The recursive definition of this data structure allows arbitrary nesting. In the sequel, two special cases are used: *flat sets*, i.e., structured sets containing merely atoms, and *flat lists*, i.e., structured lists containing merely flat sets.

The following notational conventions for variables hold: flat sets are denoted by $A, B, C, \ldots$, structured sets by $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \ldots$, flat list by $\mathsf{A}, \mathsf{B}, \mathsf{C}, \ldots$ and atoms by $a, b, c, \ldots$. There are no variables used for structured lists.

Function *order* is invoked by the wrapping function depicted in Figure 4.30. After termination of the algorithm, a post-processing step on $\boldsymbol{T}'$ produces all optimal linearizations (see below).

```
1.  begin order(S, ≤)
        S: set of type identifiers
        ≤: partial ordering
2.      if S contains less than 3 elements then return S
3.      assign the set of all unmarked maximal elements to M
4.      assign a set of lists to L in such a way that each list contains one set
        corresponding to a subhierarchy rooted at an element of M
5.      mark all elements of M
6.      assign all elements of S to S' which are not member of any subhierarchy
7.      foreach element A of L do
8.          remove A from L
9.          if there exists an element B of L such that
            there is a B_j of B and an A_i of A with nonempty intersection then
10.             if A ∘ B is defined then
11.                 replace B by A ∘ B in L
12.             else there is no solution
13.             end
14.         else
15.             while there exists an unmarked maximal element x such that
                there are at least 2 elements of A having a nonempty intersection
                with the subhierarchy rooted at x do
16.                 if A * S_≤x is defined then
17.                     refine A with S_≤x, i.e., with the subhierarchy rooted at x
18.                     mark x
19.                 else there is no solution
20.                 end
21.             end
22.             call order (recursively) for each element of A and
                add the list of all results to S'
23.         end
24.     end
25.     return S'
26. end order
```

Figure 4.29 Pseudocode for function *order*

```
D ← ∅
T' ← order(T, ≤)
```

Figure 4.30 Wrapper for function *order*

Prior to the formal definition of the function *order* in Figure 4.31 an informal pseudocode representation is given in Figure 4.29. The actual execution of *order* is illustrated by an example given below. At this point we have to define

```
1.  begin order(S, ≤)
2.      if |S| < 3 then return S end
3.      M ← max(S \ D, ≤)
4.      L ← ⋃_{m∈M} {(S_{≤m})}
5.      D ← D ∪ M
6.      S' ← S \ ⋃_{m∈M} S_{≤m}
7.      while ∃A ∈ L do
8.          L ← L \ {A}
9.          if ∃B ∈ L : ⋃ A_i ∩ ⋃ B_j ≠ ∅ then
10.             if A ∘ B is defined then
11.                 L ← L \ {B} ∪ {A ∘ B}
12.             else abort
13.             end
14.         else
15.             while ∃x ∈ max(⋃ A_i \ D, ≤) : |{A_i | A_i ∩ S_{≤x} ≠ ∅}| > 1 do
16.                 if A * S_{≤x} is defined then
17.                     A ← A * S_{≤x}
18.                     D ← D ∪ {x}
19.                 else abort
20.                 end
21.             end
22.             S' ← S' ∪ {(order(A_1, ≤), order(A_2, ≤), ···, order(A_{|A|}, ≤))}
23.         end
24.     end
25.     return S'
26. end order
```

Figure 4.31 Recursive construction of all optimal linearizations

the exact meaning of all operators used in *order*. The following operations and symbols are used:

- $\{\}$, $()$ and $\emptyset$ denote the set constructor, the list constructor, and the empty set, respectively.
- Set operators defined on flat sets are union ($\cup$), difference ($\setminus$), cardinality ($|\,|$), intersection ($\cap$) and set membership ($\in$).
- The operators $\cup$, $\setminus$ and $\in$ are also defined for the top level of structured sets.
- $|A|$ denotes the number of flat sets contained in the flat list A, denoted by $A_1, A_2, \ldots, A_{|A|}$.

- *max* yields a subset of a partially ordered set A in such a way that all elements in the subset are maximal elements of A and none of them is a minimal element of A, i.e.,

$$\max(A, \leq) \mapsto \{a \in A \mid \nexists a' \in A : a < a' \wedge \exists a'' \in A : a'' < a\}$$

- $\circ$ concatenates two *overlapping* flat lists, i.e., flat lists with common types in their respective sets. More precisely, two flat lists A and B overlap if and only if $\bigcup A_i \cap \bigcup B_i \neq \emptyset$. All sets in such a list have to be nonempty and pairwise disjoint.

  It should be noted that $\circ$ is defined if and only if $!\exists(i,j) : A_i \cap B_j \neq \emptyset, i \in \{1, |A|\}, j \in \{1, |B|\}$.[1] Informally, each of the two sets containing the common types has to be at one end of its enclosing list to enable concatenation. If this holds, there are the 4 cases which are depicted in Figure 4.32 (empty sets are removed from the concatenation result). Example:

i	j	$A \circ B$
$\|A\|$	1	$(A_1, \cdots, A_{\|A\|-1}, A_{\|A\|} \setminus B_1, A_{\|A\|} \cap B_1, B_1 \setminus A_{\|A\|}, B_2, \cdots, B_{\|B\|})$
$\|A\|$	$\|B\|$	$(A_1, \cdots, A_{\|A\|-1}, A_{\|A\|} \setminus B_{\|B\|}, A_{\|A\|} \cap B_{\|B\|}, B_{\|B\|} \setminus A_{\|A\|}, B_{\|B\|-1}, \cdots, B_1)$
1	1	$(A_{\|A\|}, \cdots, A_2, A_1 \setminus B_1, A_1 \cap B_1, B_1 \setminus A_1, B_2, \cdots, B_{\|B\|})$
1	$\|B\|$	$(A_{\|A\|}, \cdots, A_2, A_1 \setminus B_{\|B\|}, A_1 \cap B_{\|B\|}, B_{\|B\|} \setminus A_1, B_{\|B\|-1}, \cdots, B_1)$

Figure 4.32 Concatenation operator (formal definition)

  ({B},{CD},{EF})$\circ$({FG},{H}) yields ({B},{CD},{E},{F},{G},{H}), whereas ({B},{CD},{EF})$\circ$({DG},{H}) is undefined, since {CD}$\cap${DG}$\neq\emptyset$ and {CD} is not placed at either end of its list.

- $*$ represents refinement. $A * B$ is defined if and only if A denotes a flat list of pairwise disjoint and nonempty sets, B denotes a flat set, $!\exists(i,j) : i < j$ and

$$\forall k, 1 \leq k \leq |A| : \begin{cases} A_k \cap B = \emptyset \text{ for } k < i \\ A_k \cap B \neq \emptyset \text{ for } k = i \\ A_k \subset B \quad \text{for } i < k < j \\ A_k \cap B \neq \emptyset \text{ for } k = j \\ A_k \cap B = \emptyset \text{ for } k > j \end{cases}$$

[1] The quantifier $!\exists$ is to be read as *there exists exactly one*

If $A * B$ is defined, the result is given by (again, empty sets are removed from the result):

$$A * B \mapsto (A_1, \cdots, A_{i-1}, A_i \setminus B, A_i \cap B, A_{i+1}, \cdots, A_{j-1}, A_j \cap B, A_j \setminus B, A_{j+1}, \cdots, A_{|A|})$$

Example: ({B},{CDE},{FG},{H}) * {DEF} yields ({B},{C},{DE},{F}, {G},{H}). ({B},{CDE},{FG},{H}) * {DEGH} is undefined, since {CDE} $\cap$ {DEGH} $\neq \emptyset$ and {H} $\cap$ {DEGH} $\neq \emptyset$ and {FG} $\not\subset$ {DEGH}.

In the wrapping procedure (see Figure 4.30) set D is initialized as empty set. Its purpose is to hold already processed type identifiers. The wrapping procedure passes T as actual parameter to the initial call of function *order*.

For a given type hierarchy $(T, \leq)$, e.g., T = {ABCDEFG} with $\leq$ given in Figure 4.33, after termination of *order* the result contained in S' can be used

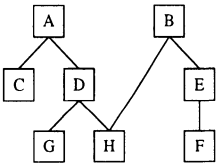


Figure 4.33 Example type hierarchy

to construct all optimal orderings. For the above example the value of S' is {({B({EF})}{H}{DG}{AC})}. The set of all optimal linearizations is constructed in the following way. Each set in the result can be represented by an arbitrary permutation of its elements, whereas each list yields only two correct representations (i.e., forward or backward sequence). In particular, sets {EF}, {DG}, and {AC} can be represented by 2! permutations each. The same is true for the set {B({EF})} containing one atomic element B and a list ({EF}) as second element. It should be noted that this list contains only one element, namely set {EF}. The list containing four elements, i.e., {B({EF})}, {H}, {DG}, and {AC} has only 2 correct representations. All in all a simple post-processing traversal yields $2 \cdot 2! \cdot 2! \cdot 2! \cdot 2! = 32$ optimal linearizations for T, e.g., (B-E-F-H-D-G-A-C), (B-F-E-H-D-G-A-C), (E-F-B-H-D-G-A-C), etc.

Applying the algorithm to the hierarchy of Figure 4.1 results in {Person, ({FacultyMember, AssistantProfessor, FullProfessor, AssociateProfessor} {Instructor} {Student})} thus giving $2! \cdot 2 \cdot 4! = 96$ optimal linearizations.

4.4.4 Implementation issues

In this section we apply the linearization algorithm for the purpose of type hierarchy indexing. In particular, the implementation of an MT-index with the help of optimal linearizations is outlined.

In general, an MT-index can be built using *any* multi-dimensional search data structure. Consequently, this outline is not focussed on any particular data structure like, for example, the BV-tree [Fre95], the hB-tree [LS90], or the hB^{Π}-tree [ELS95]. The only data structure requirement is a non-degenerating behavior in case of data skew (see Section 3.4).

One possible data structure setup could look like this: in any disk page of the value structure, the *key values component* contains a particular value combination of the indexed object properties. This component is followed by a list of object identifiers such that each identifier refers to an object having the attribute values given in the key values component. It should be noted that, in this context, the type identifier can be handled like any other attribute value, i.e., as part of the key values component. The optimal linearization algorithm guarantees minimum length query intervals in this type dimension.

The execution of query requests with the help of an MT-index involves two phases:

- a traversal of the access structure (either kept in main memory or on mass storage) collecting the set of relevant disk page addresses and
- a processing of the corresponding set of mass storage transfer operations, i.e., checking all tuples stored in the fetched disk pages and discarding all tuples not qualifying for the query request.

An important advantage of this kind of indexing framework is that *exactly the same* search structure technology could be applied to maintain *one* data structure for *all* relevant attributes of Person, e.g. income, weight, name, and so on. Considering for example

Q_9 select x from facultyMember x
where x.income < 50000 and x.weight < 100

the execution needs associative access to attribute income as well as to weight. As already pointed out, in single key approaches, the OODBMS is forced to

maintain two distinct search data structures, thus spending considerably more storage space for index maintenance and considerably more index scan time.

The splitting strategy for the type domain can be adapted to a concrete query profile. The domain split potential is determined by the data volume: the minimum data page occupancy (e.g., 0.66) and the given raw data volume determine the number of data pages, which in term determines the number of domain splits (see Section 7.2). Consequently, the only tunable parameter is the relative number of splits allocated to each domain. A larger number of splits in one domain implies a smaller number of splits in one or more of the other domains.

With respect to the limited total number of domain splits an MT-index could increase the number of splits in the type domain (thus increasing the degree of type grouping)

- if the fraction of queries referring to subhierarchies is large compared to the fraction of queries referring to the entire indexed hierarchy and
- if the number of types in the qualified subhierarchies is typically small compared to the total number of types in the indexed hierarchy.

Using a state-of-the-art multi-dimensional search data structure the implementation of the MT-index is straightforward. Figure 4.34 shows an hB-tree (see Section 3.4.4) implementation. The three rectangles represent internal hB-tree nodes. The leaf nodes (denoted by capital letters) are not shown in this figure. With respect to leaf node organization (value structure) we present two alternatives. These alternatives are orthogonal to possible leaf node data structures like linked lists or k-d trees as proposed in [LS90]. Figure 4.35(a) shows a leaf node organization using a directory. For each occurring combination of attribute values and type identifier the list of corresponding OIDs is stored. The counter is used to calculate the offset to the next values/type combination. Figure 4.35(b) shows a leaf node organization without directory. In this case leaf nodes contain one record per indexed object with OID, type, and k attribute values. Informally, the advantage of a leaf node directory decreases with increasing k and increasing attribute domain cardinalities, since particular combinations become less likely, i.e., lists of OIDs become shorter.

Continuing the example of Figure 4.34, Figure 4.36 depicts the resulting data space partitioning with the corresponding high level representation of the hB-tree. Additionally, a closer look at this figure leads to another technical issue: temporarily sacrificing the minimum node occupancy, one could split the

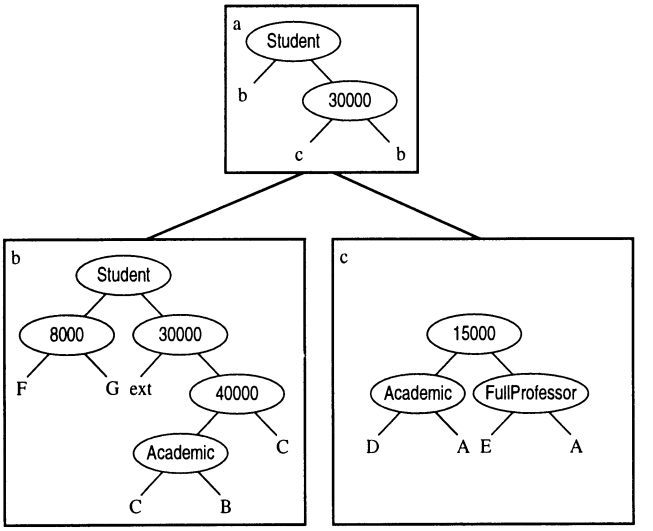


Figure 4.34 The hB-tree used as MT-index

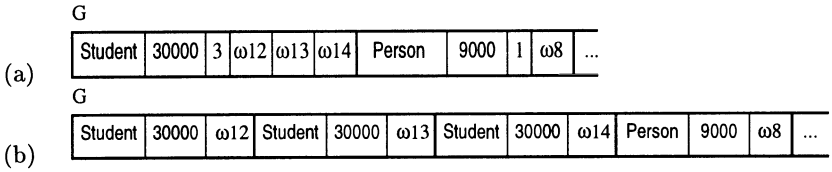


Figure 4.35 Leaf node organizations

type domain to the full extent, i.e., one region boundary per type *in advance*. Clearly, such a pre-splitting scheme allows for the omission of the type identifiers from the leaf node records thus saving index space.

4.5 BIBLIOGRAPHY

[KKD89] introduces the *class hierarchy index* and compares it to the *single class index* with respect to storage consumption and query performance.

H-trees and their application in deductive and object databases are discussed in [LLOH91]. The respective algorithms for nesting, insertion and deletion are given in [LOL92] and [OHLT96], together with a performance comparison of the H-tree and the class hierarchy index.

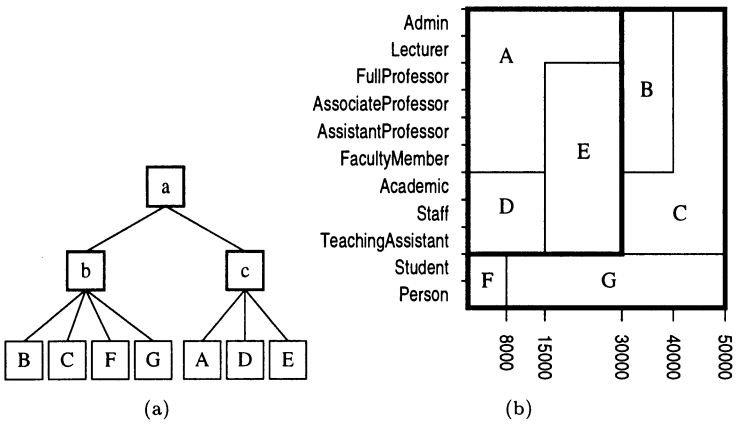


Figure 4.36 Visualization and data structure representation of the hB-tree

The basic idea of *class division* is presented in [KRVV93] and further developed in [RK95] where the performance for class division is compared to the class hierarchy index and to a multi-dimensional approach using R-trees.

The use of *multidimensional search structures* for type hierarchy indexing has been discussed in [KRVV93], [RK95], [RS94], [KM94b], and [CGO97]. [MP96, MP97b] present an algorithm for the necessary linearization of multiple inheritance hierarchies. [MP97a] compares the *multikey type index* to the class hierarchy index and the H-tree with respect to storage space and query performance. [CGO97] introduces a multi-dimensional data structure called χ-tree and compares its query performance to the CH-index.

Two other approaches try to overcome the problems of key grouping and type grouping, namely the *CG-tree* [KM94b] and the *hcC-tree* [SS94]. Both extend B^+-trees with multiple lists to organize OIDs according to set/type membership. A hcC-tree for a set of types T stores the OIDs of these types in $|T|$ linked lists of pages (*class chains*), one for each type, sorted according to the indexed property. One additional list (*hierarchy chain*) stores the OIDs of all types in a redundant organization. The pages of all chains are referenced by a common B^+-tree. This B^+-tree contains for each occurring value of the indexed property a set of pointers to those pages of each chain which contain OIDs with corresponding property values (if existing). The different chain types are used to efficiently process different query types: the class chains are used for point queries referring to a single type and for range queries referring to one or more (but not all) types. Range queries referring to the whole hierarchy and point queries referring to more than one type are answered using the hierarchy chain.

[SS94] presents the algorithms for searching, insertion and deletion in hcC-trees and compares their query performance to the CH-tree and the H-tree.

Abstracting from implementation details, a CG-tree can be viewed as a generalization of an hcC-tree. It is designed to index multiple sets which are not necessarily characterized by an inclusion relationship as defined for type extents. Consequently, there is one linked list of pages of OIDs for each indexed set. If queries often refer to groups of sets, the OIDs belonging to these sets can be stored together in *one* linked list instead of maintaining a linked list for each set in the group. In this context, the hcC-tree supports exactly one such group, namely the hierarchy chain containing the OIDs of all indexed types. [KM94b] presents the algorithms for maintaining CG-trees and compares the query performance of the CG-tree, the H-tree, and the class hierarchy index.

5

AGGREGATION PATH INDEXING

5.1 PROBLEM DESCRIPTION

In relational databases tuple sets (represented by tables) are linked by the application of the so called join operator. Consequently, any relational query involving more than one table usually contains explicit joins. For example let us assume a query involving Staff and Department, in particular, the retrieval of the names of staff persons working in the Process Automation department. Comparing the SQL-style query

```
select staff.name from staff, department
where staff.worksIn = department.iD
  and department.name = 'Process Automation';
```

to its OQL counterpart

```
select s.name from staff s
where s.worksIn.name = "Process Automation"
```

we observe an explicit join clause staff.worksIn = department.iD in the relational context in contrast to the implicit joinjoin,implicit between the types Staff and Department denoted as path expression s.worksIn.name. For the relational case we additionally assumed that the attribute worksIn of the table staff carries a department ID. Both explicit and implicit join operations have to be supported by index data structures. In absence of index data structures each of these joins would result in a large number of sequential scans per tuple or object set. In particular, for each pair (A, B) of sets to be joined each member of A has to be compared to each member of B yielding a so called *nested loop join*. For large tuple or object sets this join processing strategy soon becomes inappropriate

T. A. Mueck et al., *Index Data Structures in Object-Oriented Databases*

with respect to query performance. Speedup in this context is achieved by appropriate index data structures which allow to exclude large fractions of A and B from the comparison as well as by the materialization of the join result (usually also accessed via an index data structure).

Figure 5.1 contains portions of the schema graph of our running example relevant for this chapter. In this schema graph we can construct paths like

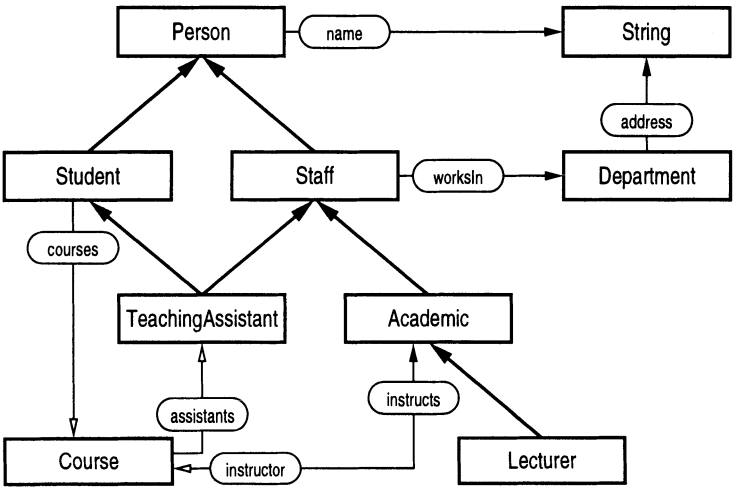


Figure 5.1 Schema graph

$P1$ Person.name

$P2$ Student.courses.instructor.worksIn.address

$P3$ Lecturer.instructs.assistants

$P4$ Student.courses.assistants.name

Prior to any discussion of path indexing issues and relevant data structures we have to introduce some notational conventions. Similar definitions can be found for example in [BM93], [KM90b], [KM94a]. Recalling T to be a type set as defined in Chapter 2, p-domain(p) denotes the domain of a property p. We will now extend the definition of p-domain for aggregation paths of arbitrary length.

Definition 5.1 (Aggregation path)
An aggregation path P is defined as $t_1.p_1.p_2.\cdots.p_n$ *with* $n \geq 1$. *The domain of an aggregation path P denoted by* p-domain(P) *corresponds to the domain of* p_n, p-domain(p_n). *An aggregation path is well formed if*

- t_1 *is a type in* T
- p_1 *is a property of type* t_1, *that is,* $t_1 \leq$ p-type(p_1), *and*
- p_i *is a property of the domain of* p_{i-1}, *that is,* p-domain($t_1.p_1.\cdots.p_{i-1}$) $\leq$ p-type(p_i) *for* $1 < i \leq n$

The above conditions imply that the domains of $p_1, \cdots, p_{n-1}$ *are object types, and that the domain of* p_n *is an object type or a literal type. Additionally, let* length(P) $= n$ *and* types(P) $= t_1 \cup \left(\bigcup_{1 \leq i < n}$ p-domain(p_i)$\right)$ *refer to the length of a path P and to the set of types referenced by the path P, respectively.*

For notational convenience we will refer to the members of types($t_1.p_1.\cdots.p_n$) by t_i meaning $t_i =$ p-domain(p_{i-1}) for $1 < i \leq n$ (see examples below). Recalling the path examples given above we conclude that $P1$, $P2$, $P3$, and $P4$ are well formed paths in the sense of Definition 5.1 with

- length($P1$) $= 1$
  types($P1$) $= \{t_1 =$ Person$\}$
  p-domain($P1$) $=$ String
- length($P2$) $= 4$
  types($P2$) $= \{t_1 =$ Student$, t_2 =$ Course$, t_3 =$ Academic$, t_4 =$ Department$\}$
  p-domain($P2$) $=$ String
- length($P3$) $= 2$
  types($P3$) $= \{t_1 =$ Lecturer$, t_2 =$ Course$\}$
  p-domain($P3$) $=$ TeachingAssistant
- length($P4$) $= 3$
  types($P4$) $= \{t_1 =$ Student$, t_2 =$ Course$, t_3 =$ TeachingAssistant$\}$
  p-domain($P4$) $=$ String

Any well-formed aggregation path corresponds to a path of the path graph (see Chapter 2). Figure 5.2 shows the path for $P4$. Substituting a particular OID

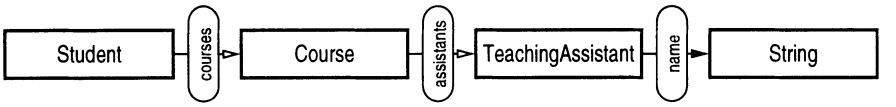


Figure 5.2 Path in the path graph for $P4$

o for the type identifier transforms an aggregation path into a path expression. Such a path expression qualifies a set of paths in the object graph, each such path is called path instantiation. The value of a path expression is the set of all values of property p_n which can be reached from o following the these paths in the object graph. A more precise description of this concept is provided in Definition 5.2.

Definition 5.2
Let $t_1.p_1.\cdots.p_n$ *denote an aggregation path* P. $o_1.p_1.\cdots.p_n$ *is called a path expression of* P *if and only if* o-type$(o_1) \leq t_1$. *The semantics of a path expression* $o_1.p_1.\cdots.p_n$ *of a path* P *are given by a function* val : $E(P) \mapsto \mathcal{P}(\text{Ext(p-domain}(P)))$ *with* $E(P) = \{o_1.p_1.\cdots.p_n | o_1 \in \text{Ext}(t_1)\}$. val *maps the set of all possible path expressions for* P *to the set of all possible value sets resulting from* p-domain(P). *Let* $[o.p]$ *denote the set of values stored in* o *for property* p. *Let* V *be an abbreviation for* val$(o_1.p_1.\cdots.p_{n-1})$. *Now* val *can be defined as*

$$\text{val}(o_1.p_1.\cdots.p_n) = \begin{cases} [o_1.p_1] & \text{if } n = 1 \\ \bigcup_{o \in V}[o.p_n] & \text{if } n > 1 \end{cases} \tag{5.1}$$

The following examples in the context of our university database are meant to illustrate these terms. Figure 5.3 shows part of the object graph of this database.

P_1 Person.name: ω_{21}.name
val(ω_{21}.name) $\Leftarrow$ [ω_{21}.name] $\equiv$ {Hurka}

P_4 Student.courses.assistants.name: ω_{14}.courses.assistants.name
val(ω_{14}.courses.assistants.name)
$\Leftarrow \bigcup$[o.name] with $o \in$ val(ω_{14}.courses.assistants)
val(ω_{14}.courses.assistants)
$\Leftarrow \bigcup$[o.assistants] with $o \in$ val(ω_{14}.courses)
val(ω_{14}.courses) $\Leftarrow$ [ω_{14}.courses] $\equiv \{\omega_{25}, \omega_{28}, \omega_{29}\}$
val(ω_{14}.courses.assistants)

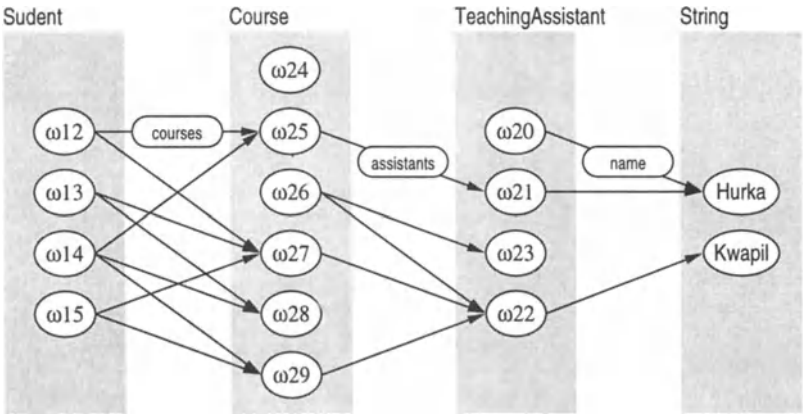


Figure 5.3 Subgraph of the object graph of the running example

$\Leftarrow \bigcup[o.\mathsf{assistants}]$ with $o \in \{\omega_{25}, \omega_{28}, \omega_{29}\}$
$\Leftarrow [\omega_{25}.\mathsf{assistants}] \cup [\omega_{28}.\mathsf{assistants}] \cup [\omega_{29}.\mathsf{assistants}] \equiv \{\omega_{21}, \omega_{22}\}$
$\mathrm{val}(\omega_{14}.\mathsf{courses.assistants.name})$
$\Leftarrow \bigcup[o.\mathsf{name}]$ with $o \in \{\omega_{21}, \omega_{22}\}$
$\Leftarrow [\omega_{21}.\mathsf{name}] \cup [\omega_{22}.\mathsf{name}] \equiv \{\mathsf{Hurka}, \mathsf{Kwapil}\}$

This kind of evaluating path expressions, i.e., starting from an object at the beginning of the path in order to find all objects at the end of the path, is called *forward traversal.* Starting the traversal with an object at the end of the path and proceeding in the opposite direction is called *backward traversal.*

A forward traversal of path $P1$ retrieves the name of a specified Person object. If thatPerson evaluates to an object that represents that person, we can query

```
select p.name from person p
where p = thatPerson
```

Since this query contains a forward traversal of path $P1$, it is also called a *forward query* with respect to $P1$. A backward traversal of $P1$ retrieves all Person objects that have a specified name, e.g.,

```
select p from person p
where p.name = "Hurka"
```

Consequently, this is a *backward query* with respect to $P1$. A forward traversal of path $P4$ retrieves the names of teaching assistants who assist courses taken by

a particular student. If thisStudent evaluates to an object that represents this student (and we keep in mind that OQL does not allow the "direct" notation of a path if it contains multi-valued properties) the corresponding OQL query can be coded as

Q_{10} select ta.name from student s, s.courses c, c.assistants ta
where s = thisStudent

The corresponding backward query retrieves all students taking any course assisted by any teaching assistant with a specified name:

Q_{11} select s from student s, s.courses c, c.assistants ta
where ta.name = "Hurka"

Summing up, index data structures designed to provide fast access to aggregation paths have to meet the following requirements:

- Support for backward as well as forward traversals: a forward traversal yields for a given OID a set of property values. If the property is an attribute its values are literals, whereas if the property is a relationship its values are OIDs. Consequently, a backward traversal starts with either a literal or an OID and results in a set of OIDs. It should be noted that conventional search structure applications focus on backward traversal starting with literals. Starting an index lookup already with an OID might seem strange at a first glance, because this OID actually provides access to all properties of the object. In the context of OODBMS, however, the properties of an object can be accessed only if the object is fetched from mass storage. So the (forward traversal) access to properties usually is less resource consuming using an index which allows forward traversals. This also holds for backward traversals of relationships for which inverse relationships have been defined. An index lookup is usually more efficient than a forward traversal of the inverse relationship after fetching the corresponding object.

- Support for multi-valued properties: an index implementation for such a property, e.g., Course.assistants, may be based either on decomposition of the property or on variable length records.

- Support for attributes as well as relationships: access data structures for literal type properties in general have to support exact match queries as well as range queries. However, access structures maintained for relationships need to handle exact match queries only. Keeping this in mind,

subspace mapping data structures qualify for attribute indexing in the first place, whereas point mapping approaches might be used for relationship indexing.

- Support for subpath access: an index created for a particular aggregation path, say Student.courses.assistants.name, should support queries directed to the start node of the path (Student) as well as to any interior node (Course, TeachingAssistant). In other words, the index should be applicable for the subpaths Course.assistants.name and TeachingAssistant.name as well.

To meet these requirements we have to consider two nearly independent issues. At the design level it has to be decided which parts of an aggregation path should be indexed. An index created for a particular aggregation path could support the access to subpaths in addition to regular forward and backward traversals. At the implementation level a search data (or a combination of search data structures) has to be chosen to meet the indexing goals as defined in the design phase. Consequently, the following section deals with some prominent design approaches without explicitly referring to concrete search data structures (see Chapter 3).

5.2 PATH DECOMPOSITION SCHEMES

5.2.1 An initial approach – Multi-Index

The indexing structure used in the GemStone OODBMS [BMO+89] was discussed in [MS86, SM91, MS90] and Φchristened multi-index in [BK89]. The general idea is to divide a path of arbitrary length into a number of subpaths with a length of one and maintain an index for each of these subpaths.

Consequently, a multi-index for a path P, $MX(P)$, consists of a set of length(P) index components, $IX(P_i)$. Each index component $IX(P_i)$ associates each occurring value of the property p_i with the set of objects whose property p_i carries this value (if p_i is single-valued) or contains this value (if p_i is multi-valued). Although the original design of the multi-index only supported single-valued properties, an extension for multi-valued properties is straightforward and was, for example, proposed in [BK89].

Definition 5.3 (Multi-Index)
The Multi-Index $MX(P)$ for the path $P = t_1.p_1.p_2.\cdots.p_n$ is defined as

$MX(P) = \{IX(t_i.p_i)$ *with* $1 \leq i \leq n\}$. *The index component* $IX(P_i)$ *for the subpath* $P_i = t_i.p_i$ *is defined as*

$$IX(P_i) = \{(o, S)|S = \{o'|o \in \text{val}(o'.p_i)\}\}$$

A multi-index for the path $P4$ = Student.courses.assistants.name consists of three index components:

MX(Student.courses.assistants.name) =
{IX(Student.courses), IX(Course.assistants), IX(TeachingAssistant.name)}

Figure 5.4 shows the index components of a multi-index on $P4$ created for the objects of Figure 5.3. In order to answer the backward query Q_{11} the following

IX(Student.courses)	
ω_{25}	ω_{12} ω_{14}
ω_{27}	ω_{12} ω_{13} ω_{15}
ω_{28}	ω_{13} ω_{14}
ω_{29}	ω_{14} ω_{15}

IX(Course.assistants)	
ω_{21}	ω_{25}
ω_{22}	ω_{26} ω_{27} ω_{29}
ω_{23}	ω_{26}

IX(TeachingAssistant.name)	
Hurka	ω_{20} ω_{21}
Kwapil	ω_{22}

Figure 5.4 Multi-index for Student.courses.assistants.name

steps have to be made:

1. Use IX(TeachingAssistant.name) to find all the teaching assistants named Hurka (this yields $\{\omega_{20}, \omega_{21}\}$)
2. Use IX(Course.assistants) to find all courses assisted by teaching assistants ω_{20} or ω_{21}. Since ω_{20} does not assist in any course the result only contains ω_{25}.
3. Use IX(Student.courses) to find all students taking the course ω_{25} which retrieves the final result of the query, $\{\omega_{12}, \omega_{14}\}$.

A multi-index for the path Student.courses.assistants.name can also be used for backward queries on any subpath of this path, e.g., Student.courses, Course.assistants.name, etc.. To characterize a specific path indexing scheme with respect to its applicability for forward and/or backward queries on a path and its various subpaths we use a method called *indexing graph* introduced in [SB96]. Figure 5.5 shows the indexing graph for a multi-index on Student.courses.assistants.

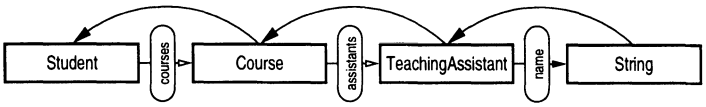


Figure 5.5 Indexing graph for the multi-index

name. An indexing graph is a subgraph of the path graph which represents the path to be indexed. It is augmented by additional edges that represent the capabilities of the used indexing scheme. If one such edge in the indexing graph runs from type t_1 to t_2, the indexing scheme in question efficiently supports traversals from objects of t_1 to objects of t_2 along the indexed path *in one step* (with one index lookup). An edge that has the same direction as the property edge connecting the respective types refers to a forward traversal, the opposite direction indicates a backward traversal.

Figure 5.5 shows that a multi-index on Student.courses.assistants.name supports backward queries on Student.courses, Course.assistants, and TeachingAssistants.name in one step. Traversals that result from concatenating n indexing edges in the indexing graph require n index scans. A backward traversal of the whole path involves three index edges and thus requires three index scans (as outlined above). Forward traversals and hence forward queries are not supported.

5.2.2 A general approach – Access Support Relations

Access support relations have been proposed in [KM90a] as a generalization of the join index (see below). [KM90b] discusses the optimization of queries containing path traversals using access support relations.

Definition 5.4 (Access support relation)
The access support relation for the path $t_1.p_1.\cdots.p_n$ *is a* $(n+1)$*-ary relation*

$$ASR(t_1.p_1.\cdots.p_n) : [S_1, \cdots, S_{n+1}]$$

The domain of attribute S_i *is the type extent of* t_i *for* $(1 \leq i \leq n)$*, the domain of attribute* S_{n+1} *is* p-domain(p_n).

For the definition of *ASR*s a set of binary *ASR*s is used, in particular, one binary *ASR* for each subpath of length one.

Definition 5.5
For each p_j $(1 \leq j \leq n)$ *of* $ASR(t_1.p_1.\cdots.p_n)$ *a binary relation* $ASR(t_j.p_j)$ *is defined as:*

$$ASR(t_j.p_j) = \{(o, o') | o' \in \text{val}(o.p_j)\}$$

As it can be seen from the definition, the content of each binary *ASR* is equivalent to the content of the corresponding multi-index component. The binary relations for $P4$ = Student.courses.assistants.name are shown in Figure 5.6. Based

ASR(Student.courses)	
ω_{12}	ω_{25}
ω_{12}	ω_{27}
ω_{13}	ω_{27}
ω_{13}	ω_{28}
ω_{14}	ω_{25}
ω_{14}	ω_{28}
ω_{14}	ω_{29}
ω_{15}	ω_{27}
ω_{15}	ω_{29}

ASR(Course.assistants)	
ω_{25}	ω_{21}
ω_{26}	ω_{22}
ω_{26}	ω_{23}
ω_{27}	ω_{22}
ω_{29}	ω_{22}

ASR(TeachingAssistant.name)	
ω_{20}	Hurka
ω_{21}	Hurka
ω_{22}	Kwapil

Figure 5.6 Binary *ASRs* for Student.courses.assistants.name

on these binary relations four different *extensions* of $ASR(t_1.p_1.\cdots.p_n)$ are defined (see below). To support forward traversals as well, the relations are kept redundantly in two data structures. One of these data structures allows access based on the first (leftmost) attribute, the other one allows access based on the last (rightmost) attribute. The data structure selection process, e.g., choosing B^+-trees or hash structures, should follow the guidelines sketched in Chapter 3.

- The canonical extension $ASR_{\text{can}}(t_1.p_1.\cdots.p_n)$ results from joining the binary *ASRs*. Thus, it contains all path instantiations that begin with an object of t_1 and end with an object of p-domain$(t_1.p_1.\cdots.p_n)$ (*complete instantiations*).

  The canonical extension of the *ASR* of our example path Student.courses.assistants.name is given in Figure 5.7. It contains all paths of the object graph in Figure 5.3 that start with an object of Student and lead to a literal of type String.

  The formal definition of $ASR_{\text{can}}(t_1.p_1.\cdots.p_n)$ can be given as

$$ASR(t_1.p_1) \bowtie ASR(t_2.p_2) \bowtie \cdots \bowtie ASR(t_n.p_n)$$

ASR_{can}(Student.courses.assistants.name)			
ω_{12}	ω_{25}	ω_{21}	Hurka
ω_{12}	ω_{27}	ω_{22}	Kwapil
ω_{13}	ω_{27}	ω_{22}	Kwapil
ω_{14}	ω_{25}	ω_{21}	Hurka
ω_{14}	ω_{29}	ω_{22}	Kwapil
ω_{15}	ω_{27}	ω_{22}	Kwapil
ω_{15}	ω_{29}	ω_{22}	Kwapil

Figure 5.7 Canonical extension of *ASR*(Student.courses.assistants.name)

where $\bowtie$ denotes the natural join operator.

- Replacing the natural join with an outer join yields the full extension $ASR_{\text{full}}(t_1.p_1.\cdots.p_n)$. It contains all the instantiations of the canonical extension plus all instantiations that

  1. terminate prematurely because of an undefined property value (in case of a single-valued property) or an empty collection (in case of a multi-valued property) and/or
  2. start with an object o of t_j with $1 < j \leq n$ because there is no object of t_{j-1} that references o through its property p_{j-1}.

ASR_{full}(Student.courses.assistants.name)			
—	—	ω_{20}	Hurka
—	ω_{26}	ω_{22}	Kwapil
—	ω_{26}	ω_{23}	—
ω_{12}	ω_{25}	ω_{21}	Hurka
ω_{12}	ω_{27}	ω_{22}	Kwapil
ω_{13}	ω_{27}	ω_{22}	Kwapil
ω_{13}	ω_{28}	—	—
ω_{14}	ω_{25}	ω_{21}	Hurka
ω_{14}	ω_{28}	—	—
ω_{14}	ω_{29}	ω_{22}	Kwapil
ω_{15}	ω_{27}	ω_{22}	Kwapil
ω_{15}	ω_{29}	ω_{22}	Kwapil

Figure 5.8 Full extension of *ASR*(Student.courses.assistants.name)

The full extension for Student.courses.assistants.name shown in Figure 5.8 contains, for example, the tuple (–, –, ω_{20}, Hurka). The corresponding

path instantiation starts with ω_{20} because teaching assistant ω_{20} does not assist in any course. Consequently, there is no Course object whose property assistants contains ω_{20}. The instantiation corresponding to the tuple (ω_{13}, ω_{28}, –, –) ends with ω_{28} because the course ω_{28} has no teaching assistants, therefore the value of its property assistants is an empty set. Finally, the instantiation corresponding to the tuple (–, ω_{26}, ω_{23}, –) has neither a proper beginning nor a proper end. It does not start with a Student object because course ω_{26} has no participants, as a consequence no Student object references ω_{26}, that is, contains ω_{26} as element of its property courses. Additionally, it ends prematurely with ω_{23} because the value of property name of teaching assistant ω_{23} is undefined.

The formal definition of $ASR_{\text{full}}(t_1.p_1.\cdots.p_n)$ can be given as

$$ASR(t_1.p_1) \overset{\frown}{\bowtie} ASR(t_2.p_2) \overset{\frown}{\bowtie} \cdots \overset{\frown}{\bowtie} ASR(t_n.p_n)$$

where $\overset{\frown}{\bowtie}$ denotes the outer join operator.

- The left-complete extension $ASR_{\text{left}}(t_1.p_1.\cdots.p_n)$ is obtained by joining the binary *ASR*s from left to right using a left outer join. It contains all instantiations that start with an object of t_1 and possibly end prematurely. The left-complete extension for Student.courses.assistants.name is shown in Figure 5.9.

ASR_{left}(Student.courses.assistants.name)			
ω_{12}	ω_{25}	ω_{21}	Hurka
ω_{12}	ω_{27}	ω_{22}	Kwapil
ω_{13}	ω_{27}	ω_{22}	Kwapil
ω_{13}	ω_{28}	—	—
ω_{14}	ω_{25}	ω_{21}	Hurka
ω_{14}	ω_{28}	—	—
ω_{14}	ω_{29}	ω_{22}	Kwapil
ω_{15}	ω_{27}	ω_{22}	Kwapil
ω_{15}	ω_{29}	ω_{22}	Kwapil

Figure 5.9 Left-complete extension of *ASR*(Student.courses.assistants.name)

The formal definition of $ASR_{\text{left}}(t_1.p_1.\cdots.p_n)$ can be given as

$$(\cdots(ASR(t_1.p_1) \overset{\frown}{\bowtie} ASR(t_2.p_2)) \overset{\frown}{\bowtie} \cdots) \overset{\frown}{\bowtie} ASR(t_n.p_n)$$

where $\overset{\frown}{\bowtie}$ denotes the left outer join operator.

- The right-complete extension $ASR_{\text{right}}(t_1.p_1.\cdots.p_n)$ is obtained by joining the binary *ASR*s from right to left using a right outer join. It contains all instantiations that end with an object of p-domain(p_n) and possibly do not start with an object of t_1. The right-complete extension for Student.courses.assistants.name is shown in Figure 5.10.

ASR_{right}(Student.courses.assistants.name)			
—	—	ω_{20}	Hurka
—	ω_{26}	ω_{22}	Kwapil
ω_{12}	ω_{25}	ω_{21}	Hurka
ω_{12}	ω_{27}	ω_{22}	Kwapil
ω_{13}	ω_{27}	ω_{22}	Kwapil
ω_{14}	ω_{25}	ω_{21}	Hurka
ω_{14}	ω_{29}	ω_{22}	Kwapil
ω_{15}	ω_{27}	ω_{22}	Kwapil
ω_{15}	ω_{29}	ω_{22}	Kwapil

Figure 5.10 Right-complete extension *ASR*(Student.courses.assistants.name)

The formal definition of $ASR_{\text{right}}(t_1.p_1.\cdots.p_n)$ can be given as

$$ASR(t_1.p_1) \bowtie\!\!\!\!\subset (\cdots \bowtie\!\!\!\!\subset (ASR(t_{n-1}.p_{n-1}) \bowtie\!\!\!\!\subset ASR(t_n.p_n)) \cdots)$$

where $\bowtie\!\!\!\!\subset$ denotes the right outer join operator.

Following from this, the X-extension $ASR_X(t_1.p_1.\cdots.p_n)$ is *applicable* for (i.e., contains all complete instantiations of) the path $t.p_i.\cdots.p_j$, if

$$t \leq \text{p-type}(p_i) \wedge \begin{cases} 1 \leq i \leq j \leq n & \text{for} \quad X = full \\ 1 = i \leq j \leq n & \text{for} \quad X = left \\ 1 \leq i \leq j = n & \text{for} \quad X = right \\ 1 = i \leq j = n & \text{for} \quad X = can \end{cases}$$

Figure 5.11 shows the indexing graph of access support relations. The labels of the indexing edges denote which extension supports the corresponding traversal. The indexing graph shows that *ASR*s efficiently support forward traversals starting at the beginning of the path to any type along the path, as well as backward traversals starting from the end of the path to any type along the path. However, they do not efficiently support any traversal starting from a type other than the first or the last type of the path (although some extensions contain the relevant data).

Suppose a query

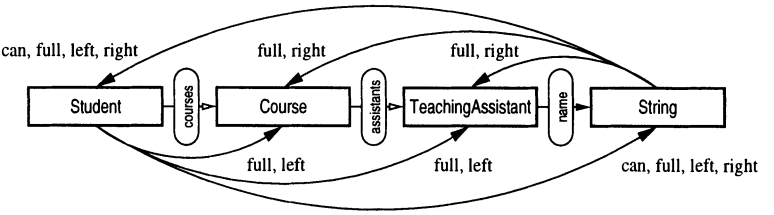


Figure 5.11 Indexing graph for different extensions of *ASR*s

Q_{12} select s from student s, s.courses c
where aTA in c.assistants

where aTA evaluates to a TeachingAssistant object. This query involves a backward traversal on the path Student.courses.assistants. According to Figure 5.11 such a traversal is not efficiently supported by any of the *ASR* extensions. In order to answer this query a *full* scan of either the full or the left-complete extension is necessary.

To overcome this deficiency a path can be split into a number of subpaths and for each subpath an *ASR* extension (called *partition*) is maintained. The set of subpath *ASR* extensions for a path is called *decomposition* of *ASR*s.

Definition 5.6
Let $ASR_X(t_1.p_1.\cdots.p_n)$ *denote an* $(n+1)$*-ary ASR extension with attributes* $S_1, \cdots, S_{n+1}$ *(*$X \in$ {can, full, left, right}*). The relations*

$$\begin{array}{lcll} ASR_X^{(1,i_1)}(t_1.p_1.\cdots.p_n) & : & [S_1, \cdots, S_{i_1}] & \text{for } 1 < i_1 \leq n+1 \\ ASR_X^{(i_1,i_2)}(t_1.p_1.\cdots.p_n) & : & [S_{i_1}, \cdots, S_{i_2}] & \text{for } i_1 < i_2 \leq n+1 \\ \cdots & & & \\ ASR_X^{(i_k,n+1)}(t_1.p_1.\cdots.p_n) & : & [S_{i_k}, \cdots, S_{n+1}] & \text{for } i_k \leq n+1 \end{array}$$

are called $(1, i_1, i_2, \cdots, i_k, n+1)$*-decomposition of* $ASR_X(t_1.p_1.\cdots.p_n)$*. The single relations* $ASR_X^{(x,y)}(t_1.p_1.\cdots.p_n)$ *are called partitions and are obtained by a projection of the relevant attributes of* $ASR_X(t_1.p_1.\cdots.p_n)$*. If a decomposition contains only binary partitions the decomposition is called binary.*

In order to support queries like Q_{12} we can construct a $(1,3,4)$-decomposition containing the partitions $ASR_{\text{full}}^{(1,3)}(P4)$ and $ASR_{\text{full}}^{(3,4)}(P4)$. The corresponding indexing graph is given in Figure 5.12.

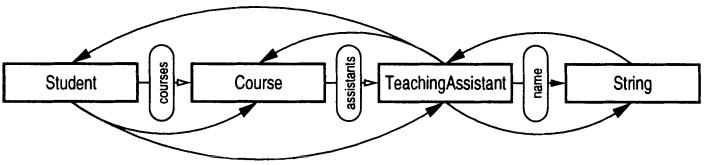


Figure 5.12 Indexing graph for a specific *ASR* decomposition

5.2.3 Specific decomposition schemes

Nested Index

The nested index introduced in [BK89] associates each object o with a set of objects that start complete path instantiations ending in o.

Definition 5.7
The nested index for the path $P = t_1.p_1.\cdots.p_n$ *is defined as*

$$NX(P) = \{(o, S)|S = \{o'|o \in \text{val}(o'.p_1.\cdots.p_n)\}\}$$

Figure 5.14 shows the data stored in a nested index for Student.courses.assistants.name. As can be seen from the definition and from the indexing graph in Figure 5.13 a nested index allows only backward traversals of the full path. This

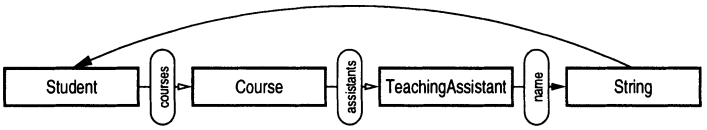


Figure 5.13 Indexing graph for the nested index

is equivalent to the backward traversal component of a canonical *ASR* extension, with the notable difference that the objects along a path instantiation are not stored in a nested index. A nested index $NX(P)$ for a path P with length$(P) = 1$ is equivalent to a multi-index for the same path.

Path-Index

A path-index [BK89] associates each object o with the set of all (not necessarily complete) path instantiations ending with o that are not a proper suffix of any longer instantiation ending with o.

NX(Student.courses.assistants.name)	
Hurka	ω_{12} ω_{14}
Kwapil	ω_{12} ω_{13} ω_{14} ω_{15}

Figure 5.14 Nested index for Student.courses.assistants.name

Definition 5.8 (Path-Index)
The path-index for the path $P = t_1.p_1.\cdots.p_n$ *is defined as* $PX(P) = \{(o, S)|S = \{o_j.o_{j+1}.\cdots.o_n\}$ *with* $1 \leq j \leq n$ *where* $o_j.o_{j+1}.\cdots.o_n.o$ *is a (not necessarily complete) instantiation of* P *ending in* o*, and there is no instantiation* $o_k.\cdots.o_j.o_{j+1}.\cdots.o_n.o$ *of* P *with* $1 \leq k < j \leq n$.

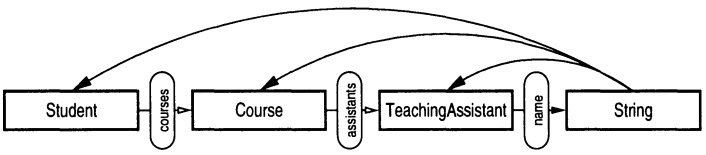


Figure 5.15 Indexing graph for a path-index

From the indexing graph for our example path $P4$ in Figure 5.15 we observe that a path-index is equivalent to the backward traversal component of a right-complete *ASR* extension (see Figure 5.16 for the data contained in a path-index). A path-index $PX(P)$ for a path P with length$(P) = 1$ is equivalent to a multi-

PX(Student.courses.assistants.name)	
Hurka	ω_{20} $\omega_{12}.\omega_{25}.\omega_{21}$ $\omega_{14}.\omega_{25}.\omega_{21}$
Kwapil	$\omega_{26}.\omega_{22}$ $\omega_{12}.\omega_{27}.\omega_{22}$ $\omega_{13}.\omega_{27}.\omega_{22}$ $\omega_{15}.\omega_{27}.\omega_{22}$ $\omega_{14}.\omega_{29}.\omega_{22}$ $\omega_{15}.\omega_{29}.\omega_{22}$

Figure 5.16 Path index for Student.courses.assistants.name

index $MX(P)$ and a nested index $NX(P)$ for the same path.

Join-Index

Join indices have been introduced in [Val87] to optimize joins in relational DBMS. In [VKC86] they are used to speed up implicit joins, i.e., path traversals, in OODBMS. A join index consists of a set of binary join indices, one for each subpath of length one of the indexed path. Each binary join index is kept redundantly in two data structures, one for accessing the first attribute, the

other for accessing the second attribute. Thus, a binary join index is equivalent to the corresponding binary *ASR*, the indexing graph of a join index is given in Figure 5.17.

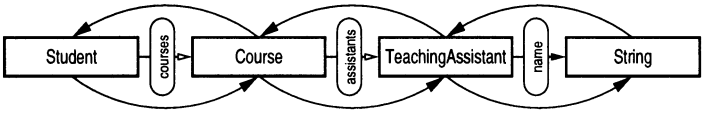


Figure 5.17 Indexing graph of a join index

5.3 BIBLIOGRAPHY

[MS86], [BMO+89], [MS87], [MS90], [SM91] elaborate on the GemStone aggregation path indexing scheme called *multi-index* in [BK89] and discuss implementation details.

[Val87] proposes *join indices* to speed up joins in relational DBMS, presents a cost model and compares the performance of joins using a join index to joins using the hybrid-hash join algorithm [DKO+84]. [VKC86] adapts join indices for path traversals in OODBMS and compares the performance of various implementation techniques. [XH94] extend join indices to *join index hierarchies*. The performance evaluation compares three variants, namely the *base join hierarchy* (equivalent to join indices and binary *ASR*s), the *complete join index hierarchy* (containing a binary join index component for every possible subpath of the indexed path), and the *partial join index hierarchy* (contains a binary join index component for specified subpaths of the indexed path), to a corresponding full *ASR*-extension.

[KM90a] and [KM92] introduce *access support relations*, provide cost models for the different extensions and compare their performance. [KM90b] presents algorithms for query optimization using *ASR*s.

[BK89] introduces the *nested index* and the *path-index*, presents cost models, and compares the performance of the two approaches to the multi-index. [Ber90] discusses query optimization using nested indices and presents cost models for different query execution plans. [BG92] and [BG93] discusses query optimization using path indices and compares them to nested indices under different query profiles and query execution plans. Implementation and query execution considerations concerning path indices are presented in [KD91].

An algorithm for optimal index configuration is presented in [Ber94]. The presented algorithm finds an optimal path partitioning together with a selection of appropriate index structures for each subpath (*path splitting*). Algorithms for index selection are also presented and compared in [CCY94].

Data structure considerations related to access support relations can be found in [LL94] (*path dictionary*) and in [LL95] (*path dictionary index*). The performance of the proposed structures is compared to path indices.

Signature files [FC84] are used to speed up forward traversals in [YLK94]. [LL92] introduces *path signatures*, [LL96] combines *path signatures* with *path dictionaries* to introduce *signature path dictionaries* and compares their performance to path signatures. Both approaches use signatures of path instantiations to speed up traversals.

A number of approaches try to combine techniques for type hierarchy indexing (see Chapter 4) with aggregation path indexing. The *nested inherited index* [Ber91, BF95] combines a nested index with a class hierarchy index and a auxiliary data structure to improve update performance. [Ber91] compares the query performance of the nested inherited index, the multi-index and the *inherited multi-index* (a combination of multi-index and class hierarchy index). [BF95] presents cost models for retrieval, insertion, and deletion and compares nested inherited index, multi-index and inherited multi-index. [CBBC94] examines the optimal index selection for a given indexing problem considering class hierarchy index, multi-index, inherited multi-index, and nested inherited index. [SB96] introduces the *generalized nested inherited index* and compares its performance to the nested inherited index, the inherited multi-index and path splitting.

Further work includes the *Generalized Index* [FLG91b, FLG91a], *Object Skeletons* [HT94], and the *Uniform Index* [Gud96].

6

COLLECTION OPERATIONS

6.1 PROBLEM DESCRIPTION

As already pointed out in Chapter 2, properties can be either single-valued or multi-valued. Instances of multi-valued properties can be sets, bags, lists or arrays of values. This is one of the most important differences to the traditional relational model requiring atomic attribute values (a central rule also known as *First Normal Form* in relational literature).

Although just recently there has been some dispute about the correct interpretation of the First Normal Form as stated by the founders of the relational model (see [Cam96, Dat96, Dar96]), the practical consequences, whether or not intended by theoreticians, are obvious: purely relational systems do not offer structured data types like sets or the like as attribute domains. In several application domains, the restriction of attribute domains to scalar data types results in rather ugly work-arounds: typically complex structures are superimposed on simple byte strings in such a way that only application programs are able to interpret the actual structure and the DBMS is not. This clearly is a circumvention of a major database design principle stating that the DBMS should store and maintain all of the structure information for any of its databases to prohibit uncoordinated structure modifications by application programmers. Consequently, even prior to the advent of OODBMS, language extensions for structured domains have been proposed (the perhaps most prominent example is the so called Non First Normal Form or NF2 proposal [SS86].

Also in the realm of object-oriented systems, the use of multi-valued properties yields a variety of new modeling features and significantly enhanced expressiveness on the schema level. However, this kind of expressiveness again results in higher complexity with respect to indexing and query optimization. An ex-

T. A. Mueck et al., *Index Data Structures in Object-Oriented Databases*

ample should help to clarify this point: if one is interested in retrieving the names of all persons who have hiking as one of their hobbies, she could post the following OQL query to the DBMS:

Q_{13} select p.name from person p
where "Hiking" in p.hobbies

A query request like this can be efficiently processed using one of the indexing approaches described in Chapter 5. In particular, if there is a path-index for attribute Person.hobbies, the corresponding data structure yields for data value "Hiking" a set of OIDs. Each OID refers to an object of type Person with a matching value in the value set of attribute hobbies.

However, if another query request should retrieve the names of all persons who have Hiking, Football and Aussie Football in their set of hobbies the retrieval situation is more complex. Persons who have the above set of disciplines as a subset of their value set in the corresponding attribute are selected by the following OQL query (*subset* query) which is structurally similar to query Q_{13} above:

Q_{14} select p.name from person p
where Set ("Hiking", "Football", "Aussie Football") <= p.hobbies

Now, using again an index on Person.hobbies to speed up the execution of the query request requires in this case two steps:

1. compute three intermediate results as follows: retrieve from the search data structure for each element s of {"Hiking", "Football", "Aussie Football"} the respective set of Person objects (more precisely, their OIDs) having s in the value set of attribute hobbies;

2. compute the final query result as follows: for each OID in the intersection of the intermediate results produced in the first step (i.e., one set of OIDs per hobby in the query request), retrieve the corresponding value of attribute name.

Example query Q_{14} shows that the overall query performance is determined by the number of entries (hobbies in the above example) in the query set since, in general, each entry causes a separate data structure traversal. Consequently, the standard index solution for Person.hobbies might be used by the query optimizer for the above request, but the resulting query performance could be poor in case of larger query sets.

For a third request which is again similar to the above examples from a user's point of view, an index on Person.hobbies cannot be used at all. The query aims at the names of all persons who do not have hobbies except any subset (also the empty set) of Hiking, Football and Aussie Football (*superset* query):

```
Q15 select p.name from person p
    where Set ("Hiking", "Football", "Aussie Football") >= p.hobbies
```

The problem with the index on Person.hobbies in this case is that the search data structure contains only references to persons with hobbies, or, in other words, persons without any hobby (the attribute value is the empty set) cannot be retrieved with the help of the index *although they qualify for the query.* Consequently, the query optimizer has to ignore the index defined for the attribute. There are several more or less complex features of OQL which allow to exclude the objects without any entry in a multi-valued attribute. A straightforward formulation for the above query request which does not qualify persons without hobbies is:

```
Q16 select p.name from person p
    where Set ("Hiking", "Football", "Aussie Football") >= p.hobbies
      and not p.hobbies.is_empty
```

In that case, the index can be used again (at least theoretically) as the following procedure demonstrates:

1. compute three intermediate results as in the example query above;
2. compute the next intermediate result as the union of the intermediate results produced in the first step (this yields the set of all OIDs referring to objects with at least one matching hobby);
3. compute the final result as follows: check for each object referenced by an OID in the previous intermediate result, if there are only matching hobbies in the value set of Person.hobbies, i.e., if there is no hobby in the value set which is not also an element of the query set; if this condition is satisfied, retrieve the corresponding value of attribute name.

In the final step, a search data structure indexing Person.hobbies can be used only if the particular structure implementation also supports forward traversals in addition to the standard backward traversal pattern (see Chapter 5). If this is not the case, all objects referenced by the OIDs found in step 2 have to be fetched from mass storage to determine the corresponding attribute values.

This could yield a severe performance degradation. Even if forward traversals are possible in a concrete index implementation, the still remaining question is if a query optimizer is able to find an appropriate query execution plan for the query specified above, i.e., an execution plan incorporating the index structure into the three execution phases outlined above.

Although all considerations refer to collections of literals so far, similar considerations hold for collections of identifiable objects. Consider a query that retrieves the names of all students who take *all* of the courses assisted by teaching assistant Kwapil:

Q_{17} select s.name from student s, teachingAssistant ta
where ta.assists <= s.courses and ta.name = "Kwapil"

An existing index for path Student.courses.assistants.name would be of limited use in this case, because it associates the string "Kwapil" with the OIDs of students who take *any* of the courses assisted by Kwapil. The same is true for a possible index on path Student.courses.assistants but not for an index on path Student.courses. The latter possibly existing index structure could be used for the execution of the above query, if in a prior execution phase the query set has been determined. The execution plan is as follows:

1. retrieve the set of courses assisted by Kwapil; this can be done by finding out the OID of the TeachingAssistant object whose attribute name has the value "Kwapil" and subsequently retrieving the value of its attribute assists; this subquery step can be optimized by a (possibly inherited) index on TeachingAssistant.name.

If this TeachingAssistant-object is ω_{22}, then the value of ω_{22}.assists (for example $\{\omega_{26}, \omega_{27}, \omega_{29}\}$) is determined as the query set and we can proceed as with query Q_{14} above, that is:

2. use the index on Student.courses to find for each element of the query set $\{\omega_{26}, \omega_{27}, \omega_{29}\}$ a corresponding set of OIDs of Person-objects whose property courses contains that element.
3. determine the set of OIDs of the qualified objects as the intersection of the three sets computed in the previous phase and determine for each of the resulting OIDs the current value of attribute name.

If we again modify query Q_{17} to retrieve the names of all students who take *only* courses assisted by Kwapil we obtain

Q_{18} select s.name from student s, teachingAssistant ta
where ta.assists >= s.courses and ta.name = "Kwapil"

The situation is the same as with query Q_{15}. Efficient query processing using the above mentioned path-index is not possible. The main reason is how index structures based on path decomposition (see Chapter 5) cope with multi-valued properties: recalling the structure of the object graph, each single-valued property of a particular object yields zero or one emerging arcs labeled with the property name. Consequently, each multi-valued property produces a (possibly empty) set of identically labeled arcs. In the standard case of a path decomposition index supporting backward traversals (i.e., the main retrieval pattern for any path index), the arcs of such a set are stored separately in the data structure. This storage organization implies that any information what OIDs or literals belong to the *same collection* (e.g., Person.hobbies or Student.courses.assistants) is lost. The situation is more favorable if the index structure also supports forward traversals. However, in this case the access item (search key) is the OID of an object having a multi-valued property, but not an element (literal or OID) of the multi-valued property itself.

Summing up, a standard path decomposition index supporting backward as well as forward traversals is useful for queries referring to multi-valued properties if we have to

- find all elements of a multi-valued property of a certain object (forward traversal): find all Course objects contained in ω_{15}.courses
- find all objects with a multi-valued property containing a certain *single element* (backward traversal): find all Student objects whose property courses contains Course object ω_{27}.

In general it does not, however, support a query like:

- find all objects having a multi-valued property with a certain *content* (backward traversal): find all Student-objects whose property courses contains only the Course-objects ω_{27} and ω_{29}.

A proposal that tries to overcome this kind of problems by indexing the *content* of a multi-valued property rather than its *single elements* is discussed in the following section.

6.2 SIGNATURE FILES FOR INDEXING MULTI-VALUED PROPERTIES

So called *signature files* as previously used in information retrieval (c.f. [FC84, FC87]) are proposed in [IKO93] as an appropriate search data structure for speeding up the execution of set operators acting on multi-valued attributes.

In the context of the indexing problem outlined above, the basic idea can be described as follows. If a multi-valued attribute like Person.hobbies should be indexed, the first step is to compute for each possible attribute value (i.e., for each element of the underlying domain from which the sets are constructed) the *element signature*. Technically speaking, such an element signature is a bit string of length F. For this bit string, m denotes the number of ones and therefore $F - m$ the corresponding number of zeros. Performance tuning in signature files is done by modifying these two parameter values (see below).

Based on the encoding of the underlying domain elements, for each set of elements a *set signature* can be computed. The set signature of a set with x elements is the bit string of length F resulting of the x-ary bitwise OR operation separately computed for each of the F bit positions in the x element signatures (see example below).

To speed up a set operation the corresponding set signature is computed and called *target signature*. This is done for each object with the multi-valued property to be indexed. Consequently, a *signature file* contains for each set s currently stored in the indexed multi-valued property p the set signature together with a list of OIDs. These OIDs refer to the objects having s as the current value for p (usually, several objects at once will have the same set as current value in the property). For example, if Person.hobbies is indexed by a signature file, the file contains the set signature for {"Hiking", "Football", "Aussie Football"} together with the OIDs of all objects with exactly these three hobbies stored in the property.

If a query request refers to the indexed multi-valued property, the set signature for the query set is built following the same procedure as described above. During index search, the query set signature is compared to all target set signatures stored in the signature file. Matching target set signatures yield *drops*. Let S_Q, S_T, $\mathrm{Sig}(S_Q)$ and $\mathrm{Sig}(S_T)$ denote a query set, a target set and their corresponding set signatures, respectively. For a query predicate $S_Q \subseteq S_T$, a target set is a drop if and only if for each bit set to 1 in the query set signature the corresponding bit in the target set signature is also set to 1. More formally,

S_T is a drop for $S_Q \subseteq S_T$ if and only if $(\text{Sig}(S_Q)\,\text{AND}\,\text{Sig}(S_T)) = \text{Sig}(S_Q)$ (bitwise AND).

It should be noted that any query predicate testing for set membership is in this context a special case, in particular a subset test predicate containing a singleton set.

A small example should further describe this indexing scheme. Figure 6.1 and Figure 6.2 show a query set together with two target sets (i.e., the set signatures of two stored objects) for query Q_{14}. Actual parameter values for F and m have been chosen somewhat arbitrarily with 8 and 2, respectively. The concrete query set signature $\text{Sig}(S_Q) = 01100011$ is the result

Query set

Query element	Signature
Hiking	00100001
Football	01000001
Aussie Football	00100010
Query signature	01100011

Figure 6.1 Query set and query signature for Q_{14} and Q_{15}

ω_{10}.hobbies (actual drop)

Target element	Signature
Hiking	00100001
Football	01000001
Aussie Football	00100010
Fishing	10010000
Target signature	11110011

ω_{11}.hobbies (false drop)

Target element	Signature
Football	01000001
Aussie Football	00100010
Soccer	01100000
Pole-vaulting	10000100
Target signature	11100111

Figure 6.2 Actual drop and false drop for Q_{14} ($S_Q \subseteq S_T$)

of F x-ary bitwise OR operations as described above. For this signature, all target set signatures $\text{Sig}(S_T)$ are retrieved from the signature file for which $(\text{Sig}(S_T)\,\text{AND}\,01100011) = 01100011$ holds. If we consider for example the target set signatures 11110011 and 11100111 (resulting from target sets ω_{10}.hobbies and ω_{11}.hobbies), both signatures render the above condition to true and are therefore drops in this terminology. However, only ω_{10}.hobbies actually qualifies for the query predicate $S_Q \subseteq S_T$. Thus we observe that some of the matching target set signatures refer to qualifying objects, others do not. Drops corresponding to qualifying objects are called *actual drops*, drops yielding objects

which are not qualifying are called *false drops*. To distinguish between actual and false drops, the objects have to be fetched from mass storage because the current values of the multi-valued attribute have to be checked.

The method guarantees that all objects qualified by the subset query predicate are found with the help of the corresponding signature file, or in other words are dropped during index search ($S_Q \subseteq S_T \Rightarrow (\text{Sig}(S_Q) \text{ AND } \text{Sig}(S_T)) = \text{Sig}(S_Q)$ holds in all cases), however, in general false drops cannot be avoided using signature files with reasonable values for parameters F and m. In fact, the false drop probability is a function of F and m: if F equals the number of elements in the underlying domain and $m = 1$, a particular position in the bit string can be used to encode a particular element of the domain and the false drop probability is zero. In all other cases (and therefore in all practically relevant cases) the probability is greater than zero.

Queries like Q_{15} can be processed with the help of a signature file in a similar manner. For query term $S_Q \supseteq S_T$, a target set S_T is a drop if and only if for each bit set to 1 in the target signature the corresponding bit in the query signature is set to 1. In other words, S_T is a drop for $S_Q \supseteq S_T$ if and only if $(\text{Sig}(S_Q) \text{ AND } \text{Sig}(S_T)) = \text{Sig}(S_T)$. Figure 6.3 contains two target sets for Q_{15}, the respective query set signature is shown in Figure 6.1. Target set signature

ω_{12}.hobbies (actual drop)

Target element	Signature
Football	01000001
Aussie Football	00100010
Target signature	01100011

ω_{13}.hobbies (false drop)

Target element	Signature
Football	01000001
Soccer	01100000
Target signature	01100001

Figure 6.3 Actual drop and false drop for Q_{15} ($S_Q \supseteq S_T$)

01100011 for ω_{12}.hobbies as well as 01100001 for ω_{13}.hobbies fulfill the above condition and therefore yield a drop. Again, we get one false drop, since the superset predicate is rendered to true only by ω_{12}.hobbies yielding an actual drop with ω_{13}.hobbies being a false drop. $S_Q \supseteq S_T \Rightarrow (\text{Sig}(S_Q) \text{ AND } \text{Sig}(S_T)) = \text{Sig}(S_T)$ holds.

Additionally set operators can be implemented by the same technique:

- as already mentioned above, the processing of the membership operator $\in$ (in) basically is a special case of the procedure for the inclusion operators $\subseteq$ (<=).

- the operators $\subset$ (<) and $\supset$ (>) are handled exactly the same way as $\subseteq$ (<=) and $\supseteq$ (>=)
- this mechanism can be also used to check whether or not two sets have the same elements, i.e., whether or not they are equal: $S_Q = S_T \Rightarrow \text{Sig}(S_Q) = \text{Sig}(S_T)$
- it is possible to check if target set and query set have a non-empty intersection (the corresponding operator is sometimes called *overlap operator*; in particular, condition $S_Q \cap S_T \neq \emptyset \Rightarrow (\text{Sig}(S_Q) \text{ AND } \text{Sig}(S_T)) \neq 0$ can be used for that purpose, if for example one would like to retrieve all teaching assistants who assist courses they take themselves:

  ```
  select ta from teachingAssistant ta
  where not (ta.assists intersect ta.courses).is_empty
  ```

When indexing collections of (identifiable) objects, the only technical difference in such a case is that element signatures are computed from OIDs rather than literals like "Pole-vaulting". From the indexing subsystem's point of view, in

ω_{22}.assists (query set)

Query element	Signature
ω_{26}	01010000
ω_{27}	00001010
ω_{29}	01000100
Query signature	01011110

ω_{15}.courses (actual drop)

Target element	Signature
ω_{27}	00001010
ω_{29}	01000100
Target signature	01001110

ω_{13}.courses (false drop)

Target element	Signature
ω_{27}	00001010
ω_{28}	00000110
Target signature	00001110

Figure 6.4 Query set, actual drop, and false drop for Q_{18} ($S_Q \supseteq S_T$)

both cases byte strings are encoded as bit strings, so there is no difference at all at the technical level. Figure 6.4 contains the query set together with two target sets for query Q_{18}. Again, we assume $F = 8$ and $m = 2$. Both target sets become a drop because $(\text{Sig}(S_T) \text{ AND } 01011110) = \text{Sig}(S_T)$ holds for both 01001110 and 00001110. ω_{13}.courses is a false drop, however, because ω_{28} is a member of this target set but not of the query set ω_{22}.assists.

6.3 BIBLIOGRAPHY

[IKO93] propose signature files for optimizing set operators. Storage overhead as well as retrieval (for $\subseteq$ and $\supseteq$) and update cost of two variants, namely the *sequential signature file* and the *bit-sliced signature file* are compared to the corresponding performance values of a nested index. Estimations of false drop probabilities for the comparison operators $\subseteq$, $\supseteq$, $\sqcap$ (set overlap), and $\equiv$ (set equivalence) can be found in [KFIO93]. [Nov94] compares the retrieval costs for the $\subseteq$ and $\supseteq$ operators with the performance of nested index and access support relations.

In [YLK94] signature files are used to speed up forward queries. For objects that are referenced by other objects through properties, signatures are computed for their property values and stored in the referencing objects. Thus, properties of referenced objects can be compared without accessing those objects and forward traversals can be aborted if it is clear from the property signature that the referenced objects do not fulfill the relevant predicates.

Signature files have also been used in the context of aggregation path indexing for *Path signatures* [LL92] and *Signature Path Dictionaries* [LL96].

7

PERFORMANCE ANALYSIS – AN EXAMPLE

This chapter contains an example for the analytical evaluation and comparison of different indexing proposals. In particular, index size and query performance of the MT-index (see Section 4.4), the CH-index (see Section 4.3.1), and the H-tree (see Section 4.2.2) are compared. As usual there is no dominating approach for all hierarchy structures and query profiles, however, we are able to provide a few rules of thumb for index selection.

As in [LOL92] and [KKD89] and most other analytical approaches our estimations are based on *best case* assumptions, e.g., maximum index page occupancy. The corresponding average case figures would not yield significantly different results (see [LS90] and [OHLT96]). A central parameter of the following models (see [KKD89] and [LOL92]) is denoted by nt and describes *attribute configuration*: each value of the indexed attribute occurs in objects of nt types. Let d and d_i denote the number of different stored values of the indexed attribute and the number of different values stored in objects of type t_i, respectively. The relationship between these three parameters is described by $d_i = d \cdot \frac{nt}{|T|}$ with $1 \leq nt \leq |T|$. We interpret the two extreme values of nt: $nt = 1 \Rightarrow d = \sum_{|T|} d_i$ and $nt = |T| \Rightarrow d = d_i$. In the first case (no overlap) each attribute value occurs in objects of exactly one type, whereas in the second case (full overlap) each attribute value occurs in objects of all types. All other parameters are described in Table 7.1. Storage space assumptions made for particular data items are given in Table 7.2. *Rec* and *Dir* refer to leaf node records and internal directories maintained in leaf nodes, respectively. In what follows, the analytical models for the single key structures are based on slightly modified material presented in [LOL92].

T. A. Mueck et al., *Index Data Structures in Object-Oriented Databases*

Param.	*Meaning*
T	Set of indexed types
$T^Q \subset T$	Set of types referenced by a query Q
k	Number of indexed attributes
$d^Q \leq d$	Number of attribute values qualified by query Q
n	Number of objects
n_i	Number of objects of type $t_i \in T$
f	fanout: number of successors of an internal node
size(x)	Size of x in bytes
e	Number of records in an index leaf node
n_{vt}	Number of objects per value per type
b_i^L	Number of index leaf nodes for t_i
b_i^I	Number of internal index nodes for t_i
b_i	Total number of index nodes for t_i

Table 7.1 Model parameters

Data item	*Size*	*Meaning*
Node	4096	Index node (internal or leaf)
Counter	2	Counter used for OIDs, node entries, ...
Att	4	Indexed attribute
OID	12	Object identifier
Offset	2	Offset within a node
TypeId	2	Type identifier
NodeId	8	Node identifier

Table 7.2 Size of data items

7.1 STORAGE SPACE REQUIREMENTS

7.1.1 Analytical model

The CH-index is a balanced multiway tree featuring a sophisticated leaf node organization: in a leaf node, OIDs are grouped by attribute value and object type. In contrast to this approach, the nesting of an H-tree corresponds to the structure of the type hierarchy, therefore a type oriented leaf node organization is obsolete. Due to the structure of the CH-index (in contrast to the H-tree),

the parameters b, b^L, and b^I are used without subscript i.

$$\text{size}(Dir) = \text{size}(Counter) + nt \cdot (\text{size}(TypeId) + \text{size}(Offset))$$
$$\text{size}(Rec^{(CH)}) = \text{size}(Att) + \text{size}(Dir) + nt \cdot (\text{size}(Counter) + n_{vt} \cdot \text{size}(OID))$$

$$\begin{aligned} e^{(CH)} &= \left\lfloor \frac{\text{size}(Node) - \text{size}(Counter)}{\text{size}(Rec^{(CH)})} \right\rfloor \\ b^L &= \left\lceil \frac{d}{e^{(CH)}} \right\rceil \\ b^I &= \left\lceil \frac{b^L}{f^{(CH)}} \right\rceil + \left\lceil \frac{\left\lceil \frac{b^L}{f^{(CH)}} \right\rceil}{f^{(CH)}} \right\rceil + \cdots + 1 \end{aligned}$$

with

$$f^{(CH)} = \left\lfloor \frac{\text{size}(Node)}{\text{size}(Att) + \text{size}(NodeId)} \right\rfloor$$

Maintain one CH-index for each indexed attribute[1] yields a total storage space consumption for a CH-index implementation of

$$b^{(CH)} = k \cdot (b^L + b^I)$$

In case of the H-tree, we have to calculate the size by means of the overall sum for all $|T|$ nested type-specific multiway trees. Consequently, we obtain for this data structure

$$\text{size}(Rec^{(H)}) = \text{size}(Att) + \text{size}(Counter) + n_{vt} \cdot \text{size}(OID)$$

$$\begin{aligned} e^{(H)} &= \left\lfloor \frac{\text{size}(Node) - \text{size}(Counter)}{\text{size}(Rec^{(H)})} \right\rfloor \\ b_i^L &= \left\lceil \frac{d_i}{e^{(H)}} \right\rceil \\ b_i^I &= \left\lceil \frac{b_i^L}{f^{(H)}} \right\rceil + \left\lceil \frac{\left\lceil \frac{b_i^L}{f^{(H)}} \right\rceil}{f^{(H)}} \right\rceil + \cdots + 1 \\ b_i &= b_i^L + b_i^I \end{aligned}$$

[1] In this model, we assume that the execution of a k-attribute query results in the traversal of k single key indices. A possible alternative is the traversal of *one* index structure and the subsequent fetching of the referenced objects. However, objects not qualifying for the query (i.e., objects with non-matching values in the other attributes) have to be discarded afterwards. The problem in this context is the significant waste of I/O bandwidth caused by the transfer of non-qualifying objects.

with

$$f^{(H)} = \left\lfloor \frac{2}{3} \cdot \frac{\text{size}(Node)}{\text{size}(Att) + \text{size}(NodeId)} \right\rfloor$$

The reduced fanout is caused by the link entries. These tuples (consuming one third of the internal nodes) are used to implement the nesting, see above. Again, we assume one H-tree per indexed attribute, and consequently estimate a storage space consumption of

$$b^{(H)} = k \cdot \sum_{t_i \in T} b_i$$

In case of an hB-tree used to implement an MT-index we have to distinguish between the abovementioned leaf node organizations (see Section 4.4.4, excluding the pre-splitting alternatives):

- grouping by attribute value and type:

$$\begin{aligned} \text{size}(Rec^{(hB)}) = {} & k \cdot \text{size}(Att) + \text{size}(TypeId) + \text{size}(Counter) \\ & + n_{vt} \cdot \text{size}(OID) \end{aligned}$$

$$\begin{aligned} e^{(hB)} &= \left\lfloor \frac{\text{size}(Node) - \text{size}(Counter)}{\text{size}(Rec^{(hB)})} \right\rfloor \\ b^L &= \left\lceil \frac{n}{e^{(hB)} \cdot n_{vt}} \right\rceil \end{aligned}$$

- no grouping:

$$\text{size}(Rec^{(hB)}) = k \cdot \text{size}(Att) + \text{size}(TypeId) + \text{size}(OID)$$

$$\begin{aligned} e^{(hB)} &= \left\lfloor \frac{\text{size}(Node) - \text{size}(Counter)}{\text{size}(Rec^{(hB)})} \right\rfloor \\ b^L &= \left\lceil \frac{n}{e^{(hB)}} \right\rceil \end{aligned}$$

In the following evaluation the first variant is chosen for the case of one indexed attribute and the second variant in case of more than one indexed attribute. At this point, a closer look at the fanout of hB-tree nodes is needed. The internal nodes are organized as k-d trees [Ben75] (see Section 3.4.1. Each k-d tree node

contains one attribute value, two node identifiers of successor nodes, and two additional bytes of maintenance information. The resulting fanout

$$f^{(hB)} = \left\lfloor \frac{\text{size}(Node)}{(\text{size}(Att) + 2 \cdot \text{size}(NodeId) + 2) \cdot r} \right\rfloor$$

contains a reduction factor r quantifying the overhead of the k-d tree organization. In particular, a number of references in the k-d tree again reference k-d tree nodes belonging to the same hB-tree node. These references do not contribute to the fanout of the hB-tree node. In [LS90] an evaluation yields $1 \leq r \leq 1.5$, in our case variations of r do not yield significantly different results. Based on this fanout we obtain

$$b^I = \left\lceil \frac{b^L}{f^{(hB)}} \right\rceil + \left\lceil \frac{\left\lceil \frac{b^L}{f^{(hB)}} \right\rceil}{f^{(hB)}} \right\rceil + \cdots + 1$$

$$b^{(hB)} = b^L + b^I$$

7.1.2 Evaluation of the index size

Using this model we present the results of two experiments. Both experiments are made for $k = 1$ and $k = 2$ (i.e., one indexed attribute in contrast to two indexed attributes) as well as for $nt = 1$, $nt = \frac{|T|}{2}$, and $nt = |T|$ (no overlap, partial overlap, full overlap). For $k = 1$ a leaf node organization (grouping by value and type) is used for the hB-tree, whereas for $k > 1$ this additional organization is omitted.

1. For fixed n_i and d, $|T|$ is increased, i.e., for a constant number of objects per type new types are added. Actual experimental values are $n_i = 10000$, $d = 10000$ and $1 \leq |T| \leq 50$. Consequently, n is increased from 10000 to 500000. The results for the CH-index, the H-tree, and the MT-index are depicted in Figures 7.1–7.3. In this figure and all following figures, solid lines represent the MT-index, dashed lines the H-tree and dotted lines the CH-index.

2. Although the setup for this experiment seems almost the same as in Experiment 1, the application context is totally different: For fixed n and d, $|T|$ is increased, i.e., for a constant number of objects the number of types is increased. This corresponds to the creation of new types and a subsequent migration of existing objects to the newly created types. Actual experimental values are $n = 500000$, $d = 10000$ and $1 \leq |T| \leq 50$. Results are presented in Figures 7.4–7.6.

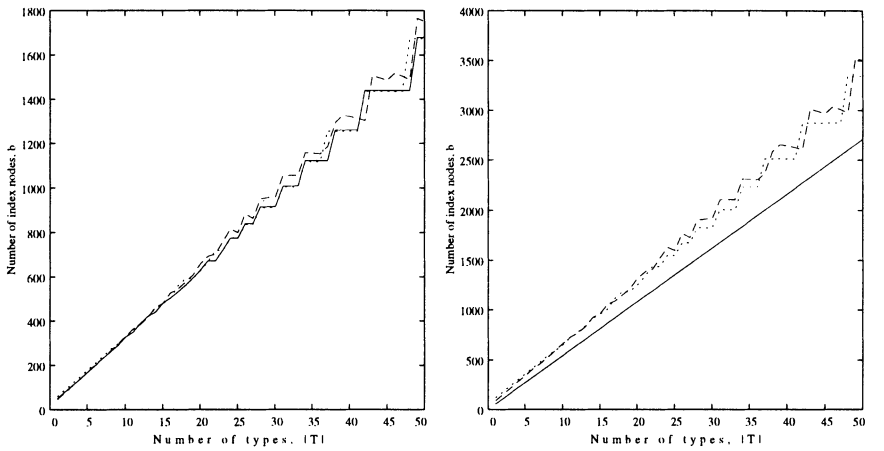


Figure 7.1 Index size: results of Experiment 1 for $nt = 1$, $k = 1$ and $k = 2$

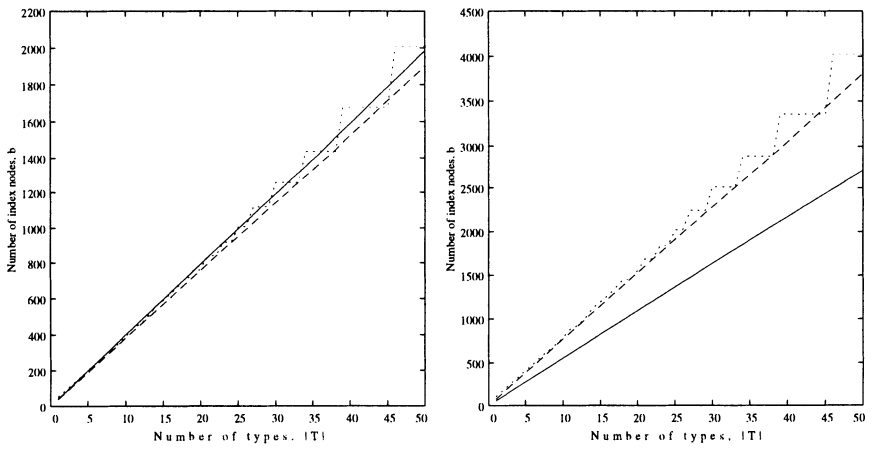


Figure 7.2 Index size (Experiment 1): $nt = \frac{|T|}{2}$, $k = 1$ and $k = 2$

Results of Experiment 1: In the one-dimensional case the storage space consumption of the three approaches is more or less the same, for overlapping attribute configurations the H-tree dominates the MT-index by about 5% in case of partial overlap and by about 8% in case of full overlap.

Unsurprisingly, for $k > 1$ the MT-index outperforms the single key approaches with respect to index size. The space overhead reduction is most appealing for the case of full overlap and less impressive in case of a non-overlapping attribute configuration.

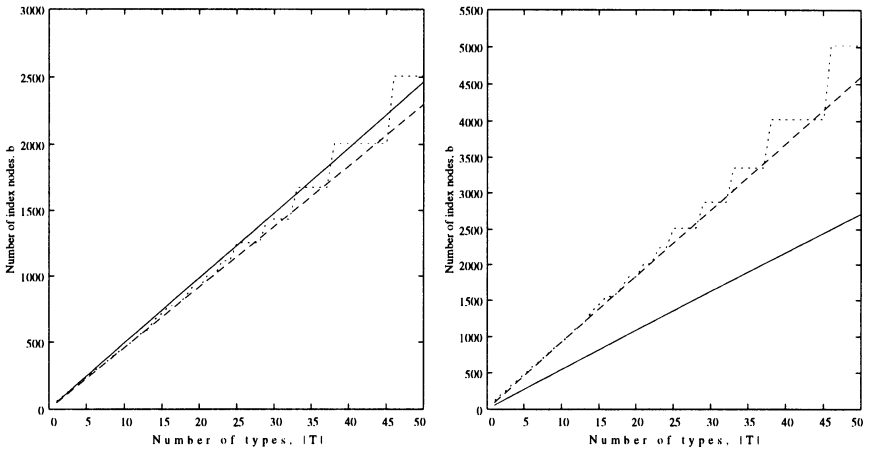


Figure 7.3 Index size (Experiment 1): $nt = |T|$, $k = 1$ and $k = 2$

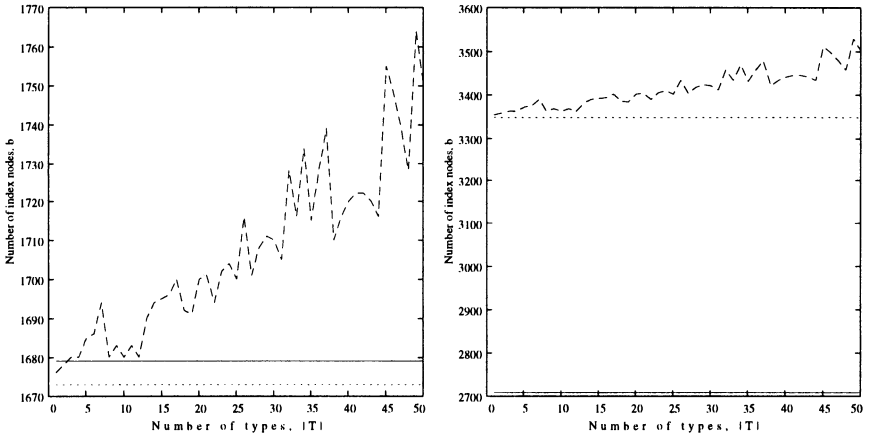


Figure 7.4 Index size (Experiment 2): $nt = 1$, $k = 1$ and $k = 2$

Results of Experiment 2: In case of increasing $|T|$ with fixed n, the influence of nt is even stronger than in the case of increasing n and $|T|$. Again, discussing the single attribute case first, the OIDs in hB-tree nodes are grouped by combinations of attribute value and type. Consequently, in the non-overlapping case the size of the hB-tree is constant, because existing attribute values and the corresponding OIDs are assigned to the newly created types, i.e., n_{vt} is constant. The same holds for the CH-index, but not for the H-tree, where the creation of new index trees causes a slightly increased storage overhead.

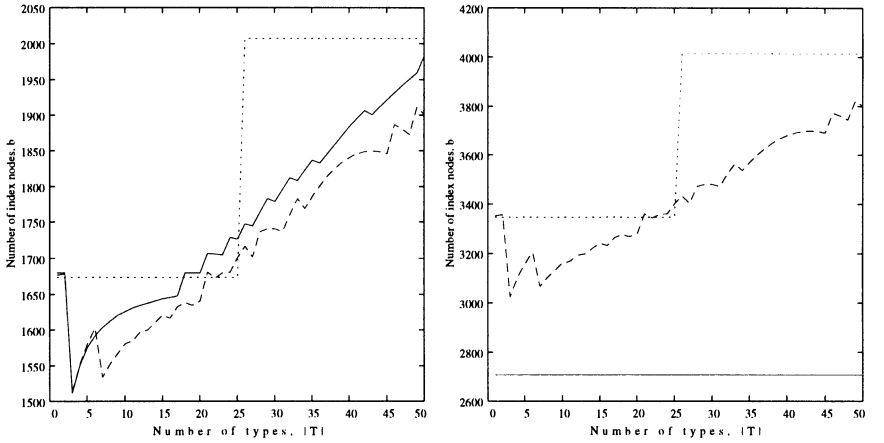


Figure 7.5 Index size (Experiment 2): $nt = \frac{|T|}{2}$, $k = 1$ and $k = 2$

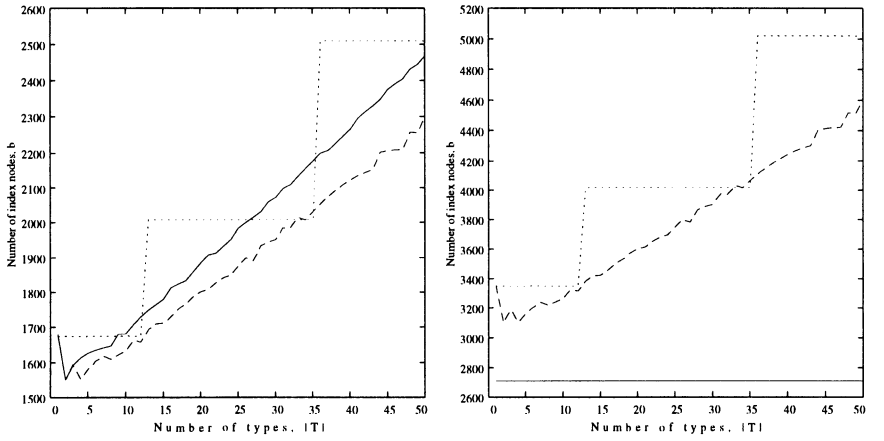


Figure 7.6 Index size (Experiment 2): $nt = |T|$, $k = 1$ and $k = 2$

In case of partial and full overlap the ranking between CH-index and H-tree changes, the MT-index is somewhere in between. In case of partial overlap the advantage of the H-tree over the MT-index is upper bound by about 5% (9% in case of full overlap). The result for the CH-index is caused by the comparatively small number of records per leaf node (6 records for $|T| \leq 12$, 5 for $12 < |T| \leq 35$ and 4 for $|T| > 35$).

The results for two indexed attributes (see right-hand side of Figures 7.4–7.6) are almost self-explanatory. The overall index size yielded by the single-key

approaches is simply k times the index size of the single-attribute case. Due to the robust directory organization of the hB-tree the index size does not increase if the number of stored OIDs does not increase. As there is no additional hB-tree leaf node organization for $k > 1$ neither the attribute configuration nor $|T|$ has any impact on the index size.

7.2 QUERY PERFORMANCE

7.2.1 Analytical model

Assuming a traversal of k single key indices for a k-attribute query, the estimated number of disk-I/O operations for a query qualifying d^Q attribute values is given by

$$k \cdot \left(height^{(CH)} + \left\lceil \frac{d^Q}{e^{(CH)}} \right\rceil \right)$$

for the CH-index and by

$$k \cdot \sum_{t_i \in T^Q} \left(height_i^{(H)}) + \left\lceil \frac{\frac{d_i}{d} \cdot d^Q}{e^{(H)}} \right\rceil \right)$$

for the H-tree. $height^{(CH)}$ ($height_i^{(H)}$) denotes the height of the CH-index (the H-tree for type t_i).

An estimation for the number of disk-I/O operations in an n-dimensional search data structure like the hB-tree is less straightforward. We use a probabilistic model outlined in [Sch93]. In particular, the number of I/O operations for a multi-attribute range query is represented by a random variable $V(S, Q, R)$ modeling the fetched data volume in a data space (normalized to $[0, 1[^n$) in terms of attribute domain structure $S(s_1, \cdots, s_n)$, query profile $Q(q_1, \cdots, q_n)$, and data space partitioning $R(r_1, \cdots, r_n)$. An example in Figure 7.7 is given for particular S, Q and R. In this representation, one tuple corresponds to a minimal rectangle in the data space (see (3, 5) in Figure 7.7(a)). The data space partitioning $R(\frac{1}{2}, \frac{1}{4})$ yields 8 mass storage transfer units (see Figure 7.7(b)). The number of disk-I/O operations depends on the actual position of a query template in the data space. For example in Figure 7.7(c), the same query template, i.e., $Q(\frac{1}{4}, \frac{6}{16})$ causes either two or three I/O operations, depending on the actual placement of the query window in the data space.

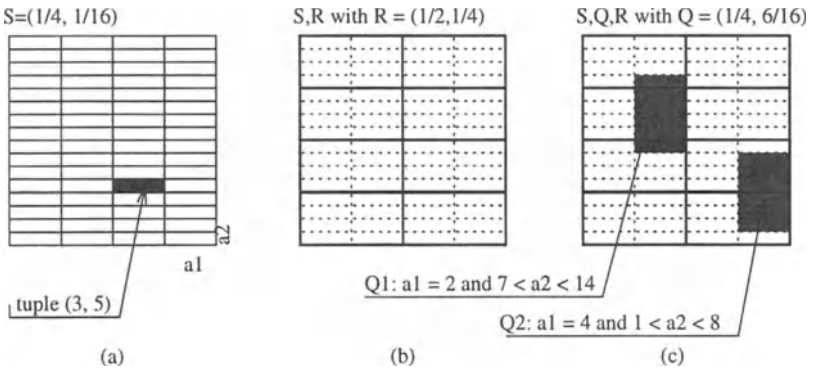


Figure 7.7 An example for an S, Q, R configuration

More precisely, let $W_j(s_j, q_j, r_j)$ denote the corresponding random variable for dimension j. A closer look at the subject yields only two actual values for W_j, in particular

$$w'_j = \left\lceil \frac{q_j}{r_j} \right\rceil r_j \quad \text{and} \quad w''_j = \left\lceil \frac{q_j}{r_j} \right\rceil r_j + r_j$$

and a probabilistic density function of

$$\mathrm{P}(W_j = b_j) = \frac{(b_j - q_j + s_j)(1 - b_j + r_j)}{r_j(1 - q_j + s_j)}, \tag{7.1}$$

$$\mathrm{P}(W_j = b_j + r_j) = 1 - \mathrm{P}(W_j = b_j) = \frac{(1 - b_j)(q_j - b_j + r_j - s_j)}{r_j(1 - q_j + s_j)} \tag{7.2}$$

with

$$b_j = \left\lceil \frac{q_j}{r_j} \right\rceil r_j.$$

Equations (7.1) and (7.2) yield an expectation value for W_j

$$\mathrm{E}(W_j) = b_j + \frac{(1 - b_j)(q_j + r_j - s_j - b_j)}{1 - q_j + s_j}, \tag{7.3}$$

i.e., the estimated interval length in dimension j as a function of (s_j, r_j, q_j).

Under certain distribution assumptions (uniformly distributed and uncorrelated raw data as well as query windows), we derive $V(S, R, Q)$ as the product over all W_j, i.e.,

$$V = \prod_{1 \le j \le n} W_j. \tag{7.4}$$

and the corresponding expectation value as

$$\mathrm{E}(V) = \prod_{1 \le j \le n} \left(b_j + \frac{(1 - b_j)(q_j + r_j - s_j - b_j)}{1 - q_j + s_j} \right) \tag{7.5}$$

with

$$b_j = \left\lceil \frac{q_j}{r_j} \right\rceil r_j .$$

Convenient approximations for equations (7.3) and (7.5) are

$$\widetilde{E}(W_j) = \begin{cases} q_j + r_j - s_j & \text{if } q_j \le 1 - r_j \\ 1 & \text{if } q_j > 1 - r_j \end{cases}$$

and

$$\widetilde{E}(V) = \prod_{1 \le l \le n} \widetilde{E}(W_j)$$

In our application context $n = k + 1$ holds for k indexed attributes, i.e., one dimension representing the type domain. The model parameters for the type domain are calculated as follows. The domain structure and the query range for the type dimension are given by $s_1 = \frac{1}{|T|}$ and $q_1 = \frac{|T^Q|}{|T|}$, the partitioning of the type domain is given by

$$r_1 = \begin{cases} \dfrac{1}{\sqrt[k+1]{b^L}} & \text{if } |T|^{k+1} \le b^L \\ \dfrac{1}{|T|} & \text{otherwise} \end{cases} \tag{7.6}$$

In other words, the standard splitting strategy is also applied to the type domain[2]. Considering a maximum split potential of $|T|$ subintervals in the type dimension and the overall split potential given by the number of data nodes b^L we obtain the two cases described in equation (7.6).

The model parameters for all other dimensions are given by $s_j = \frac{1}{d}$, $q_j = \frac{d^Q}{d}$, and $r_j = \frac{1}{\sqrt[k]{r_1 \cdot b^L}}$. Recalling that $\widetilde{E}(V)$ is an approximation for the expected volume in $[0,1[^n$ we obtain $\left\lceil b^L \cdot \widetilde{E}(V) \right\rceil$ I/O operations for data nodes. The number of index node I/O operations ranges from $height^{(hB)}$ nodes in the case of a point query to b^I nodes for a range query, assuming the unlikely case, that all internal nodes have to be read during query execution. For the worst case we therefore obtain $b^I + \left\lceil b^L \cdot \widetilde{E}(V) \right\rceil$ disk-I/O operations for range query execution.

[2]We do not include the pre-splitting scheme mentioned in Section 4.4.4 in this analytical model.

7.2.2 Evaluation of query performance

We describe three experimental setups and discuss the resulting figures. Experiments 1 and 2 deal with range queries, whereas Experiment 3 focuses on exact match (or point) queries. The evaluation is done for $k = 1$ and $k = 2$ assuming fully overlapping attribute configuration ($nt = |T|$).

1. range query; number of qualified types fixed; query range varying: for $n = 500000$, $d = 10000$, $|T| = |T^Q| = 10$ the number of I/O operations is calculated for increasing query ranges. Is this context the query range is varied between 10% and 100% of the attribute domain, i.e., $0.1 \leq \frac{d^Q}{d} \leq 1$. The corresponding results are depicted in Figure 7.8.

2. range query; query range fixed; number of qualified types varying: for $n = 500000$, $d = 10000$, $\frac{d^Q}{d} = 0.2$ (i.e., 20% of the attribute domain qualified), $|T| = 10$ the number of I/O operations is calculated for different sets of qualified types T^Q with $1 \leq |T^Q| \leq 10$. Results are given in Figure 7.9.

3. exact match query; number of qualified types varying: for $n = 500000$, $d = 10000$, $d^Q = 1$, $|T| = 10$ the number of I/O operations is calculated for different sets of qualified types T^Q with $1 \leq |T^Q| \leq 10$. Results are given in Figure 7.10.

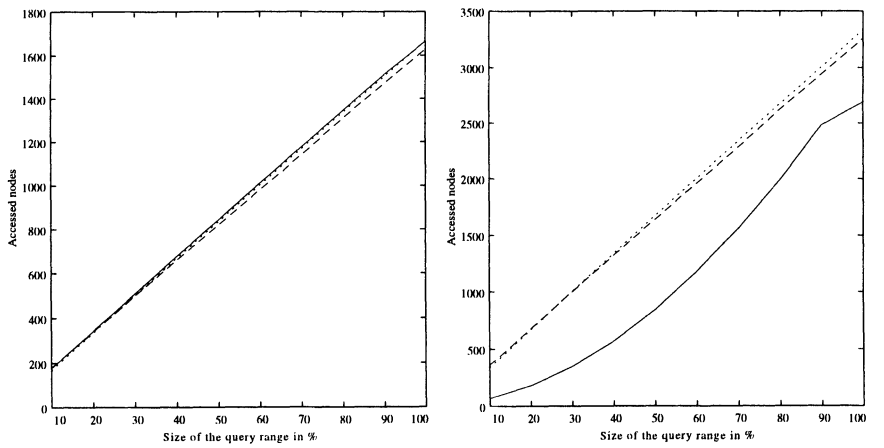


Figure 7.8 Query performance: varying range, $k = 1$ and $k = 2$

Results of experiment 1: In the one-dimensional case the results for all three approaches are almost the same. Compared to the single-key approaches

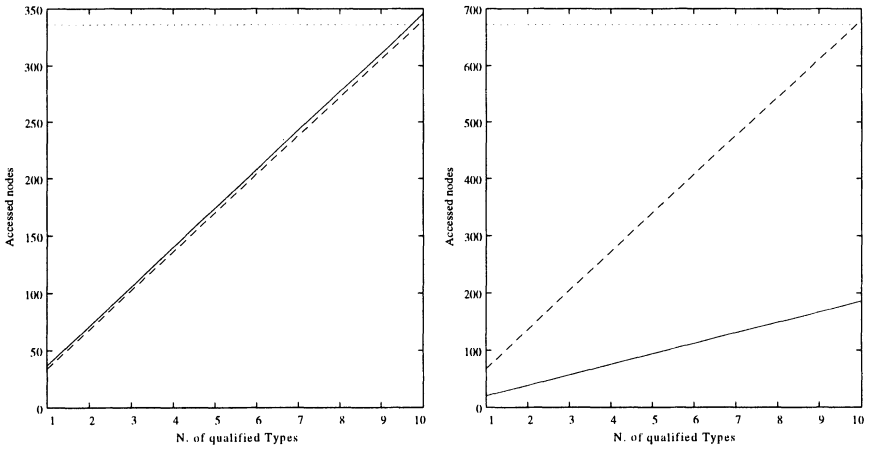


Figure 7.9 Query performance: varying $|T^Q|$, $k = 1$ and $k = 2$

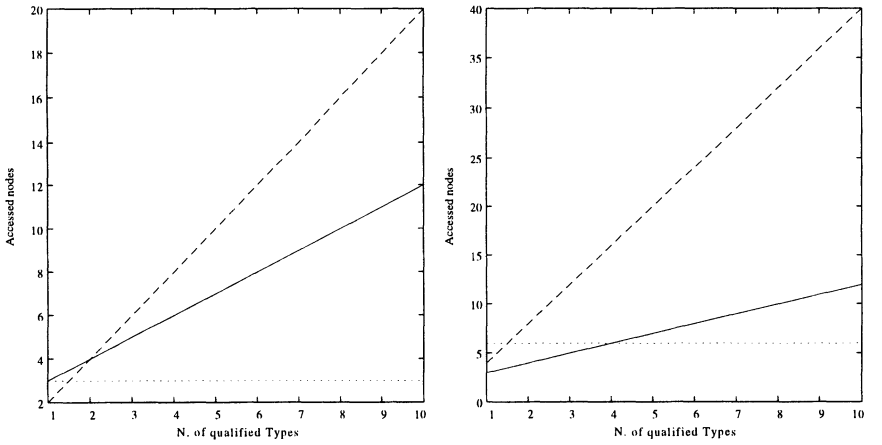


Figure 7.10 Query performance: exact match, varying $|T^Q|$, $k = 1$ and $k = 2$

the MT-index causes a 3% (1%) increase in the number of I/O operations compared to the H-tree (CH-index). In the case of $k = 2$ the MT-index has a performance advantage of up to 80% for small query ranges. Very large query intervals, e.g., $\frac{d^Q}{d} = 0.5$ still yield almost 50%.

Results of Experiment 2: In contrast to the previous experiment the second range query setup produces significantly different results for the CH-index on the one hand and for the H-tree and the MT-index on the other hand. For

arbitrary k, the CH-index is characterized by a constant number of I/O operations. The reason is that OIDs of *all types* are stored in one data structure without sufficient support for type specific access. As a result, all leaf nodes containing qualified attribute values have to be read. Subsequently, OIDs of non-qualified types have to be discarded. In this context the H-tree has a clear advantage, because for each type a distinct tree is maintained and only trees of qualified types have to be scanned. An interesting point is that the MT-index without type-separated trees performs similar to the H-tree, loosing only about 3% I/O performance. We can draw a first "non-obvious" conclusion: the type dimension approach in the MT-index framework is a competitive alternative to the maintenance of type-separated trees, e.g. H-trees. For multi-attribute configurations the performance advantage of the MT-index is most appealing for small query ranges (70% advantage over the H-tree for $\frac{d^Q}{d} = 0.2$) and deteriorates slightly in case of larger query intervals.

Results of Experiment 3: With respect to exact match queries (not considered in [LOL92, OHLT96]) the results are completely different. In case of the CH-index a single tree traversal is sufficient to answer an exact match query, whereas in the H-tree again $|T^Q|$ trees have to be traversed. For large $|T^Q|$ the MT-index performs significantly worse than the CH-index, however, in comparison to the H-tree, an up to 40% performance gain for $k = 1$ (up to 70% for $k = 2$) can be observed.

Summing up the evaluation of the query performance we have to distinguish between range queries and exact match queries. With respect to range queries we observe that for $k = 1$ and $|T| = |T^Q|$ the three compared approaches produce similar results. For $k = 1$ and $|T| < |T^Q|$ the H-tree and the MT-index outperform the CH-index. For $k > 2$ the hB-tree implementation of the MT-index dominates both previous approaches. In case of exact match queries the conclusion is that for $k = 1$ as well as for $k = 2$ the CH-index is the best choice. The H-tree suffers from the traversals of $|T^Q|$ type-separated trees; the MT-index performance is somewhere in between.

REFERENCES

[ABD+89] Malcolm Atkinson, François Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The object-oriented database system manifesto. In DOOD'89 [DOO89], pages 223–240.

[BDK92] François Bancilhon, Claude Delobel, and Paris Kanellakis, editors. *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[Bee89] Catriel Beeri. Formal models for object-oriented databases. In DOOD'89 [DOO89], pages 405–430.

[Bee90] Catriel Beeri. A formal approach to object-oriented databases. *Data & Knowledge Engineering*, 5(4):353–382, October 1990.

[Ben75] J.L. Bentley. Multidimensional binary search trees used for associative searching. *CACM*, 18(9):509–517, September 1975.

[Ben79] J.L. Bentley. Multidimensional binary search trees in database applications. *IEEE Transactions on Software Engineering*, 5(4):333–340, July 1979.

[Ber90] Elisa Bertino. Optimization of queries using nested indices. In *Advances in Database Technology - EDBT '90. Proc. Int. Conf. on Extending Database Technology*, volume 416 of *LNCS*, pages 44–59, Venice, Italy, March 1990. Springer-Verlag.

[Ber91] Elisa Bertino. An indexing technique for object-oriented databases. In *Proc. Seventh Int. Conf. on Data Engineering*, pages 160–170, Kobe, Japan, April 1991. IEEE CS Press.

[Ber94] Elisa Bertino. Index configuration in object-oriented databases. *VLDB Journal*, 3(3):355–399, July 1994.

[BF94] François Bancilhon and Guy Ferran. ODMG-93: The object database standard. *IEEE Bulletin of the Technical Committee on Data Engineering*, 17(4):3–14, December 1994.

[BF95] Elisa Bertino and Paola Foscoli. Index organizations for object-oriented database systems. *IEEE Transactions on Knowledge and Data Engineering*, 7(2):193–209, April 1995.

[BG92] Elisa Bertino and Claudia Guglielmina. Optimization of object-oriented queries using path-indices. In *RIDE-TQP, Second Int. Workshop on Research Issues in Data Engineering: Transaction and Query Processing*, Phoenix, AZ, February 1992.

[BG93] Elisa Bertino and Claudia Guglielmina. Path-index: An approach to the efficient execution of object-oriented queries. *Data & Knowledge Engineering*, 10(1):1–27, February 1993.

[BK89] Elisa Bertino and Won Kim. Indexing techniques for queries on nested objects. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):196–214, June 1989.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1(3):173–189, 1972.

[BM91] Elisa Bertino and Lorenzo Martino. Object-oriented database management systems: Concepts and issues. *IEEE Computer*, 24(4):33–47, April 1991.

[BM93] Elisa Bertino and Lorenzo Martino. *Object-Oriented Database Systems.* Computer Science Series. Addison-Wesley, 1993.

[BMO+89] Robert Bretl, David Maier, Allen Otis, Jason Penney, Bruce Schuchardt, Jacob Stein, E. Harold Williams, and Monty Williams. The GemStone data management system. In Kim and Lochovsky [KL89], pages 283–308.

[Cam96] R. Camps. Domains, relations and religious wars. *ACM SIGMOD Record*, 25(3):3–9, September 1996.

[Cat94a] Roderick Geoffrey Galton Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufmann Publishers, San Mateo, California, 1994.

[Cat94b] Roderick Geoffrey Galton Cattell, editor. *The Object Database Standard: ODMG-93 (Release 1.1).* The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, San Mateo, California, 1994.

[Cat96] Roderick Geoffrey Galton Cattell, editor. *The Object Database Standard: ODMG-93 (Release 1.2).* The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, San Mateo, California, 1996.

[CBBC94] Sunil Choenni, Elisa Bertino, Henk M. Blanken, and Thiel Chang. On the selection of optimal index configuration in object-oriented databases. In ICDE'94 [ICD94], pages 526–537.

[CCY94] Sudarshan S. Chawathe, Ming-Syan Chen, and Philip S. Yu. On index selection schemes for nested object hierarchies. In VLDB'94 [VLD94], pages 331–341.

[CDLR89] Sophie Cluet, Claude Delobel, Christophe Lécluse, and Philippe Richard. Reloop, an algebra based query language for an object-oriented database system. In DOOD'89 [DOO89], pages 313–332.

[CDLR90] Sophie Cluet, Claude Delobel, Christophe Lécluse, and Philippe Richard. Reloop, an algebra based query language for an object-oriented database system. *Data & Knowledge Engineering*, 5:333–352, 1990.

[CGO97] Chee Yong Chan, Cheng Hian Goh, and Beng Chin Ooi. Indexing OODB instances based on access proximity. In ICDE'97 [ICD97], pages 14–21.

[Cod70] E.F. Codd. A relational model for large shared data banks. *CACM*, 13(6):377–387, June 1970.

[Dar96] Hugh Darwen. In reply to domains, relations and religious wars. *ACM SIGMOD Record*, 25(4):6–7, December 1996.

[Dat96] C.J. Date. A response to r. camps' article "domains, relations and religious wars". *ACM SIGMOD Record*, 25(4):3–5, December 1996.

[DKO+84] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and A. Wood. Implementation techniques for main memory database systems. In *SIGMOD'84, Proc. of Annual Meeting*, pages 1–8, Boston, MA, June 1984.

[DOO89] *Deductive and Object-Oriented Databases. Proc. of the First Int. Conf. on Deductive and Object-Oriented Databases (DOOD)*, Kyoto, Japan, December 1989. Elsevier Science Publishers.

[DOO91] *Deductive and Object-Oriented Databases. Proc. of the Second Int. Conf., DOOD*, volume 566 of *LNCS*, Munich, Germany, December 1991. Springer-Verlag.

[ELS95] Georgios Evangelidis, David Lomet, and Betty Salzberg. The hB^{Π}-tree: A modified hB-tree supporting concurrency, recovery and node consolidation. In *Proc. of 21th Int. Conf. on Very Large Data Bases*, pages 551–561, Zürich, Switzerland, September 1995. Morgan Kaufmann Publishers.

[FAC+89] D.H Fishman, J. Annevelink, E. Chow, T. Connors, J.W. Davis, W. Hasan, C.G. Hoch, W. Kent, S. Leichner, P. Lyngbaek, B. Mahbod, M.A. Neimat, T. Risch, M.C. Shan, and W.K. Wilkinson. Overview of the Iris DBMS. In Kim and Lochovsky [KL89], pages 219–250.

[Fal86] Christos Faloutsos. Multiattribute hashing using gray codes. In *Proc. of 1986 ACM SIGMOD Int. Conf. on Management of Data*, pages 227–238, Washington, D.C., May 1986. ACM Press.

[FC84] Christos Faloutsos and S. Christodoulakisi. Signature files: An access method for documents and its analytical performance evaluation. *ACM TODS*, 2(4):267–288, October 1984.

[FC87] Christos Faloutsos and S. Christodoulakisi. Description and performance analysis of signature file methods for office filing. *ACM Transactions on Office Information Systems*, 5(3):237–257, July 1987.

[FLG91a] Farshad Fotouhi, T.G. Lee, and William I. Grosky. The generalized index model for object-oriented database systems. In *Proc. of the 10th Annual Int. Phoenix Conf. on Computers and Communication (IPCCC)*, pages 302–308, 1991.

[FLG91b] Farshad Fotouhi, T.G. Lee, and William I. Grosky. An indexing model for complex object hierarchies in object-oriented databases. In *Database and Expert Systems Applications (DEXA)*, pages 221–226, Berlin, Germany, 1991. Springer-Verlag.

[FNPS79] Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible hashing - a fast access method for dynamic files. *ACM TODS*, 4(3):315–344, September 1979.

[Fre87] Michael Freeston. The BANG file: A new kind of grid file. In *Proc. of ACM SIGMOD 1987 Annual Conf.*, pages 260–269, San Francisco, CA, May 1987. ACM Press.

[Fre89] Michael Freeston. Advances in the design of the BANG file. In *Proc. of the 3rd Int. Conf. on Foundations of Data Organization and Algorithms*, volume 367 of *LNCS*, pages 322–338, Paris, France, June 1989. Springer-Verlag.

[Fre95] Michael Freeston. A general solution of the n-dimensional B-tree problem. In SIGMOD'95 [SIG95], pages 80–91.

[Gud96] Ehud Gudes. A uniform indexing scheme for object-oriented databases. In ICDE'96 [ICD96], pages 238–246.

[Heu92] Andreas Heuer. *Objektorientierte Datenbanken: Konzepte, Modelle, Systeme.* Addison-Wesley, 1992.

[HT94] Kien A. Hua and Chinmoy Tripathy. Object skeletons: An efficient navigation structure for object-oriented database systems. In ICDE'94 [ICD94], pages 508–517.

[ICD94] *Proc. Tenth Int. Conf. on Data Engineering*, Houston, TX, February 1994. IEEE CS Press.

[ICD96] *Proc. Twelfth Int. Conf. on Data Engineering*, New Orleans, LA, March 1996. IEEE CS Press.

[ICD97] *Proc. Thirteenth Int. Conf. on Data Engineering*, Birmingham, UK, April 1997. IEEE CS Press.

[IKO93] Yoshiharu Ishikawa, Hiroyuki Kitagawa, and Nobuo Ohbo. Evaluation of signature files as set access facilities in OODBs. In *Proc. of the 1993 ACM SIGMOD Int. Conf. on Management of Data*, volume 22 of *SIGMOD Record*, pages 247–256, Washington, DC, June 1993. ACM Press.

[KD91] Ullrich Keßler and Peter Dadam. Auswertung komplexer Anfragen an hierarchisch strukturierte Objekte mittels Pfadindexen. In *Proc. der GI-Fachtagung Datenbanksysteme in Büro, Technik und Wissenschaft*, volume 270 of *Informatik-Fachberichte*, pages 218–237, Kaiserslautern, Germany, March 1991. Springer-Verlag. in german.

[KFIO93] Hiroyuki Kitagawa, Yoshiaki Fukushima, Yoshiharu Ishikawa, and Nobuo Ohbo. Estimation of false drops in set-valued object retrieval with signature files. In *Proc. of the 4th Int. Conf. on Foundations of Data Organization and Algorithms*, volume 730 of *LNCS*, pages 146–163, Chicago, IL, October 1993. Springer-Verlag.

[Kim90] Won Kim. *Introduction to Object-Oriented Databases.* Computer Systems Series. The MIT Press, Cambridge, Massachusetts, 1990.

[Kim95] Won Kim, editor. *Modern Database Systems: The Object Model, Interoperability, and Beyond.* ACM Press and Addison-Wesley, 1995.

[Kim94] Won Kim. Observations on the ODMG-93 proposal. *ACM SIGMOD Record*, 23(1):4–9, March 94.

[KKD89] Won Kim, Kyung-Chang Kim, and Alfred Dale. Indexing techniques for object-oriented databases. In Kim and Lochovsky [KL89], pages 371–394.

[KKS90] Michael Kifer, Won Kim, and Yehoshua Sagiv. Querying object-oriented databases. In SIGMOD'90 [SIG90], pages 393–402.

[KL89] Won Kim and Frederick H. Lochovsky, editors. *Object-Oriented Concepts, Databases, and Applications.* Addison-Wesley, 1989.

[KM90a] Alfons Kemper and Guido Moerkotte. Access support in object bases. In SIGMOD'90 [SIG90], pages 364–374.

[KM90b] Alfons Kemper and Guido Moerkotte. Advanced query processing in object bases using access support relations. In *Proc. 16th Int. Conf. on Very Large Databases*, pages 290–301, Brisbane, Australia, August 1990. Morgan Kaufmann Publishers.

[KM92] Alfons Kemper and Guido Moerkotte. Access support relations. *Information Systems*, 17(2):117–145, 1992.

[KM94a] Alfons Kemper and Guido Moerkotte. *Object-Oriented Database Management: Applications in Engineering and Computer Science.* Prentice-Hall, Englewood Cliffs, NJ, 1994.

[KM94b] Christoph Kilger and Guido Moerkotte. Indexing multiple sets. In VLDB'94 [VLD94], pages 180–191.

[KRVV93] Paris C. Kanellakis, Sridhar Ramaswamy, Darren E. Vengroff, and Jeffrey S. Vitter. Indexing for data models with constraints and classes. In *Proc. of the Twelfth ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 233–243, Washington, DC, May 1993. ACM Press.

[LL86] Witold Litwin and David B. Lomet. The bounded disorder access method. In *Proc. Second Int. Conf. on Data Engineering*, pages 38–48, Los Angeles, CA, February 1986. IEEE CS Press.

[LL87] Witold Litwin and David Lomet. A new method for fast data searches with keys. *IEEE Software*, 4(2):16–24, March 1987.

[LL92] Wang-Chien Lee and Dik Lun Lee. Signature file methods for indexing object-oriented database systems. In *Proc. of the 2nd Int. Computer Science Conf.*, pages 616–622, Hong Kong, December 1992.

[LL94] Dik Lun Lee and Wang-Chien Lee. Using path information for query processing in object-oriented database systems. In *Proc. of the Conf. on Information and Knowledge Management*, pages 64–71, Gaithersburg, MD, November 1994.

[LL95] Wang-Chien Lee and Dik Lun Lee. Combining indexing technique with path dictionary for nested object queries. In *Proc. of the Fourth Int. Conf. on Database Systems for Advanced Applications (DASFAA)*, pages 107–114, Singapore, April 1995. World Scientific Publishing.

[LL96] Dik Lun Lee and Wang-Chien Lee. Signature path dictionary for nested object query processing. In *Proc. of the IEEE Int. Phoenix Conf. on Computers and Communication (IPCCC)*, March 1996. to appear.

[LLOH91] Chee Chin Low, Hongjun Lu, Beng Chin Ooi, and Jiawei Han. Efficient access methods in deductive and object-oriented databases. In DOOD'91 [DOO91], pages 68–84.

[LOL92] Chee Chin Low, Beng Chin Ooi, and Hongjun Lu. H–trees: A dynamic associative search index for OODB. In *Proc. of the 1992 ACM SIGMOD Int. Conf. on Management of Data*, pages 134–143, San Diego, CA, June 1992. ACM Press.

[LS90] David B. Lomet and Betty Salzberg. The hB-tree: A multiattribute indexing method with good guaranteed performance. *ACM TODS*, 15(4):625–658, December 1990.

[Mel94a] Jim Melton, editor. *ISO-ANSI Working Draft, Database Language SQL/Foundation (SQL3).* March 1994.

[Mel94b] Jim Melton. Object technology and SQL: Adding objects to a relational language. *IEEE Bulletin of the Technical Committee on Data Engineering*, 17(4):15–26, December 1994.

[MM94] Frank Manola and Gail Mitchell. A comparison of object models in ODBMS-related standards. *IEEE Bulletin of the Technical Committee on Data Engineering*, 17(4):27–35, December 1994.

[MP96] Thomas A. Mueck and Martin L. Polaschek. Indexing type hierarchies with multikey structures. In *Proc. of the 7th Int. Workshop on Persistent Object Systems (POS 7)*. Morgan Kaufmann Publishers, May 1996. in print.

[MP97a] Thomas A. Mueck and Martin L. Polaschek. The multikey type index for persistent object sets. In ICDE'97 [ICD97], pages 22–31.

[MP97b] Thomas A. Mueck and Martin L. Polaschek. Optimal type hierarchy linearization for queries in OODB. In *Proc. of the Fifth Int. Conf. on Database Systems for Advanced Applications (DASFAA)*, pages 225–234, Melbourne, Australia, April 1997. World Scientific Publishing.

[MS86] David Maier and Jacob Stein. Indexing in an object-oriented DBMS. In *Proc. 1986 Int. Workshop on Object-Oriented Database Systems*, pages 171–182, Pacific Grove, CA, September 1986. IEEE CS Press.

[MS87] David Maier and Jacob Stein. Development and implementation of an object-oriented DBMS. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, Series in Computer Systems, pages 355–392. The MIT Press, Cambridge, MA, 1987.

[MS90] David Maier and Jacob Stein. Development and implementation of an object-oriented DBMS. In Stanley B. Zdonik and David Maier, editors, *Readings in Object-Oriented Database Systems*. Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[NHS84] Jürg Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM TODS*, 9(1):38–71, March 1984.

[Nov94] Boris Novikov. Indices for set-theoretical operations in object bases. In *Proc. of the Int. Workshop on Advances in Databases and Information Systems - ADBIS*, pages 208–216, Moscow, May 1994.

[OHLT96] Beng Chin Ooi, Jiawei Han, Hongjun Lu, and Kian Lee Tan. Index nesting – an efficient approach to indexing in object-oriented databases. *VLDB Journal*, 5(3):215–228, August 1996.

[OM84] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proc. of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 181–190. ACM Press, 1984.

[OO85] Esen A. Ozkarahan and Mohamed Ouksel. Dynamic and order preserving data partitioning for database machines. In *Proc. 11th Int. Conf. on Very Large Data Bases*, pages 358–368, Stockholm, Sweden, August 1985. Morgan Kaufmann Publishers.

[RK95] Sridhar Ramaswamy and Paris C. Kanellakis. OODB indexing by class-division. In SIGMOD'95 [SIG95], pages 139–150.

[RS94] Sridhar Ramaswamy and Sairam Subramanian. Path caching: A technique for optimal external searching. In *Proc. of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 25–35, San José, CA, May 1994. ACM Press.

[Sal94] Betty Salzberg. *File Structures: An Analytical Approach.* Prentice-Hall, Englewood Cliffs, NJ, 1994.

[SB96] Boris Shidlovsky and Elisa Bertino. A graph-theoretic approach to indexing in object-oriented databases. In ICDE'96 [ICD96], pages 230–237.

[Sch93] Manfred Schauer. *Adaptive Clusterbildung in Mehrattributsuchstrukturen.* Dissertation, Univ. Wien, February 1993. in german.

[SIG90] *Proc. of the 1990 ACM SIGMOD Int. Conf. on Management of Data*, volume 19 of *SIGMOD Record*, Atlantic City, NJ, May 1990. ACM Press.

[SIG95] *Proc. of the 1995 ACM SIGMOD Int. Conf. on Management of Data*, San José, CA, June 1995. ACM Press.

[SLT91] Marc H. Scholl, Christian Laasch, and Markus Tresch. Updateable views in object-oriented databases. In DOOD'91 [DOO91], pages 189–207.

[SM91] Jacob Stein and David Maier. Associative access support in GemStone. In Klaus R. Dittrich, Umeshvar Dayal, and Alejandro P. Buchmann, editors, *On Object-Oriented Database Systems*, Topics in Information Systems, chapter 20, pages 323–339. Springer-Verlag, Berlin Heidelberg, 1991.

[SÖ90] D.D. Straube and M.T. Özsu. Queries and query processing in object-oriented database systems. *ACM Transactions on Information Systems*, 8(4):387–430, October 1990.

[SS86] Hans-Jörg Schek and Marc H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.

[SS94] B. Sreenath and S. Seshadri. The hcC-tree: An efficient index structure for object oriented databases. In VLDB'94 [VLD94], pages 203–213.

[Sto96] Michael Stonebraker. *Object-Relational DBMSs.* Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[SZ89] Gail M. Shaw and Stanley B. Zdonik. Object-oriented queries: Equivalence and optimization. In DOOD'89 [DOO89], pages 281–295.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume I. Computer Science Press, Rockville, MD, 1988.

[Val87] Patrick Valduriez. Join indices. *ACM TODS*, 12(2):218–246, June 1987.

[VKC86] Patrick Valduriez, Setrag Khoshafian, and George Copeland. Implementation techniques of complex objects. In *Proc. Twelfth Int. Conf. on Very Large Databases*, pages 101–110, Kyoto, Japan, August 1986. Morgan Kaufmann Publishers.

[VLD94] *Proc. Twentieth Int. Conf. on Very Large Databases*, Santiago, Chile, September 1994. Morgan Kaufmann Publishers.

[XH94] Zhaohui Xie and Jiawei Han. Join index hierarchies for supporting efficient navigations in object-oriented databases. In VLDB'94 [VLD94], pages 522–533.

[Yao77] S.B. Yao. Approximating block accesses in database organizations. *CACM*, 20(4):260–261, 1977.

[YLK94] Hwan-Seung Yong, Sukho Lee, and Hyoung-Joo Kim. Applying signatures for forward traversal query processing in object-oriented databases. In ICDE'94 [ICD94], pages 518–525.

INDEX