An Object-Oriented Database System for Efficient Information Retrieval Applications

by

QiFan Chen

Dissertation submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

APPROVED:

Dr. Edward A. Fox, Chairman

Dr. Lenwood S. Heath

Dr. J.A.N. Lee

Dr. J. Terry Nutter

Dr. Monte B. Boisen

March, 1992

Blacksburg, Virginia

An Object-Oriented Database System for Efficient Information Retrieval Applications

by

QiFan Chen

Dissertation submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

APPROVED:

Dr. Edward A. Fox, Chairman

Dr. Lenwood S. Heath

Dr. J.A.N. Lee

Dr. J. Terry Nutter

Dr. Monte B. Boisen

March, 1992

Blacksburg, Virginia

An Object-Oriented Database System for Efficient Information Retrieval Applications

by

QiFan Chen

Committee Chairman: Dr. Edward A. Fox

Department of Computer Science

(ABSTRACT)

This dissertation deals with the application of object-oriented database techniques to the problem of storage and access of information retrieval (IR) data, especially data that can be organized as a graph, such as a thesaurus encoded in semantic networks, or hypertext collections. Even traditional IR models can use graph representations of documents and concepts.

This dissertation reports the development of an object-oriented model called the LEND (Large External object-oriented Network Database) model. This model contains not only features found in a typical object-oriented model but also those that specifically are designed for graph-structured data. A query language is provided facilitating the specification of graph-oriented queries.

A prototype LEND system has been implemented to test the model on realistic graph-structured data. It adopts an open system architecture and design, and is easily extensible, like the LEND model itself. The research result of suitable data structures and algorithms (a class of minimal perfect hashing functions) for the efficient implementation of the LEND model is also reported. These data structures and algorithms enable retrieval of a node or a set of nodes in an optimal fashion. Placement of a large graph on a disk is studied as well. The method developed permits efficient traversal of graphs.

ACKNOWLEDGEMENTS

I would like to thank Dr. Edward A. Fox for his insight on the data model issues and tremendous support through this research. I would also like to thank Dr. Lenwood Heath for guidance regarding data structure and algorithm research. I owe thanks to Dr. J. Terry Nutter for valuable discussions about the SNePS system. I am grateful to all my committee members, especially Dr. Edward A. Fox, for careful review of the dissertation.

I would also like to thank my wife Mrs. Heng Zhang, my parents and brother for their enormous encouragement and understanding.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

This dissertation studies object-oriented database techniques for the storage and access of information retrieval (IR) data, especially data whose structure is a graph, such as a thesaurus encoded in semantic networks and hypertext built on top of documents. Even traditional IR models can use a graph representation of documents and concepts. The promise of adapting object-oriented database techniques includes:

- fast prototyping;
- integrated search environment;
- efficient data access;
- compact and shared data storage; and
- easy extension with new data types.

To build such a system, it is essential to develop an object-oriented model for the graph structured data and to prototype a system that supports the model. To show that the model can deliver performance close to that of specialized IR systems or their components that support the operations studied in the dissertation, it is necessary to identify data structures

and algorithms suitable for the data involved. This dissertation has achieved all the above goals.

This dissertation reports the development of an object-oriented model called the LEND model. Here LEND stands for Large External object-oriented Network Database. The model contains not only features found in a typical object-oriented model but also those that are specifically designed for graph data. A query language is available facilitating the specification of graph-oriented queries.

This dissertation has put heavy emphasis on research regarding suitable data structures and algorithms for the efficient implementation of the LEND model. This is because the data (graphs) concerned are large. They must be placed on secondary storage units. These data structures and algorithms enable retrieval of a node or a set of nodes in an optimal fashion. Placement of a large graph on a disk is also studied. The method permits efficient traversal of graphs.

1.1 Overview

This chapter begins with a survey of typical IR search models and then points out many of the storage and access problems existing in typical research or commercial information retrieval systems. The main drawback lies in the fact that documents and related data are stored in a manner dependent on the search routines for different IR models. As a result, duplication of data is frequently seen and it is difficult to combine results of different IR models, in part because of complex interconnections of documents and the related data. The chapter also discusses the problem with storing externally thesauri encoded as semantic networks. There is great promise that thesauri can improve IR system effectiveness. However, a thesaurus directly encoded in a semantic network would take more space than that available in the main memory of a modern computer. It is necessary that an efficient external storage system helps solve the problem. A similar problem exists for hypertext data.

1.2 IR Models

A variety of models and systems have been developed to deal with the problem of finding documents that satisfy a given information need. As the following brief survey reveals, these models require different representations of documents, use different functions to compute the degree of match, and fit various types of information need.

1.2.1 (Extended) Boolean Model

The Boolean model is the simplest retrieval model and has been widely used in bibliographic retrieval systems. The model, using a document term list representation, permits the specification of terms linked by binary operators **and**, **or** and **not** in queries.

The advantage of the model results from its ability to describe precisely the information need. This rigid specificity is useful in applications where exact matching in both content and structure of (part of) documents are desired. However, in applications where a partial matching that retrieves neither too many nor too few is asked for, the Boolean model behaves poorly because it is unable to rank the selected documents. The Boolean model also requires the document to be represented in a term set form which implies that semantic relations among the terms are not used in the model.

Various strategies have been considered to extend the Boolean model. Among them are ones capable of answering proximity queries and ones capable of ranking output documents. For the later category, the most studied and effective one is the p-norm method [SFW83]. Given a set of terms $A_1, A_2, \ldots, A_n$, suppose a document $\mathbf{D}$ is represented as

$$\mathbf{D} = (d_{A_1}, d_{A_2}, \ldots, d_{A_n})$$

where d_{A_i} stands for the weight of term A_i in $\mathbf{D}$. A query $\mathbf{Q}$ is partially represented as

$$\mathbf{q} = (a_1, a_2, \ldots, a_n)$$

where a_i stands for the weight of term A_i in $\mathbf{Q}$. Then the model generalizes the Boolean

or-query $Q_{\mathbf{or}}$ to

$$Q_{\mathbf{or}(p)} = [(A_1, a_1) \quad \mathbf{or}^p \quad (A_2, a_2) \quad \mathbf{or}^p \quad \ldots \quad \mathbf{or}^p \quad (A_n, a_n)]$$

and the Boolean **and**-query $Q_{\mathbf{and}}$ to

$$Q_{\mathbf{and}(p)} = [(A_1, a_1) \quad \mathbf{and}^p \quad (A_2, a_2) \quad \mathbf{and}^p \quad \ldots \quad \mathbf{and}^p \quad (A_n, a_n)].$$

The model computes the similarity for the document **D** and the query **Q** in a **or**-query as

$$sim(\mathbf{D}, \mathbf{Q}_{or(p)}) = \left(\frac{a_1^p d_{A_1}^p + a_2^p d_{A_2}^p + \ldots + a_n^p d_{A_n}^p}{a_1^p + a_2^p + \ldots + a_n^p} \right)^{\frac{1}{p}}$$

and that for the document **D** and the **and**-query as

$$sim(\mathbf{D}, \mathbf{Q}_{and(p)}) = 1 - \left(\frac{a_1^p (1 - d_{A_1})^p + a_2^p (1 - d_{A_2})^p + \ldots + a_n^p (1 - d_{A_n})^p}{a_1^p + a_2^p + \ldots + a_n^p} \right)^{\frac{1}{p}}.$$

Three nice properties are associated with the p-norm model. First, the standardized Boolean operators **and**, **or** and **not** are available in the model and can be used as usual. This is particularly important for normal Boolean systems users because they do not have to learn a totally different query language. Second, term weights can be used to reflect the importance of terms in both the document and the query. This leads to a quantitative measurement for the degree of matching and to the production of ranked output. Last, by changing the value of p one may obtain the effects of the vector model, the standard Boolean model and more generally the extended Boolean retrieval model. Specifically, when p is set to 1 the model reduces to the vector model (explained in subsection 1.2.2) where the Boolean operators are not in effect, and as p approaches infinity, the model approaches a Boolean model where the Boolean operators have a strict interpretation.

1.2.2 Vector Model

The vector model [SWC75] was invented to make possible the measurement of degree of relevance of a document to a query represented in the same form. This is an advantage over

the pure Boolean model because the document-query matching can be computed mathematically to yield a real-valued number associated with each match. To users, this means that retrieved documents can be ranked to reveal the relative importance of each document as judged by the system. Since representations and manipulation are entirely within the vector space, this model can be implemented easily, with freedom allowing the design of special data structures and use of concurrent computation.

The vector model describes terms, documents and queries as vectors and puts these objects into a linear vector system called Euclidian space. In the space there exists a set of linearly independent vectors (called the basis) upon which every object can be uniquely represented as a linear combination of vectors in the basis. Scalar product of two vectors, normalization of nonzero vectors and projection of one vector onto another are important operations in such a space.

In normal application of the vector space concept to IR, terms drawn from an indexing vocabulary are usually assumed pairwise orthogonal and serve as the basis, although this assumption has ignored term-term correlation and is restrictive [RW86]. Using the basis, a document $\mathbf{D}$ and a query $\mathbf{Q}$ can be formally written as:

$$\mathbf{D} = \sum_{i=1}^{n} a_i t_i$$

and

$$\mathbf{Q} = \sum_{i=1}^{n} q_i t_i.$$

The a_i's and q_i's are the components of $\mathbf{D}$ and $\mathbf{Q}$ along the basis t_i's respectively, that can be estimated from frequency data. The similarity of $\mathbf{D}$ and $\mathbf{Q}$ can be computed as the scalar product of $\mathbf{D}$ and $\mathbf{Q}$, written as

$$\mathbf{D} \cdot \mathbf{Q} = \sum_{i=1}^{n} \sum_{j=1}^{n} a_i q_i t_i t_j = \sum_{i=1}^{n} a_i q_i.$$

The above simple vector model has been further generalized in [WZRR87]. The generalized model proposed strictly defines what a basis is for an IR application vector space

and provides a theoretically sound and elegant solution to compute term-term correlation within that space. The experimental evaluation of the model shows that it performs better than the simple vector model in terms of recall and precision.

1.2.3 Probabilistic Model

The probability model ([RJ76] and [SM83]) uses an attribute list representation together with Bayesian inference to measure the degree of matching of a query **Q** to a document **D**. The basic idea of the model lies in the observation that when **D** has a partial matching of terms against **Q**, there is a probability of **D** being relevant, denoted by $P(rel|\mathbf{D})$.

Using Bayes' theorem, $P(rel|\mathbf{D})$ can be written as

$$P(rel|\mathbf{D}) = \frac{P(\mathbf{D}|rel)P(rel)}{P(\mathbf{D})} = \frac{P(\mathbf{D}|rel)P(rel)}{(P(\mathbf{D}|rel)P(rel) + P(\mathbf{D}|nrel)P(nrel))}.$$

$P(rel)$ and $P(nrel)$ are a priori probabilities and usually are estimated in terms of the portion of relevant or nonrelevant documents against the total number of documents. $P(\mathbf{D}|rel)$ and $P(\mathbf{D}|nrel)$ are respectively the probabilities that a relevant or a nonrelevant document **D** is retrieved. If we assume that the relevance of **D** to **Q** is associated with the relevance of each term in **D** to **Q** and that all terms are independent, then $P(\mathbf{D}|rel)$ and $P(\mathbf{D}|nrel)$ can be calculated using the following formulas:

$$\begin{aligned} P(\mathbf{D}|rel) &= \prod_{t_i \in \mathbf{D}} P(t_i|rel); \\ P(\mathbf{D}|nrel) &= \prod_{t_i \in \mathbf{D}} P(t_i|nrel). \end{aligned}$$

$P(t_i|rel)$ can be estimated by the ratio of the number of relevant documents containing t_i to the total number of relevant documents. $P(t_i|nrel)$ can be estimated similarly. The basis for these estimates varies depending on the knowledge available from previous searches, feedback on the current search, or statistics about the collection in general.

1.2.4 Inference Network Model

The inference network model, proposed and verified by Turtle and Croft [Tur91, TC91], bases its work on Bayesian inference networks. An inference network in the model is composed of two components: a document network and a query network. In the first component, documents and their document content representations are encoded while in the second the information need existing in a user's mind, the queries explicitly expressing the need and the query representation details are encoded. Figure 1.1 depicts such a network. All nodes in the network represent binary-valued events. An evaluation of the network computes the probability that the information need is met (event is true) given the event that the document d_i is observed. Iterating over all documents, it is possible to rank the documents in the collection.

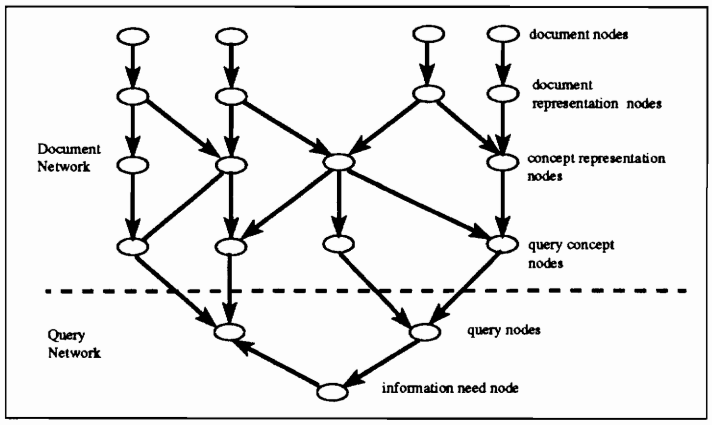


Figure 1.1: Basic Document Inference Network

The key part of the model is to determine the function computing $P(q|p_1, \ldots, p_n)$ for node q given probabilities at all parent nodes $\{p_1, \ldots, p_n\}$. The function should fit the needs of information retrieval and be cost-effective to compute and store. The model has proposed five such functions: *and*, *or*, *not*, *sum* and *weighted sum*, shown below.

$$P_{and}(q = true) = \prod_{i=1}^{n} p_i$$

$$P_{and}(q = false) = 1 - \prod_{i=1}^{n} p_i$$

$$P_{or}(q = true) = 1 - \prod_{i=1}^{n} (1 - p_i)$$

$$P_{or}(q = false) = \prod_{i=1}^{n} (1 - p_i)$$

$$P_{not}(q = true) = 1 - p$$

$$P_{not}(q = false) = p$$

$$P_{sum}(q = true) = \frac{\text{number of parent nodes denoting true event}}{\text{number of parent nodes}}$$

$$P_{sum}(q = false) = 1 - P_{sum}(q = true)$$

$$P_{weight_{sum}}(q = true) = \frac{(\sum_{i=1}^{n} w_i p_i) w_q}{\sum_{i=1}^{n} w_i}$$

$$P_{weight_{sum}}(q = false) = 1 - \frac{(\sum_{i=1}^{n} w_i p_i) w_q}{\sum_{i=1}^{n} w_i}$$

P_{and}, P_{or} and P_{not} can be used to simulate Boolean and probabilistic operators, while $P_{weight_{sum}}$ can be used to simulate within-document term frequency tf, inverse document frequency idf and the well-known weighting scheme $tf \times idf$.

When stored in the canonical link matrix form, the space complexity of the first four functions listed above is $O(1)$ and that of the last is $O(n)$. The time complexity of the first function is $O(1)$ and that of the rest is $O(n)$.

1.2.5 SNePS Supported Thesauri Storage/Access

SNePS

SNePS [Sha79, Sha89] is a knowledge representation and manipulation system. A SNePS semantic network is a directed graph in which nodes represent concepts and the arcs indicate

nonconceptual binary relations between concepts. Nodes in SNePS are of two primary types: molecular nodes and base nodes. Molecular nodes can be further divided into asserted ones and dominated ones, where the former assert propositions believed by the system and the latter stand for propositions or structured objects as part of a proposition. Base nodes are either constants or variables. Predicate calculus can also be expressed in SNePS.

Although SNePS is a loosely typed system, there are special requirements for a directed graph to be considered an acceptable SNePS network [Sha79, Mor85]. Thus, no more than one arc (and its inverse) can join a pair of nodes, and no node should be isolated. Constraints regarding structure updates are also important to prevent removing a relation to a concept (a node) on which another concept (another node) is defined.

Using SNePS to Represent Lexicons

It has been observed that a mismatch between the language used by authors and inquirers is a common problem in the information retrieval process. Automatically bridging different language usage seems necessary for effective on-line searches. Results of experiments on expanding queries by adding related words from different sources supported this point, showing improvement over the original queries. In particular, Fox [Fox80] shows related words derived manually based on lexical-semantic relations can achieve greater effectiveness measured by recall and precision. Since manual derivation of lexical relations is both expensive and problematical, Fox and Nutter [FN87] proposed building a large English lexicon in the form of a semantic net to eliminate this problem.

Current implementations of SNePS are able to handle networks which fit in main memory. But if the data collection is too large to be held internally, database support for effective access and efficient indexing are necessary. More importantly, since path-based inference is a useful mechanism for lexical relation studies, its database support issue should be addressed and given priority. Some ideas on this will be given in Chapter 3.

1.2.6 Hypertext Models

Flexible and random access to documents has recently become possible through powerful workstations, high-resolution graphics displays, increased network communications, and decreased cost for larger storage. This new access freedom is delivered especially well by hypertext systems that allow users to access documents either linearly or nonlinearly. This capability has long been demanded for documents with clear physical and logical separations and with complex structures such as dictionaries. Concurrent and coordinated access to documents at the user level is another necessity when the consumption of source documents requires well-defined steps and procedures.

Hypergraph Model

The hypergraph model, proposed in [Tom89] for page-oriented hypertext databases, applies standard database techniques to handle commonality in the structure, set-based access, and user-specific views for hypertext systems. The model, still based on directed graphs, is formalized using a 6-tuple $D = (N, P, R, V, L, E)$, where N is the node set, P the page set, R the reader set, V the mapping from N to P, L the label set and E the hyperedge set.

The model separates the text content and connection of text spans into two sets (P and N respectively) and uses a function (V) to carry out the mapping (from N to P). The interpretation of the contents is done by readers in R.

The hyperedge set E connects nodes in N. A hyperedge is defined as a set of arcs in a conventional graph that connect a common source node to a set of target nodes. The labels of the edges and target nodes are specified by a label subset in L and a node subset in N. The model relies on the notion of hyperedges to describe commonality among edges. In fact, the model proposes operators that use arcs with the same label (this set of arcs is a subset of a hyperedge) to achieve various goals.

Another benefit that could be obtained from hyperedges is the designation of current nodes. A set of current nodes, derived from a hyperedge, sets a basis for traversal operations

in a hypertext database. The use of current nodes also makes possible concurrent access such as multiple window browsing in a hypertext database.

To personalize a hypertext database, user-specific views are defined following the same formalization rule as for hypertext databases. A view is associated with a database by some additional view edges making connections over user-specific view nodes and database nodes. The operation set proposed by the model includes querying a state, creating a view and modifying a state.

"Dexter Workshop" Model

The "Dexter Workshop" model [Ore89, HS90] is an attempt to coordinate or standardize hypertext models. Its goal is to provide a descriptive language neutral to all systems, to construct a layered architecture for incremental function specification, and to encourage the standardization and interoperability of hypertext systems.

Since relativity is the foundation, the model uses abstractions in its descriptions. Its constructs include "component" for either nodes or links, "specifier" for general functions that eventually return a component typed value, and "UID" (unique identifier) for the identification of components. Run time support is also included in the specification of components, but it is not part of the model.

These constructs are further linked by functional mappings to form the data manipulation part of the model. The `datatype` function produces a composite node (a component) from a set of (more basic) component nodes. The `resolver` generates a UID from a specifier. The `accessor` returns a component from a UID. And finally a hypertext is defined as `components` + `resolvers` + `accessors` + `datatypes`.

The data manipulation part of the model includes operations to create, delete and modify components, to get the source and target ends of a link, and to obtain all incoming and all outgoing links for a node.

1.3 Research Summary

1.3.1 Hypotheses

This research concerns the following hypotheses.

- The object-oriented database technique helps the sharing of IR data among different IR search models;
- The object-oriented database technique helps the integration of IR search models;
- Minimal perfect hash functions (MPHFs) can be found efficiently and are useful to IR data search;
- IR data whose structure is a graph can be searched efficiently using MPHFs.

1.3.2 Research Contributions

This research leads to a formal object-oriented database model. The model is composed of three components: the Base Component, the Graph Component and the IR Application Component. The base component contains set, tuple, list and other constructs that can be found in typical object-oriented database models. The graph component adds node, arc, path and graph constructs. Sets, tuples or lists of these constructs are also possible. The IR component demonstrates how IR applications can be modeled by features combined from the first and the second component. A formal specification of the model is also given, along with the semantics for methods. Lastly, the query language and query examples are described.

The model has been prototyped on an object-oriented database system. The system has three layers corresponding to the storage subsystem and three components of the LEND model. The system follows the normal reference model and is organized into three layers: storage, object and application. Within each layer, class hierarchies are used to model various system components. The result is an open system to which new components such as data structures or algorithms can be easily added.

The design meets the needs of efficient storage and access of the graph structured data. This is illustrated by the page storage class, the LRU replacement policy, the node and arc classes and the graph class. A query language for graph structured data is described. With the help of the language, it is easy to form a query that specifies a subgraph. The processing of graph queries is also discussed.

This research also makes contributions to the development of algorithms for finding good minimal perfect hash (MPHF) functions. These constructed functions require small space to specify them, and can produce a hash address with effort proportional to the length of the key presented. One reference to the hash table by using the hash address can definitely answer the question: Does a record with that key exist? The proposed algorithms are able to process very large key sets (e.g., over 3 million) with a practical amount of computation time. The application of MPHF to trie compaction is also demonstrated. It is shown that the method is capable of compacting tries with different pointer distributions. The degree of compaction is only related to the number of nodes in the trie. The quality of the compaction is an e-approximation where $e = 2$. Variations of the MPHF algorithms are studied, including order-preserving and bin hashing. These hash functions have been tested and proved effective in indexing data in the LEND system.

This research also studies the physical organization of graph storage, i.e., how graphs, especially the large ones, are placed onto disk-type physical storage devices and how indices are established for such placed graphs. The assumption is that all graphs are subject to random path access, such as browsing a hypertext data set or performing a path-based inference in a semantic network. The organization allows efficient random retrieval access.

1.4 Overview of the Dissertation

Chapter 2 presents the the LEND model (LOM). Chapter 3 covers the prototype system implementing the LOM. Chapter 4 describes a set of algorithms developed for finding good minimal perfect hash (MPHF) functions for information retrieval applications. Chapter 5

describes the physical organization of graph storage. Chapter 6 discusses conclusions and future work.

Chapter 2

An Object Model for IR Applications

This chapter describes an object-oriented database model, the LEND model, suitable for IR data sharing, IR search model integration and graph structured data access. In section 2.1, conventional data models are surveyed. Section 2.2 describes one integration of IR models with the network database model, and one database model proposed for enhancing the modeling flexibility of document retrieval systems. The LEND model is fully discussed in section 2.3.

2.1 Data Models Survey

Relational

The relational model [Cod70] is one of the most famous models for conventional database applications. The model is based on a simple notation involving relations or tables to record both entities and relationships. Each relation is a subset of the Cartesian product of the list of attribute domains. If an entity is represented, the domain's values are used to characterize it. If a relationship among entities is represented, the domain of relation(s)

about the relationship are the key attribute values of those entities. Since any number of domains could be used, n-ary relationships are expressible.

A rich set of operations is available in the model through the relational algebra or calculus. They are implementation-independent, non-procedural, expressive and easy to use. On the other hand, since there are potentially many ways to execute a non-procedural query, some of which require unnecessary work, optimizing the processing of a query by selecting a proper execution plan is necessary for good performance.

Since the design of relations usually makes use of the theory of normalization, the model preserves functional dependency relationships in the tables but will ignore other relationships. These relationships, including their semantics and applicable operations, must be remembered by the users. As a direct consequence, the model is poor in its capability to directly represent as well as query complex entities. This large drawback of the model has led to the invention of various semantic models. Also, the recursive query can't be expressed in the model.

Network

The network data model [TL82], proposed by CODASYL's DBTG group, supports typed entities decomposed into a set of fields and binary relationships connecting these entities. Unlike in the relational model, each binary relation is restricted to be one-one or many-one to allow an economical multi-list implementation of the model. Since one-one and many-one are the only relationships representable, a many-many relation has to be decomposed into two many-one relations linked by an intermediate record type.

Operations on a network defined in the DBTG report consist of those on records and those on arcs. In particular, the latter reflects the underlying multi-list implementation and is fairly navigational. Since operations in the network model are record-based as opposed to set-based in the relational model, records can be searched through keys or reached by arc traversal. Arcs do not have the same role as records have in the multi-list implementation because they cannot be searched. This makes the network representation more complex

than the relational counterpart. Also, the operations are only on single records or arcs, not on composite entities consisting of nodes and arcs.

2.1.1 Semantic Data Models

Semantic data models [HK87, PM88b] have evolved with the aim of capturing semantics in connection with the data modeling process. Compared with conventional models offering record level modeling capability, semantic models directly address the needs of object level modeling capability. At this level, the static properties of data such as characteristics and relationships are represented by useful abstractions such as aggregation, generalization, and association. Repeated information is handled by derivation, and inheritance is naturally accompanied by abstractions. Dynamic properties such as transition definition, error recovery and prerequisites appear also in the semantic models. Through object level modeling, independence at the object level is achieved.

As summarized in [HK87] and [PM88b], four major advantages are associated with semantic models: economy of expression, integrity maintenance, modeling flexibility, and modeling efficiency. Economy of expression is obvious because object and relationship notions are directly used. This contrasts with the relational model, which requires remembering meanings of tables and establishing relationships through join operators. Integrity maintenance is also feasible in semantic models because the data is viewed from well above the physical record level, for example at the object level. Modeling flexibility stands for variation in viewing the data. This capability is obtained through use of abstractions in the semantic models. Modeling efficiency is the ability to model data quickly and effectively. Since operations can be defined for different abstractions and can be implemented efficiently, users are free to use these built-in operations.

In the following subsections, several semantic models are briefly surveyed.

Entity-Relationship (E-R) Model

The E-R model [Che76] provides a tool for conceptual schema design. It supports features for easily incorporating semantic information into the schema design process. Entities and relationships are the two important types of constructs in the model. Here an entity is formed in an aggregated way from attributes and a relationship may involve explicit one-to-one, one-to-many or many-to-many constraints. The existence of dependent relationships can also be represented.

Object-Oriented Model

Object-oriented methodology [KL89] has been heavily investigated for the study of programming languages and software systems where dynamic and static properties are both important. Object-orientation exploits encapsulation to package these two types of properties into one framework called objects. Abstraction through instantiation and inheritance is used to achieve reusability for static representations as well as dynamic handling. Since the technique is the major subject of study of this thesis, we discuss object-oriented models and systems in the later sections and chapters.

2.2 IR Database Models

It has been recognized that database technology can be beneficial to IR systems by promoting the modeling capability, increasing the sharability of representations and retrieval methods, and obtaining standardized features of concurrency, recovery and protection. This awareness stems from concern with the fact that normal IR systems primarily have been implemented for the purposes of efficient storage and retrieval of documents [CP85]. Integrating the IR systems with the database systems would certainly achieve a good degree of interoperability among IR systems. In the following sections, we review two such integrations.

2.2.1 Network Structured Integration

This integration [CP85] was carried out on a CODASYL system with the aim of comparing a database implementation against a stand-alone implementation. The study focused on the incurred cost, sacrifice of efficiency and the new search strategies when a database implementation is considered.

Two types of nodes and three types of links were considered in the experiment. They are the document and term nodes and the document-link, term-term and document-document links. During the construction of the network, specially designed nearest neighbor generation and physical grouping algorithms were used to store nodes potentially relevant to queries together. Two cut-off methods were also introduced to reduce the construction time.

Implementing such representations on a CODASYL system presents quite an efficiency problem because the doc-term links are many to many. The conventional CODASYL decomposition schema for many-many relations will cause too many disk accesses. A revised schema is used to solve the problem, at the expense of the redundant storage of document and term identifiers.

Based on such a network, two major experiments on cluster search and estimated probability search strategies were conducted. The experimental results showed that the network implementation is comparable to stand-alone implementations with respect to disk access and the number of similarity calculations. However, the CODASYL network implementation does suffer from long processing time due to repeated execution of the uncompiled record access statement, and huge extra storage overhead (150% of the actual data).

2.2.2 Array Model

This model [MA87] is designed with the aim of enhancing the modeling flexibility of document retrieval systems. The promise lies in the use of the array representation that facilitates conceptual level description of objects. Basically, an array in the model contains a set of nonhomogeneous elements, even arrays. This simple definition of arrays makes it possible to

describe a real world object such as a document through an array, with sufficient capability to capture the hierarchical structure of the object.

Other modeling power of the array model includes the description of various abstraction mechanisms such as classification, generalization, and aggregation.

Since the paper only deals with the model itself, consideration of efficient implementation has not been covered.

2.3 LEND Object Model

This section informally describes an object data model for the storage and access of IR model, SNePS networks and hypertext applications. A formal desciption will be given in section 2.4. The model is divided into three components: a base component, a graph component and an IR component. See Figure 2.1 for an illustration of the class hierarchy for the three components. The base part provides classes for primitive and composite objects. The graph part uses the base model to develop a graph type of universe. The graph notation helps modeling of hypertext and semantic network applications. The LEND IR part constructs IR retrieval models by using the first two components.

2.3.1 Base Component

Objects in this part include root objects, primitive objects and composite objects. To help discussion, we concentrate on the classes by which these objects are defined.

Root Class

The root class serves as the base for all LEND classes. It allows a LEND object to contain a LEND object identifier and to own a set of member functions, which other LEND classes can override or redefine. The most important member functions include *to_string*, *from_string* and *id_EQ*, *value_LE* and *value_EQ*. The first two convert an object to or from its string format. The string could be in either ASCII form or binary form, specified by an additional argument to these two functions. The binary form is used by the storage system of the LEND database system for disk representation purpose. The ASCII form helps import and export LEND objects to the outside world. The *id_EQ* function compares two objects' identifiers, while *value_LE* and *value_EQ* compare their values. They are used by the LEND database system for object storage, indexing and access. Since the six functions have distinct meanings for different classes, the details (their semantics) will be given in section 2.4. New classes can be accepted by the LEND system as long as they properly define these functions. Two identification functions *parent_classes* and *this_class* are also defined. The first returns a set of the names of the parent and all ancestor classes as ASCII strings, while the second returns the name of this class.

Primitive Classes

The primitive classes include integer, real, and character string. The integer class encodes an integer, the real class encodes a real number and the character string class encodes a character string. Since operations to them are well-known, they are not discussed here.

Composite Classes

The composite classes include set, list and tuple classes. The elements of the sets, lists or tuples can be object identifiers or objects themselves. Three functions for iterating over components of a composite object are defined. They are *first*, *get_component* and *next*. The function *count* returns the number of components.

2.3.2 Graph Component

Graph Root Class

This class serves as the root for classes defined in the Graph Component. Only objects of subclasses to the graph root class can participate in graph related representation or manipulation. In particular, IR applications are represented using graphs.

Node Class

The node class encodes member function signatures for objects to be used as nodes. Only objects belonging to subclasses of the node class can be connected using directed arc objects.

Directed Arc Class

The directed arc class encodes binary relationships between node objects. A directed arc is represented as a triple composed of the source node identifier, the label object identifier and the sink node identifier. A directed arc object may contain additional information quantifying or specifying the connections. An undirected arc can be modeled as two directed arcs.

Path Class

The path class encodes paths in a graph. A typical representation of a path in the LEND object model (LOM) is a sequence of node identifiers and label identifiers. Another alternative can be a sequence of directed arc identifiers.

2.3.3 IR Component

This section describes IR objects. The abstract representation of IR entities as objects allows compact representation and sharing of IR entities among different IR models. Figure 2.2 illustrates a sample conceptual IR graph. In the figure, there are four types of node classes and ten types of arc classes. "Structural" or "citational" relationships can exist among documents. The "Concept" class has content representatives (concepts) representing the content of documents. Hence, we have the arc class "Representing". Semantic relationships among these concepts exist. The "Anchor" class represents some arbitrary components within documents. "Hypertext" arcs connect anchors together to form hypertext. The "query" class is about queries issued against node and arc objects, such as "Concept", "Document" and "Anchor" shown in the figure.

Document Class

Document objects generalize the notion of documents. Document objects contain information that a user is interested in. An IR model helps the user retrieve documents that are likely to be relevant. As documents may vary in terms of representation media (text, audio, video), function (journal, memo, book), internal structure (bibliographical records, memo) or written style, classification may be employed. With respect to the IR models considered, a document object is viewed as an entity and bears an identifier. Its content is represented by concept nodes, and the exact media representation can be stored appropriately within the document object boundary.

Concept Class

Concept objects are used to embody the contents of documents. Automatic key extraction, automatic phrase extraction, manual assignment and natural language processing are the common means to obtain concept objects. These techniques can generate different concept representations in the form of strings.

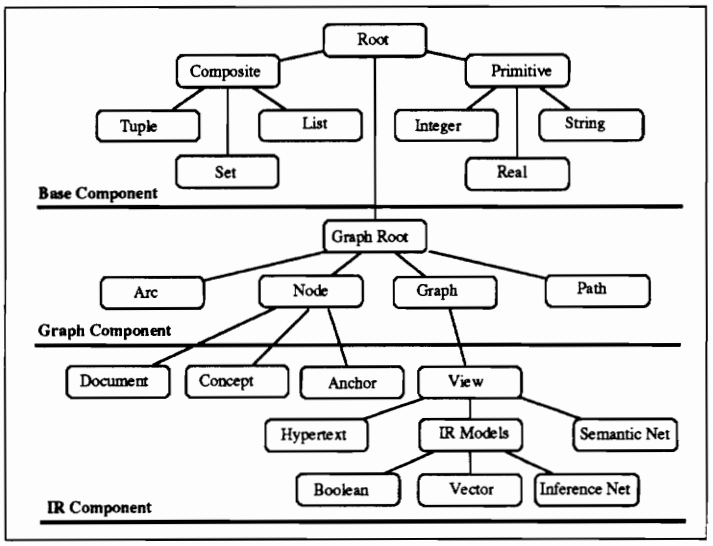


Figure 2.1: Three Components of the LEND Model

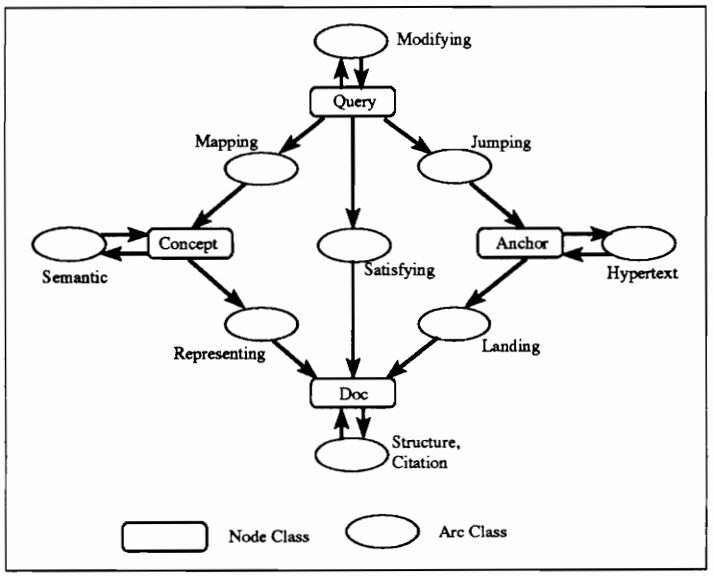


Figure 2.2: A Sample Conceptual IR Graph

Similar to the document objects, each concept object has an identifier. The string part of concepts however can be succinctly stored. This is done by introducing a level of indirection where identical text strings are replaced by their identifiers.

Anchor Class

Anchor objects serves as "anchors" which explicitly identify a portion of a document object. "Anchors" connect different portions of documents together through hypertext links in a meaningful way. "Anchors" are used for browsing.

Query Class

Query objects are introduced to encapsulate the details of heterogenous queries used by distinct IR models. IR model queries typically identify a subset of relevant document objects. Hypertext queries locate a set of "anchors" and finally a document subset.

IR Arc Class

IR arc objects encode the relationships between documents and concepts, between documents, between anchors, and between anchors and documents. Semantic relationships exist among these node objects, as shown in Figure 2.2.

The reason that relationships are encoded using links instead of being defined as constituents of node objects such as document or representation objects is that the former approach facilitates sharing a set of documents and concepts among multiple IR models. Note that subclassing document and concept classes does not work either as it needlessly duplicates their representations.

Boolean Retrieval Model Class

The Boolean model is easily represented using a bipartite graph $G_{Boolean}$ where document object nodes are on one side and the representation nodes are on the other, as shown in Figure 2.3. The relation that a term (a concept) represented by node rep_i occurs in a

document represented by doc_j is expressed by joining node rep_i and node doc_j through an arc. Suppose the query in the model is in disjunctive normal form. We associate with each conjunct in the query a set of term nodes rep_i occurring in the conjunct. Finding the answer to the conjunct is equivalent to identifying a maximal bipartite subgraph G_{sub} in $G_{Boolean}$ in which each document node in G_{sub} connects to all term nodes in the conjunct.

Vector Retrieval Model Class

Similar to the Boolean model, a vector model can be represented in bipartite graph form, as shown in Figure 2.4. The graph G_{vector} has additional relation arcs connecting document node doc_i and representation node rep_i. The weights are encoded in the arcs, which describe the degree of importance of the term in the document. A query in the model is described as a node adjacent to nodes of terms appearing in the query. Each edge from a query node to a term node bears a query weight denoting the importance of the term in the query. Computation of the rank of the matched documents is carried out through multiplication of each query term weight with the document-term weight.

2.4 Specification of the LEND Model

This section gives a formal specification of the LEND model. The style is similar to that of Straube and Özsu in [SÖ90] and to that of Lécluse, Richard and Velez in [LRV90]. The specification first introduces the primitive sets. Then it shows the representation of objects in the LEND model. Finally it gives the semantics of important methods based on the object representations.

2.4.1 Primitive Sets

The following sets form the basis of the specification.

Domain Set: D is a finite set containing n domains D_i, $D = \bigcup_{i=1}^{n}\{D_i\}$. Sample domains include integer, real number, and character string.

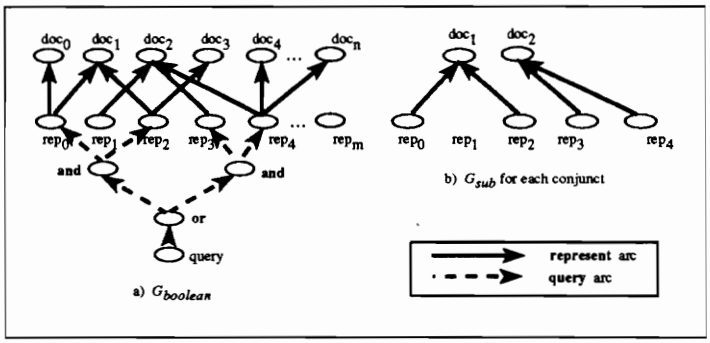


Figure 2.3: A Boolean Retrieval Graph Representation

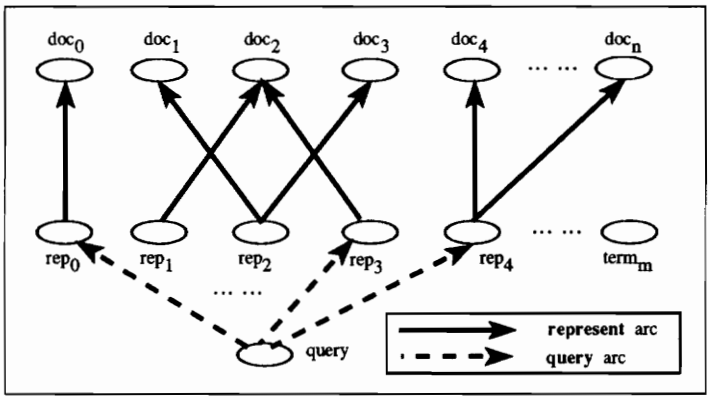


Figure 2.4: A Vector Retrieval Graph Representation

Attribute Set: Attribute set A is a countably infinite set of symbols. Attributes are used to represent properties of objects [Wan89].

Identifier Set: ID is the object identifier set. An identifier $id \in ID$ is a pair $< c, i >$, where both c and i are in the integer set Z. c is called the class code and i the instance code. Two functions $cls\_code()$ and ins_code are associated with id: $id.cls\_code() = c$ and $id.ins\_code() = i$.

Type Name Set: Type name set TN is a countably infinite set of symbols. In particular, each domain D_i has a type name d_i, $d_i \in TN$.

Class Name Set: CN is the countably infinite set of class names.

Method Name Set: MN is the countably infinite set of method names.

Note that all sets are disjoint.

2.4.2 Values

There are five types of values. The entire set of values is denoted by V.

Atomic Value: A value $v \in D_i$ is an *atomic value.*

Identifier Value: A value $id \in ID$ is an *identifier value.*

List Value: A value of the form $[v_1, v_2, \ldots, v_n]$, where $v_i \in V$, is called a *list value.*

Set Value: A value of the form $\{v_1, v_2, \ldots, v_n\}$, where $v_i \in V$, is called a set value.

Tuple Value: A tuple of the form $< a_1 : v_1, a_2 : v_2, \ldots, a_k : v_k >$, where $a_i \in A$ and $v_i \in V$, is called a *tuple value.*

2.4.3 Objects

A LEND object o is a pair (id, v), where $id \in ID$ and $v \in V$. To help discussion, we define three functions on o: $o.iden() = id$, $o.value() = v$, $o.ref()$ = all identifiers found in $o.value()$. Let Θ denote a set of LEND objects. Function $objs()$ returns normal objects in Θ. A set of LEND objects Θ is consistent iff:

- $\forall$ distinct objects o and $o' \in \Theta$, if $o.iden().cls\_code() = o'.iden().cls\_code()$, then $o.iden().ins\_code() \neq o'.iden().ins\_code()$. This condition assures the uniqueness of instance codes of a class.
- $\forall o \in \Theta$, $ref(o) \subseteq \bigcup_{o' \in \Theta} o'.iden()$. This condition states that all object identifiers must have corresponding objects.

2.4.4 Classes

A class captures the common features of a set of objects. It defines the value as well as the member functions for these objects. Both private and inheritable values can be defined, by using the basic, identifier, set, list and tuple values.

A method is composed of a signature part and an implementation part. The former has a function name, a set of arguments and a result, while the latter realizes the mapping from arguments to the result through computation, lookup or a combination of both. A method m defined by class c is defined as $m = (n, f, b)$, where n is the method name $\in MN$, $f : S \rightarrow T$ the mapping from domain S to range T, and b is the behavior (semantics) of f. The pair (n, f) is called the signature of m. The set of methods defined by c is denoted MT. The definition of f will be given after we explain the inheritance and objects associated with c.

A child class can be constructed by inheriting features from a parent class p. We use $subc(p)$ to denote the set of all child classes of p. For p and one of its child classes c, if the value portion of p is in c, c inherits the representation; if a method signature of p can be used by c, c inherits the method. Representation and method inheritance are allowed in LOM

for modeling flexibility. Although representation inheritance violates the encapsulation of p with respect to $subc(p)$, it presents no major problems to the IR applications of interest, as the depth of a relatively static class hierarchy in these applications may not be very deep. LOM allows the method inheritance to be multiple inheritance. The *parent_classes* function returns only names of the parent classes from which method signatures are inherited.

A class c is associated with a set of objects, each of which contains a value defined by the value type and is manipulated only by the set of methods owned by c. Following [SÖ90], we denote as $ext(c)$ the set of all objects of c, and as $ext^*(c)$ all instances of c and $subc(c)$. In set notation, the latter can be expressed as $ext^*(c) = ext(c) \cup \{\bigcup_{c' \in subc(c)} ext(c')\}$.

Given class c and its two associated object sets $ext(c)$ and $ext^*(c)$, S and T in signature part $f : S \rightarrow T$ of method m of c can be defined. S is the cartesian product:

$$S = S_1 \times S_2 \times \ldots \times S_n,$$

where S_1 is $ext(c)$, and S_2, S_3, ..., S_n, $T \subseteq \Theta \bigcup V$. S_2, S_3, ..., S_n are the arguments to the method, and T is the return value of m.

Now, we can define LEND class c. c is a tuple $c = (nm, P, vt_p, vt_h, MT_p, MT_s)$, where nm is the class name, P is the parent class name, vt_p is the private value type, vt_h is the inherited value type, MT_p is the inherited method set, and MT_s is the method set defined by c (including those used to override the inherited methodes). $MT = MT_p \bigcup MT_s$. In addition, a set of LEND classes should be maintained in a consistent state. A class set is consistent iff all class names are distinct and all method names defined on each class are distinct.

2.4.5 Database

A LEND database is a set of consistent classes forming a single hierarchy rooted at the LEND root class and a set of consistent objects associated with these classes. The operations on the objects are methods defined in the classes.

2.4.6 Methods for Important LEND Classes

In this section, we describe the values, method signatures and behaviors (semantics) for important LEND classes.

Root Class

The method set for the root class `Root` is shown in Table 2.1. Method *this_class* returns a string form of the class name. Methods *cls_code* and *ins_code* return the class and instance code of a root object, respectively. These two functions will be inherited by all classes $\subseteq$ *subc*(`Root`). *parent_classes* returns a set containing all parent class names. In the case of `Root`, this set is empty. Method *id_EQ* checks for equality of the identifiers of two objects. *value_LE* and *value_EQ* make less than and equal ($\leq$) and equal ($=$) comparisons of two objects with respect to their values. The semantics of value comparison operations depends on each specific class. Methods *to_string* and *from_string* convert the LEND object to and from strings. A parameter from the set $\{ASCII, BINARY\}$ controls the kind of string converted.

The value and behavior of `Root` is shown below. Argument o appearing in *id_EQ*, *value_LE* and *value_EQ* member functions must be a LEND object.

Value: $id =< c, i >$

Behavior:

$$
\begin{aligned}
cls\_code() &= c \\
ins\_code() &= i \\
parent\_classes() &= \{\} \\
this\_class() &= \{Root\} \\
id\_EQ(o) &= \begin{cases} TRUE & \text{if } c == o.c \text{ AND } i == o.i \\ FALSE & \text{otherwise} \end{cases} \\
value\_LE(o) &= FALSE
\end{aligned}
$$

$$value\_EQ(o) = FALSE$$

Constraint: No value part

Composite Class

This class modifies all methods of the `Root` class except *cls_code* and *ins_code*. New methods *first*, *get_component* and *next* allow iteration over components in the composite. Table 2.1 shows the signatures of these methods. Note that the method *get_component* returns an object as opposed to a value. The returning value approach is adopted in O_2's Reloop query language [CDLR90]. We argue here that our approach is the right way to promote encapsulation even at the component level and to facilitate data type (defining new class) extension. Methods `insert` and `delete` update the composite object. The method *count* returns the number of components in the composite object. Since this class serves as the base class for other composite classes, its member functions are never called. Hence, we do not describe the behavior of the iteration and count member functions.

Value: $id =< c, i >$

Behavior:

$$parent\_classes() = \{\texttt{Root}\}$$
$$this\_class() = \{\texttt{Composite}\}$$

Constraint: ext(`Composite`) = {}

Tuple Class

The tuple class contains a tuple value. This class modifies all methods of the `Root` and `Composite` classes except *cls_code* and *ins_code*. The new method *attr_to_num* (shown in

Table 2.1) returns the numerical index of a component that can be indexed by its attribute name (type *String*). In the following, o_{bad} is used to denote an object used to indicate method execution error. Function *to_object* converts a value to a value object. Specifically, it converts a normal identifier to an identifier object `id_object`, which contains only an identifier value.

Value: $id =< c, i >$, and a tuple value $v =< a_1 : v_1, a_2 : v_2, \ldots, a_n : v_n >$

Behavior:

$$
\begin{aligned}
parent\_classes() &= \{\texttt{Root}, \texttt{Composite}\} \\
this\_class() &= \{\texttt{Tuple}\} \\
first() &= 1 \\
get\_component(i) &= \begin{cases} v_i & \text{if } i \in [1 \ldots n] \text{ and } v_i \text{ is an object} \\ to\_object(v_i) & \text{if } i \in [1 \ldots n] \text{ and } v_i \text{ is a value} \\ o_{bad} & \text{otherwise} \end{cases} \\
next(i) &= \begin{cases} i+1 & \text{if } i < n \\ 0 & \text{otherwise} \end{cases} \\
count() &= n \\
att\_to\_num(name) &= \begin{cases} i & \text{if } name == a_i \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

Constraint:

$$v_i \in \Theta \bigcup V, \forall i \in [1 \ldots n]$$

$$n \geq 0$$

Set Class

This class modifies all methods of the `Root` and `Composite` classes except *cls_code* and *i_ code*. Well-known operations *intersect*, *union* and *difference* are included. All three

functions take an extra parameter determining whether the value or object comparison operation should be applied.

Value: $id =< c, i >$, and a set value $v = \{v_1, v_2, \ldots, v_n\}$

Behavior:

$$
\begin{aligned}
\mathit{parent\_classes}() &= \{\texttt{Root}, \texttt{Composite}\} \\
\mathit{this\_class}() &= \{\texttt{Set}\} \\
\mathit{first}() &= 1 \\
\mathit{get\_component}(i) &= \begin{cases} v_i & \text{if } i \in [1 \ldots n] \text{ and } v_i \text{ is an object} \\ \mathit{to\_object}(v_i) & \text{if } i \in [1 \ldots n] \text{ and } v_i \text{ is a value} \\ o_{bad} & \text{otherwise} \end{cases} \\
\mathit{next}(i) &= \begin{cases} i+1 & \text{if } i < n \\ 0 & \text{otherwise} \end{cases} \\
\mathit{count}() &= n
\end{aligned}
$$

Constraint:

$$v_i \in \Theta \bigcup V, \forall i \in [1 \ldots n]$$

$$n \geq 0$$

List Class

This class modifies all methods of the `Root` and `Composite` class except *cls_code* and *ins_code*. A new method *compose* combines two lists into one. Methods *insert* and *delete* update the lists. In the following, l denotes the argument list object to the *compose* function. Its value is expressed as $v^l = [v_1^l, v_2^l, \ldots, v_n^l]$. The *compose* method is useful to join together two paths in a hypertext graph where the last part of the one matches the first part of the other.

Value: $id =< c, i >$, and a list value $v = [v_1, v_2, \ldots, v_n]$

Behavior:

$$
\begin{aligned}
parent\_classes() &= \{\texttt{Root}, \texttt{Composite}\} \\
this\_class() &= \{\texttt{List}\} \\
first() &= 1 \\
get\_component(i) &= \begin{cases} v_i & \text{if } i \in [1 \ldots n] \text{ and } v_i \text{ is an object} \\ to\_object(v_i) & \text{if } i \in [1 \ldots n] \text{ and } v_i \text{ is a value} \\ o_{bad} & \text{otherwise} \end{cases} \\
next(i) &= \begin{cases} i+1 & \text{if } i < n \\ 0 & \text{otherwise} \end{cases} \\
count() &= n \\
compose(l) &= \begin{cases} [v_1, v_2, \ldots, v_n, v_2^l, \ldots, v_n^l] & \text{if } v_n \text{ and } v_1^l \text{ are objects AND} \\ & \text{if } v_n.id\_EQ(v_1^l) == TRUE \\ o_{bad} & \text{otherwise} \end{cases}
\end{aligned}
$$

Constraint:

$$v_i \in \Theta \bigcup V, \forall i \in [1 \ldots n]$$

$$n \geq 0$$

Graph Root Class

This class forms the root for all classes used in the Graph Component. It inherits the identifier representation and all methods from `Root`.

Value: Same as the `Root` class

Table 2.1: Method Signatures for Classes in Base Component

Class	Name	Method Signatures
Root	cls_code	$Root \rightarrow Num$
	ins_code	$Root \rightarrow Num$
	id_LE	$Root \times Root \rightarrow Boolean$
	id_EQ	$Root \times Root \rightarrow Boolean$
	value_LE	$Root \times Root \rightarrow Boolean$
	value_EQ	$Root \times Root \rightarrow Boolean$
	to_string	$Root \times \{ASCII, BINARY\} \rightarrow String$
	from_string	$Root \times String \times \{ASCII, BINARY\} \rightarrow Boolean$
	parent_classes	$Root \rightarrow StringList$
	this_class	$Root \rightarrow String$
Composite	first	$Composite \rightarrow Num$
	get_component	$Composite \times Num \rightarrow Root$
	next	$Composite \times Num \rightarrow Num$
	insert	$Composite \times Root \rightarrow Composite$
	delete	$Composite \times Root \rightarrow Composite$
	count	$Composite \rightarrow Num$
Tuple	attr_to_num	$Tuple \times String \rightarrow Num$
Set	intersect	$Set \times Set \times \{VALUE, OBJECT\} \rightarrow Set$
	union	$Set \times Set \times \{VALUE, OBJECT\} \rightarrow Set$
	difference	$Set \times Set \times \{VALUE, OBJECT\} \rightarrow Set$
List	compose	$List \times List \rightarrow List$
	insert	$List \times Root \times Num \rightarrow List$
	delete	$List \times Num \rightarrow List$

Behavior:

$$\begin{aligned} \textit{parent\_classes}() &= \{\texttt{Root}\} \\ \textit{this\_class}() &= \{\texttt{Graph\_Root}\} \end{aligned}$$

Constraint: ext(`Graph_Root`) = {}

Node Class

This class serves as the node class. It inherits the identifier and all methods from `Graph_Root`.

Value: Same as `Root` class.

Behavior:

$$\begin{aligned} \textit{parent\_classes}() &= \{\texttt{Root}, \texttt{Graph\_Root}\} \\ \textit{this\_class}() &= \{\texttt{Node}\} \\ \textit{sinks}(\textit{name}) &= \text{a set of } \texttt{Node} \text{ objects } (\texttt{N\_Set}) \text{ connected to} \\ & \quad \text{this } \texttt{Node} \text{ object through label } \textit{name} \end{aligned}$$

Constraint: ext(`Node`) = {}

Label Class

This class serves as the label class. It inherits the identifier and all methods from `String` (as representation).

Value: $id =< c, i >$ and a string s from the string domain.

Behavior:

$$\begin{aligned} get\_label() &= \text{s} \\ parent\_classes() &= \{\texttt{Root}, \texttt{Graph\_Root}\} \\ this\_class() &= \{\texttt{Label}\} \end{aligned}$$

Constraint: The strings cannot be empty.

Directed Arc Class

This class is the directed arc subclass of `Graph_Root`. It modifies all methods of the `Graph_Root` class except *cls_code* and *ins_code*. A tuple value serves as the representation.

Value: $id =< c, i >$, and a tuple value $v =< SOURCE : n_1, LABEL : n_2, SINK : n_3 >$

Behavior:

$$\begin{aligned} parent\_classes() &= \{\texttt{Root}, \texttt{Graph\_Root}\} \\ this\_class() &= \{\texttt{Arc}\} \\ first() &= 1 \\ get\_component(i) &= \begin{cases} n_i & \text{if } i \in [1,2,3] \\ o_{bad} & \text{otherwise} \end{cases} \\ next(i) &= \begin{cases} i+1 & \text{if } i \in [1,2] \\ 0 & \text{otherwise} \end{cases} \\ count() &= 3 \end{aligned}$$

Constraint:

$$n_1 \text{ and } n_3 \in ext^*(Node)$$

$$n_2 \in ext^*(Label)$$

Path Class

This class is the subclass of `Graph Root` and contains a `List` value. The method set of the path class is shown in Table 2.2. This class modifies all methods of the `Graph_Root` class except *cls_code* and *ins_code*. Modified methods *get_source_node* and *get_sink_node* return the source and sink node object of the path. Method *append* appends two paths together; *pass_node* and *pass_label* test whether a node object or a label object exists on the path.

Value: $id =< c, i >$, and a list value $v = [n_1, m_2, n_3, m_4, \ldots, m_{2n}, n_{2n+1}]$

Behavior:

$$parent\_classes() = \{\texttt{Root}, \texttt{Graph\_Root}\}$$

$$this\_class() = \{\texttt{Path}\}$$

$$first() = 1$$

$$get\_component(i) = \begin{cases} n_i & \text{if } i \in \{1, 3, \ldots, 2n+1\} \\ m_i & \text{if } i \in \{2, 4, \ldots, 2n\} \\ o_{bad} & \text{otherwise} \end{cases}$$

$$next(i) = \begin{cases} i+1 & \text{if } i \in [1, n-1] \\ 0 & \text{otherwise} \end{cases}$$

$$pass\_node(o) = \begin{cases} TRUE & \text{if } o.id\_EQ(n_i) \text{ and } i \in \{1, 3, \ldots, 2n+1\} \\ FALSE & \text{otherwise} \end{cases}$$

$$pass\_label(o) = \begin{cases} TRUE & \text{if } o.id\_EQ(m_i) \text{ and } i \in \{2, 4, \ldots, 2n\} \\ FALSE & \text{otherwise} \end{cases}$$

$$compose(p_2) = \begin{cases} [n_1, \ldots, n_{2n+1}, m_2^2, \ldots, n_{2n+1}^2] & \text{if } n_{2n+1}.id\_EQ(n_1^2) \\ o_{bad} & \text{otherwise} \end{cases}$$

$$count() = 2n + 1$$

$$node\_count() = n$$

$$label\_count() = n - 1$$

Constraint:

$$n \geq 0$$

$$n_i \in ext^*(Node), \forall i \in \{1, 3, 5, \ldots, 2n+1\}$$

$$m_i \in ext^*(Label), \forall i \in \{2, 4, \ldots, 2n\}$$

Node Set Class

The method set for the node set class (called `N_Set`) is shown in Table 2.2. One new method $sinks(Label)$ is introduced. It returns a new node set containing all nodes connected through $Label$.

Value: $id =< c, i >$, and a set value $v = \{n_1, n_2, \ldots, n_n\}$

Behavior:

$$parent\_classes() = \{\texttt{Root}, \texttt{Composite}, \texttt{Set}\}$$

$$this\_class() = \{\texttt{Node\_Set}\}$$

$$sinks(name) = \bigcup_{i=1}^{n} \{n_i.sinks(name)\}$$

Constraint:

$$n_i \in ext^*(Node) \quad \forall i \in [1 \ldots n]$$

Table 2.2: Method Signatures for Classes in Graph Component

Class	Name	Method Signatures
Node	sinks	$Node \times String \rightarrow N\_Set$
Label	get_label	$Label \rightarrow String$
Arc	create	$Node \times Label \times Node \rightarrow Arc$
Path	create	$Arc \rightarrow Path$
	pass_node	$Path \times Node \rightarrow Boolean$
	pass_label	$Path \times Label \rightarrow Boolean$
	compose	$Path \times Path \rightarrow Path$
	node_count	$Path \rightarrow Num$
	label_count	$Path \rightarrow Num$
N_Set	sinks	$N\_Set \times Label \rightarrow N\_Set$
Arc_Set	create	$N\_Set \times Label \times N\_Set \rightarrow N\_Set$
	sources	$Arc\_Set \rightarrow N\_Set$
	sinks	$Arc\_Set \rightarrow N\_Set$
P_Set	create	$Arc\_Set \rightarrow P\_Set$
	forward	$P\_Set \times Num \rightarrow N\_Set$
	backward	$P\_Set \times Num \rightarrow N\_Set$
	iterate	$P\_Set \times N\_Set \times N\_Set \times Num \times Num \rightarrow P\_Set$
	closure	$P\_Set \times N\_Set \times N\_Set \rightarrow P\_Set$
	compose	$P\_Set \times P\_Set \rightarrow P\_Set$

Directed Arc Set Class

The directed arc set class (called `Arc_Set`) is the direct subclass of set `Set`. The method set for the class is shown in Table 2.2. Methods *sources* and *sinks* return source or sink nodes for all arcs in the arc set.

Value: $id =< c, i >$, and a set value $v = \{a_1, a_2, \ldots, a_n\}$

Behavior:

$$\begin{aligned} parent\_classes() &= \{\texttt{Root}, \texttt{Composite}, \texttt{Set}\} \\ this\_class() &= \{\texttt{Arc\_Set}\} \\ sources() &= \bigcup_{i=1}^{n} \{a_i.get\_component(1)\} \\ sinks() &= \bigcup_{i=1}^{n} \{a_i.get\_component(3)\} \end{aligned}$$

Constraint:

$$a_i \in ext^*(Arc), \quad \forall i \in [1 \ldots n]$$

Path Set Class

The method set for the path set class (denoted as `P_Set`) is shown in Table 2.2. New method *iterate* produces a new path set whose elements are paths recursively concatenated by paths in the argument path set. The second and third parameters are node sets determining the source and sink nodes of the final paths. The last two Num parameters set the lower and the upper iteration bounds. Method *compose* finds all possible paths that can be appended together from any two paths in the two parameter path sets.

Value: $id =< c, i >$, and a set value $v = \{p_1, p_2, \ldots, p_n\}$

Behavior:

$$
\begin{aligned}
parent\_classes() &= \{\texttt{Root}, \texttt{Composite}, \texttt{Set}\} \\
this\_class() &= \{\texttt{Path\_Set}\} \\
forward(i) &= \bigcup_{i=1}^{n} \{p_i.get\_component(i)\} \\
backward(i) &= \bigcup_{i=1}^{n} \{p_i.get\_component(p_i.count() - i)\} \\
iterate(n_1, n_2, i_1, i_2) &= \text{a path set } p\_set \text{ where} \\
& \quad p\_set.forward(1).id\_EQ(n_1) == TRUE \text{ AND} \\
& \quad p\_set.backward(1).id\_EQ(n_2) == TRUE \text{ AND} \\
& \quad p\_set.v = \{q_x \mid q_x \text{ is formed by appending} \\
& \quad \text{at least } i_1 \text{ and at most } i_2 \text{ paths from } v\} \\
compose(p\_set_1) &= \text{a path set } p\_set \text{ where} \\
& \quad p\_set.v = \{q_x \mid q_x = p_i.append(p_j), \\
& \quad i \in \{1 \ldots n\},\ p_j \in p\_set_1\}.
\end{aligned}
$$

Constraint: None

SNePS Related Classes

SNePS classes model a restricted set of SNePS path constructs. In LOM, a SNePS path is considered as a sequence of labels. The path corresponds to a set of paths defined in the Graph Component, where each of them bears the labels shown in the label sequence. A SNePS path o is modeled in LEND as an object having a tuple value containing two attributes: $< PATH : x, SET : y >$, where x is the label sequence (list) and y is the physical path set.

Consider Figure 2.5 (taken from [FNA+88] but with an additional arc **TAX** connecting **goat** to **animal**), which shows a simplified SNePS representation of lexical relations involving the word *Sheep*. For the SNePS path (**TAX**, **PART**, **TAX**), we obtain two physical

instances

$$p_1 = (\texttt{hair}, \texttt{TAX}, \texttt{wool}, \texttt{PART}, \texttt{sheep}, \texttt{TAX}, \texttt{animal})$$

$$p_2 = (\texttt{hair}, \texttt{TAX}, \texttt{wool}, \texttt{PART}, \texttt{goat}, \texttt{TAX}, \texttt{animal})$$

Thus, the SNePS path (TAX, PART, TAX) is modeled as a tuple $< PATH : x, SET : y >$, where x = [TAX, PART, TAX], and $y = \{p_1, p_2\}$.

Six classes are provided for the storage and access of SNePS path-based operations. These six classes are SNePS label (SN_Label), SNePS node (SN_Node), SNePS arc (SN_Arc), SNePS path (SN_Path), SNePS path list (SN_List), and SNePS path set (SN_Set). The method signatures of three nontrivial classes (SN_Path, SN_List, and SN_Set) are shown in Table 2.3. The actual semantics of these methods are similar to those of identically named functions presented in the SNePS's user manual [Sha89]. The specification for the SNePS_Path class is listed below. To help with explanation, we denote as $reverse(p)$ the reverse of a LEND path p; as $forward(1)$ the first node in the path p; and $backward(1)$ the last node in p.

Value: $ID =< c, i >$, and a tuple value $v =< PATH : x, SET : y >$ where

x is the label sequence $[l_1, l_2 \ldots, l_n]$ and

y is the physical path set $\{p_1, p_2 \ldots, p_m\}$

Behavior:

$$parent\_classes() = \{\texttt{Root}, \texttt{Graph\_Root}\}$$

$$this\_class() = \{\texttt{SNePS\_Path}\}$$

$$converse() = q \text{ where } q.x = [m_n, \ldots, m_2, m_1] \text{ and } q.y = \bigcup_{i=1}^{m} reverse(p_i)$$

$$kstar() = \begin{cases} q \text{ where } & q.x = [KSTAR] \\ & q.y = \{\text{paths that are iterations} \\ & \text{of } r \in o.y \text{ zero or more times}\} \end{cases}$$

$$kplus() = \begin{cases} q \text{ where } & q.x = [KPLUS] \\ & q.y = \{\text{paths that are iterations} \\ & \text{of } r \in o.y \text{ one or more times}\} \end{cases}$$

$$irreflexive() = \begin{cases} q \text{ where } & q.x = [IRREFLEXIVE] \\ & q.y = \text{subset of } p.y \text{ such that} \\ & q.y.p_i.forward(1).id\_EQ(q.y.p_i.backward(1)) \\ & == FALSE \end{cases}$$

$$d\_restrict(n,q) = \begin{cases} \text{itself} & \text{if } q \text{ is from } n_1 \text{ to } n \\ o_{bad} & \text{otherwise} \end{cases}$$

$$r\_restrict(n,q) = \begin{cases} \text{itself} & \text{if } q \text{ is from } n_{2n+1} \text{ to } n \\ o_{bad} & \text{otherwise} \end{cases}$$

Constraint:

$$l_i \in ext(SN\_Label) \quad \forall i \in [1 \ldots n]$$

The specification for the **compose** method of SNePS path list **SN_List** is listed below.

Value: $id =< c, i >$, and a list value $v = [o_1, o_2, \ldots, o_n]$ where o_i is a SNePS path, $\forall i$

Behavior:

$$parent\_classes() = \{\texttt{Root}, \texttt{Composite}, \texttt{List}\}$$

$$this\_class() = \{\texttt{SN\_List}\}$$

$$compose() = \begin{cases} q \text{ where } & q.x = [COMPOSE] \text{ and} \\ & q.y = \{\text{path appended by} \\ & \text{paths } o_i.y.p_j \\ & \forall i \in [1 \ldots n] \text{ and } \exists j \in [1, \ldots, |o_i.y|]\} \end{cases}$$

Constraint:

$$o_i \in ext(SN\_Path) \quad \forall i \in [1 \ldots n]$$

The specification for the **and** and **or** method of SNePS path set **SN_set** is listed below.

Value: $id =< c, i >$, and a set value $v = \{o_1, o_2, \ldots, o_n\}$

Behavior:

$$parent\_classes() = \{\texttt{Root}, \texttt{Composite}, \texttt{Set}\}$$

$$this\_class() = \{\texttt{SN\_Set}\}$$

$$and() = \begin{cases} q \text{ where } & q.x = [AND] \text{ and} \\ & q.y = \{\text{path} = [u, v],\ u \in \bigcap_{i=1}^{n} o_i.q.forward(1) \\ & \text{and } v \text{ in } \bigcap_{i=1}^{n} o_i.q.backward(1)\} \end{cases}$$

$$or() = \begin{cases} q \text{ where } & q.x = [OR] \text{ and} \\ & q.y = \{\text{path} = [u, v],\ u \in \bigcup_{i=1}^{n} o_i.q.forward(1) \\ & \text{and } v \text{ in } \bigcup_{i=1}^{n} o_i.q.backward(1)\} \end{cases}$$

Constraint:

$$o_i \in ext(SN\_Path) \quad \forall i \in [1 \ldots n]$$

Hypertext Class

The Hypertext class captures the transition relations between documents, anchors and queries. The signatures of these transition methods are listed in Table 2.3, where *Ach_Set* stands for an anchor set, *AA_Set* for the anchor to anchor hypertext link set, and *DA_Set* for the document-anchor link set. *Query* is the query object used to identify a set of initial anchors as in the method *jump* or to select a new set of anchors as in the method *traverse*. The display construct of a hypertext system is not modeled, as it is not a data storage issue.

IR Model Class

The method **retrieve** identifies all document objects satisfying the query *Query* and returns them in the list *Doc_List*. The method *feedback* produces a new query by taking the original query, the document list retrieved and the numerical judgement list. Both methods are shown in Table 2.3.

Boolean Model Class

A new method for the Boolean model class is shown in Table 2.3. Method *create* constructs a Boolean model object from a document node set *Doc_Set*, a representation node set *Rep_Set* and a representation to document link set *DR_L_Set*.

Vector Model Class

The vector model class is similar to the Boolean model class, except that its *create* method takes a special link set *DR_L_Set* whose link elements encode the weight information.

2.4.7 Summary

In this section, we formally present the LEND model. The model includes three parts: the Base Component, the Graph Component, and the IR Component. The constructs in the Base Component can be found in typical object-oriented database systems. The constructs in the Graph and IR Components are unique to LEND. The model provides the following important features: unique object identifiers, composite objects (tuples, sets, or lists), class hierarchy, object encapsulation, method attachment to objects, node objects to lay a foundation for arbitrary references, and arcs and paths to realize arbitrary references.

The object identifiers make data sharing possible. Composite objects help with modeling of complicated document structures. The class hierarchy enables the reuse of existing classes for the definition of new classes. Method attachment facilitates computation modeling (i.e., similarity computation in the vector model). Node, arc and path constructs support SNePS

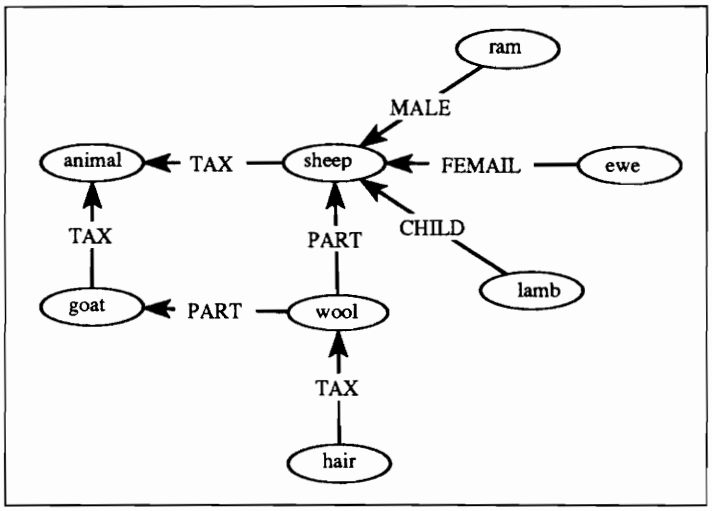


Figure 2.5: SNePS Representation Example

Table 2.3: Method Signatures for Classes in IR Component

Class	Name	Method Signatures
SN_Path	create	$SN\_Arc \rightarrow SN\_Path$
	converse	$SN\_Path \rightarrow SN\_Path$
	kstar	$SN\_Path \rightarrow SN\_Path$
	kplus	$SN\_Path \rightarrow SN\_Path$
	irreflexive	$SN\_Path \rightarrow SN\_Path$
	d_restrict	$SN\_Path \times SN\_N\_Set \times SN\_Path \rightarrow SN\_Path$
	r_restrict	$SN\_Path \times SN\_N\_Set \times SN\_Path \rightarrow SN\_Path$
SN_List	compose	$SN\_P\_List \rightarrow SN\_Path$
SN_Set	and	$SN\_P\_Set \rightarrow SN\_Path$
	or	$SN\_P\_Set \rightarrow SN\_Path$
HyperText	create	$Doc\_Set \times Ach\_Set \times AA\_Set \times DA\_Set \rightarrow HyperText$
	jump	$HyperText \times Query \rightarrow Ach\_Set$
	traverse	$HyperText \times Ach\_Set \times AA\_Set \times Query \rightarrow Ach\_Set$
	scrutinize	$HyperText \times Ach\_Set \times DA\_Set \rightarrow Doc\_Set$
	back_to_anchor	$HyperText \times Doc\_Set \times DA\_Set \rightarrow Ach\_Set$
IR_Model	retrieve	$IR\_Model \times Query \rightarrow Doc\_List$
	feedback	$IR\_Model \times Query \times Doc\_List \times Num\_List \rightarrow Query$
B_Model	create	$Doc\_Set \times DR\_L\_Set \times Rep\_Set \rightarrow B\_Model$
V_Model	create	$Doc\_Set \times DR\_L\_Set \times Rep\_Set \rightarrow V\_Model$

and Hypertext applications.

2.5 Specification of the Query Language

2.5.1 The Query Language

Table 2.4 lists the partial BNF syntax for the language. Assume G is a directed IR graph with node set $N(G)$ and arc set $E(G)$. A query Q written in the query language describes a subgraph of G that is of interest. The processing of Q either retrieves the subgraph or returns an empty graph, if no such subgraph exists. In the following paragraphs, we describe the meaning of various constructs in the language.

The "Paths" component represents a set of paths, each being a sequence of alternating nodes and labels. The *virtualPaths* operator just specifies paths using a sequence of nodes and labels. Three methods *compose*, *iterate* and *closure* defined on the path set `P_Set` are available in the language. They have semantics as defined in section 2.4.

The "Nodes" component represents a set of possible nodes in G. Basically, there are three methods of constructing such a node set. The first is to mention the node set directly by its name. In the case that the set is also the entire instance set of a class, the class name can be used. The second is to use the construct $X.sinks(L)$. The defined node set contains all nodes connected to the neighbor nodes in X through arcs labeled L. The third method is through selection. The select expression Sel_Exp qualifies nodes to be returned.

A Sel_Exp expression evaluates components of nodes within $Nodes$. The expression is made up of component terms in the form of $Component\ OP\ Value$. The *Component* construct specifies the component. In the language, attributes of tuples and elements of lists nested arbitrarily deep can be the components. The way in which an attribute component in an object is selected can also be found in [CDLR90] and [SK91]. In [SK91], it is called a path expression. The list notation $forward()$ and $backward()$ is new in LEND. The OP is the legal comparison operator applicable to the component. $Value\_Object$ is the value participating in the comparison.

Table 2.4: Partial BNF Syntax for the Pattern Language

`Paths`	::=	Path_Name |
		virtualPaths(**Nodes**, **Label**, **Nodes** [**Tail_Exp**]) |
		Paths.compose(**Paths**) |
		Paths.iterate(**Nodes**, **Nodes**, **Num**, **Num**) |
		Paths.closure(**Nodes**, **Nodes**)
`Nodes`	::=	Nodes_Name |
		Nodes.sinks(**Label**) |
		Nodes.select(**Select_Exp**) |
		Paths.forward(**Num**) |
		Paths.backward(**Num**)
`Label`	::=	Label_Name
`Tail_Exp`	::=	, **Label**, **Nodes** [**Tail_Exp**]
`Select_Exp`	::=	**Select_Exp** AND **Select_Exp** |
		Select_Exp OR **Select_Exp** |
		NOT **Select_Exp** |
		(**Select_Exp**) |
		Component OP Value_Object
`Component`	::=	*attribute_name* |
		forward(**Num**) | *backward*(**Num**) |
		[.**Component**]
`OP`	::=	== | <

The "Label" component is relatively simple. A label can be constructed from a label name.

The arc set construct is not available in the language as its role can be fulfilled by a node set's *sinks(Label)* operator and the path set construct. Well-known operations *intersect*, *union* and *difference* for "Paths" and "Nodes", operating on two set valued components, are in the language but are not shown. Also, objects can be created in the language and assigned to object variables. Invocation of methods on either the objects or the object variables is permitted.

2.5.2 Query Examples

This section lists several queries formulated using the pattern language.

Q_1 Find all documents that are cited by chapters of documents in document set X. Assume there are two node sets *docs* and *chapters*, and two arc sets *doc_chapter* and *cite*. *doc_chapter* connects *docs* to *chapters*, and *cite* representing citations connects from *chapters* to *docs*. Let $X \subseteq docs$.

Query X.sinks("doc_chapter").sinks("cite").

Note This query can be expressed using two joins in a relational database.

Q_2 Find all documents using concepts extracted from chapter titles by using vector searching. The vector model query is Q. Assume *chapters* is connected to *titles* through *has_title* and *titles* is connected to *concepts* through *has_concepts*. The weighted arc set is *weights*.

Query

```
paths p = virtualPaths("docs", "has_chapter", "chapters", "has_title", "titles",
"has_concept", "concepts");
nodes concept_subset = p.backward(1);
```

V_Model vector_model = newVector("docs", "weights", concept_subset);
vector_model.retrieve(Q);

Note This query can't be expressed in a traditional relational database, as computation (by invoking method *retrieve*) is needed.

Q_3 List concepts extracted from the titles and text bodies of documents in document set X that are also in concept set Y. Assume another node set *text_bodies* which is connected to *chapters* through *has_body*. *text_bodies* is connected to *concepts* through *has_concepts* arcs.

Query

chapter chapter_subsets = X.sinks("has_chapter");
paths p1 = virtualPaths(chapter_subsets, "has_concepts", "concepts");
paths p2 = virtualPaths(chapter_subsets, "has_body",
"text_body", "has_concepts", "concepts"));
p1.sinks().intersect(p2.sinks().intersect(Y));

Note This query can be expressed in relational database systems using join, select, and intersection operations.

Q_4 Find all synonyms of word x that can be reached in at most 4 steps from x. Assume arc set *synonym* connects node set *words* to itself.

Query

paths p = virtualPaths("words", "synonym", "words");
p.iterate({x}, "words", 1, 4).sinks();

Note This query can be clumsily expressed in relational database systems using 3 join operations. Some project and select operations are needed also.

Q_5 Find all words that cite word x as their synonyms. Assume words in the dictionary are stored in the node set *words* and the inverse relation of *synonym* is denoted as

synonym−. Each word in *words* has a tuple value containing the attribute < *word* : *String* >.

Query

words.select(word == "x").sinks("synonym-");

Note This query can be expressed in relational database systems using join, select, and project operations.

Q_6 Find all paths of length at most 10 between word x and y. The labels on the paths should be "synonym".

Query

paths p = virtualPaths("words", "synonym", "words");
p.iterate("words".select(word == "x"), "words".select(word == "y"), 1, 10);

Note This query can be clumsily expressed in relational database systems, using 9 joins.

Q_7 Find all entry terms with respect to word x in the controlled vocabulary. Assume all entry terms are in the node set *entry*, all controlled words are in the *controlled* node set, and the two sets are connected by the arc set *has_entry*.

Query

words.select(word == "x").sinks("has_entry");

Note This query can be expressed in relational database systems using join, select, and project operations.

Q_8 Find all controlled words having a narrower sense than entry term x. Assume the arc set *sub_super* connects the controlled word set *controlled*. We need to find first the controlled words relative to x. Then we search downward in a hierarchy from these controlled words.

Query

```
Nodes n = words.select(word == "x").sinks("has_entry");
paths = virtualPath(n, "sub_super", "controlled");
paths.iterate(n, "controlled").sinks();
```

Note This query can't be expressed in relational database systems, as transitive closure computation is required.

2.6 Summary

In this chapter, several important IR models are reviewed. Various problems with current IR systems are pointed out. An object-oriented model is constructed that allows integrated storage and access for all of these important IR models. A formal specification of the model and important method signatures and behaviors are given.

The LEND model captures two essential aspects of IR applications: being aggregated and being connected. The first aspect is typical in perceiving things as sets or lists of primitive or compound items. The second aspect is typical when thinking of arbitrary reference relationships between those items. The IR component makes use of the base and the graph components. The base component helps to express the retrieval results and the graph component helps to share document, representation and other objects among different IR models. The computation involved in retrieval is captured at the method level.

A pattern language designed for Graph Component access is presented. It is set based but includes many features that are graph-oriented. Queries involving paths can be expressed by the language. Several queries are shown to demonstrate that the language is useful for IR applications.

The LEND model is also object-oriented. This allows functions to be tightly bundled with the data. Representation and method inheritance makes it easy to reuse existing classes and to incorporate new classes.

Chapter 3

LEND SYSTEM DESIGN

This chapter describes the LEND system, an object-oriented database system prototyping the LEND model, from the point of view of system design and implementation. LEND has three layers: the lowest or storage layer, the intermediate object layer, and the topmost application layer. Similar to existing reference models, lower layers provide functions needed by upper layers. The three layers are described in sections 3.1, 3.2 and 3.3, respectively. The layers are implemented in C++ [Str86]. To help with the explanation, some important class profiles are abbreviated and included.

3.1 Storage Layer

The storage layer provides storage and access facilities to LEND binary string objects on a `storage object`. Storage objects are implemented on UNIX files, main memory, or partitions of a device that supports paged access. The major functions of this layer are to deliver binary string objects to the intermediate level layer (object layer) upon a `readString()` request and to save binary string objects in the store upon receiving a `writeString()` or `appendString()` request. Based on various types of storage media, additional functions can be added.

As discussed in [ZM90] (Chapter 4, pp. 237-241), there are four major issues in design-

ing a storage component for an object-oriented database system: the degree of exposure of semantics of objects to the storage component, the physical placement of objects, the implementation of object identifiers, and objects' disk and memory format.

3.1.1 The Degree of Exposure

The first issue concerns how much the storage layer should know about objects, specifically the identifiers embedded in the objects. Acquaintance with object semantics helps garbage collection, object prefetching and constraint maintenance. On the other hand, viewing objects as binary strings can keep the storage layer immune from object model evolution and help with simplifying the implementation. Thus, the LEND storage layer takes the latter approach, as garbage collection and dynamic constraint maintenance are not necessary for relatively static IR data. The intermediate results generated during pattern language query processing will be remembered as objects and finally removed. Object prefetching is directly handled through a graph partitioning algorithm. Although the semantics of objects is ignored, the layer always assumes that the object identifier is stored in the first few bytes of the binary string. Further, the length of the binary string is not recorded if it can be inferred from the class identified by the identifier.

Since objects are viewed as binary strings, the functionality of the storage layer can be simplified into several function calls. The basic calls are `readString()`, `writeString()` and `appendString()`. All are defined on the storage class and use a pseudo index object (abbreviated as `Pix`) to remember position information. A `Pix` is also a class and can be expanded for specific storage devices. A storage object can contain an array of pointers pointing at device management policy routines, such as how to perform caching and buffering. Alternatively, management policy can be organized into a hierarchy and proper policy instances can be invoked using C++'s dynamic binding feature.

3.1.2 Object Placement

The second issue regards the placement of objects on disk. Physically, objects can be represented with or without composition. Composite representation means constituents are tied to their composite object, while decomposition puts similarly typed constituents together. Object placement will save these representations on disk to minimize the number of I/O accesses. As graph structured data is the primary concern, this layer makes use of a clustering algorithm. The placement decision made by the algorithm is based on the entire structure of a graph.

3.1.3 Object Identifier Implementation

The third issue is about the implementation of object identifiers. An identifier can be coded as the physical location, as the relative physical location of an object with respect to a page or a segment, or as a logical symbol independent of physical storage. The physical address approach makes it impossible for the object to move, but allows direct object reference. The logical address method achieves just the opposite effect. The current implementation uses logical identifiers (i.e., the class code and instance code pair). An identifier hierarchy can be utilized to incorporate other possibilities.

3.1.4 Object Format

The last design issue is in connection with objects' disk and memory format. If they differ, some translation is necessary. The process can be initiated when the object is fetched or referenced. Partial translation is also possible for only the components in the object referenced. Translation makes it possible to isolate desired objects from pages that other objects reside on. Thus, an object cache can be separated from a page cache. The translation overhead is the disadvantage of the approach. The mono-format approach avoids the overhead but makes the objects fetched more closely related to the page cache. Earlier versions of object-oriented database systems (e.g., GEMSTONE [BOS91]) use dual formats. There is

a trend toward use of mono-format. The current LEND implementation takes the dual format approach. Objects are initialized when they are first fetched.

3.1.5 Additional Design Decisions

In designing the layer, three additional goals are considered: extensibility, low-overhead and reasonable performance.

The first goal concerns adding new storage devices to the layer. Since C++ allows class derivation, this principle is easy to realize. When a new device is added, a new C++ class and related management policy routines should be encoded.

The second goal deals with reduction of any unnecessary overhead in storing and accessing binary strings. Copying (long) binary strings is one example that would severely impair object delivery speed. Another case is when a binary string is very long and the corresponding large object does not need to be instantiated completely. In this circumstance, a sequence of reads of shorter binary strings is the proper solution. In general, overhead presents itself in different forms within different storage classes. No solution is ideal for all cases. With the object-oriented implementation, designers of each class can focus on situations leading to the worst overhead problems and provide ways to eliminate them.

The last goal is about maintaining reasonable performance for access to binary string objects. As current external storage devices are best characterized as linear, and prefer localized access, indexing and clustering are the most common solutions. LEND provides a set of indexing data structures and algorithms ideal for data that are relatively static and large in quantity. They are basically hash methods with very low storage cost and superior performance. One disk access is always guaranteed when this scheme is applied to disk storage objects. The memory storage uses AVL trees. The clustering itself falls into another set of classes. Graph bisection algorithms are used for thesauri that are encoded in semantic networks.

Figure 3.1 depicts the major components of the storage layer. The storage object manager manages three types of storage object: UNIX file storage, page storage and memory

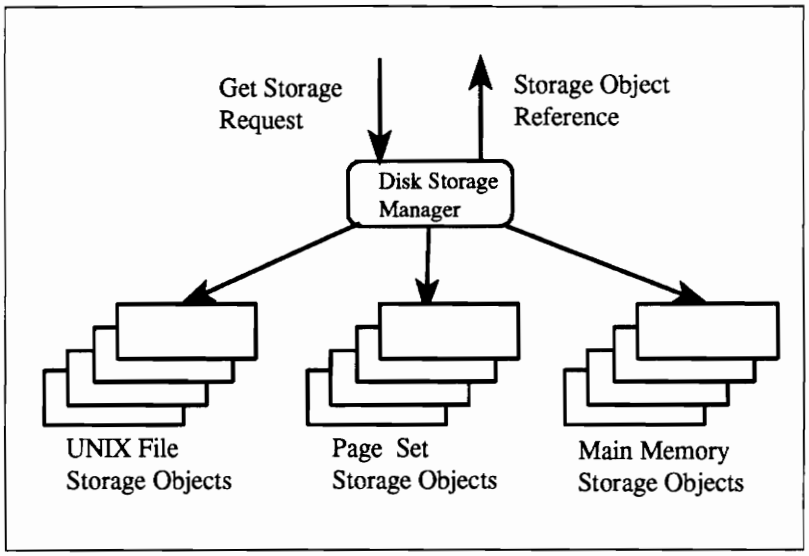


Figure 3.1: Storage Layer Components

storage. UNIX file storage objects are ordinary UNIX files. Page storage objects guarantee that the page is the unit of I/O transfer. Memory storage objects hold all objects in the heap space provided by the C++ run time system.

The major request to the manager is to create a new storage object or to look-up an existing storage object, and to return it to the upper layer (in a form of object reference). Upon a successful creation or look-up, the manager returns the reference to the storage object to the caller. The rest of I/O requests of the caller will be directly against the storage object.

In the following sections, existing classes in the storage layer will be described. Focus will be given to the public interface design and the motivation behind choosing the current implementation over other candidates.

3.1.6 LRU Class

A LRU (least recently used) replacement policy for a group of items is enforced based on the reference history of the items. The class manages a fixed size set of actively accessed items (called a window) and a set of inactively accessed items that can be arbitrarily large.

```
class lru : public replace_policy {

public:
  lru(lru*& r_front, lru*& r_back);
  ~lru();

// promote() moves the cell itself to the front of the window.
  lru* promote(lru*& w_front, lru*& w_back,
               lru*& rest_front, lru*& rest_back,
               int& window_sz, int max_wd_sz);
};

class lru_handler  // lru collection handler. It manages
 {                 // the window(active) and the inactive lists.

public:
  lru* window_front = 0;  // window front and back pointer
  lru* window_back  = 0;
  lru* rest_front = 0;       // inactive list front and back pointer
  lru* rest_back  = 0;
  int window_sz = 0;         // current window size
  int max_window_sz = 0; // max window size

  lru_handler(int  init_max_window_sz) ;
  ~lru_handler() {};
};
```

Figure 3.2: LRU Policy Class Profile

A newly referenced item will be added to the active set and has highest priority not to be moved away. If the active set is full at the time of insertion, the least recently accessed item will be purged. The item is stored back in the inactive set. Sometime, it is useful to remember those purged (or inactive) items, as they can be reintroduced to the window as they are referenced.

The LRU policy is implemented using a doubly linked list data structure to allow constant manipulation time. One list is used for the active items and another for the inactive items. The class profile of the items and the policy are shown in Figure 3.2. Note that the `LRU` class only defines the cells of the doubly-linked lists. Actual items are bound to the cells using class derivation.

3.1.7 Buffer Class

Buffers in LEND hold binary strings. The class profile is illustrated in Figure 3.3. Compared with the `streambuf` class defined by Stroustrup in [Str86], the LEND buffer class encapsulates all fence pointers: the base or left pointer `base`, the right pointer `eptr` and the content start pointer `aptr`. The base pointer indicates the start of the allocated buffer area; the left pointer indicates the left boundary of the content in the buffer; and the right pointer indicates the right boundary. A number of member functions (`get()` and `put()`) are provided to write or read the buffer for instances of different C++ primitive types. An option parameter specifies the format (binary or ASCII) of binary strings. The polymorphic `put` and `get` functions promote safe input and output, as C++ selects the right copy of each function with respect to the type of the argument. These functions are defined `inline` whenever possible to assure good performance. Also, they directly return underflow or overflow status values as appropriate for simplicity. The `underflow()` and `overflow()` functions defined in [Str86] can be added when more flexible error handling is desired.

3.1.8 Page Class

The page class (see Figure 3.4) defines a special case of the buffer class. A page object is a fixed size memory chunk that stores the following items: a count, a string pointer array and a set of binary strings. The count indicates the number of strings in the page; each pointer in the array points to a string. The count takes the first few bytes of the chunk; the pointer array follows. Binary strings are stored at the other end of the chunk and grow toward the pointer array. To refer to a string, it is necessary to present the page number and the index of the pointer within the pointer array. A page object is useful to access a set of binary objects where the reference locality property is likely to hold, since a page is fetched as an entity from the disk.

Figure 3.4 also demonstrates a derived page class that is combined with an `LRU` class. The combined class defines a page frame. A set of such frames will be managed by the LRU

```
enum buf_option { BINARY, ASCII };

class buffer {

protected:
  int bufsz = 0;     // allocated chunk size
  char *base = 0;    // base address of the chunk
  char *eptr = 0;    // end of the buf content
  char *aptr = 0;    // beginning of the buf content

public:
  buffer(int = LBUFSIZ);
  ~buffer();

// buffer management functions
  virtual void reset() { eptr = aptr = base; }; // reset the buffer to empty
  int expand(int);                              // expand buffer chunk

// buffer status functions
  int buf_sz() const { return bufsz; };            // memory chunk length
  int content_sz() const { return eptr - base; }; // content length

// get functions
  int get(char& y);                                    // get a char
  int get(unsigned short& y, buf_option = BINARY);  // get a short
  int get(unsigned int& y, buf_option = BINARY);    // get an int
  int get(char*, int) ;                                // get a string

// put functions
  int put(char) ;                                   // put a char
  int put(unsigned short, buf_option = BINARY) ;  // put a short
  int put(unsigned int, buf_option = BINARY) ;    // put an int
  int put(char*, int) ;                             // put a string

// show the buffer
  friend ostream& operator <<(ostream&, buffer&);
};
```

Figure 3.3: Buffer Class Profile

```
enum page_status_t { CLEAN, WRITTEN };

class page : private buffer
{
protected:
  int page_num = -1;   // page number
  page_status_t status  = CLEAN;

public:
  page(int buf_sz = PAGSIZ);
  virtual ~page() ;

// get and put functions
  int get( Pix i, buffer& buf) const;  // get ith byte string from page and save it in buf.
  Pix put( buffer& buf );              // put string in the buf into the page. The index of
                                       // the string pointer is returned.

  int count() const;    // return number of objects on the page
  int page_num();       // return the page number

// iteration functions
  Pix first() const;    // go over all binary strings on the page
  void next(Pix&) const;

// IO function and friend class
  friend ostream& operator<<(ostream&, page&);
};

class lru_page : private lru, public page
{
public:
  lru_page(int buf_sz = PAGSIZ);
  virtual ~lru_page() ;
};
```

Figure 3.4: Page Class Profile

policy.

3.1.9 Abstract Storage Class

This class defines a buffer and a set of common member function signatures for the UNIX file, page set and memory storage subclasses to inherit. The profile of the class is shown in Figure 3.5. Note `readString()`, `writeString()` and `appendString()` are virtual and abstract. They should be defined for each subclass. Note also that `readString()` has two versions: one that uses the abs_storage class' internal buffer and the other that uses a buffer provided by the caller. The second version allows callers to use their own buffers. Spreading buffers among callers is useful when these callers perform concurrent activity.

```
class abs_storage {

protected:
   int _status; // status of the storage.
   buffer buf;  // buffer used by the storage.

public:
  abs_storage(int init_buf_sz = LBUFSIZ);  // constructor. buffer size should be provided.
  virtual ~abs_storage() ;                 // destructor.

// IO functions
  virtual int readString (Pix& loc, buffer*&, int len) = 0; // read using storage buffer.
  virtual int readString (Pix& loc, buffer&, int len) = 0;  // read using caller's buffer.
  virtual int writeString(Pix& loc, buffer&) = 0;           // write.
  virtual int appendString(buffer&) = 0;                    // append.

// buffer request
  buffer& get_buf(int i) ;       // get storage's buffer.
  void reset() ;                 // clear storage buffer.

// status functions
  int OK() const ;                          // storage status check.
  virtual int bytes() = 0;                  // total bytes in the storage.
  virtual const char* myNickName() = 0;  // name of the storage.
};
```

Figure 3.5: Abstract Storage Class Profile

Direct control over buffering is of particular importance in connection with large objects, such as for multimedia applications.

`get_buffer()` exports the storage buffer and expands the memory chunk when necessary. Other objects can fill the buffer and return it for actual I/O operations. The storage buffer is never deleted and only undergoes expansion by copying when a larger buffer area is required.

3.1.10 UNIX File Storage Class

This class manages storage objects that can be accessed as ordinary UNIX files. In designing this class, two situations that may cause extra overhead are considered. The first is that the UNIX system imposes an upper bound on the number of files that can be opened at the same time by a process. For example, under Ultrix 4.1 on a DECstation model 3100, $n = 60$, and on a NeXTstation, $n = 252$. Since file opening is costly compared to readings and writings, UNIX file storage objects are managed under a LRU policy. The value of n is

used as the active set size. Files recently accessed are kept open while those least referenced are more likely to be closed.

The second situation concerns whether the buffer area should be directly exposed to the object layer. Both advantages and disadvantages exist. In general, exporting buffers eliminates buffer copy but weakens the buffer protection, while hiding buffers achieves the opposite effects. LEND adopts a solution favoring buffer exportation while at the same time providing a reasonable degree of buffer protection.

3.1.11 Page Set Storage Class

This class manages a set of pages stored on disk. The storage contains a set of page frames in which pages can be cached. Currently, cached pages are managed using the LRU policy. Other management policies are possible. To access a binary string through `readString` and `writeString`, it is necessary to know its page number and the pointer index within the page. These two quantities are packed in the location argument `loc` of the three functions. A UNIX file storage object is used to physically hold all pages in the set. The UNIX file is expanded if more pages are requested.

Page storage makes it possible to implement a client/server architecture where pages are the units of transfer, such as that in the O_2 system [Deu91]. Such an architecture minimizes the number of I/Os when the content of a page is useful, like a page of text.

3.1.12 Technical Comparison

Kotteman, Gordan and Stott in [KGS91] describe a storage and access manager (called James) for ill-structured data such as semantic networks, production rules and frames, with key objectives "to maximize the flexibility" and "to simplify and to unify navigation".

The James model provides tokens and groups of tokens. A token stores a data entry, while a token group bundles a set of tokens together. Additional linking mechanisms are available to connect tokens storing identical data entries together. To implement such a conceptual model, James maintains a set of token declaration entries, each storing a pointer

to a token (a variable length string). Token groups are modeled by consecutively stored token declarative entries to facilitate rapid access. The linking mechanism is implemented by a doubly linked list. Thus, each token declaration entry owns an additional two pointers for the doubly linked list. The variable length strings are kept on a heap space. A B* tree is used to index all variable length strings. The advantage of the model lies in its simplicity. The disadvantage comes from the heavy use of pointers (3 pointers per token). Also, the model limits the clustering of tokens to the group level and does not provide indexing support within token groups.

Compared with James, LEND's storage layer provides much more functionality and flexibility. More consideration is given to performance in LEND's storage layer. For example, the page storage allows a semantic network to be embedded onto a set of pages. With proper clustering, a page contains a node object as well as likely the surroundering neighbors. LEND is also designed with the goal of efficient space usage in mind. For example, arcs with the same names can be abstracted into an arc class so that the arc name can be remembered by the class and not physically stored.

The indexing structure of LEND is also different from that of James. While James primarily relies on a fixed set of data structures (token declarations, B* trees and doubly linked lists), LEND provides indexing hierarchies and a set of hashing functions. With the former, a B* tree can be easily added, and with the latter optimal access performance is guaranteed.

Moss in [Mos90] reports a Mneme store model designed for cooperative, information-intensive tasks commonly observed in CAD, CASE and office automation environment. Its design concepts include objects, files, pools, and strategies.

A Mneme object is a byte string, including an object identifier (`oid`), several tagging bits and a size count. In addition, the embedded `oid` list within the object is enumerable by the store. Mneme files support a modularized object space and are the units of backup, recovery, garbage collection and transfer. Objects reside in files and their `oids` are localized to these files. Cross-file reference through `oid` involves forwarding protocol

routines. Mneme handlers are the means of indirect reference to objects. They reside in memory. Once established, a handler contains the `oid`, the object size and a pointer to the data. Lastly, Mneme pools are logical partitions of objects in a Mneme file for management convenience. Mneme strategies are enforced by management routines handling tasks such as object clustering, prefetch, buffering, caching and storage allocation. These routines are invoked through strategy numbers associated with pools.

Mneme is intended to support persistent programming activities; LEND's storage layer is designed for information retrieval applications. Although both are organized as libraries, LEND takes advantage of C++ to allow more succinct modeling of storage layer entities. LEND has the `storage` hierarchy notion. With careful subclassing, normal files on different operating systems can become subclasses in the hierarchy and can be used to store LEND objects. Mneme's strategy routines combined with its pool concept are equivalent to LEND's notion of page frames managed under certain policies (say, LRU). These policies can conceptually form hierarchies themselves, such as the page replacement policy hierarchy, or page prefetch policy hierarchy. LEND treats static graph clustering more as a computational issue, detailed in Chapter 5.

LEND also differs from Mneme in the format of objects stored. First, in the storage layer, objects are represented as byte strings. The storage layer neither imposes tags nor prefixes attributes to the string. The `oid` of a normal LEND object will occupy the first few bytes of the string. Value identifiers are never externally stored because reference through the value identifier is not required. Second, the size count is not stored if the size can be inferred from the type of the object while reading. The choice is made when the object class is written. Third, `oid` lists encoding reference relationships are not embedded in a node object in LEND. Rather, relations are described by arc classes, though the arc objects can be stored close to the node objects, and the clustering algorithm (described in Chapter 5) maximizes such cases. Storing arcs separately from nodes provides the flexibility of adding more types of arcs to the object graph without adjusting the class definition of the node objects. In information retrieval, this is crucial to add a new retrieval model to the existing

node objects, such as document and concept representation nodes.

3.2 Object Layer

This layer is composed of classes that can be used as primitive classes for direct application modeling, or as base classes for deriving subclasses when specific modeling needs arise. At run time, this layer accepts object identifiers or key objects (i.e., objects with internal state serving as keys), transforms them to locations of the desired binary strings, issues requests to the storage layer for reading in binary strings, initializes objects from the binary strings (i.e., restores the object's internal value and associates the value with member functions), and finally delivers objects to the upper layer. The key task is to activate objects from their binary string formats. In a typical object-oriented database system, this may include format conversion, enforcement of encapsulation and association of methods. This layer also caches initialized objects.

The increasingly popular C++ is chosen as the implementation language and also the language in which classes are defined or derived. As an implementation language, C++ brings the benefit of ease of learning of LEND, fine interleaving of applications (in C or C++) with database calls, computational power (see [LLOW91]) and an open system architecture. As a class definition language, C++ makes it easy to realize the constructs of the LEND model, including object identifiers, attributes, encapsulation, methods, classes and the class hierarchy.

Section 3.2.1 describes the structure of a typical object manager and the architecture of the overall object layer. Section 3.2.2 explains the design and implementation of some important classes defined in the base and graph components of the LEND model. To facilitate discussion, G is used to denote a directed graph in which an IR entity is encoded. The node set of G is $N(G)$ and the arc (edge) set of G is $E(G)$.

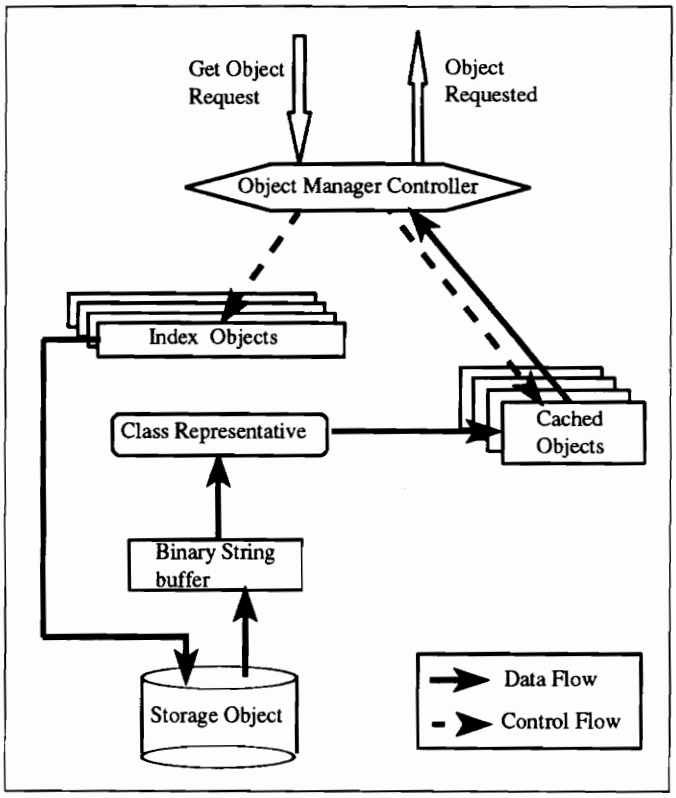


Figure 3.6: Object Manager Components

3.2.1 Object Manager and Object Layer Architecture

Figure 3.6 shows the structure of an object manager. An object manager manages all object instances of a class (whole set) or a composite object in LEND. Specifically, it is responsible for restoring objects from binary strings, caching the restored objects, and looking up cached objects. The manager contains a storage object, a set of index objects, a class representative and an object cache.

The storage object associated with the manager stores objects in the binary string format. All objects managed by the manager are stored in the storage object. The storage object also provides I/O services for binary strings which may reside in the main memory or on the disk. The storage object is acquired by calling the storage object manager in LEND's lower layer. Note that the same storage object may be shared by several object

managers. This allows objects of different classes to be clustered.

The index objects associated with the object manager help it quickly locate the binary strings. In particular, the index object transforms object identifiers to locations. If the object identifier itself is represented as a physical address, an identity mapping index object can be employed. To allow fast access to an index object, the object manager arranges all index objects in an indexed array. The numerical index of a particular index object depends on the class of objects the manager manages.

The class representative is a normal object belonging to a particular class. Its responsibility is to restore other objects in the class from binary strings, by using two standard methods **from_string** and **copy** of the representative. The semantics of the method **from_string** is to restore the object itself, using the binary string parameter. The semantics of the **copy** method is to make a copy of itself. The process of initialization can be described as follows. First, the **from_string** method of the class representative is called to initialize the representative. Next the **copy** method is called to duplicate the representative. This duplicated copy is the object desired by the upper layer.

The activation method is simple, safe and extensible. Assuming the **copy** and **from_string** methods are well written for the class, the restored object contains state that was converted to the binary string. Also, method attachment and encapsulation enforcement are automatically achieved by C++. Whenever a new class is added, as long as those two member functions are defined, code in the object layer does not need to change. Another advantage of this approach is that the size information can be procedurally remembered by the class representative, if the size of all objects of the class is equal. In such a case, the size count is not required to be physically stored.

Having described the structure of an object manager, we next discuss the overall architecture of the object layer, as shown in Figure 3.7. The object layer maintains a vector of object managers for all classes and composite objects existing in the database. Each object manager is responsible for initializing objects of a particular class or a composite object. To locate an object manager, an INDEX is used as shown in the upper part of the figure.

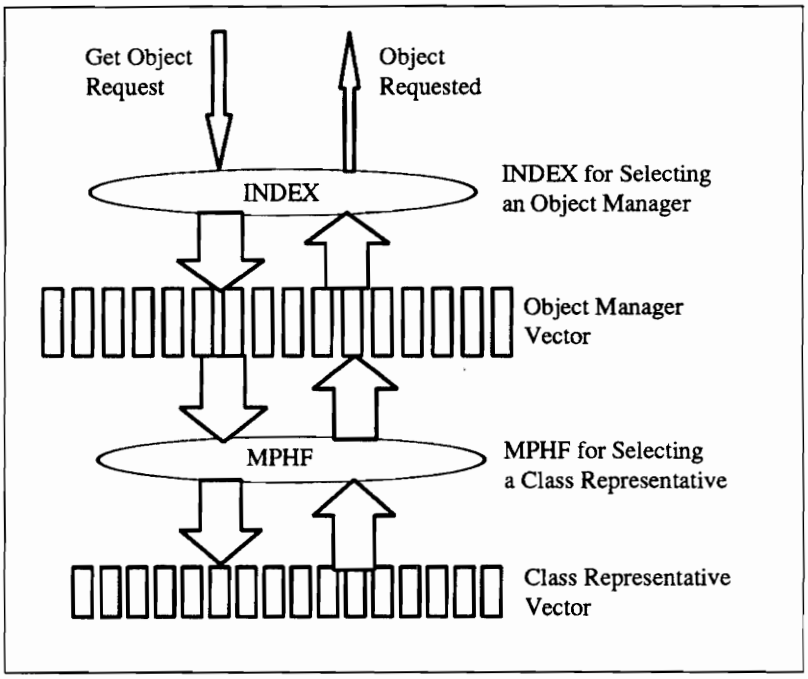


Figure 3.7: Object Layer Architecture

This INDEX maps the identifier of the class or the composite object to the desired object manager. This INDEX should be able to handle insertion and deletion of object managers.

More than one object manager can use representatives belonging to one class. To reduce duplication, similar class representatives can be abstracted out of the managers. An object vector is introduced for the purpose. The vector stores a unique copy of the class representative for a class. A MPHF (see Chapter 4) is utilized to quickly locate the class representative, given its class identifier.

In the following, we illustrate how the object layer functions. Assume that there are three classes currently in LEND: a tuple class T, a set class S and an object identifier class OID. An OID object contains an object identifier. Class T has 3 members: t_1, t_2 and t_3. Class S has two members: s_1, and s_2. The content of s_1 is $\{t_1\_oid, t_2\_oid\}$, where t_i_oid is an object of class OID, and contains the identifier of t_i, $i = 1, 2$. There are three object managers: OM_T, OM_S and OM_{OID}, corresponding to class T, S and OID respectively.

There also are three class representatives: CR_T, CR_S and CR_{OID} for the three classes. Suppose the first request is to retrieve t_2_*oid* from set s_1, and the second is to fetch t_2 using t_2_*oid*.

To accomplish the first task, we identify object manager OM_S because it is capable of locating s_1, either from the storage object or from its object cache. Suppose s_1 is not in the cache. Then we fetch the binary string encoding s_1 from the object storage. CR_S is then initialized using the string. Later, CR_S copies itself to restore the s_1 object. s_1 will be cached in OM_S's object cache. An object manager then will be established for s_1, as s_1 is a composite object.

To retrieve t_2_*oid* from s_1, we present t_2_*oid* as a query object to s_1. s_1 will use some index to locate t_2_*oid*, and fetch the binary string from the location. CR_{OID} will be activated to generate an OID object, say x. We then compare x with t_2_*oid*. Suppose they are identical. We then proceed to cache x in s_1's object cache.

The final task is to access object t_2. This is easy because we can present t_2_*oid* as a query object to OM_T. OM_T will use some index to locate a binary string encoding value of t_2. CR_T will be used to produce the exact copy of t_2, and t_2 finally will be returned.

3.2.2 Important Classes

Root Class

This class serves as the base class for all other LEND classes. It defines the LEND object identifier `oid` and a set of public method signatures that other LEND classes make use of. The `oid` encodes information about whether the identifier is physical or logical. A physical identifier is one that gives the physical location of the object, while a logical identifier is a symbolic location. Frequently, an index converts the logical identifier to the physical identifier. The value part `value` stores the value of the object or the minimal information necessary to obtain the complete value. The location of an object is also saved in the `oid` portion. The `status_tag` encodes various status information, such as the status of the

```
class root
 {
protected:
  oid_t oid;            // id of the root
  value_t value;        // value of the root
  status_t status_tag   // tag for the root

public:
  root(int init_option= 0);
  root(object&) ;
  virtual ~root();

// id and value comparison functions
  virtual Boolean id_EQ(root&) const;
  virtual Boolean value_LE(root&) const;
  virtual Boolean value_EQ(root&) const;

// import and export functions
  virtual int toString(buffer&, buf_option = BINARY) ;     // from buffer
  virtual int fromString(buffer&, buf_option = BINARY) ;   // to buffer

// copy function
  virtual root& copy() const;
  virtual root& reference() const;

// status inquiry functions
  Boolean valueObject() const ;                          // value object?
  virtual Boolean OK() const ;                           // in good shape ?
  virtual const char* myClassName() const;               // class name
  virtual const char** myParentClassNames() const;       // parent class names
  virtual const class_code_t class_code() const;         // class code of the object
  virtual const instance_code_t instance_code() const;   // instance code of the object
};
```

Figure 3.8: Root Class Profile

object and whether the object is a reference copy of another object. The profile of the root class is shown in Figure 3.8.

In the current implementation, the `oid` part consists of an `unsigned short` typed instance code and a pointer to the class description. The description, containing an `int` typed class code, the class name and the next available instance code, is shared by all objects in the class. On most UNIX machines, this configuration gives rise to a 16 bit long class code and a 32 bit long instance code. The total number of classes is 2^{16} (=65,536) and each class can allow $2^{32} - 1$ (= 4294967295) instances. The object address is of `Pix` type. `Pix` is a class generalizing the physical location notion.

The "next available instance code" field records the next available instance code. Since a class description is shared by all objects belonging to a class, maintenance of this variable

is easy.

In the following paragraphs, we discuss the public member functions.

`Constructor and Destructor`: This group includes two constructors and a destructor. The first constructor takes an optional initialization argument to signify whether the object is born normally or is a value. A value object has similar representation (value part) and behavior as an ordinary object, except that the uniqueness of its instance code is not enforced among all value objects of the same class. Value objects can be used as values, such as to form non-sharable, encapsulated constituents of composite objects. The default value (0) indicates that a value object is desired.

`Comparison`: This group contains four comparison functions: one for identifier comparison (`id_EQ`) and two for value comparison (`value_LE` and `value_EQ`). The semantics of these functions has been discussed in Chapter 2.

`Export and Import`: This group involves two functions `to_string()` and `from_string()`. The first writes the object into a binary string while the second initializes the object from a binary string. In both cases, a `buffer` object is utilized to hold the binary string.

`Copy and Reference`: This group enables an object to be duplicated or shared through the use of functions `copy()` or `reference()`. The first function makes a copy of the object, while the second returns a reference to the object. These functions are useful in writing generic code for index and set classes. In using the two functions, it should be kept in mind that costly `copy()` duplicates the object, while inexpensive `reference()` returns a reference to the object. The delete operation can be applied to the reference object. The value portion of the original object is not removed.

`Object Status`: Functions in this group are used to reveal the status of the object. Common functions include `OK()`, `myClassName()`, `myParentClassNames()`, `c_code()` and `i_code()`. The first, when returning a Boolean `TRUE`, indicates that the object is fine

and can take member function calls. The second returns the class name while the third returns a set of parent class names. The last two return the class and instance codes of the object, respectively.

`Class-specific Functions`: This group contains all functions that are specific to the class. Since they are not part of the virtual functions used by the system, they can take any form. Their implementation depends entirely on the individual class.

Primitive Class

Primitive classes include **`integer`**, **`real`** and **`string`**. Primitive objects serve as the components of other composite objects or directly as the value part of nodes in $N(G)$. Internally, these objects do not contain any logical or physical pointers to encode arcs in $E(G)$.

There are two reasons behind including primitive types in the LEND hierarchy. The first is that C++ treats **`int`**, **`float`** and **`char string`** as built-in primitive types. They do not participate in any class hierarchy. The second is that the object manager and index class contain generic code working only for objects belonging to the LEND class hierarchy. Generic code is compact in size and easy to maintain.

Composite Class

The profile of the class is shown in Figure 3.9. Since there should be no instances associated with the class, this class is an **`abstract`** class. Note that the iteration control variable is defined using the **`Pix`** class rather than as an integer. The argument option determines whether a copy or a reference to a component is returned.

Set Class

The set class forms the basis for object grouping and permits set-oriented querying. The profile of the class is listed in Figure 3.10.

```
enum get_component_option_t { COPY, REFERENCE };

 class composite: public root
{

public:
  composite(int init_option = 0) ;
  virtual ~composite() ;

  virtual Pix first() = 0;
  virtual root& get_component(Pix, get_component_option_t = REFERENCE) = 0 ;
  virtual  void next(Pix&) = 0;

  virtual int count() = 0;
};
```

Figure 3.9: Composite Class Profile

```
enum residency_option_t {  MEMORY, DISK };
enum transiency_option_t  { TRANSIENT, PERSISTENT };

class  set : public composite
{

public:
  set(residency_option_t = MEMORY,
      transiency_option_t = TRANSIENT,
      root& bound_object);
  ~set();

// retrieval functions
  virtual PixAVLSet& get_pixes(int  index_position,  root& query_object) ;

// update functions
  virtual Pix insert(root&) ;
  virtual int  delete(Pix) ;

// iteration functions
  virtual Pix first() ;
  virtual const root& get_component(Pix, get_component_t = REFERENCE) ;
  virtual void next(Pix&) ;

// status inquiry functions
  virtual int count() const ;

// status change functions
  void to_persistent();
  void to_transient();

// I/O function
  friend ostream& operator <<(ostream& s, abs_set&) ;
};
```

Figure 3.10: Set Class Profile

Upon a successful initialization, the set can be subjected to both iteration and set queries. The iteration can be performed by using the `first`, `get_component` and `next` operations. Set queries can be carried out in two steps. First, use the `get_pixes()` function to obtain a set of `Pixes`. The function takes two arguments. The first is an integer k indicating which index object in the index array of the set element object manager should be used. The second is the query object containing a query value. The returned `Pixes` contain the location of objects that match with the query object. The next step is to use `get_component` to fetch individual matching objects by scanning over the `Pix` set.

Tuple Class

This class provides a general way to describe composite objects that have a fixed number of components. The tuple value is implemented by two arrays: the name array and the root object pointer array. The former is used to remember the attribute names while the latter is used to remember attribute values. The pointer implementation for tuple values makes it possible to accept normal as well as value objects as attribute values.

List Class

This class implements lists. An easy implementation exists which makes use of the set class. The ith element e of the list can be encoded by the tuple $< index : i, list\_element : e >$.

Data Access Classes

Data access classes contains objects to quickly locate object. These objects fall into three categories: hash function, hashing disk indexing and AVL memory indexing. Contrary to typical object-oriented database systems, the data access component in LEND is more closed to the object layer than to the storage layer. This is because most index classes contain generic member functions. They take LEND objects as keys. Generic index code is compact and easy to maintain.

The hash functions are organized into a hierarchy for different hash functions. MPHF, OPMPHF and PBH are all included. The major function of a hash object is to map a key to an integer. To save space, key sets are not remembered in the hash objects. False drops will happen when a key not in the original key set is presented to the hash function. Validation should be performed by reading the actual objects.

The index objects map a key to a set of locations, where objects can be read. Index objects can be memory based and disk based. The current implementation includes three hash function data structures: MPHF, OPMPHF, PBHF (the details of these data structures will be covered in Chapter 4). These hash functions map keys into integers. A data structure realizing integer to location set mapping is then utilized.

The AVL index class uses an AVL tree to index memory resident objects. The tree always stays in memory. Ordinary AVL trees keep keys inside tree nodes. To LEND, this means that the degree of encapsulation of objects indexed will suffer, as keys will be exported. Another two disadvantages are that keys will be duplicated in AVL trees, and different AVL trees handling different types of keys may have to be provided. The problem has a nice solution by using C++'s dynamic binding capability, which can select the right copy of a member function to invoke at run time.

Based on this, LEND stores object pointers instead of keys inside the AVL trees. When the AVL tree makes comparisons, it follows the pointers and, with the help of dynamic binding, chooses the right copies of comparison functions. The approach preserves the encapsulation for the objects indexed, eliminates key duplication, and requires only one AVL tree class. Since both the AVL tree and objects indexed reside in memory, this approach will not impair much the performance of tree lookup, as only one additional memory resident pointer is chased for each comparison.

The technique is not suitable for disk based search trees, as an extra I/O is required to read in the key in each search step.

Directed Arc Class

This class provides directed arc objects useful for the purposes of association or connection. The basic memory representation of an arc is composed of three object identifiers: for the arc object itself, for the source object, and for the destination. An additional one can be added for the label.

A directed arc has two disk formats: stored as a single arc object (`format`$_1$); or stored as two pointers associated with the source and the sink node objects (`format`$_2$). The former facilitates the maintenance of relationship integrity while the latter allows efficient graph traversals.

When stored in `format`$_1$ and when the page set storage object is used, an arc object contains the page number of both the source and destination node objects, and two marks indicating whether all similarly typed arcs reside on the same page with respect to the sink and the sink node. Chapter 5 details the usage of the page numbers and marks.

If stored in `format`$_2$, the page numbers of the adjacent node objects are kept with the two pointers. If all pointers of a node object can't fit into a page, they are separated onto several pages. The exact encoding is maintained by a pointer list object.

3.3 Application Layer

The layer provides interfaces with applications. Four intended applications are the LEND query language, the F^{3}L high-level process communication language, SNePS' access request for semantic networks resident on disk, and C++ programs. The LEND query language (specified in section 2.5) allows inquiries on node objects as well as connections among node objects. The C++ interface is the simplest as LEND system can be used as a database library. The remainder of this section is devoted to the processing of SNePS' data requests and the LEND query language.

3.3.1 SNePS Operator Process

This section discusses the possible implementation of some important SNePS operators by using services available from the object layer. Path-based inference operators listed in Table 2.3 are of interest. In the following, we go over each operator. The focus is on the data structure and related algorithms. Recall that a SNePS path is modeled as a tuple with two attributes. The first is the label sequence (a list) and the second is the set of LEND paths. Each path in the set bears the label sequence. Let sp be a SNePS path. We denote $sp.x$ as the label sequence, $sp.y$ as the LEND path set. The number of labels in the label sequence is $|sp.x|$ and the number of LEND paths in $sp.y$ is $|sp.y|$

Converse Operator

The `converse` operator is defined for both SNePS arcs and SNePS paths. We discuss the arc version first.

In SNePS, an arc coexists with its converse (i.e., its twin). The `converse` operation maps an arc to its twin. To save space, the twin can be physically stored once. Let *phy_arc* be the arc actually stored. Then the representation of the SNePS twin arcs only need contain the identifier of *phy_arc* and the direction information. Another useful trick about the representation deals with the SNePS arc identifier. Since `converse` yields an arc that already exists, it is advantageous to know the identifier of the resulting arc without making a database lookup. This can be easily achieved by dividing the instance code space of the SNePS arc class into two parts, and using a bit in the code to record the subspace of the arc. By setting or resetting the bit, the `converse` operator can be executed in $O(1)$ time.

The path version of the `converse` operator is a little bit complicated. There are two implementation choices: either to materialize (materialization approach) the resulting path set or to maintain the set virtually (view approach). The former makes a new, separated path set while the latter maintains a pointer to the sibling path set. If the path set is virtually stored, deletion of the original path set affects the virtual one. To avoid dangling

of the virtual path set, the deletion operation should include a sibling path set checking procedure. Similar to what was discussed for SNePS arcs, bit tagging can be used to obtain a resulting path's identifier quickly.

If the path set is managed virtually, finding the converse of a SNePS path takes $O(1)$ time for book-keeping.

Kstar Operator

Let *sp* be a SNePS path. The *Kstar* on *sp* generates a set of paths, each the concatenation of zero or more LEND paths owned by *sp*. The problem is the well-known transitive closure problem. In graph theoretic terms, the problem calls for the computation of all reachable nodes from a set of nodes in a directed graph G. Graph-based algorithms exist for the problem. The computation involves first coalescing nodes in strongly connected components, then topologically sorting the condensed acyclic graph, and finally adding to a node the successor sets of its immediate successors, in reverse sorting order. See [AJ90]. The time complexity is $O(|N(G)||E(G)|)$.

Since the graph is stored on disk, it is critical to know those nodes that have been visited in the computation. The optimal case is to keep the information resident in memory. A hash table indexed by a MPHF is useful in achieving the goal. Within such a hash table, an entry contains a bit related to a node in the graph. The two states of the bit record the visiting status of the node. Assume the faster MPHF algorithm (see section 4.3) is used and it produces a 3 bit/key MPHF, then a total of 4 bits suffices to encode a node. For a 10 million node graph, the space usage is about 4.7 MB, which can fit in the main memory of a modern workstation.

Kplus Operator

Let *sp* be a path. The *Kstar* on *sp* generates a set of paths each of which is the concatenation of one or more instances of *sp*. *Kplus* is a variant to the *Kstar* operator. Let $A(x)$ be the answer to operator x. Then the answer to *Kplus* is the set difference: $A(Kstar)$ - S.

AND Operator

This operator takes a set S of SNePS paths and returns a new SNePS path sp. A LEND path p that connects node u to v is in $sp.y$ iff all SNePS paths in S connect u to v.

Analogous to `Compose`, this operator can be implemented in two ways. The materialization approach involves multiple set intersections, where each set corresponds to a SNePS path, and contains pairs of source and sink nodes that the SNePS path connects.

The view approach takes $O(|S|)$ for book-keeping work.

OR Operator

This operator takes a set S of SNePS paths and returns a new SNePS path sp. A LEND path p that connects node u to v is in $sp.y$ iff some SNePS path in S connects u to v.

Analogous to `Compose`, this operator can be implemented in two ways. The materialization approach involves multiple set unions, where each set corresponding to a SNePS path contains pairs of source and sink nodes that the SNePS path connects.

The view approach takes $O(|S|)$ for book-keeping work.

3.3.2 Query Language Process

As the query language is expected to be used for browsing through G, it was decided that an interpreter is capable of processing the language in a timely fashion. The choice of an interpreter does not mean to sacrifice the performance. Instead, it is felt that establishment of suitable indexes, and proper clustering of storage for G, will make the processing easy and fast. A compiler version can be added later.

Compared with relational and other object-oriented database queries, the LEND query language has three distinct features which affect its processing. First, it uses graph concepts. Components of a graph such as paths and arcs are explicitly expressible using the language. Arcs are stored in the database and paths may be materialized as well. In either case, indexes can be established. As such, finding a best strategy to assure the existence of a

path with a certain property can be reduced to the selection of a proper index, similar to the strategy used in the ObjectStore system [LLOW91], as opposed to the expensive optimization process for the equivalent join problem found in a relational query.

Second, in contrast to typical object-oriented databases where objects are embedded within each other [LLOW91, Kim90], LEND expresses the embedding relationships through explicit semantic arcs. "Nodes" and "arcs" in the query language have corresponding node objects and arc objects in the database. They can be physically stored to facilitate either graph traversal access or selections on node or arc sets. It is important for the interpreter to know different storage structures and to take advantage of their properties and statistics. The storage structure information can be easily obtained from set objects through member functions calls. Given that information, it is the interpreter's job to figure out the set of pages to be fetched and in what order. If buffer space is limited, the interpreter should consider choice of the page replacement policy. The proposed work of Lavinus [Lav91] ties in with this search strategy selection problem.

Lastly, the query language allows the definition of variables which may hold temporary query results, and permits their use to formulate other queries. The introduction of variables is necessary as they allow a complicated query to be formed incrementally, which will increase the flexibility of the query language. To the interpreter, these variables add a certain amount of burden since answers are needed for questions such as: where to save temporary results and whether indexes should be set up.

In the following, we describe the initial query processing work. The query interpreter is divided into three components: the lexical parser, the syntax parser and the executor. The combination of the first two components generates a parse tree. Parse tree nodes are C++ constructs allowing initially rapid prototyping and later extension to other language constructs. Each query construct category such as `path` and `nodes` has a corresponding tree node class. Each tree node of these classes owns a pointer to the value it stands for and a member function `exec()`. The executor examines and reshapes the parse tree for the purpose of efficient execution. Finally, the member function `exec()` of the root node of the

tree is invoked. It recursively calls the overloaded `exec()` member functions of the child nodes to finish the execution job.

3.4 Summary

In this chapter, the architecture and various components of the LEND system are described. The LEND system follows the normal database reference model and is organized into three layers: storage, object and application. Within each layer, class hierarchies are used to model various system components. The result is an open system where new components such as data structures or algorithms can be easily added.

The design meets the needs of efficient storage and access of graph structured data. This is illustrated by the page storage class, the LRU replacement policy and the node and arc classes. The processing of SNePS operators and the pattern language is also discussed.

Chapter 4

HASHING METHODS

This chapter describes a set of algorithms developed for finding good minimal perfect hash (MPHF) functions for information retrieval applications. These constructed functions require a small amount of space for their specification and can produce a hash address with effort proportional to the length of the key presented. One reference to the hash table by using the hash address can definitely answer the question: does the record with that key exist? The proposed algorithms are able to process very large key sets (e.g., over 3 million) with a practical amount of computation time. The MPHFs produced can be used as data indexing aids.

The chapter begins with a review in section 4.1 of state-of-the-art approaches to finding minimal perfect hash fucntions. It then points out that the key elements in successfully identifying MPHFs are the search space and time tradeoff.

Section 4.2 describes a bipartite graph MPHF finding algorithm. The mapping, ordering and searching framework is presented in detail, which serves as the basis for other algorithms covered in the chapter. Section 4.3 reports on a new MPHF finding algorithm. It shows that a slightly changed mapping step can yield smaller MPHFs while still using a similar amount of searching effort. The time can also be cut down significantly if pattern matching heuristics are used. Section 4.4 explains the use of MPHF methods for `trie` data structure

compaction. The quality of the compaction is an e-approximation [HS78] where $e = 2$.

Section 4.5 reveals the extension of normal MPHF algorithms for finding order preserving MPHFs (OPMPHFs). OPMPHFs are ideal for searches where it is necessary to maintain the order of keys. A general space lower bound is derived for this kind of function. The rest of the section details an OPMPHF finding algorithm extending the bipartite graph algorithm. Some analytic results are included to justify the selection of the searching space.

The last section in the chapter (i.e., section 4.6) reports another extension of MPHF algorithms for finding perfect bin hash functions (PBHFs), where a bin may hold more than one key. PBHFs are useful for page or block oriented access.

4.1 Introduction

4.1.1 Hashing

Let S be a collection of records each of which has a (unique) associated *key*, say k, selected from U, a (usually finite) universe of keys. The cardinality of U is $N = |U|$. While some researchers assume that U is the set of integers

$$U = \{1 \ldots N\},$$

U in this chapter is considered more realistically to be the set of character strings having some finite maximum length. Clearly this is appropriate for keys that are words or names in any natural or artificial language, or that are elements of some descriptor or identifier set, possibly involving phrases as well, or ASCII representations of objects of various data types.

The actual set of keys used in a particular database at a fixed point in time is $S \subset U$ where typically $|S| \ll |U|$. The cardinality of S is $n = |S|$. Records are stored in a *hash table* T having $m \geq n$ locations (or *slots*), indexed by elements of $Z_m = \{0, 1, \ldots, m-1\}$. The utilization of space in T is measured by the *load factor*, $\alpha = n/m$. Depending on the application, T may be in primary memory, magnetic disk, optical disc, or recorded on some other device; in all cases it is desirable for T to be as small as possible and for us to be able to quickly find the appropriate slot in T for any given key k.

The retrieval problem is to locate the record corresponding to a key $k \in U$ or to report that no such record exists. This is done by *hashing*, i.e., applying a *hash function* h that is computable in time proportional to the size of the key k, and examining the slot in T with address $h(k)$. If there are two keys $k_1, k_2 \in S$ such that $h(k_1) = h(k_2)$, then there is a *collision* of k_1 and k_2. Much effort is expended in traditional work [AHU85] on hashing in resolving collisions. Collisions force more than one *probe* (reading a slot) of T to access some keys. If h is a 1-1 function when restricted to S, h is called a *perfect hash function* (PHF) since there is no need to waste time resolving collisions. A PHF h allows retrieval

of records (objects) keyed from S in one access, which is clearly optimal in terms of time. For any form of hashing, optimal space is attained when the hash table is fully loaded, i.e., when $\alpha = 1$; this is called a *minimal hash function.* The best situation, then, is to have a *minimal perfect hash function* (MPHF) where $\alpha = 1$ and there are no collisions, so h is a 1-1 onto mapping when restricted to S.

Hashing has been a topic of study for many years, both in regard to practical methods and analytical investigations [Knu73a]. Recently there has been renewed interest in hashing due to the development of techniques suitable for dynamic collections [ED88]. A less extensive literature has grown up, mostly during the last decade, dealing with perfect hash functions; that subarea is studied in this chapter.

4.1.2 Mapping to Integers

Given a key k from a static key set S of cardinality n, selected from a universe of keys U with cardinality N, the objective is to find a function h that maps each k to a distinct entry in the hash table T containing m slots. Certainly function h must map each key k to an integer. In the simplest cases, keys are positive integers bounded above by N. This situation was assumed, for example, by Sprugnoli [Spr78], Jaeschke [Jae81], and Fredman, Komlós, and Szemerédi [FKS84].

The more general case where keys are strings (since clearly integers can be represented as strings of digits or strings of bytes, for example) is considered in this chapter. The usual approach where keys are strings is to associate integer values with all or some of the characters in the string, and then to combine those values into a single number. Chang [Cha84] uses four tables based on the first and second letters of the key. Cichelli [Cic80] uses the length of the key and tables based on the first and last letters of the key. Note, however, that the length of a key, its first letter, and its last letter are sometimes insufficient to avoid collisions; consider the case of the words 'woman' and 'women' in Cichelli's method.

Cercone et al. [CKB83] enhance the discriminating power of transformations from strings to integers by generating a number of letter to number tables, one for each let-

ter position. Clearly, if the original keys are distinct, numbers formed by concatenating fixed length integers obtained from these conversion tables will be unique. In practice, it often suffices to simply form the sum or product of the sequence of integers.

While in some schemes (e.g., [Cic80]) the resulting integer is actually the hash address desired, in most algorithms, the h function must further map from the first integer value(s) produced into the hash table.

4.1.3 Existence Proofs

One might ask if a MPHF h for a given key set exists. Jaeschke [Jae81] proves this and indeed presents a scheme guaranteed to find such a function (though his method requires exponential time in the number of identifiers). Assuming that the task is to map a set of distinct positive integers $\{k_1, k_2, \ldots, k_n\}$ bounded above by N without collisions into the set of m indices of T, then an array of N entries suffices, as demonstrated in [FCHD89]. However, the load factor is low since usually $N \gg n$.

Another way to view this situation is to realize that perfect hash functions are rare in the set of all functions. Knuth [Knu73a] observes that only one in 10 million functions is a perfect hash function for mapping the 31 most frequently used English words into 41 addresses. The task then can be viewed as one of searching for rare functions, and of specifying them in a reasonable amount of space.

4.1.4 Space to Store PHF

Using an argument due to Mehlhorn [Meh82] (see also [Mai83]), it is shown in [FCHD89] that an approximate lower bound of ≈ 1.4427 bits per key is needed to represent an arbitrary MPHF. It is important to realize that for arbitrary key sets this cost is *required* for one-probe access, no matter what scheme is employed. In addition to the lower bound, Mehlhorn also gives a method of constructing MPHFs of size $O(n)$ bits. However, the construction requires exponential time and therefore is not practical.

It is typical to state the size of PHFs in terms of the number of computer words used,

each of $\log_2 n$ bits. Then Mehlhorn's lower bound is $\Omega(n/\log_2 n)$ computer words, and his practical upper bound is $O(n \log_2 n/\log_2 n) = O(n)$ computer words. The initial algorithm (section 4.2) achieves MPHF size of less than $O(n)$ computer words, and the newest algorithm (section 4.3) approaches the theoretical lower bound.

4.1.5 Classes of Functions

There are several general strategies for finding perfect hash functions. The simplest one is to select a class of functions that is likely to include a number of perfect hash functions, and then to search for a MPHF in that class by assigning different values to each of the parameters characterizing the class.

Carter and Wegman [CW79] introduced the idea of a class H of functions that are $universal_2$, i.e., where no pair of distinct keys collide very often. By random selection from H, one can select candidate functions and expect that a hash function having a small number of collisions can be found quickly. This technique has also been applied to dynamic hashing by Ramakrishna and Larson [RL89].

Sprugnoli [Spr78] proposes two classes of functions, one with two and the other with four parameters, that each may yield a MPHF; searching the parameter values of either class is feasible only for very small key sets. Jaeschke [Jae81] suggests a reciprocal hashing scheme with three parameters that is guaranteed to find a MPHF, but it is only practical when $n \leq 20$. Chang [Cha86] proposes a method with only one parameter, though its value is likely to be very large, and requires a function that assigns a distinct prime to each key; however, he gives no algorithm for that function, so the method is only of theoretical interest. A practical algorithm finding perfect hash functions for fairly large key sets is described in [CHK85]. They illustrate the tradeoffs between time and size of the hash function, but do not give tight bounds on total time to find PHFs or experimental details for very large key sets.

The above-mentioned 'search-only' methods may (if general enough, and if enough time is allotted) directly yield a perfect hash function when the right assignment of parameters

$$\boxed{MAPPING \rightarrow ORDERING \rightarrow SEARCHING}$$

Figure 4.1: Method to Find Perfect Hash Functions

is identified. However, analysis of the lower bounds on the size of a suitable MPHF suggests that if parameter values are not to be virtually unbounded, then there must be a moderate number of parameters to assign. However, in the algorithms of Cichelli [Cic80] and of Cercone et al. [CKB83] two important ideas can be discovered: using tables of values as the parameters, and using a mapping, ordering, and searching (MOS) approach (see Figure 4.1). While their tables seem to be too small to handle very large key sets, in the general case, tables are useful and the MOS approach is also an important contribution to the field of perfect hashing.

In the MOS approach, the construction of a MPHF is accomplished in three steps. First, the Mapping step transforms the key set from the original universe to a new universe. Second, the Ordering step places the keys in a sequential order that determines the order in which hash values are assigned to keys. The Ordering step may partition the key set, yielding an ordering of subsets of consecutive keys. Each subset is called a *level*, and the keys of each level must be assigned their hash values at the same time. Third, the **Searching** step attempts to assign hash values to the keys of each level. If the Searching step encounters a level that it is unable to accommodate, it backtracks, sometimes just to an earlier level, assigns new hash values to the keys of that level, and tries again to assign hash values to later levels. Sager's method is a good example of the MOS approach.

4.1.6 Sager's Method and Improvement

Sager in [Sag84, Sag85] proposes a formalization and extension of Cichelli's approach. Like Cichelli, he assumes that a key is a character string. In the Mapping step, three auxiliary hash functions are defined on the original universe of keys U

$$h_0 : \quad U \rightarrow \{0, \ldots, m-1\}$$

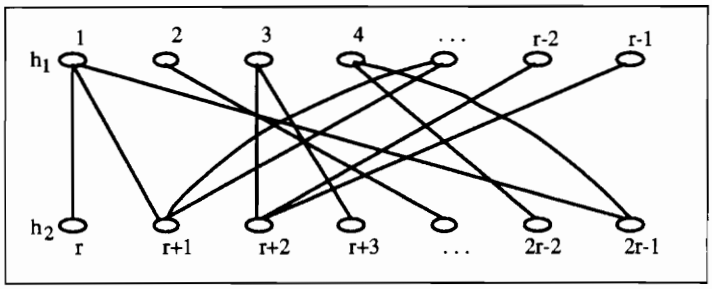


Figure 4.2: Dependency Graph

$$h_1 : \ U \rightarrow \{0, \ldots, r-1\}$$

$$h_2 : \ U \rightarrow \{r, \ldots, 2r-1\}$$

where r is a parameter (typically $\leq m/2$) that ultimately determines how much space it takes to store the perfect hash function (i.e., $|h| = 2r$). These auxiliary functions compress each key k into a unique identifier

$$(h_0(k), h_1(k), h_2(k))$$

which is a triple of integers in a new universe of size mr^2. The class of functions searched is

$$h(k) = \Big(h_0(k) + g(h_1(k)) + g(h_2(k))\Big) \pmod{m}$$

where g is the function whose values are selected during the search.

Sager studies a graph that represents the constraints among keys. Indeed, the Mapping step goes from keys to triples to a special bipartite graph, the *dependency graph*, whose vertices are the $h_1()$ and $h_2()$ values and whose edges represent the words. The two parts of the dependency graph are the vertex set $\{0, \ldots, r-1\}$ and the vertex set $\{r, \ldots, 2r-1\}$. For each key k, there is an edge connecting $h_1(k)$ and $h_2(k)$; that edge carries the label k. See Figure 4.2.

In the Ordering step, Sager employs an ordering heuristic based on finding short cycles in the graph. This heuristic is called *mincycle*. At each iteration of the Ordering step, the mincycle heuristic finds a set of unselected edges in the dependency graph that occur in as

many small cycles as possible. The set of keys corresponding to the set of edges constitutes the next level in the ordering.

There is no proof given that a minimum perfect hash function can be found, but mincycle is very successful on sets of a few hundred keys. Mincycle takes $O(m^4)$ time and $O(m^3)$ space, while the subsequent Searching step usually takes only $O(m)$ time. The time and space required to implement mincycle are the primary barriers to using Sager's approach on larger sets.

Sager chooses values for r that are proportional to m. A typical value is $r = m/2$. In the case of minimal perfect hashing ($m = n$), it requires $2r = n$ computer words of $\log_2 n$ bits each to represent g. This is somewhat more than Mehlhorn's lower bound of $1.4427n/\log_2 n$ computer words. The *ratio* $(2r/n)$ must be reduced as low as possible, certainly below 1. After further investigation a modified $O(m^3)$ algorithm [FCHD89] was developed. With this algorithm it is possible to find MPHF's for sets of over a thousand words.

4.1.7 Summary of Related Work

Hashing has been used in many applications and, with recent development of dynamic hashing techniques, has witnessed a resurgence of interest [ED88]. However, most dynamic hashing involves low load values, on the order of 0.6 to 0.9, and requires resolution of collisions. In most dynamic hashing schemes, hash addresses identify buckets or bins wherein a number of records can be stored, and which are usually only partially full. The work of Gonnet and Larson does improve upon the typical approach, however, allowing high load factors, through the use of a small amount of extra storage used in buckets [GL88].

An examination of previous schemes for perfect hashing shows that many are generally only applicable to small sets, or require a prohibitive amount of space to store. Some approaches require input keys to already be integers in some restricted range, while others are really two level searches where extra storage in each bucket supports the second level of search. An example is the work of Brain and Tharp, which extends Cichelli's approach but still cannot handle very large key sets [BT90].

The most competitive alternative approach comes from the work of Fredman, Komlós, and Szemérdi [FKS84] and others who have built on their work (e.g., [SS89]). These methods assume a universe that is the set of integers $\{1, \ldots, N\}$ in contrast to the assumption of keys that are arbitrary character strings. Membership queries are accommodated in constant time, and the hash function requires $n + o(n)$ space. Construction has random expected time $O(n)$. The algorithms described in the following two subsections have tighter bounds, and are able to work on very large key sets.

4.2 Bipartite Graph MPHF Finding Algorithm

This section reports the first MPHF finding algorithm using a bipartite graph as working domain. The work has been reported in [FHCD92] and [FCHD90].

4.2.1 Key Concepts of the Algorithm

There are three insights due to Lenwood Heath [FCHD90] allowing the new algorithm to run in linear expected time.

1. Randomness should be exploited whenever possible. A small probability of failure is traded for an outstanding average case performance.

2. The vertex degree distribution is highly skewed; this can be exploited to carry out the Ordering step in a much more efficient manner.

3. Assigning g values to a set of related words can be viewed as trying to fit a pattern into a partially filled disk, where it is important to enter large patterns while the disk is only partially full.

Since Mapping and Searching have expected time $O(n)$, the time complexity of our method is $O(n)$ in the expected sense. Because **minimal** perfect hash functions are the primary concern, it is assumed that $m = n$, and so n alone is used. All results easily generalize to the case $m \geq n$.

Randomness

The first use of randomness is in the Mapping step where a good set of triples is obtained to serve as identifiers for the original word strings. To assure the distinctness of triples, a random number for each key (mod n) is obtained by making use of all of the information in the key to give maximum discrimination. The pseudo-random number generator selected, to allow machine independence and to ensure good behavior, is due to Park and Miller

[PM88a]. The random integers generated by their random number generator are between 0 and $2^{31}-2$. Now consider the use of these random numbers to obtain the desired triples. First, three tables ($table_0$, $table_1$, $table_2$) of random numbers are constructed, one for each of the functions h_0, h_1, and h_2. Each table contains one random number for each possible character at each position i in the key. Let a key be the character string $k = k_1 k_2 \ldots k_y$. Then the triple is computed using the following formulas:

$$\begin{aligned} h_0(k) &= \left(\sum_{i=1}^{y} table_{0i}(k_i)\right) \bmod n \\ h_1(k) &= \left(\sum_{i=1}^{y} table_{1i}(k_i)\right) \bmod r \\ h_2(k) &= \left(\sum_{i=1}^{y} table_{2i}(k_i)\right) \bmod r + r. \end{aligned}$$

Assuming the triples $(h_0(k), h_1(k), h_2(k)), k \in S$, are random, it is possible to derive the probability that the triples are distinct. Let $t = nr^2$ be the size of the universe of triples. The probability of distinctness for n triples chosen uniformly at random from t triples is

$$p(n,t) = \frac{t(t-1)\cdots(t-n+1)}{t^n} = \frac{(t)_n}{t^n}.$$

By an asymptotic estimate from Palmer [Pal85],

$$p(n,t) = \frac{(t)_n}{n} \sim \exp\left\{-\frac{n^2}{2t} - \frac{n^3}{6t^2}\right\}$$

and for typical values of $r = cn/\log_2 n$, where c is a constant,

$$p(n,t) \approx \exp\left\{-\frac{n^2(\log_2 n)^2}{2n(cn)^2}\right\} = \exp\left\{-\frac{(\log_2 n)^2}{2c^2 n}\right\} \approx 1 - \frac{(\log_2 n)^2}{2c^2 n}$$

so that $p(n,t)$ goes to 1 quite rapidly with n. A suitable set of triples can be found almost always the first time, and even with very good likehood of success when h_0, h_1, and h_2 are all determined using permutations (different views) of one random table.

Vertex Degree Distribution

The second key concept is that the distribution of degrees of vertices in the dependency graph is decidedly skewed, and indeed has a number of interesting properties. In particular,

it can be shown that in a random dependency graph most vertices have low degree, which leads to small levels in the ordering. For a particular vertex v and the edges incident on it, the probability that a particular edge from edge set E is incident on v is $p = 1/r$. Let X be the random variable that equals the degree of v. Then (see [Fel68]) X is binomially distributed with parameters n and p. Since the case of large n is of interest, the Poisson approximation to the binomial distribution applies:

$$\Pr(X = d) \approx \frac{e^{-\lambda}\lambda^d}{d!}$$

where $\lambda = np = n/r$. From the Poisson approximation, the expected number of vertices of degree d is given by

$$2r\Pr(X = d) \approx \frac{2re^{-n/r}\left(\frac{n}{r}\right)^d}{d!}$$

When $r = n/2$ ($\lambda = 2$), the expected number of vertices of degree 0, 1, 2, 3, and 4 are approximately 0.27r, 0.54r, 0.54r, 0.36r, and 0.18r, respectively. The skewed distribution of vertex degrees provides the inspiration for a new Ordering heuristic. Instead of ordering the *edges* (keys) of the dependency graph as mincycle does, the new heuristic orders the *vertices*. Let $v_1, v_2, \ldots, v_{2r}$ be any ordering of the vertices of the dependency graph. For each v_i, there is a set of edges $K(v_i)$ that go from v_i to vertices earlier in the ordering

$$K(v_i) = \{(v_i, v_j) \in E | j < i\}.$$

This set of edges is also a set of keys, and every key occurs in exactly one $K(v_i)$. $K(v_i)$ may be empty, but cannot be larger than the degree of v_i (it is often smaller). The ordering of the set of keys into levels is just the ordering of the nonempty $K(v_i)$.

As discussed in the next section, it is desirable to have levels that are as small as possible and to have all large levels early in the ordering. This suggests that a vertex of larger degree should be processed earlier than a vertex of smaller degree. This also suggests that a vertex whose K set is (currently) larger should be chosen next in the ordering over a vertex whose K set is (currently) smaller. Finally, it is imperative that the Ordering heuristic be able

to choose the next vertex in the ordering quickly and simply. The new Ordering heuristic evolved from these insights gained by examining the skewed distribution of vertex degrees and from the imperative of choosing the next vertex quickly.

Fitting into a Disk

At each iteration of the Searching step, one level of the ordering is to be placed in the hash table. Each level is the key set $K(v_i)$ corresponding to a vertex v_i. For purposes of illustration, assume that $v_i \in \{r, \ldots, 2r-1\}$. Each key $k \in K(v_i)$ has the same h_2 value $h_2(k) = v_i$ and, therefore, will have the same $g \circ h_2$ value $g(h_2(k)) = g(v_i)$. By assumption, the $g \circ h_1$ value of k is already selected. Since all h_0 values are already defined, $h(k)$ is determined by the selection of $g(v_i)$. Consider the sum of the two values already known for k. Let

$$b(k) = h_0(k) + g(h_1(k)).$$

Then

$$h(k) = (b(k) + g(v_i)) \pmod{n}.$$

The $b(k)$ values for all keys $k \in K(v_i)$ yield offsets from $g(v_i) \pmod{n}$ to the hash values of the keys.

The set of $b(k)$ values constitutes a *pattern* (mod n). The pattern may be viewed as being in a circle of n slots and subject to *rotation* by the amount $g(v_i)$. The hash table is viewed as a disk with n slots, some of which may already be filled. To successfully assign hash values to the keys in $K(v_i)$, the Searching step must determine an offset value $g(v_i)$ that puts all the $b(k) + g(v_i)$ values in empty slots of the hash table *simultaneously*. This process we refer to as fitting a pattern into a disk.

Finding hash values for a set of j related words corresponds to finding suitable g values so that the pattern of size j can be placed into the disk, with each of the j words fitting into an empty slot. Clearly, when $j = 1$ this is possible as long as there is an empty slot.

Further, regardless of the size of j, it is always possible to fit a pattern into an empty table. We would expect, therefore, to be able to find a hash function if vertices of large degree are handled when the disk is mostly empty, and if when the table is starting to get full, remaining vertices are of low degree, preferably degree 1. In [FCHD89] it is derived that the probability of fitting a pattern of size j into a m slots disk with f slots occupied already is:

$$\Pr(\text{fit}) \quad = \quad 1 - e^{-\mu}$$

where

$$\mu = \left(\left(1 - \frac{f}{m} \right)^{j-1} (m - f) \right).$$

Note that if f is a function of m such that $f < (1 - \epsilon)m$, for some constant $\epsilon > 0$, then $\mu \to \infty$ and $\Pr(\text{fit}) \to 1$. For our purposes, this means that the disk must be slightly less than full ($f < (1 - \epsilon)m$) when the last pattern of size $j > 1$ is placed.

4.2.2 Algorithm

The bipartite graph algorithm is an extension of the work by Sager [Sag85] and builds on the new insights described in earlier sections. To aid in subsequent discussion, the terminology introduced by Sager and later extended is summarized in Table 4.1.

The algorithm for selecting h has three steps: Mapping, Ordering, and Searching.

The Mapping Step

The Mapping step takes a set of n keys and produces the three auxiliary hash functions h_0, h_1, and h_2 (see section 4.2.1). These three functions map each key k into a triple

$$(h_0(k), h_1(k), h_2(k)).$$

Because the ultimate MPHF must distinguish any two of the original keys, it is essential that these n triples be distinct. As discussed in section 4.2.1, if h_0, h_1, and h_2 are random

Table 4.1: Summary of Terminology

U =	universe of keys
N =	cardinality of U
k =	key for data record
S =	subset of U, set of keys in use
n =	cardinality of S
T =	hash ordering, with slots numbered $0, \ldots, (m-1)$
m =	number of slots in T
h =	function to map key k into hash ordering T
$\|h\|$ =	space to store hash function
r =	parameter specifying the number of vertices in one part of the dependency graph
h_0, h_1, h_2 =	three separate random functions easily computable over keys
g =	function mapping $0, \ldots, (2r-1)$ into $0, \ldots, (m-1)$
t =	number of levels in the ordering

functions, it is very likely that the triples will be distinct. The h_1 and h_2 values are used to build a bipartite graph called the *dependency graph.*

Half of the vertices of the dependency graph correspond to the h_1 values and are labeled $0, \ldots, r-1$. The other half of the vertices correspond to the h_2 values and are labeled $r, \ldots, 2r-1$. There is one edge in the dependency graph for each key in the original set of keys. A key k corresponds to an edge labeled k between the vertex labeled $h_1(k)$ and the vertex labeled $h_2(k)$. Notice that there may be other edges between $h_1(k)$ and $h_2(k)$, but those edges are labeled with keys other than k. If the value $h_0(k)$ is associated with the edge k, then all the information that the Ordering and Searching steps need to construct a MPHF is present in the dependency graph.

Two data structures describe the dependency graph, one for the edges (keys) and one for the vertices (h_1 and h_2 values). Both are implemented as arrays. The **vertex** array is

```
vertex:  array [0..2r-1] of record
    firstedge:  integer;
    degree:  integer;
```

```
    g:  integer;
    end
```

firstedge is the header for a singly-linked list of the edges incident on the **vertex**. **degree** is the number of vertices incident on the vertex. **g** is the g value for the **vertex**, which is assigned in the Searching step. The **edge** array is

```
edge:  array [1..n] of record
    h0, h1, h2: integer;
    nextedge1: integer;
    nextedge2: integer;
    end
```

$\mathbf{h_0}$, $\mathbf{h_1}$, and $\mathbf{h_2}$ contain the h_0, h_1, and h_2 values for the **edge** (key). Also, $\mathbf{nextedge_i}$, for side i $(= 1, 2)$ of the graph (corresponding to $\mathbf{h_1}$, $\mathbf{h_2}$, respectively), points to the next edge in the linked list whose head is given by **firstedge** in the **vertex** array.

Figure 4.3 details the Mapping step. Let $\mathbf{k_1}, \mathbf{k_2}, \ldots, \mathbf{k_n}$ be the set of keys. The h_0, h_1, and h_2 functions are selected (1) as the result of building tables of random numbers as described in section 4.2.1. The construction of the dependency graph in (2) and (3) is straightforward. In (3) when edges are added to the appropriate linked list, values for $\mathbf{nextedge_i}$ in the **edge** array are updated as needed. (4) examines sets of edges having the same $\mathbf{h_1}$ value to check for distinct ($\mathbf{h_0}$, $\mathbf{h_1}$, $\mathbf{h_2}$) triples; since vertex degrees are small, (4) takes expected time that is linear in n. In the rare circumstance that distinct triples are not produced (5), new random tables are generated, defining new $\mathbf{h_0}$, $\mathbf{h_1}$, and $\mathbf{h_2}$ functions. The probability that random tables must be generated more than twice is exceedingly small. Therefore, the expected time for the Mapping step is $O(n)$.

The Ordering Step

The Ordering step explores the dependency graph so as to partition the set of keys into a sequence of levels. The step actually produces an ordering of the vertices of the dependency

(1) build random tables for h_0, h_1, and h_2
(2) **for** each $v \in [0 \ldots 2r-1]$ **do**
    `vertex[v].firstedge` $= 0$
    `vertex[v].degree` $= 0$
(3) **for** each $i \in [1 \ldots n]$ **do**
    `edge[i].`h_0 $=$ $h_0(k_i)$
    `edge[i].`h_1 $=$ $h_1(k_i)$
    `edge[i].`h_2 $=$ $h_2(k_i)$
    `edge[i].nextedge`$_1 = 0$
    add `edge[i]` to linked list with header `vertex[`$h_1(k_i)$`].firstedge`
    increment `vertex[`$h_1(k_i)$`].degree`
    `edge[i].nextedge`$_2 = 0$
    add `edge[i]` to linked list with header `vertex[`$h_2(k_i)$`].firstedge`
    increment `vertex[`$h_2(k_i)$`].degree`
(4) **for** each $v \in [0 \ldots r-1]$ **do**
    check that all edges in linked list `vertex[v].firstedge` have
        distinct (h_0, h_1, h_2) triples.
(5) **if** triples not distinct **then**
    repeat from step (1).

Figure 4.3: The Mapping Step

graph (at least those that do not have degree zero). From the vertex ordering, the sequence of levels is easily derived. If the vertex ordering is $v_1, \ldots, v_t$, then the level of keys $K(v_i)$ corresponding to a vertex v_i, $1 \leq i \leq t$, is the set of edges incident both to v_i and to a vertex earlier in the ordering. More formally, if $0 \leq v_i \leq r-1$, then let $x = 1$, $y = 2$, while if $r \leq v_i \leq 2r-1$, then let $x = 2$, $y = 1$.

$$K(v_i) = \{k_j | h_x(k_j) = v_i, h_y(k_j) = v_s,\ s < i\}.$$

The rationale for the vertex ordering is discussed in section 4.2.1.

An analogy with Prim's algorithm [Sed88] for constructing a minimum spanning tree will help illuminate the heuristic for ordering vertices. At each iteration of Prim's algorithm, an edge is added to the minimum spanning tree that is lowest in cost such that one endpoint of the edge is in the tree and the other endpoint is not. Of course, when an edge is added to the tree, so is a vertex. One implementation of Prim's algorithm maintains the unexamined edges that have at least one endpoint in the tree in a heap so that the cheapest edge can always be selected in logarithmic time.

The ordering heuristic initiates the ordering with a vertex v_1 of maximal degree. At each iteration of the Ordering step, a previously unselected vertex v_i is added to the ordering. v_i is selected from among those unselected vertices that are adjacent to at least one of $v_1, \ldots, v_{i-1}$; from among these vertices, v_i is selected to have maximal degree. If there are no such unselected vertices and there remain unselected vertices of nonzero degree (i.e., another connected component needs to be processed), then select any vertex of maximal degree to be v_i. The algorithm maintains the unselected vertices that are adjacent to selected vertices in a heap **VHEAP** ordered by degree. Figure 4.4 gives the Ordering step.

The heap operations are **initialize** (start an empty heap), **insert** (add a vertex to the heap), and **deletemax** (select a vertex of maximum degree and remove it from the heap). Each heap operation can be accomplished in $O(\log n)$ time (since $r = O(n)$). Because the vertex degrees of a random dependency graph are (mostly) small, there is an optimization possible to speed the heap operation. In this optimization, **VHEAP** is implemented as a series

```
initialize(VHEAP)
v1 = a vertex of maximum degree
mark v1 SELECTED
for each w adjacent to v1 do
    insert(w, VHEAP)
i = 2
while some vertex of nonzero degree is not SELECTED do
    while VHEAP is not empty do vi = deletemax(VHEAP)
        mark vi SELECTED
        for w adjacent to vi do
            if w is not SELECTED and w is not in VHEAP then
                insert(w, VHEAP)
        i = i + 1
```

Figure 4.4: The Ordering Step

of stacks and one bounded size heap. Most vertices have degree 1, 2, 3, or 4. One stack is provided for each degree, so that the size of the heap is kept below a constant. Usually, the heap VOHEAP contains all vertices of degree ≥ 5. When a vertex w is to be added to VHEAP, the degree of w is checked. If the degree is ≤ 4, then w is added to the appropriate stack; otherwise, w is inserted in VOHEAP. All list operations take constant time. The time for the Ordering step is thus actually linear.

There is one issue not addressed in Figure 4.4: the dependency graph may not be connected. Typically, the dependency graph consists of one large connected component and a number of smaller components. By choosing v_1 as a vertex of maximal degree, the algorithm is almost certainly choosing v_1 in the large component. Therefore, the algorithm selects the large component first. After that, it must process the remaining components in the same fashion. The algorithm maintains a list of those vertices not yet selected and can easily find an unselected vertex of maximal degree. Therefore, the Ordering step is able to order all vertices of degree > 0.

In another implementation, the Ordering step is redefined to take advantage of the cycle

structure of the graph [FCHD89]. In each component, a maximal subtree is identified that attaches to the remainder of the component through a single cut point. It is easy to verify that each edge of such a subtree can always be made to appear in a level of size 1. These subtrees can be identified in linear time and placed at the end of the ordering. This improves the probability of success for the Searching step.

The Searching Step

The Searching step takes the levels produced in the Ordering step and tries to assign hash values to the keys, a level at a time. Assigning hash values to $K(v_i)$ amounts to assigning a value to $g(v_i)$, as is indicated in section 4.2.1. To this end, a `hash-table` data structure is defined

```
hash-table: array [0..n − 1] of record
    key: integer
    assigned: boolean
    end
```

where **key** is the index to the key that has hash value i, and **assigned** is a flag as to whether the hash value i has been assigned to any key yet.

When a value is to be assigned to **vertex**[i].**g**, there are usually several choices for **vertex**[i].**g** that place all the keys in $K(v_i)$ into unassigned slots in hash-table. The analysis and the empirical results mentioned in section 4.2.1 indicate that an acceptable value for **vertex**[i].**g** should be picked *at random* rather than, for example, picking the *smallest* acceptable value for **vertex**[i].**g**. In looking for a value for **vertex**[i].**g**, the Searching step uses a random probe sequence to access the slots 0, . . . , n − 1 of the hash-table.

Figure 4.5 gives the algorithm for the Searching step. A random probe sequence of length n is chosen in step (3). The probe sequence actually used in the implementation has only a weak claim to randomness; that is, just a small amount of randomness seems sufficient to make a good Searching step. At the beginning of the Searching step, the current

```
(1)     for i ∈ [0...n − 1] do
            hash-table[i].assigned = false
(2)     for i = 1 to t do
(3)         establish a random probe sequence s_0, s_1, ..., s_{n−1} for [0...n − 1]
            j = 0
            do
                collision = false
                if v_i ∈ [0...r − 1] then w = 1 else w = 2
                for each k ∈ K(v_i) do
(4)                 h(k) = edge[k].h_0 + vertex[edge[k].h_{3−w}] + s_j   (mod n)
(5)                     if hash-table[h(k)].assigned then
                            collision = true
(6)             if not collision then
                    for each k ∈ K(v_i) do
                        hash-table[h(k)].assigned = true
                        hash-table[h(k)].key = k
                    else
                    j = j + 1
(7)                 if j > n − 1 then
                        fail
            while collision
```

Figure 4.5: The Searching Step

implementation chooses a set of 20 small primes (or fewer if n is quite small) that do not divide n. Each time (3) is executed, one of the primes q is chosen at random to be s_1 and is used as an increment to obtain the remaining $s_j, j \geq 2$. Thus, the random probe sequence is

$$0, q, 2q, 3q, \ldots, (n-1)q.$$

A more robust random probe sequence would choose the increment q at random from $0, \ldots, n-1$ such that the greatest common division of q and n is 1. As just mentioned, such a robust sequence does not appear to be necessary.

A detail that is omitted from Figure 4.5 is the action taken when the Searching step is unable to insert a level into the hash table (**fail** in (7)). This is such a rare occurrence that

for a large enough value of n and an appropriate choice of r, it is very unlikely to occur even once in the execution of the algorithm. Therefore, one reasonable response to this rare event is to restart the algorithm from the beginning with new random tables for h_0, h_1, and h_2. The current implementation actually uses a simple backtracking scheme. It assigns new g values to earlier vertices and then tries to complete the g function for the entire graph. More sophisticated backtracking schemes are possible, but, as mentioned above, it is unclear whether the added effort is justified.

4.2.3 Experimental Results

To support the claims regarding the theory and practice relating to our approach to MPHF determination, [FCHD89] gives collected distribution data regarding the vertex degrees in dependency graphs and level sizes in the ordering. For consistency, all timings listed below are obtained on a Sequent Symmetry with 32 megabytes of main memory, running on a single (80386) processor running at about 4 MIPS, reporting UNIX "times()" results.

Table 4.2 shows timings (in seconds) for each phase of the algorithm, omitting only time required to double check that an MPHF has been found. Total time is given, along with two metrics for the size of the hash function, when $r = 0.3n$. The words per key value is set at 0.6; the number of stacks used is 12; and the machine is Sequent. First, since less than $O(n)$ words/key is claimed, words/key is fixed at 0.6. Second, since $O(n)$ bits/key is the lower bound, bits/key is used as measurement scale. Note that key set sizes are given as successive powers of 2, from 2^5 through 2^{19}, so it is easy to see the time required is $O(n)$. It also has been found that fewer words/key are needed as the key set size increases, suggesting $< O(n)$ words/key.

In summary, the algorithm processes large key sets well, and while there is some variation in processing time because of the probabilistic nature of the operations, with an appropriate value for r the algorithm finds a MPHF with high probability.

Table 4.2: Timing Results for the Bipartite Graph Algorithm

n	bits/key	Mapping	Ordering	Searching	Total
32	3.0	0.37	0.02	0.03	0.42
64	3.6	0.75	0.03	0.07	0.85
128	4.2	0.60	0.05	0.08	0.73
256	4.8	0.97	0.05	0.18	1.20
512	5.4	1.23	0.08	0.35	1.67
1024	6.0	1.50	0.17	0.67	2.33
2048	6.6	2.42	0.30	1.67	4.38
4096	7.2	3.47	0.62	3.13	7.22
8192	7.8	5.53	1.27	5.92	12.72
16384	8.4	9.87	2.52	12.05	24.43
32768	9.0	18.78	5.05	24.62	43.45
65536	9.6	35.68	10.20	50.02	95.90
131072	10.2	69.70	20.15	101.08	190.93
262144	10.8	137.97	40.30	201.57	379.83
524288	11.4	275.25	81.23	406.58	763.07

Large Key Sets

The largest sets of keys we have been able to handle directly with the initial algorithm are a collection of over 420,878 French words and a set of $2^{19} = 524,288$ names taken from the Online Computer Library Center (OCLC) catalog, since our data structures have been tuned to require a maximum of $9n$ computer words. The Sequent machine has sufficient primary memory for this to work well. For the French words, the graph used has $r = .25n$, so $|h|$ required 0.5 words/key. The Ordering step was modified to use 12 stacks, one for each vertex degree between 1 and 12. This modification led to a fast Ordering time of 53 seconds, while the time for the Searching step was 608 seconds. The total time, including the Mapping step, for the algorithm to find a MPHF for the 420,878 words was 812 seconds. For the OCLC keys, total time was 763 seconds.

For very large key sets, the primary limitation on the efficiency of the algorithm comes from the size of main memory. If little of the dependency graph can fit in the main memory of a virtual memory machine, then swapping occurs with a large proportion of the references

to the dependency graph. This is because the graph is a random one and, therefore, violates the Principle of Locality.

To accommodate very large key sets, the dependency graph is partitioned into disconnected subgraphs of manageable size. With the modified implementation, a MPHF for a key set consisting of 1.2 million words was constructed on the Sequent Symmetry. The dependency graph was partitioned into 6 subgraphs of approximately equal size. In one run $r = .30n$, so $|h|$ required 0.60 words/key. The total time to construct the MPHF was 4804 seconds.

4.2.4 Summary

This section gives theoretical and experimental validation of the bipartite graph algorithm for finding minimal perfect hash functions, suitable for key sets ranging in size from small to very large (over a million). The beauty of the algorithm lies in that:

- A bipartite graph is used as an encoding vehicle, with which the concepts such as levels of keys and the ordering of levels are developed;
- Any pattern set can be guaranteed to contain distinct members as the g values in earlier levels can be properly chosen to avoid member duplications in the later levels;
- Random fit of patterns is superior to Sager's incremental fit. It makes it possible to compute the probability of success of the searching step and hence the entire algorithm.

4.3 A Faster MPHF Finding Algorithm

The bipartite graph algorithm described in Section 4.2 makes use of the dependency graph where vertices, generated in the mapping step, are the range of the h_1 and h_2 functions, and edges represent keys. The ordering step heuristically arranges the vertices in the dependency graph to get a vertex sequence. The subsets of keys induced by the sequence are fit into the hash table in the searching step. The algorithm is the first known capable of computing MPHFs for very large sets while keeping the specification space of the computed MPHFs within the bound of $\log n$ bits per key.

A limitation of the algorithm also derives from the ordering heuristics in that large subsets may fit into the table later than some smaller ones. It seems impossible to generate a sorted level sequence simply by walking through nodes of the bipartite graph. This causes an increase in the difficulty of fitting when a small number of subsets - hence a small function - is intended. Another drawback of the algorithm is that each vertex contributes $\log n$ bits to the MPHF specification. For vertices associated with small key subsets appearing earlier in the sequence, which have little information content, this is a large space overhead.

Another algorithm described in [FCHD90] uses a simple random function to obtain the subsets of integers. To order subsets of integers, one sorts the subsets in decreasing order by size. The heuristic works because larger subsets are more difficult to fit than the smaller ones into a partially fitted table, and in general the emptier the table the easier it is to fit a subset. Experiments show that the heuristic works well when the integers are binomially distributed into a sufficient number of subsets. The MPHF specification space of the approach can go below 3 to 4 bits per key. However, it is still not clear whether the binomial distribution of subsets is the best choice. Also a large searching effort is required as the tentative hash address of a key is computed from a random number generator, which is inefficient when the hash table is almost full.

4.3.1 Algorithm Description

In this section, a faster MPHF finding algorithm is described. It is able to compute a MPHF that can be specified with 2 to 3 bits per key. This is achieved by exploiting a deliberately skewed distribution with respect to the key subset sizes in the mapping stage. Key addresses are obtained by rotating a set of random numbers called a pattern. The rotation allows an acceleration method to be employed to cut down the searching effort. As a result, the new algorithm takes less time to find, and less space to specify a MPHF than the method of section 4.2. Thus, the algorithm can be considered better. Pearson's method [Pea90] is extended for generating random numbers from key strings. The new method provides good random numbers. As a result, 256 bytes is enough to specify our random number tables in this algorithm. This contrasts to $384l$ bytes used in the bipartite graph algorithm, where l is the maximal length of the key in the key set. The specific MOS framework underlying the algorithm is presented in the following section.

The MOS Framework

The basic approach in [FCHD90] is to treat the problem as a search for desired functions in a large search space s. In actuality, preparatory Mapping and Ordering steps are needed so that fast Searching can take place. The overall scheme is illustrated in Figure 4.6. Mapping transforms the problem of hashing keys into a different problem, in a different space. Ordering paves the way for searching in that new space, so that locations can be identified in the hash table. Hashing then involves mapping from keys into the new space, and using the results of searching to find the proper hash table location. From that perspective, the key results in the earlier [FCHD90] and this algorithm are as follows.

- Search space s requires at least $1.4427n$ specification bits (at least $2^{1.4427n}$ distinct values must be in the space).
- Finding an MPHF is a search problem that determines the proper value in s for an instance S (which is the key set).

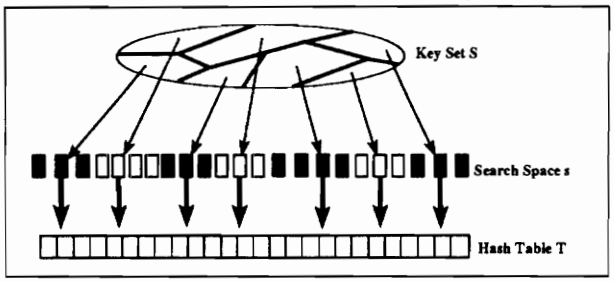


Figure 4.6: Illustration of the Key Concepts

- S is related to s through partitioning both S and s into subsets S_i and s_i, for $i = 0, 1, 2, \ldots$

In the rest of this section, we discuss the Mapping, Ordering and Searching steps.

The Mapping stage accomplishes several important goals. First, the n keys must be mapped to integer values, in the range $0 \ldots n-1$. This is done by pseudo-random hash function h_{10} which will almost certainly map several keys into the same address, and leave other addresses without any keys.

$$h_{10} \quad : \quad S \rightarrow \{0, \ldots, n-1\}$$

See Figure 4.7 for an illustration of the process.

Second, we wish to shrink the range of integer values from n to b so that later we need only search for b values. Finding an MPHF which has specification size close to the lower bound can be accomplished when b is roughly $2n/\log_2 n$. We can accomplish this by composing h_{10} with another function that will map into the range $\{0 \ldots b-1\}$.

However, in the process we can, if we are clever, accomplish a third goal. In particular, we wish to separate the keys into two major groupings. Our second function, then, is really accomplished by two functions that operate upon disjoint portions of $\{0 \ldots n-1\}$.

$$h_{11} \quad : \quad \{0, \ldots, p_1 - 1\} \rightarrow \{0, \ldots, p_2 - 1\}$$

$$h_{12} \quad : \quad \{p_1, \ldots, n-1\} \rightarrow \{p_2, \ldots, b-1\}$$

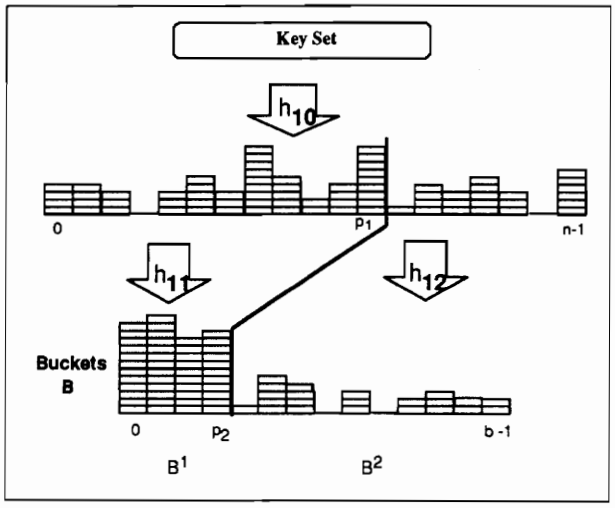


Figure 4.7: Mapping Stage

These, together with h_{10}, accomplish the mapping from keys to buckets.

$$bucket(k) = \begin{cases} h_{10} \circ h_{11} & \text{if } h_{10}(k) < p_1 \\ h_{10} \circ h_{12} & \text{otherwise} \end{cases}$$

Thus, the mapping function $bucket(k)$ is composed of three functions illustrated in Figure 4.7: h_{10} randomly distributes keys into an auxiliary integer set $\{0, n-1\}$, while h_{11} and h_{12} in turn randomly deliver them into B, in particular into the unequal size subsets B^1 and B^2. Note that h_{11} and h_{12} depend on two parameters p_1 and p_2. Good values for these two parameters are experimentally determined to be around $0.6n$ and $0.3b$, respectively.

What this means is that roughly 60% of the keys (since $p_1 = 0.6n$ and h_{10} is likely to be relatively uniform at a coarse level) are mapped into roughly 30% of the buckets (since $p_2 = 0.3n$), i.e., $B^1 = \{0, \ldots, p_2 - 1\}$. In effect, we are forcing the buckets produced by h_{11} to each hold many keys. This is fine, since our earlier work with searching indicates that large groups of keys can be managed if dealt with early in the search process.

At the same time, the other say 40% of the keys are "spread" by h_{12} into 70% of the buckets, i.e., $B^2 = \{p_2, \ldots, s - 1\}$, yielding fewer keys per bucket. This is handy since

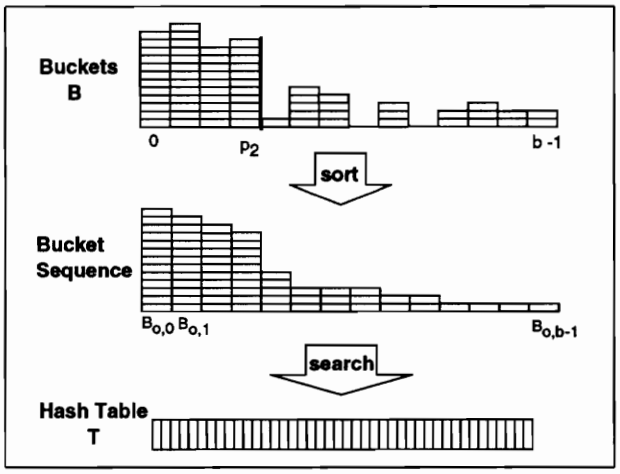


Figure 4.8: Ordering and Searching Stage

during searching it is desirable to have small groups of keys processed towards the end of the operation.

In summary, the Mapping phase, illustrated in Figure 4.7, accomplishes our goals of mapping to integers, compressing the range of integers, and separating big from small groupings of keys.

During the Ordering stage, illustrated in the top portion of Figure 4.8, we use the organization developed during Mapping to prepare for Searching. The key features of this stage are as follows. A similar strategy has been discussed in [TY79] and [FHCD92].

- Buckets are ordered by decreasing sizes to obtain the bucket sequence:

$$\{B_{o,0}, B_{o,1}, \ldots, B_{o,b-1}\}.$$

  (where the subscript o designates ordered buckets as opposed to initial buckets)

- Bucket sorting can be used as the maximal number of keys in B is known.

The Searching stage involves choosing a $\log_2 n + 1$ bit parameter value $g()$ for each of the buckets, so that each key in each bucket can be mapped by the finally constructed hash function, h, to a previously unused slot in the hash table T.

Essentially, the group of keys in a bucket must all be "fit" at the same time, since they are mutually constrained by virtue of the earlier processing that put them in the same bucket. Choosing the parameter value for the bucket must assure that its "pattern" of entries can be "fit" into open slots in T. As we try different $g()$ values, we "rotate" the pattern until we find a good fit.

Thus, the Searching process maps keys in each bucket $B_{o,i}$ to T via the function h:

$$h_{20}: \quad S \times \{0,1\} \rightarrow \{0,\ldots,n-1\}$$
$$h(k): \quad \{h_{20}(k,d) + g(B_{o,i})\} \; mod \; n,$$

where

- h_{20} is a pseudo-random function mapping keys in each $B_{o,i}$ to distinct values in $\{0, n-1\}$. Recall that $\log_2 n + 1$ bits are allocated to each bucket. A designated bit d in these bits is used by h_{20} as part of the seed. As 0 and 1 can be the value for d, h_{20} can generate two different sets of integers for keys in $B_{o,i}$. This adds a degree of freedom to the searching, avoiding failures by changing the d values as needed. An integer r global to all buckets has also been used as part of the seed to h_{20}. Should some bucket fail to be mapped to distinct integers, a new value for r is tried. With the help of r, the same bucket sequence can be reused. When large key sets are processed, generating another sequence of buckets can be costly. In the following, we use the term "pattern set" P_i for the set of values of h_{20} corresponding to keys in $B_{o,i}$.
- Each $g(B_{o,i})$ takes $\log_2 n$ bits.
- $g(B_{o,i})$ rotates the pattern set for a fit.
- $g(B_{o,i})$ can be selected by aligning an item in the pattern with an empty slot in T. This is an important heuristic to improve efficiency.

This final hashing function $h()$ has simple form and is easily computable for any key in S. It is formed as the sum of two values, and only calls for a single lookup to obtain the g value.

1).Input the key set;
2).Partition the key set into subsets K_is;
3).Sort K_i's into a non-increasing sequence KS;
4).Fit each K_i into the hash table based on KS;
5).Output the MPHF specification.

Figure 4.9: Outline of the Faster Algorithm

Based on the above discussion of the MOS framework, the new algorithm can be described, as shown in Figure 4.9.

The Implementation

The previous discussion leads to a simple algorithm which can be implemented easily.

To efficiently compute pattern sets, keys are best kept in memory. However, this may cause I/O thrashing when the key set is quite large. In the implementation, two operating modes are encoded. The first is the memory mode in which all keys of a small to relatively moderate size key set are in memory. The second, disk mode, stores keys on the disk and fetches each K_i when necessary. To explore the fact that keys within each K_i have reference locality, the key set is sorted initially based on KS. The offset of each K_i is kept in main memory. Thus, at searching stage, incremental file seeks are performed to read in K_i. The sorting algorithm used is a refinement of bucket sorting. Chunks of keys (the size of a chunk is a parameter set by the user) are sorted in main memory and flushed out onto the disk. Several merging passes are followed to obtain the final sorted set.

The implementation of the random number table for generating h_{10}, h_{11} and h_{12} is based on [Pea90]. As explained in that paper, simply XORing characters with the content of a 256 entries tables suffices to produce good random numbers within the range $[0, 255]$. As keys of printable ASCII characters are desired in the faster algorithm, the number of entries in the table is reduced to 128. To generate a random value larger than 127, several XOR operations are applied to the key string; each returns a different 7-bit portion of the final

```
// tbl is a 128 character array containing an arbitrary permutation of {0, 127}.
// num_slices counts the number of 7 bit slices in the pseudo-random
//     number.
// array h[] has num_slices entries. It records the partial results
//     for each 7 bit slice.
// variable mask holds the octal value 127.

int EP(char* string, int seed)
{
// initialize the h[] arrays.
  char c = string[0];
  for ( int i = 0; i < num_slices; i++ ) {
    h[i] = ( c + i +seed ) & mask;
    h[i] = tbl[ h[i] ];
  }

// compute values for all 7 bit slices, using Pearson method.
for ( i = 0; i < num_slices; i++ )
   for ( int j= 1; j < strlen(string); j++ )
     h[i] = tbl[ ( h[i] ^ string[j] ) & mask];

// combine all 7 bit slices into a pseudo random number
int sum = h[0];
for ( i = 1; i < no_slices; i++ ) {
    int x = h[i];
    x = x << ( i * 7 );
    sum += x;
  }

// make the pseudo random number positive
sum = abs(sum);
return sum;
}
```

Figure 4.10: Extended Pearson Method

random number.

Since the Pearson method yields an 8 bit pseudo-random number from a character string and in our case a larger pseudo-random number is desired from the same character string, the character string is modified to generate a different 7-bit portion. This is done by incrementing (mod 128) the ASCII value of the first character in the character string. Figure 4.10 lists the program that implements the extended Pearson method.

To assure the randomness of the new approach, a series of chi-square tests were conducted, as reported in Table 4.3. The randomness of three approaches is compared: Random

Table 4.3: Chi-Square Tests on the Randomness of the Three Methods (k=64)

n	Random	7-bit Pearson	Table
1024	accept	accept	accept
2048	accept	accept	accept
4096	accept	accept	accept
8192	accept	accept	accept
16384	suspect	accept	accept
32768	accept	accept	accept

Table 4.4: Running Time Summary of the Three Methods (k=64)

n	Random	7-bit Pearson	Table
1024	0.15	0.30	0.28
2048	0.32	0.62	0.67
4096	0.58	1.17	1.37
8192	1.38	2.25	2.81
16384	2.53	4.60	5.45
32768	6.18	9.73	10.60

(random numbers generated from Park and Miller's random number generator [PM88a]), 7-bit Pearson (as described above) and Table (as described in Section 4.2). Each method is applied to six sets of English words. The resulting random numbers are classified into $k = 64$ categories following Knuth [Knu73b]. They are: $0 - 1$ or $99 - 100$ as reject; $2 - 5$ or $95 - 98$ as suspect; $6 - 10$ or $90 - 94$ as almost suspect and $11 - 89$ as accept. Table 4.3 indicates that all methods are of equally good quality in producing good random numbers.

Table 4.4 lists the time (in seconds) each method spent. The random method requires the least time since it does not scan through each key string. It is expected that any approach with high disambiguating power should pass through each key string in order to extract enough information. 7-bit Pearson and Table methods are examples of such an approach. The 7-bit Pearson is marginally faster than Table in all cases except the 1024-word case, mostly due to the XOR operation.

The searching step uses a variant of a well-known pattern matching algorithm [Boy77]. Recall that a fit implies that each member in the pattern set P_i matches an empty slot.

Therefore, an arbitrary member in P_i can be aligned with an empty slot, and testing can then determine whether the rest of the members fit into other empty slots. A proper alignment yields a proper g value. We define:

$$x \equiv \text{the index of the empty slot}$$
$$u \equiv \text{the member of } P_i \text{ to be aligned with } x.$$

The rotation offset or $g(B_{o,i})$ is computed as $(n + x - u) \bmod n$. The method gives considerable speedup when key sets are of moderate to large size.

Figure 4.11 illustrates an index data structure, along with the hash table. In the diagram, there are three arrays called `randomTable`, `mapTable` and `hashTable`. The first array is used to remember currently empty slots in the hash table. As it is preferable for each P_i to fill the hash table in a random fashion, this array initially contains a random permutation of the hash addresses in the range $[0, n-1]$. The pointer `filledCount` for `randomTable` is initially 0. It is an invariant that any slots to the right side of `filledCount` (inclusive) are empty and any ones to the left are filled. This property guarantees only empty slots are searched to fit P_i. After fitting P_i, the empty slots used by P_i need to be excluded from the right side of `filledCount`. This is achieved by making use of the array `mapTable`.

The `mapTable[0, n-1]` array contains pointers pointing at `randomTable` such that, for any filled slot x in `hashTable`, `randomTable[mapTable[`x`]]` $\equiv x$. Thus, given an empty slot x in the hash table, we can locate its position in the `randomTable` array through `mapTable`. Suppose a slot v of `hashTable`, which is referred to in location y in `randomTable`, needs to be filled and the invariant needs to be maintained after the filling action. Then we can just switch the pointers corresponding to `mapTable[randomTable[filledCount]]` and `mapTable[`y`]` and advance `filledCount` right one position. See the positions of the two differently shaded boxes in the topmost part of Figure 4.11. When $|P_i| > 1$, a sequence of switches is required.

Thus, if no backtracking is involved in the above algorithm, as it usually the case, total time is $O(n)$.

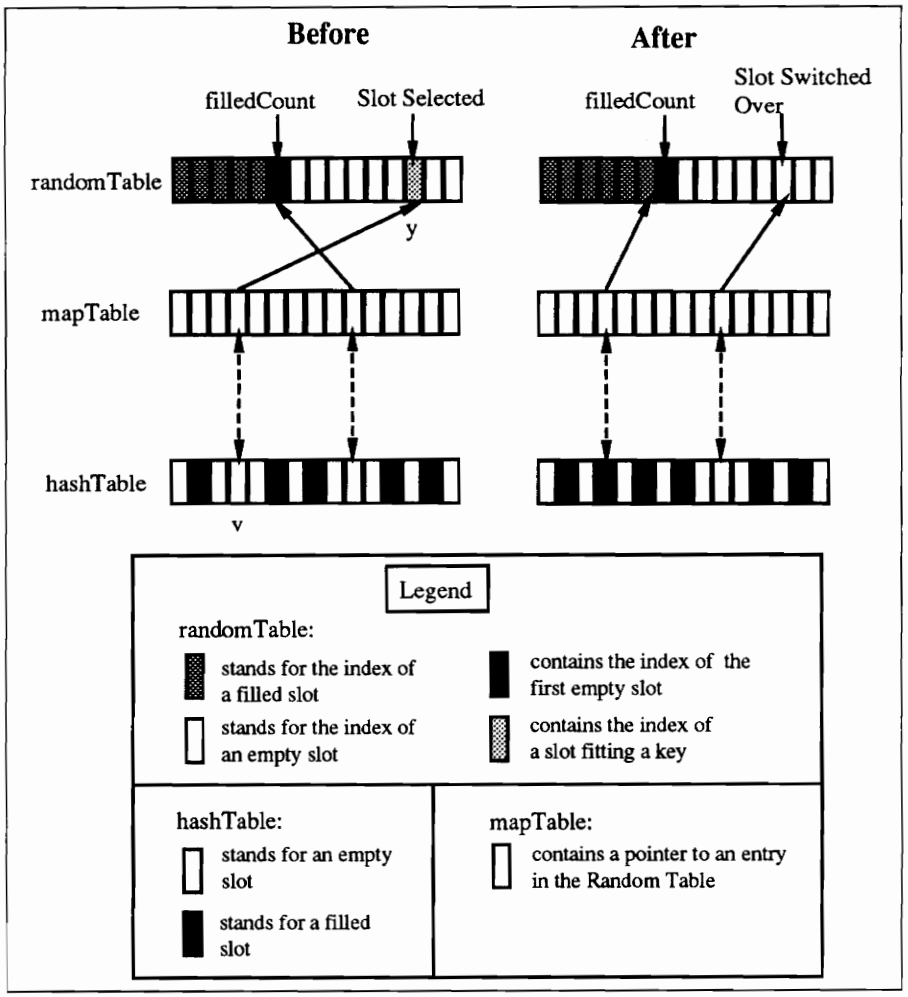


Figure 4.11: Index Data Structure and Filling of a Hash Table Slot

4.3.2 Analysis

In this section, the distributions of keys in B_1 and B_2, the probability of keys in K_i colliding with each other, and expected cost for K_i to fit the hash table, are analyzed.

Distribution of $|K_i|$

Let X_i be the random variable that equals the size of K_i in B_1 and Y_j be the random variable that equals the size of K_j in B_2. The probability that x keys fall into K_i is

$$\Pr(X_i = x) = \binom{p_1}{x}(\frac{1}{p_2})^x(1-\frac{1}{p_2})^{p_1-x}$$

X_i is a binomial distribution, with parameters p_1 and $1/p_2$. Similarly, the probability that y keys fall into K_j is

$$\Pr(Y_j = y) = \binom{n-p_1}{y}(\frac{1}{b-p_2})^y(1-\frac{1}{b-p_2})^{n-p_1-y}$$

Y_j is also a binomial distribution, with parameters $n-p_1$ and $1/(b-p_2)$.

When n is large and since p_2 is linearly proportional to n, both $\Pr(X_i = x)$ and $\Pr(Y_j = y)$ can be approximated by the Poisson:

$$\Pr(X_i = x) \approx \frac{e^{-\lambda_1}\lambda_1^x}{x!}$$
$$\Pr(Y_j = y) \approx \frac{e^{-\lambda_2}\lambda_2^y}{y!}$$

where $\lambda_1 = \frac{p_1}{p_2}$ and $\lambda_2 = \frac{n-p_1}{b-p_2}$. The expected value of X_i is λ_1 and that of Y_j is λ_2.

Probability of Generating Useful Pattern

Previously (section 4.3.1) it was mentioned that $\log_2 n + 1$ bits are allocated for each K_i, out of which 1 bit is for pattern generation and $\log_2 n$ bits are for pattern rotation. It is interesting to calculate the probability that a particular K_i of size x fails to yield a useful pattern under such an arrangement and the probability that all K_is can be successfully

mapped to their patterns with one application of $h_1()$. Here a usable pattern P_i is defined as a set containing distinct integers: $P_i \subseteq \{0 \ldots n-1\}$.

Let B_{jr} be the rth key subset of size j and Q_{jr} be the indicating random variable:

$$Q_{jr} = \begin{cases} 1 & B_{jr} \text{ can yield a useful pattern} \\ 0 & B_{jr} \text{ can't yield a useful pattern} \end{cases}$$

The probability that a usable pattern can be generated is:

$$\Pr(Q_{jr} = 1) = \frac{n_j}{n^j} + \frac{n_j}{n^j}(1 - \frac{n_j}{n^j})$$

which can be approximated to

$$\approx 2(e^{\frac{j^2}{2n}} - e^{\frac{j^2}{n}}).$$

using the formula $(n)_j/n^j \approx O(1)e^{\frac{-j^2}{2n}}$ ([Pal85], Appendix III).

$\Pr(Q_{jr} = 1)$ approaches 1 when $j \ll n$, which is the case in the algorithm.

Expected Cost of Fitting a Pattern

Next, the cost of fitting a pattern of size j into an n slots hash table T with f slots already filled is approximated. The total number of slot subsets of size j from T is $\binom{n}{j}$, out of which only $\binom{n-f}{j}$ can fit the pattern. Imagine $\binom{n}{j}$ subsets as $\binom{n}{j}$ balls in a bag, where $\binom{n-f}{j}$ are white balls and $\binom{n}{j} - \binom{n-f}{j}$ are black balls. The cost of fitting the pattern is equivalent to repeatedly drawing balls from the bag until the first white ball is seen, without putting back previously drawn black balls. Let V_j be a random variable equal to the number of draws to obtain the first white ball in such an experiment, and denote $\binom{n}{j} - \binom{n-f}{j}$ as B_j and $\binom{n-f}{j}$ as W_j. We have

$$\Pr(V_j = z) = \prod_{r=0}^{z-2}(\frac{B_j - r}{B_j + W_j - r})(\frac{W_j}{B_j + W_j}).$$

When j is small and n is large, the fitting process can be approximated as a Bernoulli experiment where the balls after being drawn are returned to the bag. Let Z_j be a random

variable equal to the number of draws to obtain the first white ball in the approximated setting. We have

$$\Pr(Z_j = z) = (\frac{B_j}{B_j + W_j})^{z-1}(\frac{W_j}{B_j + W_j}).$$

The expectation of Z_j is

$$\begin{aligned} \mathrm{E}(Z_j) &= \sum_{z=1}^{n} z\Pr(Z_j = z) \\ &\approx \sum_{z=1}^{\infty} z\Pr(Z_j = z) \end{aligned}$$

as large hash table size is of interest.

Denoting $(\frac{B_j}{B_j+W_j})$ by q and $(\frac{W_j}{B_j+W_j})$ by p, we have

$$\begin{aligned} \mathrm{E}(Z_j) &= \sum_{z=1}^{\infty} z\Pr(Z_j = z) \\ &= \sum_{z=1}^{\infty} zq^{z-1}p \\ &= \frac{1}{p} \\ &= \frac{B_j + W_j}{W_j} \\ &= \frac{\binom{n}{j}}{\binom{n-f}{j}}. \end{aligned}$$

This simple closed form formula can be used in the algorithm to predict the number of rotations for a fit. If the predicted value is too large (i.e., $> n$), then there is no point in actually performing the fitting. Table 4.5 compares the computed expected values of variable Z_j with those generated from simulations. The parameters are selected to be close to actual situations. Except for the two cases ($n = 1024$, $f = 200$ and $j = 31$, and $n = 4096$, $f = 815$ and $j = 25$), the formula predicts the number of rotations quite accurately. The accuracy of predication decreases when the f value is larger, compared to that of n. An informal explanation is that a larger f value makes it difficult to find a fit. The Bernoulli

Table 4.5: Comparison of Average Rotations for Fits with Expected Values

j	n = 1024, f = 40		n = 1024, f = 120		n = 1024, f = 200	
	Avg.	Exp.	Avg.	Exp.	Avg.	Exp.
4	2.05	1.17	2.50	1.65	2.85	2.39
13	2.55	1.68	5.55	5.11	20.80	17.18
22	3.40	2.43	17.85	16.00	113.00	126.00
31	3.80	3.50	54.75	50.68	565.65	943.00
j	n = 4096, f = 163		n = 4096, f = 489		n = 4096, f = 815	
	Avg.	Exp.	Avg.	Exp.	Avg.	Exp.
7	2.20	1.33	2.80	2.44	4.80	4.73
13	2.40	1.70	4.80	5.23	13.65	17.97
19	2.45	2.17	11.20	11.26	67.20	68.43
25	2.60	2.77	26.05	24.25	203.85	261.08
j	n = 8192, f = 327		n = 8192, f = 654		n = 8192, f = 981	
	Avg.	Exp.	Avg.	Exp.	Avg.	Exp.
9	2.40	1.44	3.00	2.11	4.10	3.15
17	2.70	2.00	5.05	4.12	11.25	8.76
25	2.80	2.77	8.65	8.03	24.70	24.38
33	4.55	3.85	21.40	15.66	62.95	67.89

Note:
j = size of subset to be fit;
n = number of keys;
f = number of slots already fit;

approximation makes it more difficult to find a fit, as a tried location set may be tested again. Thus the expected value is much larger than the actual one.

Percentage of Saving

As explained in section 4.3.1, an auxiliary data structure can be used to land an element of a pattern on an empty hash table slot. In such a case, the fitting process can be modeled as fitting the remaining $j-1$ elements in the pattern into a hash table of $n-1$ slots with

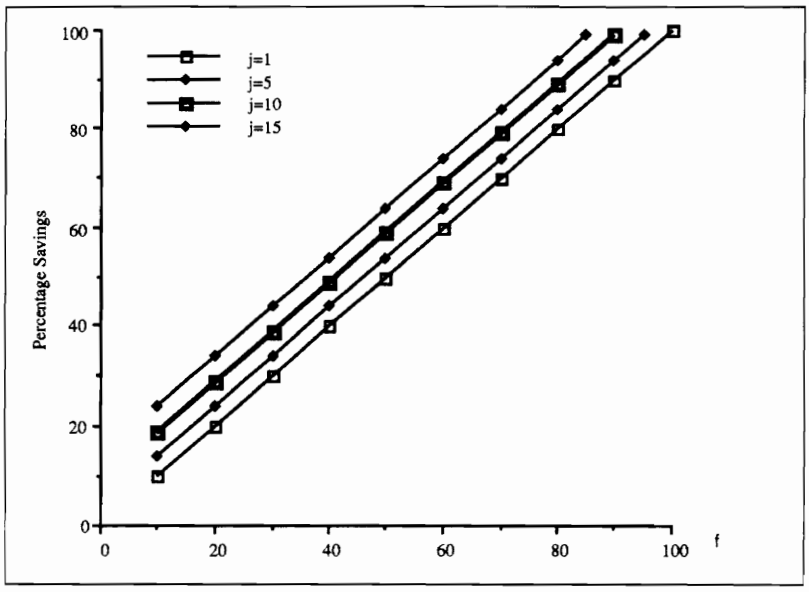


Figure 4.12: Plotted Percentage of Savings (n = 100)

$f-1$ already filled slots. The expected number of rotations for a fit becomes:

$$\begin{aligned} \mathrm{E}(Z_j') &= \frac{\binom{n-1}{j-1}}{\binom{(n-1)-(f-1)}{j-1}} \\ &= \frac{\binom{n-1}{j-1}}{\binom{n-f}{j-1}}. \end{aligned}$$

$\mathrm{E}(Z_j') = 1$ when $j = 1$. When $j > 1$, the saving arising from the alignment is:

$$\begin{aligned} \frac{\mathrm{E}(Z_j) - \mathrm{E}(Z_j')}{\mathrm{E}(Z_j)} &= 1 - \frac{n-f-j+1}{n} \\ &= \frac{f+j-1}{n} \end{aligned}$$

which grows larger as the hash table becomes full or the pattern size is larger. Figure 4.12 illustrates the percentage of savings as a function of f and j for a fixed $n = 100$.

4.3.3 Experiment

This section presents the results of running the new algorithm. Table 4.6 describes the performance of the algorithm on different ket sets, running in main memory mode. The

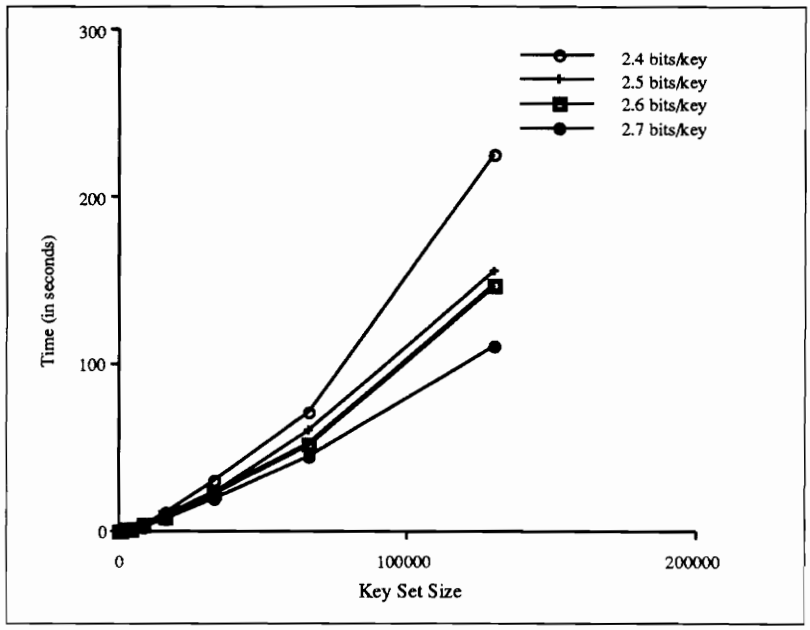


Figure 4.13: Total Time vs. Key Set Size

desired space is specified at 2.4, 2.5, 2.6 and 2.7 bits per key. Parameters p_1 and p_2 are set to 0.6 and 0.3 respectively. The operating mode is main memory mode. Figure 4.13 plots the data in Table 4.6.

Table 4.7 shows the time for a large 420,878 key set when bits per key values are varied from 2.35 to 2.9. The algorithm operates in the disk mode and the sorting of bucket sizes is done by splitting the key set into two subsets (chunks). Figure 4.14 plots the total time vs. various bits per key values. It can be seen that searching time decreases steadily as the bits per key value increases and increases sharply when that same value decreases to values close to the theoretical lower bound.

Table 4.8 lists the time for a 3,875,766 key set with bits per key values are set from 2.4 to 3.5. Figure 4.15 plots the running time vs. bits per key values using Table 4.8 data. Note that the faster algorithm was able to find an MPHF for the 3.8 million key set in about 6 hours on a NeXTstation, with 2.7 bits per key. This translates into about a megabyte of space needed to store the MPHF specification for one of the largest real

Table 4.6: Running Time Summary of the Faster MPHF Algorithm

	2.4 bits/key				2.5 bits/key			
Keys	Map	Order	Search	Total	Map	Order	Search	Total
64	0.01	0.00	0.04	0.05	0.01	0.00	0.03	0.04
128	0.01	0.00	0.03	0.04	0.01	0.01	0.05	0.06
256	0.03	0.00	0.05	0.08	0.02	0.01	0.05	0.08
512	0.05	0.01	0.15	0.21	0.06	0.01	0.10	0.17
1024	0.10	0.00	0.24	0.35	0.12	0.00	0.19	0.31
2048	0.21	0.01	0.59	0.80	0.22	0.00	0.43	0.66
4096	0.44	0.02	1.14	1.60	0.44	0.01	0.91	1.36
8192	0.86	0.02	2.98	3.84	0.87	0.02	2.59	3.49
16384	1.72	0.02	8.79	10.53	1.72	0.33	7.34	9.11
32768	4.28	0.06	25.33	29.67	4.29	0.07	18.23	22.59
65536	8.75	0.12	62.55	71.42	8.58	0.12	51.47	60.17
130198	17.15	0.21	208.06	225.82	17.12	0.25	137.86	155.45
	2.6 bits/key				2.7 bits/key			
Keys	Map	Order	Search	Total	Map	Order	Search	Total
64	0.00	0.00	0.04	0.05	0.00	0.00	0.04	0.05
128	0.00	0.00	0.05	0.05	0.02	0.01	0.03	0.05
256	0.00	0.00	0.03	0.08	0.03	0.01	0.04	0.08
512	0.05	0.00	0.06	0.16	0.04	0.00	0.08	0.14
1024	0.11	0.00	0.23	0.30	0.05	0.01	0.28	0.40
2048	0.21	0.01	0.44	0.66	0.21	0.02	0.39	0.62
4096	0.43	0.02	0.97	1.42	0.27	0.17	0.86	1.30
8192	0.87	0.03	2.33	3.23	0.85	0.02	2.03	2.90
16384	1.72	0.04	6.15	7.93	1.71	0.03	5.18	6.92
32768	4.28	0.06	18.15	22.51	8.71	0.06	13.75	18.09
65536	8.58	0.13	43.18	51.88	8.71	0.12	37.04	45.88
130198	17.06	0.24	130.02	147.21	17.12	0.24	93.89	111.25

Note: for all runs,
Mode = main memory;
Machine = DECstation 5000;
Language = GNU G++ 1.37;
Time (CPU) is in seconds.

Table 4.7: Running Time Summary (keys = 420,878)

Bits/Key	Mapping	Ordering	Searching	Total
2.35	102.67	258.82	1410.72	1772.20
2.40	102.90	259.73	1245.65	1608.28
2.45	102.62	258.48	1080.73	1441.83
2.50	102.57	259.30	952.02	1313.88
2.55	102.75	259.65	134.85	1196.28
2.60	102.73	259.92	748.22	1110.87
2.65	102.65	259.98	674.03	1036.67
2.70	102.75	261.10	608.05	971.90
2.75	102.70	261.43	561.10	925.23
2.80	102.78	261.47	511.42	875.67
2.85	102.73	262.08	471.47	836.28
2.90	102.95	263.67	444.72	811.33
3.00	103.95	270.00	391.63	765.58
3.50	104.23	272.57	270.52	647.32
4.00	104.03	276.33	226.80	607.17
4.50	104.13	281.00	209.62	594.75
5.00	105.95	285.08	198.85	589.88

Note: for all runs,
Number of Chunks = 2;
Machine = NeXTstation with 12 MB memory;
Language = cc++ 1.36.4, GNU g++ library 1.39;
Time (CPU) is in seconds.

key set we could obtain, suggesting that the algorithm is quite feasible for use on modern workstations. These results are better than our previous best reported findings with the second algorithm reported in [FCHD90], which yielded an MPHF with 4.58 bits per key, running on a Sequent (68030 processor) for about 9 hours.

Figure 4.16 plots the level sizes and the number of rotations for an $n = 1024$ key set. Figure 4.17 is for an $n = 4096$ key set. In both figures, the X axis gives the sorted K_i sequence. The tick numbers on the axis indicate the indices of K_i's in the sequence. The left and right Y axes represent respectively the number of rotations and the sizes of K_i's. The saw-tooth shaped curve is about the actual numbers of rotations used by different K_i

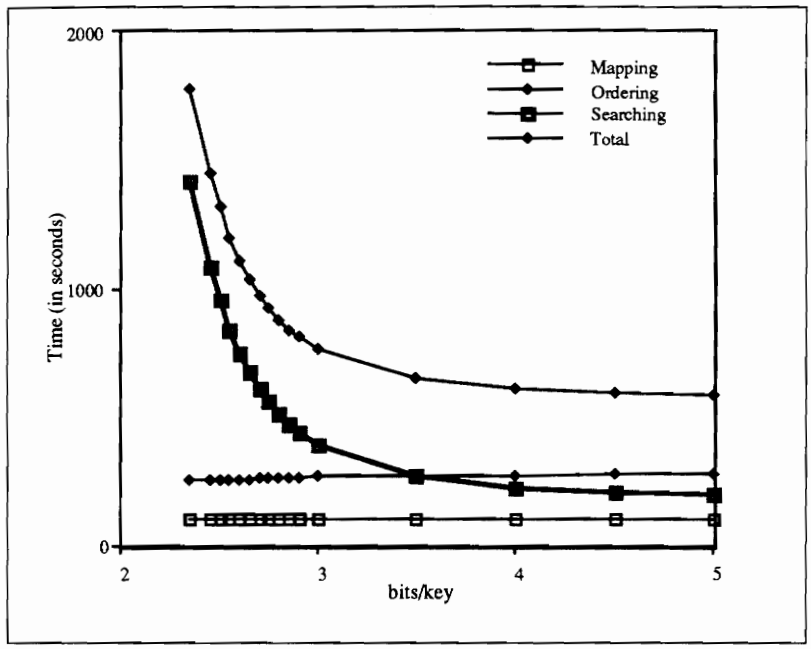


Figure 4.14: Running Time Summary for a 420,878 Word Set

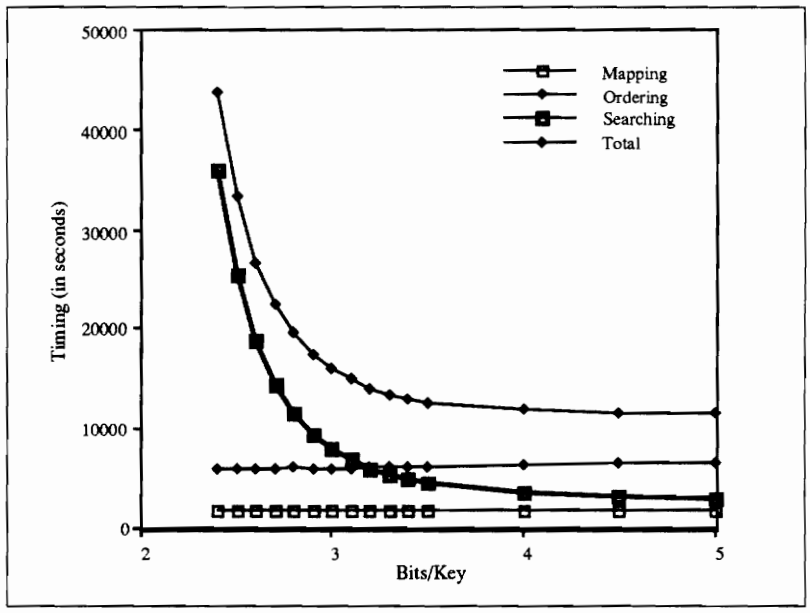


Figure 4.15: Running Time Summary for a 3,875,766 Word Set

Table 4.8: Running Time Summary (keys = 3,875,766)

Bits/Key	Mapping	Ordering	Searching	Total
2.4	1889.57	5927.55	35888.80	43705.90
2.5	1885.97	5936.38	25520.70	33343.10
2.6	1886.75	5977.77	18938.40	26802.90
2.7	1887.42	6047.78	14485.80	22421.00
2.8	1897.43	6170.10	11601.50	19669.10
2.9	1893.63	6088.02	9524.37	17506.00
3.0	1904.75	6107.53	8083.08	16095.40
3.1	1893.73	6119.43	6997.67	15010.80
3.2	1885.23	6140.98	6109.87	14136.10
3.3	1883.62	6224.05	5436.15	13543.80
3.4	1885.55	6197.42	4957.92	13040.90
3.5	1885.83	6191.40	4585.70	12662.90
4.0	1894.35	6421.37	3638.32	11954.00
4.5	1895.17	6530.93	3181.98	11608.10
5.0	1894.87	6698.38	2999.28	11592.50

Note: for all runs,
Number of Chunks = 5;
Machine = NeXTstation with 64MB memory;
Language = cc++ 1.36.4, GNU g++ library 1.39;
Time (CPU) is in seconds.

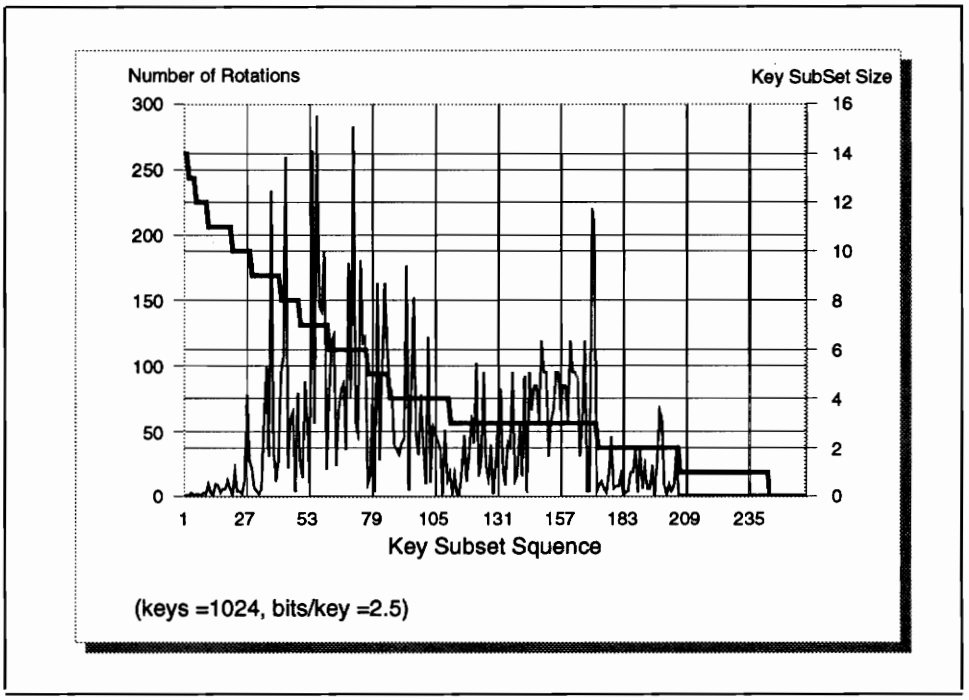


Figure 4.16: Key Set Sizes against the Number of Rotations (1024 keys)

sorted into a non-increasing order shown in the staircase shaped curve. It can be seen that at both ends of the sequence, the numbers of rotations are small. In Figure 4.16 these ranges are $[1 \ldots 10]$ and $[180 \ldots 230]$, and in Figure 4.17 the ranges are $[1 \ldots 5]$ and $[700 \ldots 819]$. The number of rotations in the tail sections for size 1 levels is exactly 1. Also, it can be observed that more rotations are needed for K_i's appearing later in the sequence than those occurring earlier with same size. This corresponds to an angle-shaped curve at each step in the staircase shaped curve.

4.3.4 A Note on the Search Step

In this section, two different searching strategies (rotation fit and random fit) are compared. A method to reduce the space used by the random fit method is sketched. Random fit method has been used in the second algorithm in [FHCD92].

Recall that a total of $\log_2 n + 1$ bits are attached to each key subset. A pattern is

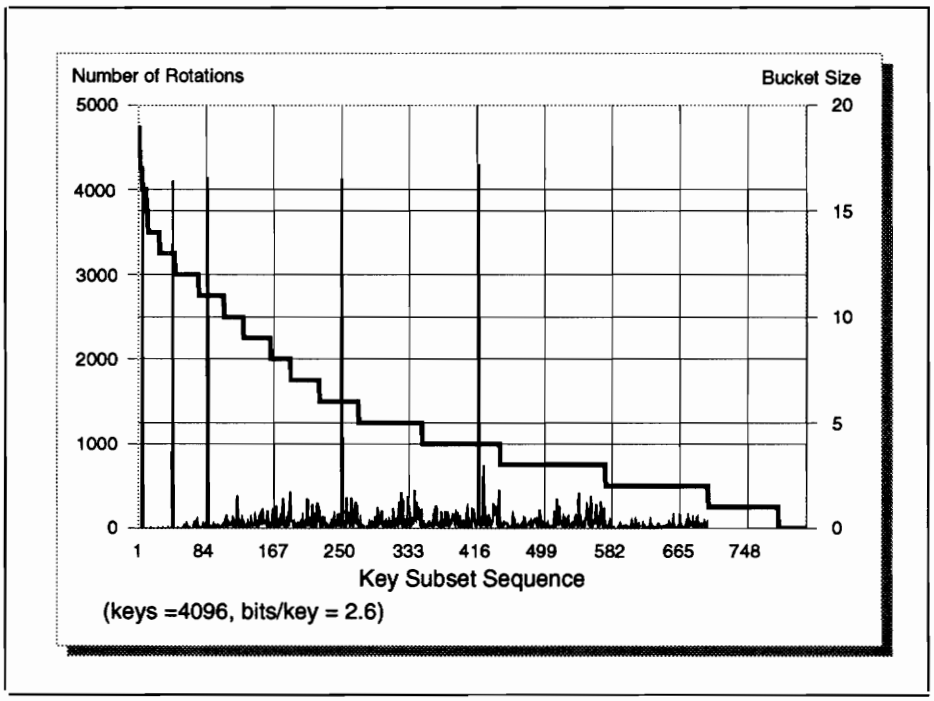


Figure 4.17: Ket Set Sizes against the Number of Rotations (4096 keys)

produced from each subset and matched against the hash table in a rotating manner. The success of obtaining the pattern is probabilistically assured by the pattern bit.

Consider Table 4.9 where the performance of the rotation fit method and the random fit method is compared. The table reports the results of two simulations of fitting patterns of sizes from 1 to 12 in the increment of 1 onto a hash table of 128 slots. The number of filled slots in the experiment is selected to be 5, 10, 15 and 20. For each parameter pair (pattern size, filled slots), a total of 50 random fit tests and rotation fit tests are performed. Each test has 7 bits to use. The random test takes 7 bits for a total of 128 rotations. The rotation test, on the other hand, splits the 7 bit into 1 pattern bit (d = 1) and 6 rotation bits. Thus, it can generate 2 different patterns, and each pattern can rotate 64 times. The `ratio` columns give average percentage of successful tests in the 50 tests. The `fails` columns list number of fails (i.e., no fit in the test) in these 50 runs.

It can be seen that both methods have almost identical percentage of fit ratios. This can

Table 4.9: Simulation of Random and Rotation Fit

Pattern Size	Filled = 5				Filled = 10			
	Random Fit		Rotation Fit		Random Fit		Rotation Fit	
	Ratio	Fails	Ratio	Fails	Ratio	Fails	Ratio	Fails
1	0.9614	0	0.9581	0	0.9155	0	0.9219	0
2	0.9190	0	0.9222	0	0.8394	0	0.8555	0
3	0.8291	0	0.8839	0	0.7520	0	0.7284	1
4	0.8125	0	0.8341	0	0.6912	0	0.6775	0
5	0.7513	0	0.7739	1	0.6145	0	0.5880	0
6	0.7080	0	0.6701	1	0.5338	0	0.5263	1
7	0.6380	0	0.5923	1	0.4764	0	0.4683	1
8	0.5811	0	0.5698	1	0.4134	0	0.3797	1
9	0.5139	0	0.5167	2	0.3627	0	0.3406	1
10	0.4656	0	0.4855	3	0.3025	0	0.2942	4
11	0.3938	0	0.4325	4	0.2547	0	0.2308	5
12	0.3575	0	0.3832	7	0.2084	0	0.2045	9

Pattern Size	Filled = 15				Filled = 20			
	Random Fit		Rotation Fit		Random Fit		Rotation Fit	
	Ratio	Fails	Ratio	Fails	Ratio	Fails	Ratio	Fails
1	0.8773	0	0.8811	0	0.8372	0	0.8480	0
2	0.7728	0	0.7489	0	0.7097	0	0.7073	0
3	0.6739	0	0.6467	0	0.5852	0	0.5811	0
4	0.5875	0	0.5364	1	0.4670	0	0.4869	0
5	0.4923	0	0.4697	0	0.3869	0	0.3922	0
6	0.4300	0	0.3980	0	0.2942	0	0.3291	0
7	0.3513	0	0.3363	2	0.2561	0	0.2439	0
8	0.2919	0	0.2770	1	0.1964	0	0.2073	0
9	0.2358	0	0.2458	1	0.1536	0	0.1427	5
10	0.1966	0	0.1741	4	0.1114	0	0.1202	6
11	0.1563	0	0.1488	6	0.0927	0	0.0880	9
12	0.1291	0	0.1213	10	0.0695	0	0.0663	11

be explained by the fact that the random test amortizes successful and failure fittings and the rotation test concentrates on the successful ones using patterns containing all distinct members. These factors have no effect on the fit ratio at all. However, if the tests are viewed as a Bernoullie experiment where the outcome is either "there is a fit" or "no fit is found", then the random method consistently outperforms the rotation method as it has 0 fails in all the runs, while the random method tends to incur fails when the pattern size grows.

However, methods exist that can quickly fit patterns using rotation tests. One approach is to make all patterns not too large or equivalently to use a slightly larger specification space. Such a constraint does not hold for the random fit method. But, as hash addresses are randomly generated from some random number generator, the searching speed may suffer. Another problem with the random fit method is the handling of the remaining size one patterns.

Assume that all K_i's for $|K_i| > 1$ can fit by using the mapping and ordering steps of the faster algorithm described in section 4.3 and a random fit searching step. For keys in singular K_i's where $|K_i| = 1$, random matching in sequence may not work as all probes of the very last key may miss the only remaining empty slot. A novel fix of the problem is to consider such K_i's collectively.

Let G be a bipartite graph in which one side U of the graph consists of all the keys in singular K_i's and the other side V has all slots in the hash table. See Figure 4.18 (a). We consider the remaining empty slots. If key u in a singular K_i can probe slot v in T, an edge is drawn between u and v. Given G, a perfect matching that enables all u's to land at all remaining empty slots is desired. Suppose u can be mapped n times (i.e., $g(K_i)$ occupies $\log_2 n$ bits), there are at most n edges emitting from each u. In the remainder of the section, we evaluate the probability of the existence of such a perfect matching.

First we examine the odds that a particular empty slot v will not be probed by a specific key node u. Let N be the random variable counting the number of times v has been probed

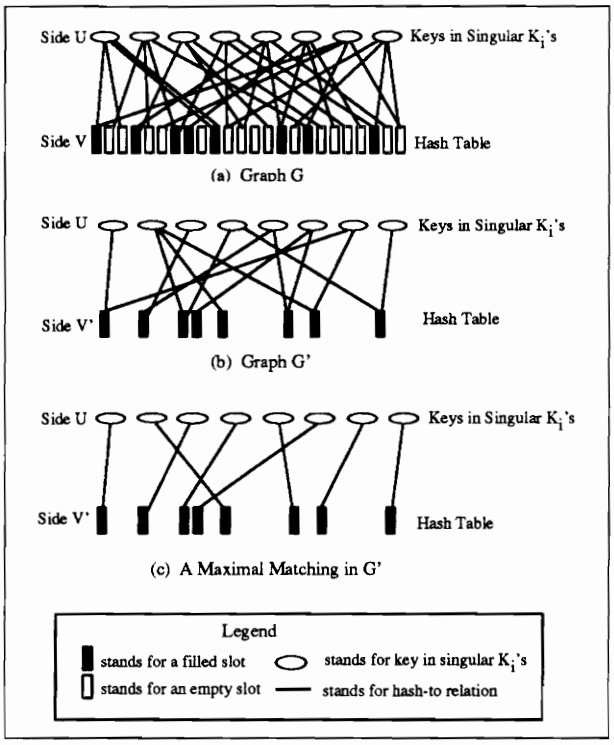


Figure 4.18: Graph G and its Induced Graph G'

by u in all n probes. The probability that all n probes will miss v is:

$$\begin{aligned} \Pr(N = 0) &= \frac{(n-1)^n}{n^n} \\ &= (1 - \frac{1}{n})^n \\ &\approx e^{-1}. \end{aligned}$$

The probability that all u's on side U miss v is:

$$\begin{aligned} \mathrm{P} &= \prod_{i=1}^{l} \Pr(N = 0) \\ &\approx e^{-l}. \end{aligned}$$

where l is the number of singular K_i's. If $l \rightsquigarrow \infty$ as $n \rightsquigarrow \infty$, P approaches 0. Therefore, it is unlikely that an empty node will not be probed from side U in G.

Next, we calculate the probability that a perfect matching exists. To facilitate discussion, we consider an induced bipartite graph G': G' has all nodes from side U of G and only empty slot nodes from side V. The new side is denoted as V'. Duplicated edges among any node pairs in G are eliminated in G'. A sample G' is shown in Figure 4.18 (b) for G drawing in Figure 4.18 (a). The existence of a perfect matching in G is equivalent to the existence of a maximal matching in G'. A maximal matching is shown in Figure 4.18 (c) for the sample G'.

As pointed out by Walkup in [Wal80], an isolated node in G' makes it impossible to find a maximal matching. The above formula shows that the probability of an empty slot not being probed approaches zero when n is large, which implies that there are no isolated nodes in the V side of G'. Since nodes on the U side of G' "initiate" the edges, all v nodes are connected. An edge exists in G' with probability

$$p = 1 - (1 - \frac{1}{n})^n.$$

This is the same probability that a key at node u is hashed at least once to the empty slot represented by node v in G'.

G' is a model A ($G_{n,p}$) random graph as discussed in [Par85]. When n is sufficiently large, p satisfies the condition:

$$p > (\log_e n + t_n)/n,$$

where t_n is an arbitrary function such that $\lim_{n\to\infty} t_n = \infty$. t_n can be set to the function $\log_e n$ in our case. Following [ER66], G' has a perfect matching almost certainly.

Algorithms to find such a perfect matching can be found in [AV79], [Mot89] and [GH90]. Since G' is quite dense due to p, a good choice is the one described in [GH90] which is of $O(n \log_e(1/p) + n)$ or almost $O(n)$ time complexity in our perfect matching case.

4.3.5 Comparison of the Faster MPHF Algorithm with B-trees

This section briefly compares the MPHFs (produced by the faster MPHF algorithm) with B-trees. Since the algorithm works best with static data, while a B-tree can handle inser-

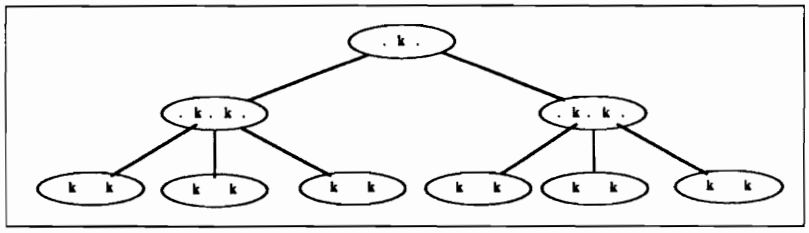


Figure 4.19: An Example B-tree

tion and deletion, the comparison is made only regarding space consumption and search efficiency. Here by space consumption we mean the extra space (in bytes) required by either the MPHF or the B-tree. By search efficiency we mean the number of keys accessed in order to know if a query key *key* is in the key set.

Following Bayer and McCreight [BM72], Knuth ([Knu73a], pp.473-480), and Landenhop and Wright [LW89], a B-tree of order $m = 2k+1$ is defined as a tree with following properties:

- k is a positive integer.
- The root has at least 1 and at most $2k$ keys.
- Every non-root node contains at least k and at most $2k$ keys.
- A non-leaf node with x keys has $x + 1$ child nodes.
- All leaves are at the same level.

Further, we assume that all keys in the tree are subject to query and pointers are not stored in leaf nodes. Figure 4.19 shows a B-tree of order 5, where a stands for a key and a dot stands for a pointer. Note that there are 3 levels in the tree. The root contains 1 key and all other nodes contain 2 keys.

To help with evaluating space utilization of B-trees, we define the following terms.

l: l is the number of levels in the tree.

N: N is the number of nodes in the tree.

E: E is the number of edges (i.e., the number of pointers) in the tree. The relation $N = E+1$ holds.

K: K is the number of keys in the tree.

For a B-tree with fixed l and k, E takes minimal value E_{min} when the root node has 1 key and all other nonleaf nodes have k keys. E takes maximal value E_{max} when all nonleaf nodes contain $2k$ keys. E_{min} and E_{max} give the lower and the upper bound on the number of pointers. They are also the bounds for the space consumption.

Assuming the B-tree is configured where $E = E_{min}$, N can be calculated as

$$1 + 2[(k+1)^{l-1} - 1]/k,$$

and the number of keys K can be computed as

$$2[(k+1)^{l-1}] - 1.$$

See [BM72], pp.174-175.

Rewriting N in terms of K, we have: $N = \frac{K-1}{k} + 1$. This formula shows that N can be sublinear to K. It also shows that N can be made very small by using large k. Of course $N = 1$ is possible, which corresponds to a degraded B-tree with only one node. Although such a B-tree consumes no extra space for pointers, the search can't be done efficiently.

Similarly, we can compute N and K for the B-tree configured at $E = E_{max}$. The number of nodes N is

$$[(2k+1)^l - 1]/2k,$$

and the number of keys K is

$$(2k+1)[(2k+1)^{l+1} - 1]/2k.$$

See [BM72], pp.175.

Table 4.10 compares the space consumption of B-trees with that of MPHFs. We assume the MPHFs under comparison takes 3 bits per key to specify. Column 1 and 2 list the

Table 4.10: Comparison of Space Consumption

		Min			Max		
k	l	E	K	E_{MPHF}	E	K	E_{MPHF}
5	3	14	71	6	132	1330	124
10	3	24	241	22	462	9260	868
25	3	54	1351	126	2652	132650	12435
50	3	104	5201	487	10302	1030300	96590
100	3	204	20401	1912	40602	8120600	761306
200	3	404	80801	7575	161202	64481200	6045112
5	4	86	431	40	1463	14640	1372
10	4	266	2661	249	9723	194480	18232
25	4	1406	35151	3295	135303	6765200	634237
50	4	5306	265301	24871	1040603	104060400	9755662
100	4	20606	2060601	193181	8161203	1632240800	18804848
200	4	81206	16241201	1522612	5368708	2147483646	67108864

various combination of k and l. Column 3, 4 and 5 show minimal number of pointers while column 6, 7 and 8 show maximal number of pointers, for each combination of k and l. We use E_{MPHF} to denote the number of pointers that would be required if K keys are indexed by a MPHF. The quantity E_{MPHF} is calculated through the formula $E_{MPHF} = 3K/32$, assuming each pointer takes 4 bytes to store. It can be seen that a B-tree uses much fewer pointers than a MPHF.

However, for the search efficiency comparison, MPHF's will win, as the complexity is always $O(2 * length(key))$ for a query key key. Searching in the B-tree involves following a search path from the root to a node where whether k is in tree or not can be answered. Within a node, we can use linear search if m is small, and binary search if m is large. The length of the search path depends on the key key and the configuration of the B-tree. Landenhop and Wright in [LW89] give a simulation method to measure average search path length. When $k = 50$ and $N = 75684$, they show the average length is 2.98.

4.3.6 Summary

This section describes a faster practical algorithm for finding minimal perfect hash functions, suitable for key sets ranging in size from small to very large.

This algorithm corrects many of the problems with the two earlier algorithms reported in [FCHD90], called Algorithm 1 and Algorithm 2. First, Algorithm 1 makes use of moderately large tables to generate integer triples. By extending Pearson's method [Pea90], the faster algorithm uses two mapping tables each containing only 128 characters. The results of the Mapping stage are sufficiently random.

Second, in Algorithm 1, the Searching phase was slow, requiring many unnecessary rotations to locate an acceptable solution. By adding an index data structure, the faster algorithm is able to reduce the searching time significantly.

Third, the faster algorithm deals with the need to reduce the size of the specification of the MPHF by radically changing the Mapping, Ordering, and Searching phases of Algorithm 1. In particular, no use is made of the bipartite dependency graph first suggested by Sager [Sag85].

Fourth, the Searching phase computes $g()$ values that properly rotate patterns until all elements of a bucket fit into the hash table. The auxiliary data structure speeds up searching by assuring we rotate the pattern in a random fashion, by avoiding rotations into previously filled slots, and by reducing the number of memory accesses for each test. Analysis yields a closed form estimate for the number of rotations needed at any given stage of the processing, allowing us to omit testing when failure is likely. Empirical studies show our estimates to be relatively accurate, indicating that Searching is generally fast.

Lastly, the faster algorithm uses the rotation fitting method during the searching step, as opposed to Algorithm 2 that uses the random fitting method. Rotation fitting makes it possible to use the index data structure.

4.4 Using MPHF to Compact Trie Data Structures

A *trie* is a useful data structure to help find information related to a set of keys sharing common prefixes; see [Bri59, Fre60]. Formally, a *trie* is a tree, whose nodes are M-place pointer vectors with fields corresponding to the digits or characters which comprise the keys. Each node on level i represents the set of all keys that start with the same sequence of i characters; the node specifies an M-ary branch, depending on the $(i+1)$st character of the key.

When the trie is organized logically in a tree fashion, the trie searching for a key is extremely fast: it depends only on the length of the key. This is in contrast to binary searching tree representation schemes which require at least $\log_2 n$ probes. Further, the time to insert and delete a key is proportional to the search time.

The above mentioned advantages of tries, however, are accompanied by high space requirements. [ASH84] proposes several algorithms to compact tries to save space. The search time of the resulting compacted trie is on the same order as an ordinary trie. Their compact trie algorithms usually achieve 80% space reduction on tries of around 40 nodes created from realistic data and 20% on tries of around 50 nodes where the numbers of pointers are generated randomly.

In this section, a new method for trie compaction is described. The new algorithm, based on the work on MPHF algorithms, compacts tries by converting them into a set of keys and a set of records where a hashing function can be built easily. After compaction, all of the null pointers in the ordinary trie are removed and the space used to specify the compact trie is about $\log_2 M + 2\log_2 N + 4$ bits per non-null pointer in the ordinary trie, where M is the length of the pointer vector in each trie node and N is the total number of nodes in the trie. Search time in the compacted trie is of the same order as the ordinary trie.

4.4.1 Related work

Knuth in [Knu73a] mentioned the use of compaction approachs to reduce trie space consumption. Ai-Suwaiyel and Horowitz in [ASH84] reveal the results of a detailed investigation of the approach. In the following, we give highlights of their research.

The core idea of the compaction method described in [ASH84] is to conceptually treat trie nodes as bit strings containing ones and zeros: A non null pointer corresponds to a 1 bit and a null pointer to a 0 bit. For a trie of N nodes each with M pointer positions, there are N such bit strings, each of M bits. To compact the original trie is to pack the N strings into a one-dimensional long bit string, called P, such that no two 1 bits have the same position. For example, two strings 10100 and 11001 can be packed into string 10111001, by placing the second string at the 4th position of the first. Arithmetically, this is to append 10100 to three 0's and to shift 11001 right three positions, obtaining 0001100, and then OR 10100000 and 0001100 together. If the bit strings are sparse (of 1's), then it is expected that many 1's in one bit string can occupy 0 positions in the other. As a result, a bit string with length less than MN bits is capable of encoding the original N strings.

A packing that will produce a long string P of all one's is optimal. As finding such a packing is NP-complete [ASH84], heuristic packing algorithms are pursued for near-optimal solutions. In [ASH84], three heuristic algorithms: COMPRESS, PAIR and LINEAR, are proposed. They differ primarily in the way a string pattern is chosen and where the pattern is tested against the partially formed bit string.

Since near-optimal solutions are produced, it is necessary to measure the quality of these solutions. Following Horowitz and Sahni ([HS78], pp.560-561), the notation *absolute*, $f(n)-$ and $e-$approximation algorithms are utilized. Let I be an instance of a problem to be solved approximately. Let $F(I)$ be the quality of the optimal solution for I and $G(I)$ be the quality of an approximate solution for the same I. Briefly, an algorithm is called an *absolute* approximate if $|F(I) - G(I)| \leq c$ for some constant c on all instances; or it is called $f(n)-$approximate if $|F(I) - G(I)| \ / \ F(I) \leq f(n)$ where $F(n)$ is not equal to 0; or

it is called $e-$approximate if it is $f(n)-$ approximate and $f(n) \leq e$, for some constant e. We denote as α the ratio $|F(I) - G(I)| \ / \ F(I)$.

Let the quality be measured as the number of bits. Then $F(I)$ is the number of bits required to optimally compact a trie instance I and $G(I)$ is the number used by a sub-optimal compaction algorithm. Since $G(I) \geq F(I)$, trie compaction is a minimization problem. $\mu = (G(I) - F(I)) \ / \ G(I)$ is defined as the space utilization of the compaction.

COMPRESS Algorithm

This algorithm works by selecting the nodes of a trie in any order and placing the next node at the first possible location in the partially formed bit string. Since at stage i of the computation $M * (i-1)$ locations may be tested, this algorithm is $O(MN^2)$.

Speaking of packing strategies, the algorithm tends to settle packed patterns close to each other. This is in contrast to the MPHF algorithm described in section 4.2 and 4.3 where there is random placement. Actually, our earlier work with MPHFs has revealed that it is much more difficult for concentrated placement strategies (i.e., packing by incrementally adjusting the g value) to produce good results.

PAIR Algorithm

This strategy works by combining pairs of nodes, then compacting pairs, and repeating the process until only a single node remains. This is a process much like working bottom-up in a complete binary tree. At each time, nodes in one level of the tree are compacted pairwise. The resulting strings are used to produce strings at parent nodes in the level above. This method takes only $O(MN \log N)$ units of time.

Since a node compacts only to its sibling in the binary tree, this heuristic practically yields sub-optimal compaction.

it is called $e-$approximate if it is $f(n)-$ approximate and $f(n) \leq e$, for some constant e. We denote as α the ratio $|F(I) - G(I)| \ / \ F(I)$.

Let the quality be measured as the number of bits. Then $F(I)$ is the number of bits required to optimally compact a trie instance I and $G(I)$ is the number used by a sub-optimal compaction algorithm. Since $G(I) \geq F(I)$, trie compaction is a minimization problem. $\mu = (G(I) - F(I)) \ / \ G(I)$ is defined as the space utilization of the compaction.

COMPRESS Algorithm

This algorithm works by selecting the nodes of a trie in any order and placing the next node at the first possible location in the partially formed bit string. Since at stage i of the computation $M * (i-1)$ locations may be tested, this algorithm is $O(MN^2)$.

Speaking of packing strategies, the algorithm tends to settle packed patterns close to each other. This is in contrast to the MPHF algorithm described in section 4.2 and 4.3 where there is random placement. Actually, our earlier work with MPHFs has revealed that it is much more difficult for concentrated placement strategies (i.e., packing by incrementally adjusting the g value) to produce good results.

PAIR Algorithm

This strategy works by combining pairs of nodes, then compacting pairs, and repeating the process until only a single node remains. This is a process much like working bottom-up in a complete binary tree. At each time, nodes in one level of the tree are compacted pairwise. The resulting strings are used to produce strings at parent nodes in the level above. This method takes only $O(MN \log N)$ units of time.

Since a node compacts only to its sibling in the binary tree, this heuristic practically yields sub-optimal compaction.

LINEAR Algorithm

This approximation algorithm works by repeatedly placing the next node at the first possible location of the previous node it fits into. As a small constant size M region is searched by the strategy for each next node, the time complexity of the approach is $O(MN)$. Like PAIR, LINEAR limits itself to a small searching space for a non-overlapping fit. It is unlikely that this approach yields a high-quality approximation solution.

The Quality of the Three Algorithms

For all three algorithms, the ratio of the size of the compacted trie to the size of the optimally compacted trie is in the range $[0, MN/(r+N-1)]$, where MN is the worst case size and $r+N-1$ is the optimal case size, for r being the number of keys stored in the trie. Let $f(n) = (mn-n-r+1)/(n+r-1)$, then it was shown in the paper that all three algorithms are $f(n)$-approximate when $m > 3$.

4.4.2 The New Algorithm

In this section, a new method of trie compacting using MPHFs is given. The trie data structure, coding scheme, quality of approximation of the method and experimental results are presented in turn.

The Trie Data Structure

The trie data structure implemented relies on the assumption that lexical data such as words in English dictionaries are the primary applications and therefore memory conservation is the major concern. To achieve this, internal nodes and leaf nodes in a trie are distinguished. A trie node in the algorithm can be in two states: *simple* or *expanded*. A *simple* node contains only a termination mark. The pointer vector is not stored. The *simple* nodes are used as leaf nodes of a trie. For an insert request, a *simple* node could be expanded into an *expanded* node for which a pointer vector must be allocated. The *expanded* nodes are

LINEAR Algorithm

This approximation algorithm works by repeatedly placing the next node at the first possible location of the previous node it fits into. As a small constant size M region is searched by the strategy for each next node, the time complexity of the approach is $O(MN)$. Like PAIR, LINEAR limits itself to a small searching space for a non-overlapping fit. It is unlikely that this approach yields a high-quality approximation solution.

The Quality of the Three Algorithms

For all three algorithms, the ratio of the size of the compacted trie to the size of the optimally compacted trie is in the range $[0, MN/(r+N-1)]$, where MN is the worst case size and $r+N-1$ is the optimal case size, for r being the number of keys stored in the trie. Let $f(n) = (mn-n-r+1)/(n+r-1)$, then it was shown in the paper that all three algorithms are $f(n)$-approximate when $m > 3$.

4.4.2 The New Algorithm

In this section, a new method of trie compacting using MPHFs is given. The trie data structure, coding scheme, quality of approximation of the method and experimental results are presented in turn.

The Trie Data Structure

The trie data structure implemented relies on the assumption that lexical data such as words in English dictionaries are the primary applications and therefore memory conservation is the major concern. To achieve this, internal nodes and leaf nodes in a trie are distinguished. A trie node in the algorithm can be in two states: *simple* or *expanded*. A *simple* node contains only a termination mark. The pointer vector is not stored. The *simple* nodes are used as leaf nodes of a trie. For an insert request, a *simple* node could be expanded into an *expanded* node for which a pointer vector must be allocated. The *expanded* nodes are

the internal nodes in the trie. To encode the two modes, a trie node uses the following data structure:

```
struct node_t {
        vector_t *v_ptr;
        char tag;
}
```

The pointer `v_ptr` will be set to zero for a *simple* node, or points at a pointer vector for an expanded node. The usage of the `tag` field is similar to that for a normal trie, i.e., a set tag implies that a key exists, made up of characters from the root to the node.

Suppose a trie contains L leaf nodes, pointers occupy 1 computer word and one `char` takes 1/4 computer word, then the saving arising from making the abovementioned distinction is

$$\begin{aligned} \Delta &= N(M + 1/4) - \{N(M + 1/4 + 1) - LM\} \\ &= LM - N \end{aligned}$$

computer words. Later it can be seen that the quantity LM is usually larger than N for real tries, which justifies making the distinction.

Coding Scheme of Compacted Tries

The new algorithm takes a distinct approach to encode the compacted trie. First, it numbers non-root internal nodes from 1 to $N - L$ and all leaf nodes as zero. The root node is always assigned number 1. Second, the algorithm decomposes the trie into a key set K and a record set R, where $|K| = |R| = N - 1$. Let ID_i be the number assigned to node i. Each non-null pointer in the original trie corresponds to one key in K and each node to one record in R. Consider the trie segment shown in Figure 4.20, and suppose the source node of pointer p is x and the sink node of p is y and the location of p in x is j. Then the key $k \in K$ for p is encoded as the pair $< j, ID_x >$. The record in R for y is a tuple $r =< j, b, ID_x, ID_y >$.

the internal nodes in the trie. To encode the two modes, a trie node uses the following data structure:

```
struct node_t {
        vector_t *v_ptr;
        char tag;
}
```

The pointer **v_ptr** will be set to zero for a *simple* node, or points at a pointer vector for an expanded node. The usage of the **tag** field is similar to that for a normal trie, i.e., a set tag implies that a key exists, made up of characters from the root to the node.

Suppose a trie contains L leaf nodes, pointers occupy 1 computer word and one **char** takes 1/4 computer word, then the saving arising from making the abovementioned distinction is

$$\begin{aligned} \Delta &= N(M + 1/4) - \{N(M + 1/4 + 1) - LM\} \\ &= LM - N \end{aligned}$$

computer words. Later it can be seen that the quantity LM is usually larger than N for real tries, which justifies making the distinction.

Coding Scheme of Compacted Tries

The new algorithm takes a distinct approach to encode the compacted trie. First, it numbers non-root internal nodes from 1 to $N - L$ and all leaf nodes as zero. The root node is always assigned number 1. Second, the algorithm decomposes the trie into a key set K and a record set R, where $|K| = |R| = N - 1$. Let ID_i be the number assigned to node i. Each non-null pointer in the original trie corresponds to one key in K and each node to one record in R. Consider the trie segment shown in Figure 4.20, and suppose the source node of pointer p is x and the sink node of p is y and the location of p in x is j. Then the key $k \in K$ for p is encoded as the pair $< j, ID_x >$. The record in R for y is a tuple $r =< j, b, ID_x, ID_y >$.

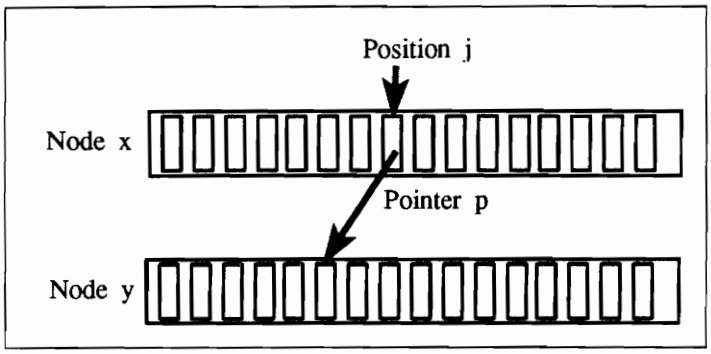


Figure 4.20: Two Nodes and One Pointer in a Trie

Here b is a bit set to 1 if the prefix from the root node to y is a word in the trie and to 0 otherwise. Since y is only pointed at by p, key k is uniquely associated with the record. Thus, a one-one mapping exists between K and R. If K is indexed by a MPHF, the record r can be quickly retrieved from the table if its key k is present.

The intuition behind the decomposition into key set and record set is as follows. The ideal compaction should remove all null pointers. One way to achieve this is to maintain all non-null pointers in a separate table. Each level of trie searching is then transformed to table lookup. The decomposition simply creates such a table, and gives ways to form keys from a sought after word and to look at appropriate records in the table. The following example illustrates the idea.

Let the word sought be $X = A_1\ A_2\ \ldots\ A_k$. We use A_i to denote the ASCII value of the ith character in X. To simulate checking location A_1 in the root node, key $< A_1, 1 >$ is formed, as the root node is numbered 1. Hashing of the key $< A_1, 1 >$ should obtain an address for a record of form $< j, b, ID_x, ID_y >$. Now, it is necessary to match j against A_1 and ID_x against 1. Success implies there is a pointer from the root to y for A_1 in the original trie, so a new key can be formed. Otherwise, X is not in the trie and the search should stop. Suppose A_1 passes the test, then the second key formed looks like $< A_2, ID_y >$. Repeat the process until either a mismatch is found or X is exhausted and the b field in the last record is 1. The latter case confirms X is in the trie.

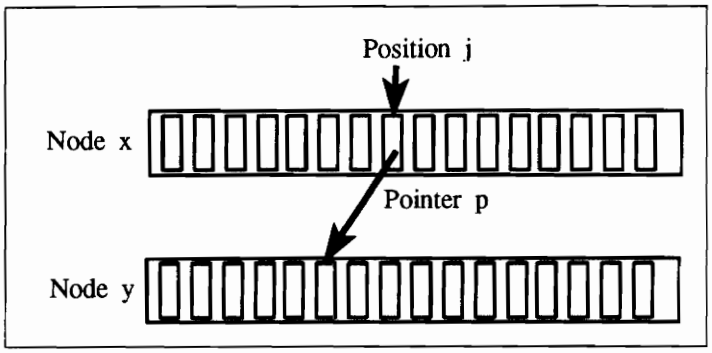


Figure 4.20: Two Nodes and One Pointer in a Trie

Here b is a bit set to 1 if the prefix from the root node to y is a word in the trie and to 0 otherwise. Since y is only pointed at by p, key k is uniquely associated with the record. Thus, a one-one mapping exists between K and R. If K is indexed by a MPHF, the record r can be quickly retrieved from the table if its key k is present.

The intuition behind the decomposition into key set and record set is as follows. The ideal compaction should remove all null pointers. One way to achieve this is to maintain all non-null pointers in a separate table. Each level of trie searching is then transformed to table lookup. The decomposition simply creates such a table, and gives ways to form keys from a sought after word and to look at appropriate records in the table. The following example illustrates the idea.

Let the word sought be $X = A_1\ A_2\ \ldots\ A_k$. We use A_i to denote the ASCII value of the ith character in X. To simulate checking location A_1 in the root node, key $< A_1, 1 >$ is formed, as the root node is numbered 1. Hashing of the key $< A_1, 1 >$ should obtain an address for a record of form $< j, b, ID_x, ID_y >$. Now, it is necessary to match j against A_1 and ID_x against 1. Success implies there is a pointer from the root to y for A_1 in the original trie, so a new key can be formed. Otherwise, X is not in the trie and the search should stop. Suppose A_1 passes the test, then the second key formed looks like $< A_2, ID_y >$. Repeat the process until either a mismatch is found or X is exhausted and the b field in the last record is 1. The latter case confirms X is in the trie.

In general, the compacted trie handles searching with ease and is of the same linear $O(k)$ time complexity, as those presented in [ASH84]. For the insertion and deletion of words, however, the capability of our method is limited. This is also true for the methods shown in [ASH84], which also are only suited to static trie applications.

The Degree of Approximation

The quality of the new method can be measured by the degree of approximation. However, as the MPHF algorithm is probabilistic, the measurement is in the probabilistic sense. For a record of form $< j, b, ID_x, ID_y >$, the components j, b, ID_x and ID_y take $\log_2 M$, 1, $\log_2 N$ and $\log_2 N$ bits, respectively. In total, the whole record set uses $(N-1)(\log_2 M + 2\log_2 N + 1)$ bits. Considering also the cost of the MPHFs, the approach uses about $(N-1)(\log_2 M + 2\log_2 N + 4)$ bits. The optimal compacted trie is $(N-1)(\log_2 N + 1)$ bits. The difference is $(N-1)(\log_2 M + \log_2 N + 3)$ bits. This difference is consistent across a wide range of N and M, as the MPHF algorithm works at a $2-3$ bits/key ratio for key sets of small to very large size. By calculating α, the approach yields $\alpha = 1 + \frac{\log_2 M+2}{\log_2 N+1}$, which is less than 2 for $\log_2 M + 1 < \log_2 N$ (a condition usually true for practical tries). Therefore, MPHF compaction is an $e-$approximation, where $e = 2$, for most tries. The utilization μ is $G(I) - F(I) \;/\; F(I) \leq 2/3$.

Empirical results

To test the method for real tries, words from the *Collins English Dictionary* [Han79] were arbitrarily selected. In the set of runs, the total memory upper bound was set to vary from 2 Mb to 20 Mb, in increments of 2Mb. This situation may arise when a trie is used to help check a text collection for controlled words. It is wise to limit the trie size so that it can fit into the main memory. Assuming $M = 128$ and a *simple* node takes 5 bytes and an *expanded* one $4 * 128 + 5 = 517$ bytes, N is roughly lying between 512 and 9,000 for the selected memory size range.

Table 4.11 shows the results of these runs. The first column gives the number of words

In general, the compacted trie handles searching with ease and is of the same linear $O(k)$ time complexity, as those presented in [ASH84]. For the insertion and deletion of words, however, the capability of our method is limited. This is also true for the methods shown in [ASH84], which also are only suited to static trie applications.

The Degree of Approximation

The quality of the new method can be measured by the degree of approximation. However, as the MPHF algorithm is probabilistic, the measurement is in the probabilistic sense. For a record of form $< j, b, ID_x, ID_y >$, the components j, b, ID_x and ID_y take $\log_2 M$, 1, $\log_2 N$ and $\log_2 N$ bits, respectively. In total, the whole record set uses $(N-1)(\log_2 M + 2\log_2 N + 1)$ bits. Considering also the cost of the MPHFs, the approach uses about $(N-1)(\log_2 M + 2\log_2 N + 4)$ bits. The optimal compacted trie is $(N-1)(\log_2 N + 1)$ bits. The difference is $(N-1)(\log_2 M + \log_2 N + 3)$ bits. This difference is consistent across a wide range of N and M, as the MPHF algorithm works at a $2-3$ bits/key ratio for key sets of small to very large size. By calculating α, the approach yields $\alpha = 1 + \frac{\log_2 M+2}{\log_2 N+1}$, which is less than 2 for $\log_2 M + 1 < \log_2 N$ (a condition usually true for practical tries). Therefore, MPHF compaction is an $e-$approximation, where $e = 2$, for most tries. The utilization μ is $G(I) - F(I) \;/\; F(I) \leq 2/3$.

Empirical results

To test the method for real tries, words from the *Collins English Dictionary* [Han79] were arbitrarily selected. In the set of runs, the total memory upper bound was set to vary from 2 Mb to 20 Mb, in increments of 2Mb. This situation may arise when a trie is used to help check a text collection for controlled words. It is wise to limit the trie size so that it can fit into the main memory. Assuming $M = 128$ and a *simple* node takes 5 bytes and an *expanded* one $4 * 128 + 5 = 517$ bytes, N is roughly lying between 512 and 9,000 for the selected memory size range.

Table 4.11 shows the results of these runs. The first column gives the number of words

in the trie. Columns 2 and 3 show the number of internal and leaf (external) nodes, respectively. Columns 4 and 5 give the memory consumption for the original and compacted tries. Columns 6 and 7 are the space utilization with respect to columns 4 and 5. The last column shows the α values. From the table it can be observed that:

- A trie data structure consumes huge amount of space even it has a moderate number of nodes. It is not practical for the number of nodes to exceed a few thousand for typical main memories.

- Space utilization of an uncompacted trie may increase slightly as the number of words grows larger. But as the table indicates, this ratio never surpasses 1% for the tested cases.

- Using the Faster MPHF algorithm can drastically reduce the space of a uncompacted trie. It is a practical approach in the sense that the final amount of space used is bounded from above by 3 times the optimal size, for tries where the condition $\log N > \log M + 1$ is met.

- The ratio $|F(I) - G(I)|/F(I)$ drops as the number of nodes increases, implying more compaction can be achieved for larger tries. This is because the function

$$\frac{|F(I) - G(I)|}{F(I)} = 1 - \frac{1}{2 + \frac{\log_2 M+2}{\log_2 N+1}}$$

  decreases when M is kept a constant and N grows larger. The minimal value is 0.5.

- The ratio of memory consumption of the uncompacted tries to the compacted tries is approximately 100 : 1. This is due to the fact that 4 byte pointers are used in uncompacted tries for maximal insert/delete flexibility while the same entities are represented in $\log_2 M + 2\log_2 N$ bits in compacted tries for maximal compaction.

Table 4.11: Comparison of Uncompact and Compact Tries

Words	No. of Nodes		Space Required		Utilization μ		Quality
	Internal	Leaf	Original (MB)	Compacted (KB)	Original	Compacted	$\frac{\lvert F(I)-G(I)\rvert}{F(I)}$
672	4049	669	1.99	7.60	0.919%	62.7%	1.68
1434	8099	1422	3.99	43.51	0.926%	62.0%	1.63
2258	12147	2225	5.99	67.76	0.931%	61.7%	1.60
3145	16195	3085	7.99	92.89	0.936%	61.4%	1.59
4045	20243	3960	9.99	118.55	0.940%	61.2%	1.58
4996	24291	4856	11.99	144.68	0.942%	61.1%	1.57
5976	28338	5767	13.99	171.17	0.945%	60.9%	1.56
6978	32386	6701	15.99	198.05	0.947%	60.8%	1.55
7952	36433	7616	17.99	225.05	0.948%	60.8%	1.55
8984	40481	8569	20.00	252.46	0.950%	60.7%	1.54

4.4.3 Technical Comparison

Binary string packing is a special case of array compaction; see [TY79] for the description of a compaction algorithm. Since bit strings (or equivalently rows in an array) are compacted string by string, the compact performance is very dependent on the number of bits and bit positions in the strings. As shown in [ASH84], sparser one-dimension arrays will be produced when M grows larger.

The MPHF compaction, on the other hand, considers each 1 bit in the string as an independent element throughout the packing process. Since the dependency relationship among 1 bits is removed, the algorithm has more freedom to move 1 bits around. Actually, the packing yields a 100% full table that contains all 1's. However, this tightest packing is achieved by paying costs to MPHF and branch information encoding. In the following paragraphs, we calculate the cost within the context of the trie data structure present in [ASH84]. We will show that the cost is a function of N, M and r.

The trie used in [ASH84] stores another set of pointers in trie nodes. Each pointer points in an entry in an auxiliary table where a unique suffix is kept. Suppose a pointer p is located at location M_i of the node N_i, and N_0, N_1, ..., N_i is the path from the trie root to N_i. Let the location in each node on the path be M_k for k in $[0, i]$. Then the word stored

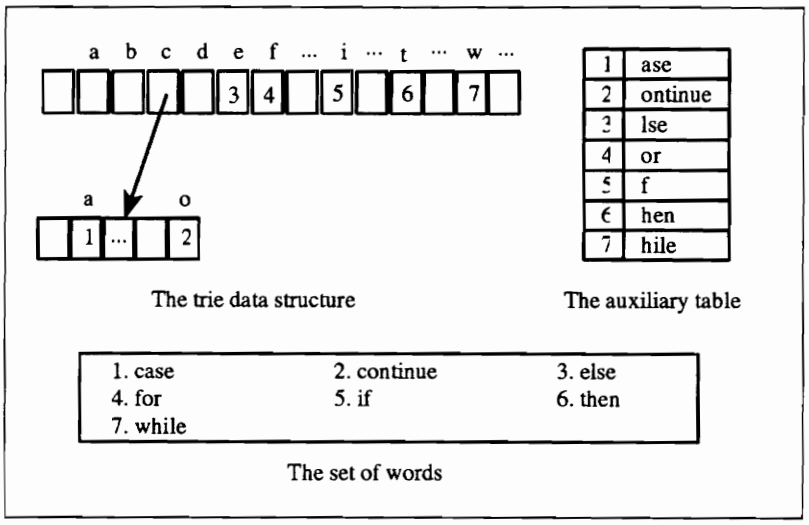


Figure 4.21: A Trie Example

in the trie is actually $M_0M_1M_2 \ldots M_i\underline{suf}$. The suffix $\underline{suf}$ is a string in the auxiliary table pointed at by p. Figure 4.21 illustrates such an example trie.

Let r be the total number of auxiliary table pointers. $N-1$ will be the number of intra-trie pointers, as there are N nodes in the trie. A MPHF compaction algorithm will set up two record tables for such a trie. The first table encodes the nodes and intra-trie pointers using the method presented in section 4.4.2, yielding a $(N-1)(3+\log_2 M+2\log_2 N)$ bits coding. The second table is for auxiliary table pointers. The table has r entries, each for an auxiliary pointer and of the form $< ID_x, Pos_j, Ptr >$. ID_x is the number assigned to the node in which the pointer exists. Pos_j is the position of the pointer within the node. Ptr is the actual pointer to the auxiliary table. Such an entry takes $r*(\log_2 N+\log_2 M+\log_2 r)$ bits to store. Also, it is necessary to using a faster MPHF to index the table on keys of form $< ID_x, Pos_j >$. The total cost is $(N-1)(3+\log_2 M+2\log_2 N)+r(3+\log_2 N+\log_2 M+\log_2 r)$ bits. The order is $O(max(r,N)\log_2 N)$.

4.4.4 Summary

In this section, a new trie compacting which consistently performs better than the methods described in [ASH84] is demonstrated. This is because the coding scheme of our method maps pointers and nodes onto a key and record space. The mapping destroys the aggregation of pointers inside nodes in the original tries. This is in contrast to a bit string compaction approach where the relative position of pointers within nodes is kept after compaction. As a result, our approach is capable of handling tries with different pointer distributions. The degree of compaction is only related to the number of nodes in the trie.

4.5 Minimal Perfect Hash Functions, Preserving Order

This section focuses on MPHFs that also have the property of preserving the order of the input key set. The work has been reported elsewhere in [FCDH91] and [FCDH90]. Because they are of special value for information retrieval applications, its usefulness is elaborated in the following paragraphs. To make it clear what is implied, consider Figure 4.22. A function must be obtained that maps keys, usually in the form of character strings or concatenations of several numeric fields, into hash table locations. In brief, the i^{th} key is mapped into the i^{th} hash table location.

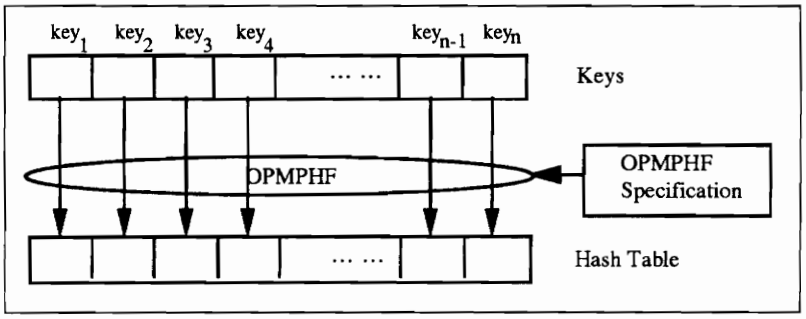


Figure 4.22: Order Preserving Minimal Perfect Hash Function

While there are numerous applications for the method, it is appropriate to consider two that are particularly well known and important for information retrieval. First, there is the dictionary. Here the object is to take a set of tokens or token strings (words, phrases, etc.) and allow rapid lookups to find associated information (number of postings of a term, the "concept number" for that entry, pointers to inverted file lists, etc.). If Order Preserving Minimal Perfect Hash Functions (OPMPHFs) can be used for this purpose, in one disk access the record of any dictionary item can be identified, and it is possible to rapidly find previous or subsequent entries as well. Thus, the dictionary can be kept in lexicographic order, and can be read sequentially or accessed directly. A second application is for accessing inverted file data. For a given term ID (identifier), it is usually necessary to find the number of postings, that is the number of documents in which the term occurs, and then to find the list of all those occurrences. All of this information has been included in a single

file accessible by an OPMPHF. Normally, for a given term ID, we obtain the document and frequency (of that term in that document) pairs for all occurrences. Assuming that document numbers have value at least 1, we use the simple trick of storing the postings data in the frequency field of an entry that has a given term ID and document number set to 0. For a given term ID, a key can be formed by concatenating the value 0 to it. We find the postings in one seek, and read the document-frequency pairs that appear directly after. Various methods using unnormalized forms of the data are possible to effect space savings. In fact, the associated value for a key that can be located using OPMPHF can actually be an arbitrary value so that variable length records can be directly addressed.

4.5.1 Three Methods to Find OPMPHFs

There are at least three ways to obtain an OPMPHF. The first two are straightforward extensions of bipartite graph and faster MPHF algorithms. The third method takes a different approach by compactly encoding pointers in the function specification space.

Method 1: Acyclic Graphs

The first method, by using the acyclic technique, involves constructing a bipartite graph G sufficiently large so that no cycles are present. This extends the work described in [FCHD90], and is based on the use of a large ratio $(2r/n)$ which makes the probability of having a cycle approach 0 (see proof in section 4.5.3). If there are no cycles, g values can be selected that preserve any a priori key order.

The algorithm for this method is basically the same as that described in section 4.3 throughout the Mapping and Ordering phases. But because G is acyclic, an ordering of non-zero degree vertices v is obtained through some kind of edge traversal (e.g., depth-first or breadth-first) of all components in G. Thus, in Figure 4.23, which shows an acyclic bipartite graph, an ordering obtained by depth-first traversal of first the left connected component and then the right might give the vertex sequence $(VS) : [v_1, v_5, v_0, v_2, v_6, v_3, v_7]$. The corresponding levels of edges are given in the edge sequence: $[\{\}, \{e_1\}, \{e_0\}, \{e_3\}, \{e_2\}, \{\}, \{e_4\}]$.

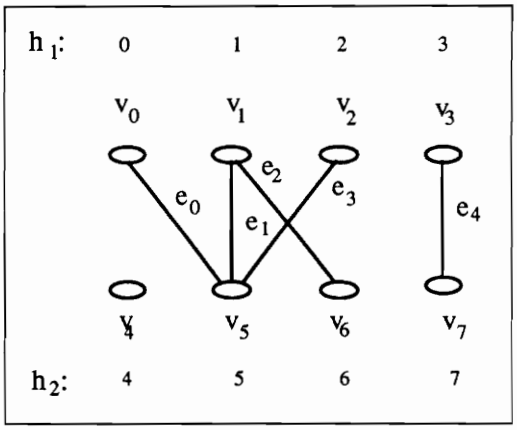


Figure 4.23: A Cycle Free Bipartite Graph

Notice that in this example, each level has at most one edge, which is only possible if G is acyclic.

During the Searching phase, a single pass through the ordering can determine g values for all keys in a manner that preserves the original key ordering. This is possible since with only one edge being handled at each level, there are no interdependencies that would restrict the g value assignments.

Although this approach is simple, it is only practical if a small acyclic graph can be found. Section 4.5.3 gives a detailed probabilistic account of the expected number of cycles in G, as a function of the ratio $2r/n$. If the average number of cycles, E(Y), approaches 0, then by Chebyshev's inequality

$$\mathrm{P}(Y \geq t) \leq \mathrm{E}(Y)/t,$$

so the probability of a particular graph having cycles approaches 0. Thus, for sufficiently large ratio (i.e., $ratio \rightarrow \infty$), it will be very unlikely that G will have cycles. However, this ratio and the size of G is very much larger than values required in the other two methods described below.

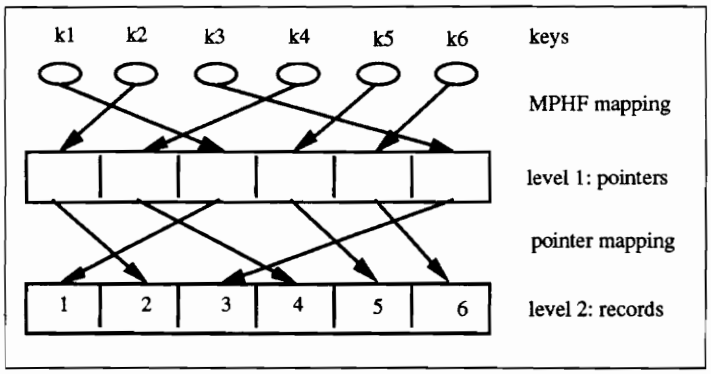


Figure 4.24: A Two Level OPMPHF Scheme

Method 2: Two Level Hashing

The second one is to use two level hashing. Here the MPHF computed through the methods discussed in section 4.2 or 4.3 is in the first level and an array of pointers is in the second. A hash value from the MPHF addresses the second level where the real locations of records are kept. The records are arranged in the desired order. If the bipartite graph MPHF algorithm is used, the first level takes $2r$ computer words, and at the second, n computer words for the OPMPHF. For large key sets, $2r \cong 0.4n$ is possible and feasible. Thus this method typically will use $1.4n$ computer words. If the faster MPHF algorithm is used, then the total space will reduce to $n(\log n + b)$ bits, for b being in 2 to 3. Figure 4.24 illustrates the two level hashing scheme.

Method 3: Using Indirection

The last method is based on the idea of using a bipartite graph G to store the additional information required to specify a MPHF that also preserves order. For n keys, if our graph has somewhat more than n vertices (i.e., if ratio > 1), then there should be enough room to specify the OPMPHF. In a random graph of this size, a significant number of vertices will have zero degree. The solution is to use indirection for some of the keys; this solution can be viewed as a combination of Methods 1 and 2. This means that some keys will be

mapped using indirection, in this case using the composition

$$order(k) = h(k) = g\Big(\{h_0(k) + g(h_1(k)) + g(h_2(k))\} \bmod 2r\Big)$$

while on the other hand, the desired location of a key that is, as before, found directly is determined by:

$$order(k) = h(k) = \{h_0(k) + g(h_1(k)) + g(h_2(k))\} \bmod n.$$

Note the g function is used in two ways, one for direct keys and the other for keys that are handled through indirection. In any case, $h(k)$ preserves the original ordering of keys (i.e., $order(k)$). As the entire g value can be kept in memory for moderate size key sets, indirection reference will not cause measurable performance overhead.

Consider the distribution of X, the degree of a vertex in G. The actual distribution is binomial and can be approximated by the Poisson:

$$\begin{aligned} \mathrm{E}(X = d) &= \{2r\, e^{-n/r}(n/r)^d\}/d! \\ \mathrm{E}(X = 0) &= 2r\, e^{-n/r} \end{aligned}$$

When $r = n$, about 13.5% of the vertices have zero-degree. If these zero-degree vertices can be used to record order information for a significant number of keys, then it is not necessary for G to be acyclic to generate an OPMPHF. In addition, one vertex from each acyclic component in G can accept indirect keys. Figure 4.25 is a brief demonstration of the idea. Note that keys associated with edges e_0 and e_1 can be indirectly hashed into zero-degree vertices v_6 and v_2. In this example it is not necessary, but if another indirect key had to be accommodated, one vertex from the set $\{v_1, v_5, v_3, v_7\}$ could be used.

In general, an edge (key) is indirectly hashed when that situation is described by information associated with its two vertices, given by $h_1(k)$ and $h_2(k)$. Usually, indirection can be indicated using one bit per vertex that is decided at MPHF building time and that is subsequently kept for use during function application time. The following section describes a particular encoding scheme.

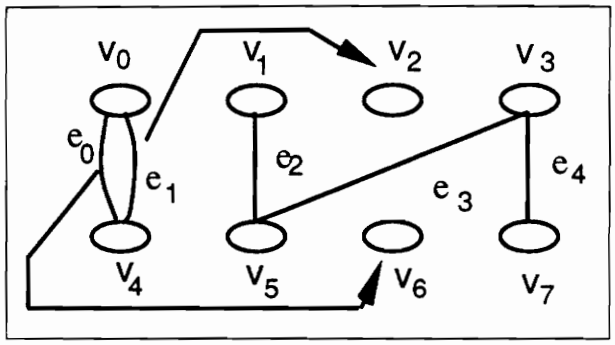


Figure 4.25: Zero Degree Vertices are Useful

4.5.2 The OPMPHF Algorithm and Data Structures

This section outlines an algorithm using one indirection bit, which is an extension of the one in section 4.2 used to find MPHFs. Recall that the OPMPHF class is:

$$h(k) = g\Big(\{h_0(k) + g(h_1(k)) + g(h_2(k))\}\ mod\ 2r\Big),$$

when the indirection bit associated with the two vertices for this key have the same value, and otherwise uses

$$h(k) = \{h_0(k) + g(h_1(k)) + g(h_2(k))\}\ mod\ n.$$

The algorithm for selecting proper g values and setting indirection bits for vertices in G consists of the three steps: Mapping, Ordering and Searching.

The Mapping Step

This step is essentially identical to that discussed in section 4.2. The only addition is that the indirection bit must be included in the vertex data structure.

The basic concept is to generate unique triples of the form $(h_0(k), h_1(k), h_2(k))$ for all keys k. $h_0(), h_1(), h_2()$ are simple pseudo-random functions. Figure 4.26(a) illustrates one assignment of h_0, h_1, and h_2 values to keys yielding the graph G shown in Figure 4.26(b). Since the final hash function should be perfect, all triples are distinct.

The h_0, h_1 and h_2 values for all keys are entered into an array *edge* defined as

```
array of [0...n − 1] of record
    h0, h1, h2:  integer;
    nextedge1:  integer;
    nextedge2:  integer;
    order:  integer
```

Here the combination h_0, h_1, h_2 field contains the triple. The nextedge$_i$ field ($i = 1, 2$) indicates the next entry in the *edge* array with similar h_i value to the current entry. It is utilized to link together all edges joined to a vertex. The order field is the desired hash location of a key.

The g function is recorded in another array *vertex* defined as follows.

```
vertex:  array of [0...2r − 1] of record
    g:  integer;
    mark:  bit;
    firstedge:  integer;
    degree:  integer
```

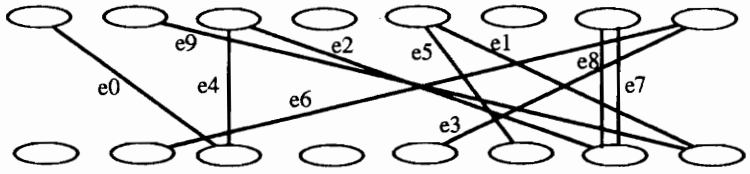
Here $vertex[i].g$ records the final g value for $h_1(k) = i$ if i is in $[0, r-1]$ or the final g value for $h_2(k) = i$ if i is in $[r, 2r-1]$. The mark field contains a bit of indirection information, as given above for either $h_1(k)$ or $h_2(k)$. Also, $vertex[i].firstedge$ is the header for a singly-linked list of the keys having $h_1(k) = i$ if i is in $[0, r-1]$ or the keys having $h_2(k) = i$ if i is in $[r, 2r-1]$. The firstedge field actually points at an entry in the edge array indicating the start of the list and nextedge$_i$ (for $i = 1, 2$) there connects to the rest of the list. The degree field is the length of the list or equivalently the degree of the vertex. Thus, the *edge* and *vertex* arrays give a representation of a bipartite graph G.

The Ordering Step

In the Ordering step it is necessary to obtain a good vertex sequence VS for use later in the Searching step. Specifically, VS specifies a sequence of the vertices so that, during

Key	h0	h1	h2	Edges in G
x-rays	0	0	10	e0
Euclidean	6	4	15	e1
ethyl ether	9	2	14	e2
Clouet	0	7	12	e3
Bulwer-Lytton	4	2	10	e4
dentifrice	0	4	13	e5
Lagomorpha	8	7	9	e6
Chungking	7	6	14	e7
quibbles	4	6	14	e8
Han Cities	2	1	15	e9

(a) The Key Set



(b) The Bipartite Graph

Figure 4.26: A Key Set and its Dependency Bipartite Graph G

searching, each related set of edges can be processed independently. For a given vertex v_i in the ordering, these related edges contained in $K(v_i)$ (i.e., at that level) are the backward edges, going to vertices that appear earlier in the ordering. Taking the bipartite graph in Figure 4.26(b) as an example, one of the 16! possible vertex sequences is

$$VS = [v_6, v_{14}, v_2, v_{10}, v_0, v_4, v_{15}, v_1, v_9, v_7, v_{12}]$$

with corresponding levels or edge sets

$$\begin{aligned}
K(v_6) &= \{\} \\
K(v_{14}) &= \{e_7, e_8\} \\
K(v_2) &= \{e_2\} \\
K(v_{10}) &= \{e_4\} \\
K(v_0) &= \{e_0\} \\
K(v_{13}) &= \{\} \\
K(v_4) &= \{e_5\} \\
K(v_{15}) &= \{e_1\} \\
K(v_1) &= \{e_9\} \\
K(v_9) &= \{\} \\
K(v_7) &= \{e_6\} \\
K(v_{12}) &= \{e_3\}
\end{aligned}$$

The graph constructed from vertices in VS plus edges in G is essentially a redrawing of G that excludes zero-degree vertices, as can be seen in Figure 4.27.

Finding a proper VS requires that vertices with many backward edges (i.e., with large $|K(v_i)|$) be processed first. The other key issue in finding a proper VS is to handle the fact that some edges will be indirect and some direct. Since the assignment of a g value for vertex v_i fully determines the hash addresses of all keys in $K(v_i)$, given that the g values of each previously visited vertex have been set, it is in general true that at most one key in

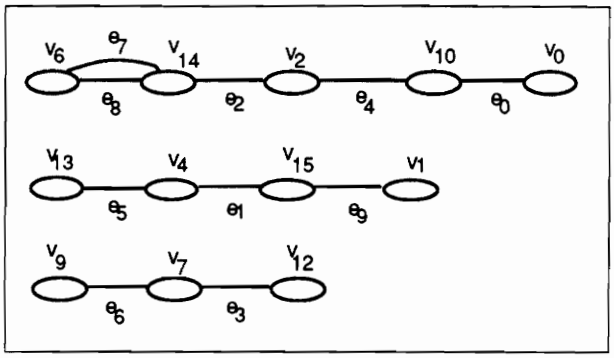


Figure 4.27: Redrawing of G Based on a VS that Excludes Zero-degree Vertices

$K(v_i)$ can be directly hashed for a fixed g value at v_i. Thus, exactly one such key should be determined for $K(v_i)$, if the Searching step is to proceed properly. In the scheme proposed, one bit (namely the mark bit) is attached to each vertex for the purpose. Then, when the hashing function is used, for key k only its two indirection bits (stored in primary memory and attached to the two vertices $h_1(k)$ and $h_2(k)$) need to be checked.

Given the need to quickly find the proper VS and to decide the proper indirection bits for vertices in VS, it is essential to obtain hints from the properties of the $K(v_i)$, such as their size. For a key k in a level where $|K(v_i)| = 1$ and v_s is the vertex at the other side, the key can be directly hashed by setting the g value at v_i to

$$g(v_i) = (order(k) - h_0(k) - g(v_s)) \ mod \ n.$$

Recall that $order(k)$ refers to the desired hash address for key k.

For keys in a level containing more than one key (i.e., $|K(v_i)| > 1$), since at most one key can be direct, hashing of the other keys requires indirection. Since in our scheme indirect hashing is indicated by the indirection bits, all such keys have those bits set accordingly and thus are indirectly hashed. In summary, a proper VS will be one that tends to maximize the number of v_i's with $|K(v_i)| = 1$ and, hence, to minimize the number of v_i's with $|K(v_i)| > 1$.

A practical way to obtain such a VS is to take into account the structure of G. Following standard graph terminology, $E(G)$ refers to the set of edges in G and $V(G)$ the set of vertices. Figure 4.28 lists other terms. Special attention must be given, though, to each connected

component C. Clearly, edges in a tree component (denoted by AC to stand for "acyclic component") of G can be directly hashed if their vertices are included in VS by a simple depth or breadth first traversal as described in section 4.5.1. For example, in the second and third components in Figure 4.27, all five edges are direct. Since any vertex in an AC can be the root for a traversal, the ordering of vertices for AC is not performed until the Searching step. At that time, the single vertex of the AC that has been used for indirection of a previous key is made the root of the traversal.

For a cyclic component (denoted by CC) such as the larger component at the top of Figure 4.27, two types of edges are distinguishable. First there are π-edges that are cut edges such that removal of each of them leaves at least one acyclic subcomponent. Denote as CC_{trees} the maximal subgraph of CC whose edges are π edges. Note that in Figure 4.26(b), $V(CC_{trees}) = \{v_0, v_2, v_{10}, v_{14}\}$ and $E(CC_{trees}) = \{e_0, e_2, e_4\}$. All edges in a CC_{trees} can be hashed directly if a vertex visiting strategy similar to that for a tree component AC is used, and the roots for visiting are vertices shared by π-edges and non π-edges. Since the existence of g values at the root is the only precondition for assignment of g values to other vertices in CC_{trees}, edges in CC_{trees} should be assigned after the non π-edges are handled.

Second, there are non π-edges of CC. Non-π-edges can be direct or indirect, based on a specific ordering of the vertices that these edges are connected to. Denote as $CC_{\overline{trees}}$ the subgraph of CC after π-edges are removed. In Figure 4.26(b), there is only indirect non π-edges with $V(CC_{\overline{trees}}) = \{v_6, v_{14}\}$ and $E_{CC_{\overline{trees}}} = \{e_7, e_8\}$. Intuitively, keys where $|K(v_i)| = 1$ should be direct and those where $|K(v_i)| > 1$ should be indirect. However, due to the way in which the indirection bits are set, some keys where $|K(v_i)| = 1$ may also be indirect.

In summary, the strategy to obtain a good VS involves first identifying ACs, $CC_{trees}s$ and $CC_{\overline{trees}}s$. The vertices in $CC_{\overline{trees}}s$, then in CC_{trees} and finally in ACs are ordered. The implementation of the algorithm combines the ordering and searching for $CC_{trees}s$ and ACs in the Searching step to save one traversal of edges in them. In the Ordering step for vertices in $CC_{\overline{trees}}s$, a vertex whose $K(v_i)$ set is (currently) larger is chosen next in the

$E(G) =$	edges of graph G
$V(G) =$	vertices of graph G
$C =$	component in G
$AC =$	C that is acyclic
$CC =$	component containing one or more cycles
$CC_{trees} =$	maximal subgraph of CC_{trees} containing only cut edges, each cutting CC into at least one acyclic subcomponent
$CC_{\overline{trees}} =$	$CC - CC_{trees}$

Figure 4.28: Graph Notation for OPMPHF Algorithm

ordering over a vertex whose $K(v_i)$ set is (currently) smaller. The ordering of vertices in $C_{trees}s$ and ACs is done purely through tree traversals.

The number of vertices of G for a fixed key set is an important factor affecting the quality of VS. First, $|V(G)|$ is theoretically bounded below by the number of keys n. Any G with smaller than n vertices has little probability of producing an OPMPHF. For G with $|V(G)| > n$, a tradeoff lies between the size of the OPMPHF and the ease of finding such an OPMPHF. Let S be the set of indirect keys. Then if G is large, $|S|$ becomes small implying both an easier indirect fit for S and a larger OPMPHF. On the other hand, a small G will result in a large S, increasing the difficulty of finding an OPMPHF, though if one is found, it will be rather small. The final constraint is that $|S|$ must be less than the total number of ACs.

Having obtained $VS(CC)$, indirection bits are marked for all vertices in the sequence. Though not necessarily yielding an optimal marking in terms of generating a minimal number of indirect edges, the method achieves satisfactory results. Suppose v_i is marked for $K(v_i)$. Without loss of generality, assume v_i is in the first side of G and v_j is a previously marked vertex adjacent to v_i.

The mark bit v_i.mark is determined in one scan of $VS(CC)$ with the heuristic rule:

a) v_i.mark = 1 if $|K(v_i)| = 0$; or

b) v_i.mark = 1 if v_j.mark = 0 and $|K(v_i)| = 1$; or

c) v_i.mark = 0 if v_j.mark = 1 and $|K(v_i)| = 1$; or

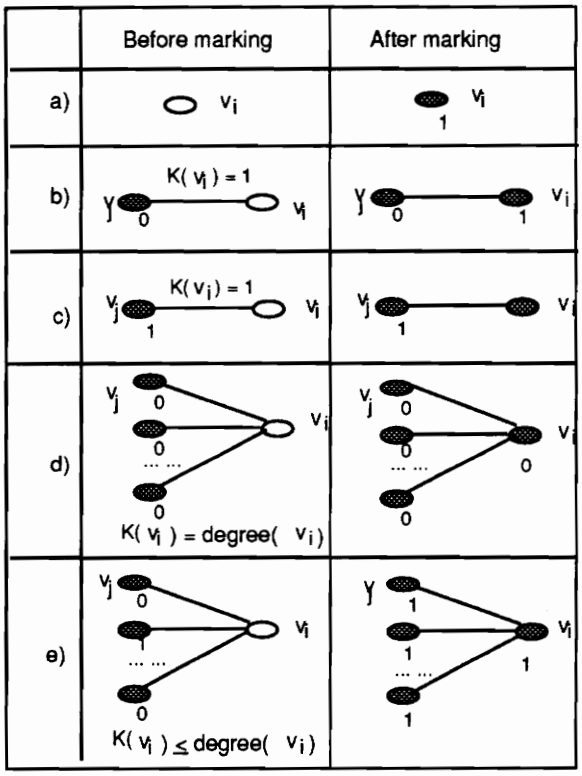


Figure 4.29: Illustration of Marking Process

d) v_i.mark = 0 if $|K(v_i)| > 1$ and all v_j.mark = 0 and $|K(v_i)| = v_i$.degree; or

e) v_i.mark = 1 if $|K(v_i)| > 1$ and set all v_j.mark = 1 if previously 0.

Figure 4.29 illustrates the marking scheme where shaded vertices are marked and the unshaded vertices are to be marked.

If v_i is on the second side, h_1 and h_2 can be switched for steps a) to e). A simple induction proof on the length i of $VS(CC)$ shows that (1) a direct edge only appears in a $|K(v_i)| = 1$ level if that edge is not forced to be indirect by e); (2) all edges in levels with $|K(v_i)| > 1$ are indirect. Here a key k is indirect if its two mark bits are identical, and direct otherwise.

The Ordering phase performs its job in three sub-steps. In Step (1), all components in G are identified by assigning component IDs (*cids*). Stack data structure VSTACK keeps

all unidentified vertices adjacent to at least one identified vertex. Each time a vertex is popped from VSTACK, it gets a *cid* and its adjacent unidentified vertices are pushed onto VSTACK. After the identification process, all zero-degree vertices will get a 0 *cid* and all other vertices get *cid*s greater than 0. Step (1) can be finished in $O(n)$ time because each non-zero vertex is in VSTACK only once, and pushing and popping operations take constant time.

Steps (2) and (3) recognize $E(CC_{trees})$ in each component by manipulating the degree field. Initially, Step (2) collects all vertices of degree one into VSTACK and sets their degree field to zero. Afterwards, Step (3) takes VSTACK and tries to find more vertices whose degree could be reduced to one. Each time a vertex is popped, the degree of all its adjacent vertices is decreased. If some of them turn into degree one vertices, then they are pushed onto VSTACK. The process will continue until no more vertices can have their degree values decreased. It can be seen that each time a vertex is popped, an edge in $E(CC_{trees})$ is found that connects the vertex to some previously popped vertex. The final non-zero vertices left are just those in $V(CC_{\overline{trees}})$. Since at most n vertices will get into VSTACK and each stack operation takes constant time, steps (2) and (3) together use $O(n)$ time.

Next, the vertices in $V(CC_{\overline{trees}})$ are subjected to an ordering in Step (4) to generate a vertex sequence $VS(CC_{\overline{trees}})$. In generating $VS(CC_{\overline{trees}})$, Step (4) uses a heap VHEAP to record vertices out of which a vertex with maximal degree is always chosen as the next vertex to be put into the sequence. The usage of VHEAP is analogous to Prim's algorithm for building a minimum spanning tree. Due to the small vertex degrees in G, step (4) takes $O(n)$ time, on average, to finish the ordering.

Based on $VS(CC_{\overline{trees}})$, Step (5) marks all vertices in the sequence to maximize the number of direct keys in $|K(v)| = 1$ levels, and forces all keys in $|K(v)| > 1$ levels to be indirect. Step (5) is linear because the number of visits to vertices in $VS(CC_{\overline{trees}})$ is bounded by the total of the degree values of those vertices.

The Searching Step

The Searching step determines the g value for each vertex so as to produce an OPMPHF. The job is done in two sub-steps. First, g values for all vertices in the $VS(CC_{\overline{trees}})$ generated by the Ordering step are decided. These g values will in turn hash all keys in $E_{CC_{\overline{trees}}}$ to vertices in ACs where they find their desired hash values. Then all the edges in $E(CC_{trees})$ and $E(AC)$ are processed to finish the searching.

The Searching phase contains 5 steps. Step (1) straightforwardly assigns g values for $VS(CC_{\overline{trees}})$. The random probe sequence $s_0, s_1, \ldots, s_{2r-1}$, a random permutation of the set $[0 \ldots 2r-1]$, gives an ordered list of potential g values to test for each vertex. Step (1) classifies three kinds of v_i in the assignment: $|K(v_i)| = 0, |K(v_i)| = 1$ and each k in $K(v_i)$ is direct, or $|K(v_i)| > 0$ otherwise. Each case is treated separately. Step (1) will use $O(n)$ time for a successful assignment. For the rare case when there is no satisfactory g value for some vertex, we start another run of the Mapping, Ordering and Searching steps.

Step (2) fits edges in $CC_{trees}s$, by a depth-first traversal. The last two steps (3) and (4) are for edges in ACs with traversal root vertices either fixed during Step (1) or in ACs that have accepted no indirect edges. Step (3) can be done in linear time. Since only one edge is directly hashed during each visit of a vertex, steps (2), (3) and (4) cannot fail.

An Example

In this section an example is shown for finding an OPMPHF for the 10 key set listed in Figure 4.26(a) and the corresponding bipartite graph in Figure 4.26(b). It can be seen from Figure 4.26(b) that G has one CC with $V(CC) = \{v_0, v_2, v_6, v_{10}, v_{14}\}$ and $E(CC) = \{e_0, e_2, e_4, e_7, e_8\}$. G also has two trees AC_1 and AC_2 consisting of vertices $V(AC_1) = \{v_1, v_4, v_{13}, v_{15}\}$ and edges $E(AC_1) = \{e_1, e_5, e_9\}$ in AC_1, and vertices $V(AC_2) = \{v_7, v_9, v_{12}\}$ and edges $E(AC_2) = \{e_3, e_6\}$ in AC_2.

When the Ordering phase is carried out for G, it identifies CC, AC_1 and AC_2 during Step (1) , and truncates π-edges in CC in steps (2) and (3), leaving a sub-graph $CC_{\overline{trees}}$ which has

Table 4.12: g Value Assignment

vertex	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
g value	5	0	5	0	8	0	8	7	0	1	5	0	6	7	8	7
mark bit	0	0	0	1	0	1	1	0	1	1	1	1	1	1	1	1

Table 4.13: The Keys and Their Final Hash Addresses

key	h_0	h_1	h_2	$h(k)$
x-rays	0	0	10	0+5+5 (mod 10) = 0
Euclidean	6	4	15	6+8+7 (mod 10) = 1
ethyl ether	9	2	14	9+5+8 (mod 10) = 2
Clouet	0	7	12	0+7+6 (mod 10) = 3
Bulwer-Lytton	4	2	10	4+5+5 (mod 10) = 4
dentifrice	0	4	13	0+8+7 (mod 10) = 5
Lagomorpha	8	7	9	8+7+1 (mod 10) = 6
Chungking	7	6	14	7+7+8 (mod 16) = 7, g(7) = 7
quibbles	4	6	14	4+8+8 (mod 16) = 4, g(4) = 8
Han Cities	2	1	15	2+0+7 (mod 10) = 9

two edges $\{e_7, e_8\}$. In Step (4), vertices adjacent to these two edges are subject to ordering, producing a vertex sequence $VS(CC_{\overline{trees}}) = \{v_6, v_{14}\}$. $VS(CC_{\overline{trees}})$ is immediately involved in a marking process in Step (5), starting at v_{14}. Since $K(v_{14}) = 0$, $v_{14}.bit = 1$. v_6 obtains the same mark (bit 1) because $K(v_6)$ is of size 2 and v_{14} has been assigned bit 1.

During the Searching phase, g values will be assigned first to vertices in $VS(CC_{\overline{trees}})$ in Step (1). v_{14} gets a random number 8. v_6 gets 3 so that two edges sharing vertices v_6 and v_{14} can be indirectly hashed to vertices v_7 and v_4. The remaining 8 edges are all direct. Vertices v_2, v_{10} and v_0 will obtain their g values in Step (2); they are all 5. Since neither AC_1 nor AC_2 has accepted any indirect edges, they are processed in Step (4). The final g assignment for all vertices is illustrated in Table 4.12. To validate the OPMPHF based on the ranking of occurrence of keys in Figure 4.26(a), Table 4.12 lists the h for each key.

4.5.3 Analysis and Experimental Validation

In this section, the space lower bound and two random variables representing properties of G are analyzed. The lower bound is interesting because it gives a lower limit on the number of nodes G should have. The two random variables are interesting because they link to the upper bound on the number of nodes in G.

A Lower Bound on the Size of OPPHFs

Following the definition of a (N, m, n) perfect class of hash functions in [Meh82], a (N, m, n) *order-preserving perfect class H of OPPHFs* is defined as a set of functions h

$$h : [0 .. N-1] \rightarrow [0 .. m-1]$$

such that for any permutation of any subset S in N of size $|S| = n$, there is an h in H such that h is an OPPHF for the permutation.

Theorem.

$$|H| \geq \frac{\binom{N}{n} n!}{\left(\frac{N}{m}\right)^n \binom{m}{n}}.$$

The proof is based on an argument similar to that found in [Meh82], in proving the lower bound for the (N, m, n) perfect class of PHFs.

Proof: Clearly, there are $\binom{N}{n}$ distinct subsets in $[0 .. N-1]$, each of size n. For each such subset S, there are $n!$ permutations (i.e., $n!$ different orderings). We need to show at most $\left(\frac{N}{m}\right)^n \binom{m}{n}$ permutations out of the total $\binom{N}{n} n!$ can be order preserving and hashed by a single fixed h in H.

Let $S = \{s_1, s_2, \ldots, s_n\}$ be a subset of size n that is hashed perfectly (that is, one-to-one) by h. Denote $h(s_i)$ by t_i. Then $T = \{t_1, t_2, \ldots, t_n\}$ is one of the $\binom{m}{n}$ subsets of $[0 .. m-1]$ of size n.

Let K be the number of subsets of size n from $[0 .. N-1]$ that are hashed perfectly by

h and that have image T. Let $|h^{-1}(t_i)| = k_i$. Then

$$K = \prod_{i=1}^{n} k_i$$

while

$$N \geq P = \sum_{i=1}^{n} k_i.$$

K is maximized when all the k_i's are equal, i.e., when $K = (P/n)^n$.

Now let T_j, $j = 1, \ldots, \binom{m}{n}$ be the possible choices of T, and let P_j denote the corresponding value of P. The total number of sets that are hashed perfectly by h is

$$Q = \sum_{j=1}^{\binom{m}{n}} (P_j/n)^n$$

This is maximized when all the P_j's are equal. This occurs when the inverse images under h of each element of $[0 \,..\, m-1]$ have equal size, that is, N/m. Then each $P_j = Nn/m$ and

$$Q = \binom{m}{n} \left(\frac{N}{m}\right)^n.$$

From this, the theorem follows.

In the case of our OPMPHF scheme, $n = m$ and $N = nr^2$. Thus

$$|H| \geq \frac{\binom{nr^2}{n}\, n!}{(r^2)^n}$$

Using the asymptotic estimate $\binom{N}{n} \approx \frac{N^m}{n!}$

$$|H| \approx \frac{(nr^2)^n}{r^{2n}} = n^n$$

or $\log_2 |H| \approx n \log_2 n$. Therefore, $O(n \log_2 n)$ bits of space are required for $|h|$ or, equivalently, the number of g values should be at least n.

Average Number of Cycles

In the following, the expected number of cycles in G, a bipartite graph having $2r$ vertices on each side and having n random edges, is computed. Let $\Pr(2i)$ be the probability of having a cycle of length $2i$ formed in a particular vertex set of $2i$ vertices, with i (where $i \geq 2$) vertices being on each side. There are $i!(i-1)!/2$ ways to form distinct cycles out of these $2i$ vertices and $\binom{n}{2i}(2i)!$ ways to select $2i$ edges to form such a cycle. The remaining $n-2i$ edges can go into G in $(r^2)^{n-2i}$ different ways. Thus in total there are $i!(i-1)!/2 \cdot \binom{n}{2i} \cdot (2i)! \cdot (r^2)^{n-2i}$ ways to form a cycle of length $2i$. Given that there are a total of $(r^2)^n$ possibilities,

$$\begin{aligned} \Pr(2i) &= \frac{\frac{i!(i-1)!}{2} \cdot \binom{n}{2i} \cdot (2i)! \cdot (r^2)^{n-2i}}{(r^2)^n} \\ &= \frac{i!(i-1)! \cdot \binom{n}{2i} \cdot (2i)!}{2r^{4i}} \end{aligned}$$

For the case $i = 1$,

$$\begin{aligned} \Pr(2) &= \frac{\binom{n}{2} \cdot (r^2)^{n-2}}{(r^2)^n} \\ &= \frac{n(n-1)}{2r^4} \end{aligned}$$

Let Z_{ij} be an indicator random variable. $Z_{ij} = 1$ if there is a $2i$ edge cycle in the j^{th} vertex set of $2i$ vertices, $Z_{ij} = 0$ otherwise. Clearly, there are $\binom{r}{i}^2$ such sets in G. Each vertex set has the same probability of having $2i$ edge cycles.

Let X_i be a random variable counting the number of $2i$ edge cycles in G.

$$X_i = \sum_{j=1}^{\binom{r}{i}^2} Z_{ij} = \binom{r}{i}^2 \cdot \Pr(2i).$$

Define $Y_c = \sum_{i=1}^{r} X_i$ as another random variable counting the number of cycles in G of length from 2 to $min(2r, n)$.

$$
\begin{aligned}
\mathrm{E}(Y_c) &= \sum_{i=1}^{r} \mathrm{E}(X_i) \\
&= \sum_{i=1}^{min(2r,n)} \binom{r}{i}^2 \cdot \Pr(2i) \\
&= \binom{r}{1}^2 \cdot \Pr(2) + \sum_{i=2}^{min(2r,n)} \binom{r}{i}^2 \cdot \frac{i! \cdot (i-1)! \cdot (2i)!}{2 \cdot r^{4i}} \cdot \binom{n}{2i} \\
&\approx \frac{n(n-1)}{2r^2} + \sum_{i=2}^{min(2r,n)} \left(\frac{(r)_i}{i!} \cdot e^{-\frac{i^2}{2r}} \right)^2 \cdot \frac{i! \cdot (i-1)! \cdot (2i)!}{2 \cdot r^{4i}} \cdot \left(\frac{n^{2i}}{(2i)!} \cdot e^{-\frac{(2i)^2}{2n}} \right) \\
&= \frac{n(n-1)}{2r^2} + \sum_{i=2}^{min(2r,n)} \frac{1}{2i} \cdot \left(\frac{n}{r} \right)^{2i} \cdot e^{-i^2 \cdot (\frac{1}{r} + \frac{2}{n})} \qquad (4.1) \\
&\leq \sum_{i=1}^{min(2r,n)} \frac{1}{2} \left(\frac{n}{r} \right)^{2i} \\
&\leq \sum_{i=1}^{\infty} \frac{1}{2} \left(\frac{n}{r} \right)^{2i} \\
&= \frac{1}{2} \cdot \frac{\left(\frac{n}{r}\right)^2}{1 - \left(\frac{n}{r}\right)^2}.
\end{aligned}
$$

Then,

$$
\mathrm{E}(Y_c) \leq \frac{1}{\left(\frac{r}{n}\right)^2 - 1}.
$$

When $\frac{r}{n} \to \infty$ (e.g., $r = n \log n$), then $\mathrm{E}(Y_c) \to 0$.

Table 4.14 shows the percentage of G's with number of edges fixed at 1024. The number of vertices varies from 2252 to 3276 with the ratio ranging from 2.2 to 3.2.

Average Number of Trees

This subsection includes a derivation of a formula counting the number of tree components in G, excluding zero-degree vertex components. Following [Aus60], the number of different trees in a bipartite graph G' is

$$
R_{ij} = j^{i-1} \cdot i^{j-1}.
$$

Table 4.14: The Fraction of G's Having Cycles

Ratio	Vertices	Fraction of G's having cycles	$E(Y_c)$
2.2	2252	0.46	2.18
2.4	2458	0.42	0.96
2.6	2662	0.41	0.58
2.8	2868	0.31	0.39
3.0	3072	0.25	0.28
3.2	3276	0.25	0.22

Here the total $i+j$ distinct vertices are split into two groups: i vertices in one and the remaining j vertices in the other. These vertices are connected by $i+j-1$ indistinguishable edges to form a tree. The formula counts the number of different such trees.

The expected number of trees of distinct edges of size from 1 to $\min(n, 2r-1)$ in a bipartite graph G with r vertices on each side is

$$E(\text{Trees}) = \sum_i \sum_j \frac{\binom{r}{i}\binom{r}{j} \cdot R_{ij} \cdot \binom{n}{i+j-1} \cdot (i+j-1)! \cdot (r^2 + i \cdot j - r \cdot (i+j))^{|n-i-j+1|}}{r^{2n}}$$

where i and j should satisfy the constraints $n-i \geq 1$ and $n-j \geq 1$ when $i+j-1 < n$, or $n-i \geq 0$ and $n-j \geq 0$ when $i+j-1 = n$. It is easy to see that the term $\binom{r}{i} \cdot \binom{r}{j}$ is the number of ways to have all the combinations of i and j vertices on both sides. The following term $j^{i-1} \cdot i^{j-1}$ is the number of different trees constructible from these $i+j$ vertices. The next term $\binom{n}{i+j-1}$ allows us to select $(i+j-1)$ distinct edges (keys) to participate in the tree. Since these keys are distinct, there are $(i+j-1)!$ ways to have the actual tree distinct. The next term $\{r^2 + i \cdot j - r \cdot (i+j)\}^{(n-i-j+1)}$ is the number of ways to have the remaining $n-i-j+1$ edges freely go into G without being adjacent to any tree vertices. The last term, the denominator r^{2n}, is the total number of ways to put n edges into G.

At present, the author is not aware of a closed form formula for E(Trees). Table 4.15 shows the average number of trees in G with various numbers of vertices and edges, and the expected values computed by the E(Trees) formula.

Table 4.15: Expected vs. Average Number of Trees (ratio=1.3)

No. Edges	E(Trees)	Average No. of Trees
16	1.78	1.23
32	2.74	2.59
64	5.28	4.93
128	10.39	9.82
256	20.19	20.09
512	39.82	38.76
1024	79.52	79.35

Observed Number of Indirect Edges

An adequate number of indirect edges is vital to a successful OPMPHF. Table 4.16 summarizes the observed components, observed trees, observed number of indirect edges generated by the one bit marking scheme, observed total number of zero degree vertices, and the observed total number of trees in G generated from a CISI vector collection of 74264 keys. The different ratios for G that were tested are 1.22, 1.3, 1.4 and 1.5.

It can be seen that most G will have only one or a few big cycle components (see last column of Table 4.14) and many smaller tree components (see last column of Table 4.15). Notice also that the number of indirect edges varies inversely with the size of G (see Table 4.16). This means more edges need to be indirect as G becomes smaller. On the other hand, a small G will have fewer vertices of zero degree or in tree components. Consequently, for the scheme to be successful G should not be too small. A rough bound on the number of allowed indirect edges (keys) for our algorithm to be successful is E(AC), i.e., the total number of zero vertices plus the total number of trees. The maximum number of indirect edges in a particular G is the total number of edges in non-tree components.

4.5.4 Test Collections and Timing Statistics

Recall that there are at least two information retrieval applications where OPMPHFs are useful. The first involves dictionary structures which has led to several experimentation with

Table 4.16: Number of Indirect Edges in a 74264 Edge Graph

Ratio	Components	Trees	Zero degree vertices	Indirect edges
1.20	4258	4257	16725	16536
1.25	4922	4921	18773	13914
1.30	5831	5830	20724	11789
1.40	7409	7407	24914	7365
1.50	9361	9360	29349	4510

dictionary key sets derived in part from the CED. Timing and other descriptive statistics from these runs with the OPMPHF algorithm are given in Table 4.17. The second deals with inverted files. Table 4.18 shows results for a set of 67,974 keys based on the inverted files data for the CISI test collection. All of these runs were made on a DECstation 5000 Model 200. Times were measured in seconds using the UNIX "times()" routine, and so are accurate up to 1/60th of a second.

Table 4.17 shows timings for runs on graphs with different numbers of edges (varying from 32 to 16,384). Notice that the timing is approximately linear in the size of the key set, as is expected from our analysis. Table 4.18 lists timings for runs on the 67,974 edge graph (CISI vector collection) with ratio varying from 1.22 to 1.5. Table 4.18 shows that as the ratio gets smaller (from 1.186 to 1.218), the Searching step takes more time to finish. This is because more indirect edges have to be packed into a smaller number of zero-degree or tree vertices. The overhead columns in Table 4.17 and Table 4.18 shows the number of bits more than what is required ($\log n$) for each key.

4.5.5 Summary

In this section, a practical algorithm for finding order-preserving minimal perfect hash functions is described. The method is able to find OPMPHFs for various sizes of key sets in almost linear time with the function size remaining within reasonable bounds. The application of the method to dictionary and inverted file construction is also illustrated. Several probabilistic analysis results on the characteristics of the random graph G are given.

Table 4.17: Timing Results for Dictionary Collection

No.	Settings		Time in Seconds			
Keys	Ratio	Overhead	Mapping	Ordering	Searching	Total
32	1.25	2.30	0.22	0	0	0.22
64	1.23	2.45	0.22	0	0	0.22
128	1.19	2.40	0.22	0	0.02	0.24
256	1.22	2.80	0.25	0	0.02	0.27
512	1.22	3.00	0.25	0	0.05	0.30
1024	1.21	3.20	0.27	0.03	0.07	0.37
2048	1.22	3.45	0.32	0.03	0.08	0.45
4096	1.20	3.45	0.42	0.10	0.57	1.28
8192	1.19	3.60	0.63	0.25	0.92	1.80
16384	1.22	4.10	1.08	0.53	1.37	2.98

Table 4.18: Timing Results for Inverted File Data

		Time in Seconds			
Ratio	Overhead	Mapping	Ordering	Searching	Total
1.186	4.0	3.67	2.28	15.87	21.82
1.193	4.1	3.72	2.27	13.92	19.90
1.199	4.2	3.62	2.25	11.20	17.07
1.205	4.3	3.72	2.25	19.05	15.02
1.211	4.4	3.73	2.27	7.90	13.90

They are useful in guiding the proper selection of various parameters and providing insights on the design of the three main steps of the algorithm.

More experiments with the algorithm are possible. One direction is to find ways to make more edges direct so that an OPMPHF can be specified in G with a smaller number of nodes. Another direction is to modify the algorithm for variable length record indexing. Given a set of records $R = \{r_1, r_2, \ldots, r_n\}$, with l_i (a positive integer) being the length of record r_i and k_i the key of r, the problem is to find a MPHF function that maps records onto a space of size $\sum_{i=1}^{n} l_i$ such that no record overlaps with others. This problem could be solved by using OPMPHF easily, with the expense of at least n computer words. Since records need not follow a prior order, the OPMPHF solution may be overly expensive. The question is: can a MPHF be found taking less than n computer words?

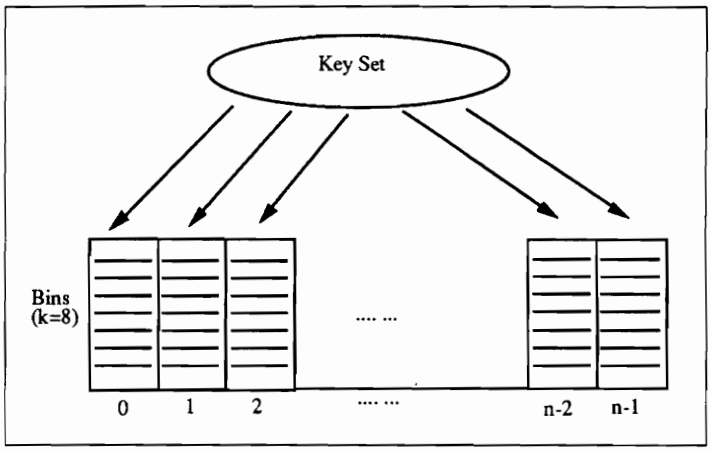


Figure 4.30: Bin Hashing

4.6 Perfect Bin Hashing

Perfect Bin Hashing Function (PBHF) is a general extension of MPHF where hash table slots are **bins** capable of holding at most k keys ($k \geq 1$). Analogously to MPHF, it is desirable to pack a set of $k * n$ keys into n bins such that each bin contains exactly k keys. Figure 4.30 illustrates the idea.

Perfect bin hashing is different from the one-dimensional packing problem studied by Johnson *et. al.* in [JDU+74]. One-dimensional packing involves a set of elements of different sizes packed into as few bins as possible such that no overflows occur in any bins. The sizes of all bins are assumed the same. Perfect bin hashing, on the other hand, takes same-sized entities and is given a fixed number of bins to work on. Also, one-dimensional packing does not worry about the element searching problem, while perfect bin hashing concerns the computation of a small hash function for rapid searching.

Bin hashing has many applications. The most obvious one is to put a collection of equally important items onto a few bins (i.e., pages). If the total cost of retrieval of a page and selection of the desired element from the page is less than for access using an MPHF, the use of bin hashing is justifiable. Furthermore, bin hashing allows a complete bin to be fetched at once which facilitates reference to other items on the same page. Mairson in

[Mai83] suggested the use of binary searching methods on elements stored in a page. The faster MPHF algorithm (section 4.3) produces small hash function specification tables that can be saved on pages. Thus, it is possible to store the MPHF specification table as well as the elements on a single page, and for intra-page access to data elements to be optimized too.

In the remainder of the section, we outline the hashing algorithm and discuss, in particular, the packing strategy.

4.6.1 Bin Hashing Algorithm Outline

Contrary to MPHF, there are more strategies for bin hashing as duplications are allowed in patterns and slots can accept more than one key. In the following, we list two possible strategies for each of the ordering and packing stages. To facilitate explanation, we denote the pattern generated from key subset K_i as a list of pairs:

$$P_i = [\{s_{i1}, c_{i1}\}, \{s_{i2}, c_{i2}\}, \ldots, \{s_{il}, c_{il}\}]$$

where l is the number of distinct elements (integers) in P_i, s_{ij} is the value of the jth distinct element in P_i, and c_{ij} is the number of times s_{ij} is duplicated in P_i. The ordering strategies considered are

Sort 1 ordering strategy. Sort the key subsets in non-increasing order of their sizes, i.e., the key of sorting for subset K_i is:

$$k = \sum_{r=1}^{l} c_{ir}.$$

Sort 2 ordering strategy. Sort the key subsets in non-increasing order. The sorting key for subset K_i is the weighted sum:

$$k = \sum_{r=1}^{l} w_r c_{ir}.$$

Here the weight w_r gives the importance of c_{ir} in the pattern. If w_r is an increasing function of c_{ir}, then a pattern that contains many duplicated elements will appear

```
1).Input the key set;
2).Partition the key set into subsets K_i's;
3).Generate pattern sets for all K_i's;
4).Sort patterns into sequence KS, using strategy Sort 1 or Sort 2;
5).Pack keys in each pattern, using strategy First fit or Best fit;
6).Output the PBHF specification.
```

Figure 4.31: Outline of the Bin Hashing Algorithm

earlier in the pattern sequence. It will participate in packing earlier. As the hash function computation places no restrictions on the sorting method, the use of weights w_rs will not increase the PBHF specification space.

There are two principle packing strategies.

First fit packing strategy. Fit the pattern when it first fits.

Best fit packing strategy. Fit the pattern such that the maximal β value for all bins is minimized. β is defined as the number of keys in a bin.

Combining these different ordering and packing strategies, we are able to give four versions of the perfect bin hash (PBHF) algorithm. See Figure 4.31.

4.6.2 Space Lower Bound Analysis

A simple derivation of the lower bound on space to specify any bin hash functions is given in [Mai83]. Here we elaborate the proof.

Denote m as the size of the key space, n the size of the key set, and k the size of a bin. Assume further that n can be divided by k, so there are n/k bins. For a given perfect bin hash function P, let i_j denote the number of keys in the key space that can be mapped to bin j under P. The total number of subsets of keys that are perfectly bin-hashed by P is:

$$t = \prod_{j=1}^{n/k} i_j.$$

Table 4.19: Space Lower Bound for Various k's

k	2	4	6	8	10	1024	4096	8192
Lower Bound	0.91n	0.58n	0.44n	0.35n	0.30n	0.0062n	0.0018n	0.00096n

Note that t is maximized when all i_j are equal to $\binom{mk/n}{k}^{n/k}$ (see [Meh82]). Since there are $\binom{m}{n}$ subsets of keys of size n, the space lower bound is :

$$\begin{aligned}
H &= \frac{\binom{m}{n}}{\binom{mk/n}{k}^{n/k}} \\
&= \frac{\frac{m^n}{n!}\frac{(m)_n}{m^n}}{\left[\frac{(mk/n)^k}{k!}\frac{(mk/n)_k}{(mk/n)^k}\right]^{\frac{n}{k}}} \\
&\approx \frac{\frac{m^n}{n!}e^{-n^2/2m}}{(mk/n)^n\left[(1/k!)\,e^{-k^2/\frac{2mk}{n}}\right]^{n/k}} \\
&\approx \frac{\left(\frac{n}{k}\right)^n (k!)^{n/k}}{n!} \\
&\approx \frac{e^n}{\sqrt{2\pi n}}\frac{(k!)^{n/k}}{k^n} \\
&\approx \frac{1}{\sqrt{2\pi n}}\left(\frac{e}{k}\right)^n (k!)^{n/k}.
\end{aligned}$$

The number of bits required is:

$$\begin{aligned}
\log_2 H &\approx n\log_2(e/k) - \log_2\sqrt{2\pi n} + \frac{n}{k}\log_2(k!) \\
&\approx n\log_2(e/k) - \log_2\sqrt{2\pi n} + \frac{n}{k}\log_2\left(\frac{k^k\sqrt{2\pi k}}{e^k}\right) \\
&\approx (n/k)\log_2\sqrt{2\pi k} - \log_2\sqrt{2\pi n}\ \textit{bits}.
\end{aligned}$$

Table 4.19 lists lower bounds as a function of n for various k's.

4.6.3 Empirical Results

In this section, the initial results of the PBHF algorithm are presented. Due to space limitations and time constraints, only the results for the algorithm using the `Sort 1` and

Table 4.20: Running Time Summary of the Bin Hashing Algorithm

Keys	Bits/Key	Bin Size	Reading	Partitioning	Searching	Total
1024	1.3190	8	0.48	0.05	0.53	1.03
2048	1.3089	16	0.53	0.12	1.67	2.32
4096	1.1300	8	0.65	0.20	1.62	2.47
8192	1.0769	16	0.87	0.38	1.07	2.29
16384	1.1786	16	1.38	0.78	1.80	3.96
32768	1.1733	32	2.17	1.57	2.97	6.71
130198	1.0589	1024	7.32	6.33	13.80	27.45

Note: for all runs,
A BHF is computed;
Time (CPU) is in seconds;
Machine = DECstation 5000, model 200;
Language = g++ (version 1.37).

the `First fit` strategies are shown in Table 4.20. The approach partitioning the key set into subsets is identical to that used in the faster MPHF algorithm. It can be seen that the bits per key values are not close enough to the lower bound. More experiments are necessary to investigate the reason and test the other strategies.

4.6.4 Summary

In this section, a proposal for extending the MPHF algorithm to the bin hash problem is described. Two ordering and two searching strategies are given. Initial results based on the use of two strategy combination demonstrate that the solution is feasible in producing small functions. More experiments are necessary to test other strategies.

4.7 Hashing Methods Summary

In this chapter the classical approaches to identifying MPHFs are first introduced. These approaches fail to find small hash functions for large key sets with a tolerable amount of computation effort. Four heuristic, probabilistic algorithms are then presented which overcome the mentioned drawbacks of classical approaches.

The mapping, ordering and searching framework is the key to the success of these four algorithms. Mapping transforms the problem into another universe. Irregularity in key sets is transformed into workable randomness in this universe. Ordering includes a hidden partition process of slicing the universe into pieces, which makes it possible to include heuristics for efficient searching. The final searching step incrementally determines the hash function in the form of a bit string. The randomness property is also utilized in the final step. It allows the measurement of the degree of success of the search.

The four algorithms are all probabilistic. The tradeoff between computation time and function space contributes to the odds of their success. A range of values for parameters used in these algorithms were identified that guarantee that with high probability they will be successful.

The described four algorithms have wide applications in indexing keys associated with other information. Chapter 3 has demonstrated their use for indexing objects in LEND. Also, they can be used to solve other problems, such as trie compaction and bin hashing. This chapter discusses the application of the MPHFs to these two problems.

Chapter 5

GRAPH STORAGE

In this chapter, we assume all graphs are in the class where the average degree of nodes is low, and access should be optimized for random path traversal. The chapter describes the physical organization of graph storage: i.e., how graphs in the class, especially the large ones, are placed onto disk-type physical storage devices and how indices are established for such placed graphs. The organization allows efficient retrieval for random path traversals: a kind of graph access where each node is equally likely as the starting point of a path and each adjacent arc is equally likely to be traversed from a node. Such placed graphs do not bias references to any special region in the graph.

The chapter begins with a section (5.2) on the partitioning of graphs. Graph partitioning algorithms of growing importance in the VLSI chip layout area, and a primary set of methods used by the database community for object layout, are reviewed.

The next section (5.3) is devoted to the method of placement of Information Retrieval graph data sets. It is shown that Fiduccia and Mattheyses' algorithm [FM82] for hypergraph bisection can be simplified for the partition of large IR graphs. The major modification of the algorithm is described.

Block and cross block arc packing concepts are then introduced in section 5.3. The purpose of packing is to assure relative space efficiency. The design of data structures

to accommodate the partitioned graph for efficient path access is detailed in the section. The data structures are implemented as a set of abstract data types in C++ to allow encapsulated and specialized encoding of useful information for quick access. The use of MPHF access methods is also illustrated.

Section 5.4 introduces a cost model for random path traversal in the partitioned graph. The formulas computing average number of page accesses for verifying the existence of a path of length l and that of obtaining all neighbors of a node, are described. In addition, an approximation of the expected number of page accesses in a relaxed situation is analytically derived.

Section 5.6 describes the graph storage experiment carried out on a real IR graph (MeSH thesaurus [HM86]). Results related to MeSH partitioning, arc and block packing and a series of performance measurements for graph access are reported.

5.1 Terminology

This section gives a short description of graph theory terminology used in the chapter.

Graph G: G is an ordered triple $(N(G), E(G), \psi_G)$ consisting of a nonempty set N(G) of nodes, a set $E(G)$, disjoint from $N(G)$, of edges (also called arcs), and an incident function ψ_G that associates with each edge of G an unordered pair of nodes of G; see [BM76], pp. 1.

Directed Graph G: G has node set $N(G)$, edge set $E(G)$ and the incident function ψ_G. ψ_G is defined as one that associates with each edge an ordered pair of nodes.

Random Graph $G_{n,p}$ of Model A: $G_{n,p}$ is called a random graph of model A iff it has n nodes and an edge exists between any two nodes with probability p; see [Pal85], pp. 6-7.

Random Geometric Graph $U_{n,d}$: $U_{n,d}$ is called a random geometric graph if it is generated in the following way. $U_{n,d}$ has n nodes spreading in a unit square. The coordinates of

the n nodes are independent numbers uniformly drawn from the interval (0, 1). An edge connects node x and y iff the Euclidean distance between the two nodes is d or less; see [JAMS89].

Bisection of Graph: Separate G into two subgraphs with almost equal numbers of nodes. Each subgraph is usually called a partition of G.

Partition of Graph: Separate G into a set of subgraphs. Each subgraph is usually called a partition of G.

Layout Plan L of Graph G: L is a layout plan if it maps subgraphs of G (obtained through partition) into a set of fixed size pages, such that no subgraphs are across more than one page. Arcs of G that are within a page boundary is called interior arcs and those that cross pages are called cross arcs. Cross arcs can be stored on pages with or separate from the subgraphs.

5.2 Literature Overview

This section surveys methods with which graphs are partitioned. The survey covers important partitioning algorithms coming out of both the database design and VLSI chip layout design areas.

5.2.1 Object Clustering Algorithms

This section surveys three object clustering algorithms. They work on graphs representing records and relations between records. The three algorithms can be characterized as constructive. The partition is achieved after the graph has been scanned once.

Bell's Clustering Algorithms

Bell *et al.* [BMPA88] investigate approaches for clustering of physical records for a CODASYL database. The goal is to store on storage devices records in close proximity that are

referred to frequently together. Two versions of the algorithm are studied. The first is for page-based clustering and the second is for page as well as cylinder-based clustering. The operation of both versions depends on a graph initially representing records as nodes and inter-record access relationships as weighted edges.

The first algorithm takes the initial graph and collapses nodes in order of decreasing priority of edge weights. Records in the two collapsing nodes are to be placed in a single page, if the formed page's capacity is not exceeded. The edge weights of nodes adjacent to the collapsing nodes are added in the resulting graph. The collapsing process continues until no further action can be taken. The second algorithm extends the first one to allow an orthogonal cylinder level clustering to take place when two pages can't be collapsed within the page boundary.

The algorithms are evaluated by simulation on a Products-Parts bipartite graph. The cardinalities of nodes on the two sides are in the ratio of 1:3. The fanout (the maximal number of parts associated with a product) varies within the range 2-10. The page sizes are from 500 to 2000 bytes. Access of the database is in the form of obtaining parts associated with a particular product. The evaluation show that both algorithms can yield record placement that are better than a random one, and the second algorithm can outperform the first. However, the gain decreases as the fanout grows larger.

Banerjee's algorithm

Banerjee *et al.* in [BKKG88] present a directed acyclic graph (DAG) clustering algorithm for CAD databases. CAD databases are best characterized as hierarchical representations of engineering artifacts such as computer chips. In the hierarchy, a component description is kept in a node and those of its subcomponents are in the child nodes. Access in a CAD database typically involves downward traversal for retrieval of descendants of a given node. It makes sense to cluster a DAG to reduce the cost of such downward traversal.

The cluster algorithm described uses three simple clustering heuristics: depth-first, breadth-first and children depth-first. The first two are well known graph traversal meth-

ods. See [HS78] pp. 263-269. The third is also a graph traversal method, defined as a procedure visiting recursively a node and then immediately all of its child nodes. Each heuristic produces a node visiting sequence in a DAG and the sequence consequently can be mapped onto a set of pages. To support insertion and deletion of nodes in the sequence, the set of pages is managed by a positional B-tree.

A simulation approach is used to evaluate the heuristics. In the experiment, all hierarchies are complete and of 9 levels. Fanouts of non-leaf nodes are set to be 2, 3, 4 and 5. The performance is measured as access cost ratios (in term of page I/O's) of two different heuristics against the traversal start position in the hierarchy. The traversals in the simulation are limited to operations where either all descendants or only direct descendants of a node are retrieved. Depth-first has the best performance of all, when all children nodes are retrieved. Children depth-first follows with at most one additional page I/O. Breadth-first is the worst as non-descendants are accessed. In the case of direct descendant node access, children depth-first usually generates the best clustering. Depth-first presents larger overhead than the children-depth-first when the start level is near the root. But the overhead diminishes when the start level is close to the leaf.

Cheng and Hurson's algorithm

Cheng and Hurson [CH91] describe an approach similar to [BMPA88] for the problem of clustering complex objects modeled by trees. As usual, the nodes represent objects and arcs represent the various semantic relations among objects. Weights can be associated with arcs to reflect importance of relations with regard to traversal. Collapsing of a weighted DAG is the key feature of the approach. Node sequences in two merging nodes are maintained throughout the clustering process. As more than one arc may share a similar weight, the merging is done level by level. Edges with identical weights are said to be at the same level. Initially, level 1 merging takes place in which nodes connected by the highest weight for arcs are merged. Level 2 merging follows, dealing with nodes connected by next highest weighted arcs, and so on.

Simulation is the evaluation tool. The experiment is designed to allow three levels of merging. The level size, i.e., the number of ith level nodes between two $(i-1)$th level nodes ranges from 10 to 200. Tested page sizes are in the range of 256 to 2048 bytes, although objects with larger size become more common. Each simulated object access is to return 10 nodes connected to the complex object node. Simulation shows that retrieval of complex objects at level one takes least time, followed by the level-two and three objects. Level-one access time is consistently around 20% of the random access time while that of level-two and three is around 50%.

5.2.2 Graph Partitioning Algorithms

This section surveys six graph clustering algorithms that can be characterized as iterative. The initial partition will be repeatedly modified. Most of them are designed for VLSI chip layout, with the objective function to minimize the number of wires across components. Values of the function are used to measure the performance of these algorithms.

Kernighan and Lin's algorithm

Kernighan and Lin [KL70] devised the first algorithm to partition weighted graphs. Formally, the problem is to bisect an edge weighted graph G such that the two resultant subgraphs have an equal number of nodes and the total weight of inter-subgraph edges is minimized. The weight on the edge connecting node x and y is denoted as c_{xy}. In the following, we use KL to denote their algorithm.

Let $\{A, B\}$ be a partial partition. KL computes for a node x in either A or B a cost $D_x = E_x - I_x$ where E_x is the summation of edge weights of x to its adjacent nodes in the opposite partition, and I_x is the summation of the edge weights of x to its adjacent nodes in the same partition. For a pair of nodes (x, y) where x is in A and y is in B, the gain f_{xy} of switching x and y, to their opposite partition, is $D_x + D_y - 2c_{xy}$. An easy calculation would show that after switching, the new cost of a node u in A is $D_u + 2c_{ux} - 2c_{uy}$ and node v in B is $D_v + 2c_{vy} - 2c_{vx}$.

A sequence of switchings thus can be performed. At the ith switching, a pair of nodes (x, y) maximizing the gain $g_i = f_{xy}$ is selected. KL determines an improved partition $\{A', B'\}$ by literally switching the first k such pairs of nodes. k is selected to maximize the sum $\sum_{i=1}^{k} g_i$.

The time required to compute the initial D values is $O(n^2)$. The selection of node pairs for switching can be simplified to the process of first sorting the D values for A and B and then picking up the two nodes associated with the largest D in the two sorted lists. The running time complexity of the simplified algorithm is $\sum_{i=1}^{n} i \log i$, which grows as $n^2 \log n$. Thus, KL runs in $O(n^2 \log n)$.

KL offers two ways to further reduce the cut set size, once an improved partition is found. The first is to apply the same switching algorithm on the improved partition with the risk of falling into a local optimum, and the second is to perturb the partition in the hope that a global optimum can be reached.

Fiduccia and Mattheyses' algorithm

Fiduccia and Mattheyses in [FM82] describe an iterative heuristic algorithm for partitioning an unweighted hypergraph HG into two relatively equal partitions. In their terms, nodes in HG are called cells and hyperedges are called nets.

The algorithm operates by moving one cell at a time to the opposite partition. The gain of the movement is computed as the number of cross partition nets eliminated. A cell with maximal gain is moved always, except if the opposite partition is too full. All cells that can be moved are free cells. A moved cell will be locked in the moved-to partition within the pass and is called a locked cell.

The authors use the following insights to achieve linear time complexity. First, only gains of those cells connected to the moving cell w should be updated. Second the updates can be replaced by a series of increments and decrements on nets adjacent to w. Specifically, suppose n is one of the nets adjacent to w, and $F(n)$ and $T(n)$ count the number of cells n connects at the "**from**" and "**to**" sides related to the moving, respectively. Before the

move, if $T(n) = 0$ then the gains of all free cells on n are increased to encourage all of those cells to move to the "**to**" side; if $T(n) = 1$, then the gains of the cells on the "**to**" side is decreased to discourage it from moving to the "**from**" side. Similarly, after moving, the "**from**" side needs to be examined. If $F(n) = 0$, then gains of all free cells on the "**to**" side are reduced; if $F(n) = 1$, then the gain of the only cell on the "**from**" side is increased. The use of localized increments and decrements limits the number of updates for a net in one pass to 4.

The efficiency of the algorithm is at the cost of maintaining complicated data structures. Further, if weighted networks are being partitioned, the current double-linked list data structure is useless, as real valued gains cannot be used to index buckets.

Krishnamurthy's algorithm

Krishnamurthy in [Kri84] extends the single gain value concept of the FM algorithm to a gain value vector.

A gain vector of a node contains a sequence of k gain values. The first value in the sequence is called a level 1 value, and in general the ith value is called the level i gain value. The level 1 gain is the cut edge reduction if the node is moved. The level 2 gain is the reduction that would be seen after the node and some other node are moved, and a level i gain is the cost that would be seen after the node and then some other $i - 1$ nodes are moved. In short, a gain vector not only records immediately the cost reduction when a cell is moved to the opposite partition, but also predicts the benefits of the movement of the cell to the movement of other cells in the adjacent nets.

Formally, a gain vector on a cell C is defined in two steps. First the binding number $\beta_A(n)$ of a net n with respect to a partition A is the number of free cells in n that are in A. Second, the ith level gain of C is measured as the set cardinality difference, where the first set contains nets x satisfying conditions $\beta_A(x) = i$ and $\beta_B(x) > 0$, and the second set contains nets y satisfying conditions $\beta_A(y) > 0$ and $\beta_B(y) = i - 1$.

Given gain vectors for all cells, a lexicographic order can be formed among them. As in

the FM algorithm, a cell with the largest gain vector is moved. The bucket data structure can also be maintained. [Kri84] shows that the time complexity of the algorithm is $O(kN)$ for k being the length of the gain vectors and N is the number of hyperedges.

Selecting a good value for k is difficult. In the paper, k is suggested to be either 2 or 3. From a time complexity point of view, larger k means more computation time. However, larger k does not necessarily yield better partitioning as higher level gains are decided by the number of cells a net has. Thus, in a network with a small number of densely connected nets, most higher level gains are zeros. They certainly do not contribute much to the partitioning.

Sanchis' algorithm

This algorithm [San89] is designed for m-way partitioning of networks, for $m \geq 2$. Use of a bisection algorithm for the m-way partition problem can yield bias caused by a bad initial partitioning. To overcome this, the algorithm tends to uniformly maximize the gains for all m partitions. Thus under the scheme, the movement of a free cell involves consideration on all non-home partitions. In the following, c is used to denote a cell and n_c the set of nets connected with c.

The definition of a gain vector is slightly changed. The binding number of net n to all partitions except partition A_k is defined as $\beta'_{A_k} = \sum_{i \neq k} \beta_{A_i}(n)$, where β_{A_i} is the binding number in the context of [Kri84]. The ith level gain associated with moving cell c from partition A_j to A_k is the cardinality difference of the two sets. The first set is $\{n \in n_c \mid \beta'_{A_k}(n) = i \wedge \beta_{A_k}(n) > 0\}$ containing all nets adjacent to c such that each has i cells in all non A_k partitions and at least one cell in A_k. The second set is $\{n \in n_c \mid \beta'_{A_j}(n) = i - 1 \wedge \beta_{A_j}(n) > 0\}$ containing all nets adjacent to c such that each net has at least one cell in partition A_j and $i - 1$ cells in all non A_j partitions. The first set measures the binding of c to $\bar{A}_k$ with respect to the partition $\{\bar{A}_k, A_k\}$, while the second measures the binding to $\bar{A}_j$ with respect to the partition $\{A_j, \bar{A}_j\}$.

After c is moved from A_j to A_k, the β' values of nets connected to c will change, which

in turn causes the gain vectors of cells on these nets to change. However, it is proved that all updates on the gain vectors can be done in constant time, independent of m, by using a cascaded bucket array data structure.

Quantitative analysis of the performance of the algorithm is difficult because many factors are involved, such as the characteristics of the networks, the selected length of levels, and the value of m. The comparison of unform partitioning with hierarchical partitioning concerns us most. The conclusion drawn by the author is that hierarchical algorithms such as the Krishnamurthy algorithm, with a small number of levels, may provide better results when computing time is limited, while the uniform algorithm with moderate number of levels will outperform hierarchical ones when enough computation time is available.

Simulated Annealing Algorithm

Johnson et al. in [JAMS89] report the use of a simulated annealing approach to the graph partition problem. The algorithm is designed to avoid falling into a local optimum (i.e., allowing the acceptance of a bad solution) and hence to reach relatively global optimization.

The simulated annealing algorithm contains two loops. The outer loop is about the falling of the "temperature" `T`, a variable controlling the probability that a bad solution is accepted. Given the initial temperature `INITT` and the final "frozen" temperature `FROZENT` and the cooling ratio r, the outer loop iterates $\log_r \frac{\texttt{FROZENT}}{\texttt{INITT}}$ times. The inner loop randomly modifies the graph partition `L` times, where `L` is the epoch length. Each time if a good modification (which reduces the cutset) is seen, then the current solution is altered according to the modification. However, if the modification is bad (which increases the cutset), the current solution is altered with probability $e^{-\delta/\texttt{T}}$, where δ is the difference of costs of the two partitions. In [JAMS89], the cost of a partition (V_1, V_2) is defined as the summation of the size of the cross partition edge set and the imbalance penalty $\alpha(|V_1| - |V_2|)^2$. The experiments are conducted on random graphs of Model A $G_{n,p}$ and random geometric graphs $U_{n,d}$. The latter is usually more suitable to real applications as they are more structured and contains clusters.

Experimental results show that the annealing algorithm dominates the KL algorithm on model A random graphs, when the computation time is not taken into account. In fact, the former is about 100 time slower than the KL algorithm on the $G_{500,0.01}$ graph. For the class of random geometric graphs $U_{n,d}$, the annealing algorithm demonstrates relatively poor performance, compared with the KL algorithm. As explained in [JAMS89], this may be due to the fact that the local optimas are distant from each other in such graph instances and the annealing algorithm has difficulty in reaching a better and distant local optimum from the current one.

[JAMS89] also compares with other competitors of the annealing algorithm besides the KL algorithm. It is pointed out that the FM algorithm, if enhanced with the "imbalance squared" penalty function and if the best vertex from either partition is allowed to move, can be as good as the KL algorithm. Another competitor is the coalesced KL algorithm. See [GB83]. The algorithm first finds a maximal matching for the graph to be partitioned. Nodes of each matching are coalesced to yield a smaller graph, on which the KL algorithm is applied. The coalesced KL algorithm can outperform the original KL algorithm with less than twice the running time.

Genetic Algorithm

Shahookar and Mazumder in [SM90] describe a genetic algorithm to sub-optimally make standard cell placement. Its power lies in its capability to maintain a pool of placement candidates and to manipulate them to obtain better candidates. This is analogous to biological evolution where fitter individuals have more chance to survive, and further increase in fitness compared to weaker ones. Following genetic terms, the pool of candidates is called the *population*, and the manipulation operators are separately called **crossover**, **mutation** and **inversion**. The algorithm works in the following manner. Initially, a random population is set up. Three operators are applied to the good candidates in the population to generate possibly better candidates. Weaker candidates are removed from the population to keep the population size constant. The evaluation and selection process can be repeatedly

used until an acceptable individual is found in the population or the allocated computation time is used up. The following function is used to measure the fitness of a candidate:

$$\frac{1}{\sum_{\text{nets}} x(i)W_h(i) + y(i)W_v(i),}$$

where $x(i)$ and $y(i)$ are the horizontal and vertical spans of net i, and $W_h(i)$ and $W_v(i)$ are the weights along the two spans.

The **crossover** operator is the major genetic operator. It takes two individual placement candidates and generates an offspring by combining constituents from both candidates. If the parent candidates represent good placements, then the offspring is expected to be even better. The **mutation** operator makes spontaneous changes to individuals in the hope that missing good components of a placement can be brought back. The **inversion** operator groups a set of components (e.g., nodes that are densely connected) in a candidate so that the **crossover** operator can produce offspring that inherit the entire component set. In the placement problem, **inversion** shuffles cells around so good cells originally apart can be close to each other. These close to each other cells have less chance to be separated, when subject to a `crossover` operator.

Experiments with the algorithm on networks on the order of hundreds of cells reveal that the cycle **crossover** operator combined with deterministic selection yields the best results. The number of individuals examined in the searching space is smaller than the examined by a simulated annealing algorithm package (TimberWolf 3.2 [SSV87]) for obtaining comparable placements. The cut ratio is in the range 9 to 50. The algorithm can cut more wire length at the beginning stage of iterations, and then decreases its performance in a moderately monotonic fashion. The run time of the algorithm is marginally less than that of TimberWolf, but still huge (about 12 CPU hours) on a moderate size (800 cell) circuit.

Table 5.1 summaries the surveyed algorithms.

Table 5.1: Summary of the Partitioning Algorithms

Algorithm name	Algorithm characteristics	Data Processed	Time Complexity
Bell	constructive	Graph	$O(n \log n)$
Banerjee	constructive	Graph	$O(n)$
Cheng	constructive	Graph	$O(n \log n)$
KL	iterative	Graph	$O(n^2 \log n)$ per pass
FM	iterative	Hypergraph	$O(n)$ per pass
Krishnamurthy	iterative	Hypergraph	$O(kn)$ per pass
Sanchis	iterative	Hypergraph	$O(mkn)$ per pass
Johnson	iterative	Graph	exponential
Shahookar	iterative	Hypergraph	exponential

5.3 IR Graph Storage

The graph storage problem studied deals with the placement of a large, relatively static IR graph composed of node objects and arc objects on disks and the accommodation of appropriate data structures for the storage of such a graph. The objective is to efficiently retrieve portions of the graph that exhibits the locality property, where the storage of the graph is relatively space efficient.

In information retrieval, common types of access include identifying semantically related words in a dictionary organized as a semantic net; browsing in a document space; or retrieving document and term representatives in the Boolean and Vector model for faster document-query matching computation. All these applications can operate more efficiently if adjacent nodes can be fetched quickly. And assuming no special preference for subsets of adjacent nodes, all of them are equally important.

The graphs appearing in the above mentioned applications have several distinctive features. They contain a large number of relatively static nodes and arcs of different kinds. The size of nodes are in the order of a few to thousands of bytes. Although conceptually larger objects are possible such as for multimedia, most other large objects can be viewed as composite objects. Physically, each component can be small in size. The overall average degree of nodes is low. Arcs are directed and non-reflexive.

We divide the problem into three subproblems: partitioning, packing and data structure design. We briefly introduce each subproblem in the following paragraphs and then give their solutions in sections 5.3.1, 5.3.2 and 5.3.3, respectively.

Recall that IR data is often relatively static. It is worthwhile to spend some processing time in exchange for better overall performance of traversal. Thus, the partitioning algorithms surveyed in section 5.2 can be employed to partition the graph into blocks such that the number of cross arcs are suboptimally minimized. The typically small node and arc sizes also permit a sufficient number of nodes and arcs to be stored on a page, so the advantage of clustering can be seen. However, the size of the graph still presents problems, as most partitioning algorithms surveyed are designed to handle instances of graphs or hypergraphs with several hundred nodes. As a partition of the graph that maximizes connectivity (locality) within each page is the goal to be achieved, the principle of locality may not be used in these algorithms. Thus, there is a need for modification of the partitioning algorithms so that their operations can be efficient even under the condition that the principle of reference locality is not well observed initially.

The resulting partition induces an initial layout plan of the graph on the disk. The plan may not be space-efficient as the load ratio of the pages may not be a control variable for the partitioning algorithm. In addition, if arcs are stored as individual objects, cross arcs will take extra pages to store. It is beneficial to pack these cross arcs onto node pages where space is still available. Finally, pages not completely full can sometimes be merged together.

Given the layout plan, modified in the above mentioned ways for space efficiency, it is necessary to design the data structures and index mechanisms so that pinpointing a node in the graph and following paths initiating from that node can be carried out quickly.

Following is a list of assumptions and requirements considered in this chapter.

- graph G is large, static and relatively sparse;
- searching of G exhibits reference locality;

- G is to be stored on a set of pages; and
- space utilization should not be overlooked.

5.3.1 Partitioning Algorithm Selection and Modification

This section discusses the choice of partitioning algorithm and its modification for large graph partitioning applications.

The base algorithm selected is the FM algorithm [FM82]. Several reasons are behind the choice. First, the algorithm runs in time proportional to the sum of the degrees in a hypergraph. In the graph case, this is equal to the sum of the degrees of all nodes in the graph, which is bounded from above by $O(|E|)$ where E is the edge set. This property of the algorithm is crucial, because IR graphs are large, and it is necessary that partitioning can be accomplished with a reasonable amount of CPU time.

Second, the algorithm can be recursively applied to obtain subgraphs of sizes that will fit into pages. Note that the constraint that two almost equal size subgraphs should be produced by a bisection can be relaxed in the context of IR graph layout. Further, many IR graphs such as MeSH are of low degree on average. Thus, it is expected that the FM algorithm can perform as well as other sophisticated algorithms.

The M-way algorithm [San89] is not considered because it involves complicated data structures that can easily consume more space than main memory can offer. Consider MeSH as an example. A calculation on total space using the formula $O(clp)$ ([San89] pp. 67) reveals that a main memory on the order of 1.44 GB is required for the MeSH graph, which is not practical for the machines we can use. The parameters in the formula are interpreted as: c, the number of cells in the net, which is equal to 287717 in MeSH; l, the length of the vector gain, which is set to 4; and p, the biggest cell degree in the input network, which is equal to 12,596 in MeSH.

Further, experimental results in [San89] show that the M-way algorithm can outperform hierarchical algorithms such as FM algorithm only when moderate to ample computation

```
for each net n on the base cell
    if T(n) = 0 then increment gains of all free cells on net(n)
    else if T(n) = 1 then decrement gain of the only T cell on net(n), if it is free

    decrement F(n)
    increment T(n)

    if F(n) = 0 then decrement gains of all free cells on net(n)
    else if F(n) = 1 then increment gain of the only F cell on net(n), if it is free
```

Figure 5.1: Gain Computation in FM algorithm

time is available. The time complexity of the algorithm is $O(lmb(\log b + pl))$ ([San89] pp. 68) for each pass. For the MeSH case, this is about 5.6×10^{14} time units, found by plugging in the following additional parameters: b, the number of partitioned blocks, is set to 4000 (assuming the page size is 8192 bytes and MeSH raw data is about 30 MB); m, the number of pins in the graph, is equal to 679,592. As no estimation on computation time is given in [San89] on further reduction of cut set size for larger networks such as MeSH, it is felt that a bisection algorithm should be the first choice.

Although a reasonable choice, the FM algorithm has several drawbacks prohibiting it from being applied to graph partition applications. The data structures in the FM algorithm are overkill for graphs, as unnecessary space and manipulation overhead are involved. The gain computation formula shown in Figure 5.1 is too complicated as all nets in the graph case have two cells. It is not necessary to separate the update operation into four steps. Also, the sizes of nets are neglected in the FM algorithm but the arcs in the IR graph case have nonnegligible sizes and, therefore, should be considered in the partitioning process.

To correct these shortcomings, the FM algorithm is modified. The following subsection details the changes. The resulting algorithm is named the SFM (Simplified FM) algorithm.

A Simplified FM Algorithm for Graph Bisection

Recall that the key idea of the FM algorithm is to maintain two subgraphs (called partitions) and move one node at a time from one partition to another, based on its gain values. Let A and B denote the two partitions under discussion and x be the node to be moved to its opposite block. Further assume x is in partition A before the move, and $deg(x, A)$ (i.e., the number of nodes in A that are adjacent to x) is the degree of x in A, and $deg(x, B)$ (i.e., the number of nodes in B that are adjacent to x) is the degree of x in B. Denote by $gain(x, B)$ the gain value of x with respect to B. Then $gain(x, B)$ can be computed using the formula:

$$gain(x, B) = deg(x, B) - deg(x, A).$$

After x is moved to B, $deg(x, B)$ and $deg(x, A)$ change, and consequently the gain values of nodes adjacent to x. Let y be such an adjacent node and S be its partition. If $S = A$, the new gain $gain'(y, S)$ increases by 2, and if y is in B, $gain'(y, S)$ decreases by 2. Thus, the gain value formula becomes:

$$gain'(y, S) = \begin{cases} gain(y, S) + 2 & \text{if } S = A \\ gain(y, S) - 2 & \text{if } S = B \end{cases}.$$

For each node moved, the adjustment work is proportional to the number of its adjacent nodes. Since a moved node is locked in the new partition in the current pass, the total amount of adjustment work is in worse case on the order of the number of arcs $O(|E|)$ in G.

The FM algorithm, for a bisection, suggests that the maximal block size be half of the total cell size plus the maximal cell size. Adding the maximal cell size is necessary to assure at least one node can move around. In connection with IR graphs, the maximal block size can be set to the summation of three terms: half of the total node size plus that of the total arc size, the maximal node size and the maximal degree multiplied by the arc size. This will guarantee that at least one node can be moved around.

The arcs contribute non-negligible sizes since they are physically stored. There are several ways to store arcs: storing arcs as a complete object or as adjacency lists. In the

following discussion, the former is assumed. The revised algorithm can be equally applied to the latter storage plan.

At node moving time, the arc size effects the partition content sizes. After x is moved from A to B, A holds fewer bytes of content, as x is moved to B and $deg(x, A)$ many arcs become cross arcs. On the other hand, the B side will contain x and $deg(x, B)$ more arcs. The change in block sizes is easily computed by checking the degree values of x, which requires a total of $O(|N|)$ work. It can be seen that the degree values play an important role in updating gain values as well as block sizes. Although they are computable by counting $deg(x, A)$ and $deg(x, B)$, it is best to keep them in a materialized form for quick reference.

Based on the discussion, it is possible to describe the data structure and the SFM algorithm. Four data structures are used in the algorithm: `node`, `arc`, `arcInfo` and `partition`. In the following, we give their definitions in C style.

The node data structure `node` is

```
struct   node_t {
         PixSLList node_list;
         int gain_value = 0;
         Pix bucket_index = 0;
         short degrees[2];
         struct {
                  unsigned idle :        1;
                  unsigned bounded :     1;
                  unsigned lock:         1;
                  unsigned visited :     1;
                  unsigned sd :          1;
                  unsigned sz :          16;
         } flags;
};
```

node_list remembers the pointers to the **arc** data structure that associates adjacent nodes together. **gain_value** is the gain value of the node if moved to the opposite partition. **bucket_index** records the location of the node in the buckets of its home partition to allow quick removal of the node from its current bucket. The array **degrees[2]** is the degrees of the node in the two partitions. The **flags** field bundles various information into a single integer: **idle** tells whether the node entry has been associated with any node instance; **bounded** indicates if the node entry has found a partition; **lock** tells whether the node entry is either free to move or locked in a partition; **visited** indicates visiting status; and **sd** and **sz** record the side and the size of the node.

The **arcInfo** data structure is

```
struct    arcInfo_t {
          int classId;
          int objectId;
};
```

class_id and **object_id** are for an arc object's identification attributes that are best carried throughout the bisection.

The **arc** data structure is

```
struct    arc {
          node_t *node_ptr = 0;
          arcInfo_t *arcInfo_ptr = 0;
          struct {
                    unsigned direct : 1;
          } flags;
};
```

The **node_ptr** and **arcInfo_ptr** fields remember the association of a node and an arc. The **direct** field keeps the orientation of the arc with respect to the node.

The **partition** data structure is

```
struct   partition_t {
         PixDLList **buckets = 0;
         int max = 0;
         int current_max_bucket = 0;
         int max_partition_size = 0;
         int current_partition_size = 0;
         int this_side = 0;
};
```

This data structure maintains an array of $2 * max + 2$ buckets in `buckets`, where max is the maximal degree of the graph. Each bucket is of doubly-linked list type (`PixDLList`). The 0th bucket is for the locked nodes (i.e., nodes moved from the other partition to this one) and the rest are for free nodes (i.e., nodes free to move to the other partition). `current_max_bucket` points at the nonempty bucket with highest gain. Nodes in this bucket have highest priority to be inspected in the next round of moves. `max_partition_size` and `current_partition_size` respectively are the maximal and current capacity of the partition. `this_side` indicates the side of the partition.

The SFM algorithm in a pseudo language is shown in Figures 5.2 and 5.3. If arcs are stored in adjacency list form the SFM algorithm still works. Note that the size of the adjacent list can be added to that of node x. Thus, the arc size in the previous discussion can be set to zero.

5.3.2 Layout Plan Modification

This section describes how the partition generated by the SFM algorithm is modified so that the quality of the partition in terms of space utilization and page I/O accesses can be further improved. The modification involves arc packing when arcs are stored as individual objects, and page packing. Note that arc packing is not necessary if arcs are stored as adjacency lists. This is because the list sizes can be included in the node size. When the

```
save_node(node& x, int& current_partition_index)
{
  if ( partition[current_partition_index] cannot hold x ) {
    current_partition_index ++ ;
    current_partition_index %= 2 ;
  }
  put x to partition[current_partition_index]'s buckets[0];
  set x visited;
}

initial_bisect()
{
  while ( exists unvisited node x ) {
    save_node(x, current_partition_index);
    while ( exists unvisited node y adjacent to x )
      save_node(y, current_partition_index);
  }
  for each node x {
    compute x's degree in each partition;
    compute x's binding value in each partition;
  }
  for each node x
    insert x from buckets[0] to buckets[x.flags.binding_value + max + 1];
}

move_a_node(node& w)
{
  move w to t by performing
   w.set_lock(yes);
   w.set_side(t);
   remove w from its current bucket to bucket[0] of opposite partition;

  for ( each unlocked node y adjacent to w ) {
    update y's new degree in both partitions;
    compute y's new binding value;
    relocate y to its new bucket;
  }
}
```

Figure 5.2: Simplified FM algorithm (1)

```
int make_a_pass()
{
  do {
    let u be the unlocked node with highest binding value in partition 0, if any;
    let v be the unlocked node with highest binding value in partition 1, if any;
    let w be either u or v such that w can fit into its opposite
              partition t and w's binding value is as large as possible;
    move_a_node(w);
  } while ( exist nodes to be moved );

  add locked nodes in each partition to the free node buckets;
}

SFM_bisect()
{
  initial_bisect()
  for ( int i = 0; i < PASSES; i++ )
    make_a_pass();
}
```

Figure 5.3: Simplified FM algorithm (2)

SFM algorithm finishes, the adjacency lists and nodes are guaranteed to fit into pages. The extra long adjacency lists that can't fit within a page will be placed onto several pages. We first discuss arc packing.

Recall that a set of partitions (subgraphs containing nodes and interior arcs) and a set of cross arcs are the output of the SFM algorithm. They are to be mapped onto pages of a disk. Since it is guaranteed to fit a page, a partition can be directly mapped into a page. The cross arcs require additional pages to store. However, as the partition pages are not necessarily full, some of the cross arcs can be packed in partition pages.

The arc packing strategy is based on the criteria that cross page arcs should be as close as possible to the objects they are connecting. Since the overhead of identifying objects in a page is small, the closeness can be relaxed to within the same page. It is best to place an arc on its source node page, since then retrieving the source and sink nodes and the arc involves two page accesses which is optimal when two nodes are stored on two separate

```
Let f be a new empty block
for each cross arc x
    let y be the block containing the source node of x
    let z be the block containing the destination node of x
    if y has space for x then
        pack x into y
    else if z has space for x then
        pack x into z
    else if f is full then create a new empty block, calling it f
        pack x into f
```

Figure 5.4: Packing Cross Arcs

pages. If the arc can't fit into the source node pages, it is also an almost optimal choice to place it on its sink node page. Two page accesses are needed to fetch the source and the sink node in this case also. An additional lookup for the arc page number using the arc index is needed as well. If the page is not purged from the buffer at run time, then the sink node is in memory as it resides on the same block as the cross arc. For the case that the arc cannot fit on either the source or sink node page, it has to go to some other page. The number of page accesses becomes three. Figure 5.4 summarizes the arc packing process.

Similar to the arc packing process, pages not 100% full can be packed together to reduce the storage space. Page packing is exactly the bin packing problem studied by Johnson *et al.* [JDU+74]. It is shown in [JDU+74] that the first or best fit strategy will yield a packing with size (i.e., the number of bins used) exceeding at most 17/10 of that of an optimal packing; and with the first or best decreasing fit strategy that takes more processing time, the size exceeds at most 11/9 of that of an optimal packing. Both first and best fit strategies (with or without sorting according to the size of items) are applicable to the page packing problem.

Thus, a layout plan modification consists of arc (when arcs are stored as individual objects) and page packing steps. The former intends to store the arc object and its connecting node object on an identical page, even though the other connecting node is not within the same page. The latter intends to reduce the number of pages in the final layout plan. In

the interest of performance, arc packing should be carried out first. Afterwards, the page packing can reduce the number of pages.

5.3.3 Storing a Partitioned Graph

Cattell in [Cat91] summarizes the possible implementations of binary and higher order relationships in an object-oriented database. They are generally pointer-based encodings. Binary relations are most commonly represented as object identifier lists within the (node) object ([Cat91], pp. 164). If the binary relation is one-to-one or one-to-many, the physical collocation storage of a node object and its (many) adjacent objects can be useful, although the method is not flexible for storing relations of other types. Higher-order relations can be encoded using intermediate objects which connect with participating (node) objects through the binary relations.

A partitioned graph is stored in LEND on a set of pages which have a direct mapping relation with the physical pages on a disk. Figure 5.5 shows the global view of an example G stored on a set of four pages, with 16 nodes and 20 arcs. Eight arcs (c, d, e, j, n, o, p and s) are cross arcs. Five indices are also shown. They help identify nodes from whence a traversal can take place. Although indices provide a quick way to locate nodes, the data structure for path traversal should be emphasized. In the following, we concentrate on two ways arcs objects are stored.

Plan 1: Storing Arcs as Individual Objects

Figure 5.6 shows the internal structure of a page for such a storage plan. The count and the pointer array are at the top of the page, while the nodes and arcs are at the bottom of the page.

When arcs are stored as individual objects, some searching is necessary to locate for a given arc type the instances of arcs of that type stored with the node on the same page, and the instances of arcs that are stored on other pages. In the following, index strategies for the two cases are discussed.

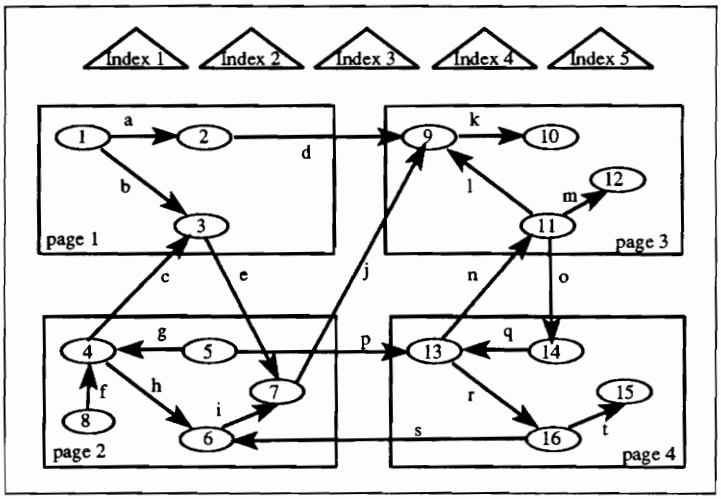


Figure 5.5: Global View of Graph Storage and Indexing

For the first case, a linear searching of objects on the page will work when the number of objects on the page is small. For the second case, a global index can be used instead which associates the object identification of a node object with the locations of arcs that are connected with the node. If arcs have direction, two sets of global indices can be used where one is for the arcs coming into the node and the other for those leaving the node. If the page number of the node adjacent to the node of interest is encoded in the arc, the cost of obtaining an adjacent node page is the summation of the cost of finding the location of the arc, the cost of bringing in the arc, and the cost of reading the adjacent node page.

Suppose the partitioning has done a good job and the degree of the node is low, then it is conceivable that the number of arcs stored out of the page is low or even zero. For a given node, it is desirable to know whether it has cross page arcs before the global index, which requires extra I/O, is consulted. This can be achieved through encoding marks in each arc. A set mark implies that all arcs are within the page boundary. For directed arcs, two marks can be employed for the source and the sink side.

Figure 5.7 illustrates the idea of marking. Figure 5.7 (a) shows a partitioned sample graph. The number of subgraphs is two. Figure 5.7 (b) demonstrates the storage of nodes and arcs of the sample graph on two pages. The idea of marking is also illustrated, assuming

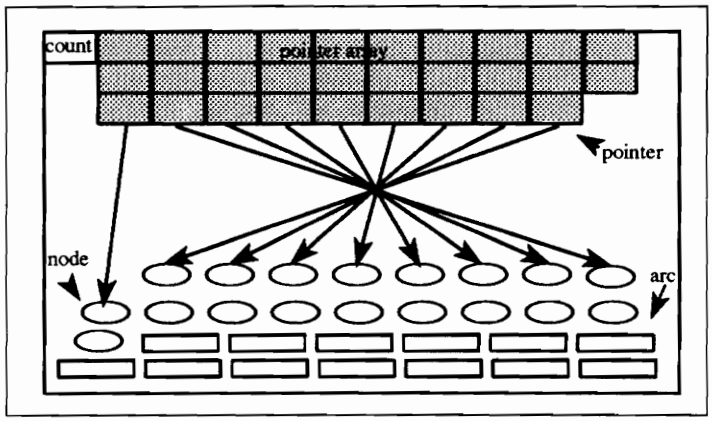


Figure 5.6: Page Layout for Arc Storage (Plan 1)

that all arcs are of similar type. The dots on arcs stand for the set marks.

Plan 2: Storing Arcs as Adjacency Lists

Figure 5.8 shows the internal structure of a page for the adjacency list storage plan. Note that the node and its adjacency list are stored consecutively. Figure 5.9 shows an example adjacency list. The list has two portions encoding **in** and **out** arcs for a node. Within each portion, there two sublists: one for the interior arcs and one for the cross arcs. Each sublist is composed of groups of identifiers of the adjacent nodes connected by similar types of arcs. Each group begins with the count indicating the number of such adjacent nodes. Next to the count is the type code for the arc type. The type code is following by identifiers of the adjacent nodes. Depending on the position of the adjacent node (within same page or on another page), object identifiers can be stored differently. For a within page adjacent node, the object identifier can be the index position of the pointer of the node. For an outside page adjacent node, the identifier can be the logical object identifier of the adjacent node, or the pair of the page number and pointer position of the adjacent node.

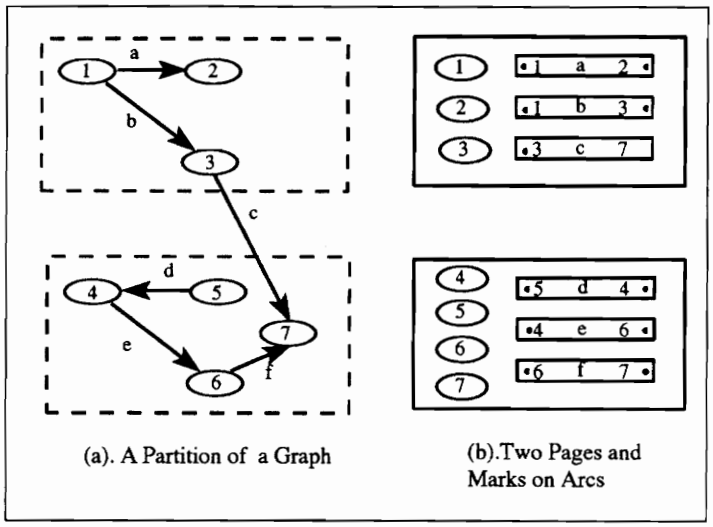


Figure 5.7: Plan 1 Storage Example

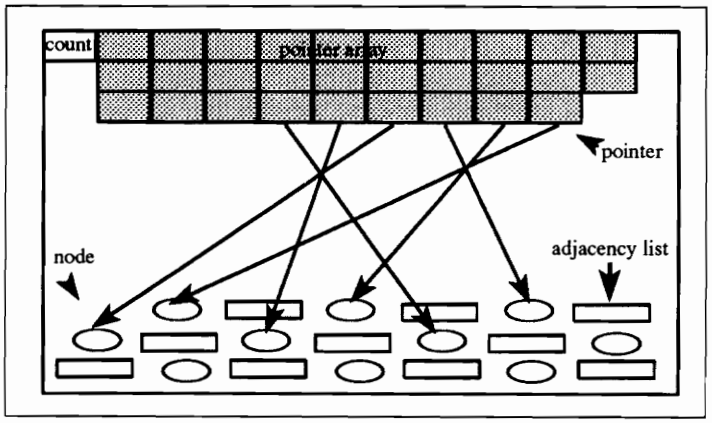


Figure 5.8: Page Layout for Arc Storage (Plan 2)

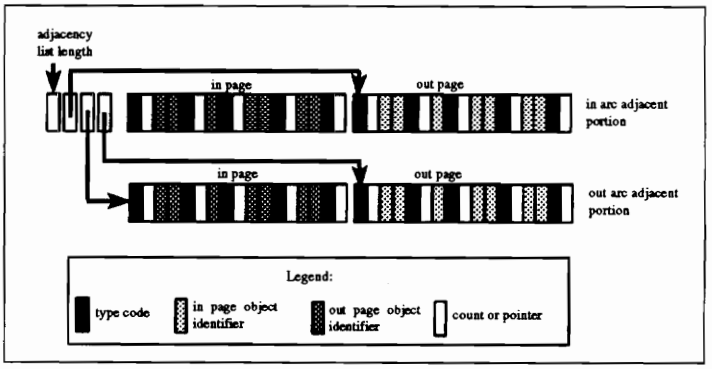


Figure 5.9: Example of an Adjacency List

Table 5.2: Four Cases When Fetching an Arc Object

Case	Page Cost	Index Cost
x, y and $a \in P_u$	0	0
x and $a \in P_u$; $y \in P_v$	1	1
$x \in P_u$; a and $y \in P_v$	1	2
$x \in P_u$; $a \in P_v$; $y \in P_w$	2	2

5.4 A Graph Access Cost Model

This section determines average I/O costs of traversal for a path in a partitioned graph. The purpose of developing the cost model is to obtain a simple cost formula with which the paging I/O of common graph traversals can be estimated. The estimation can be beneficial to the query parser. The cost estimation can also be used in the comparison of performance of the partitioned storage plan with that of a random storage plan.

5.4.1 Arcs Stored as Individual Objects

Recall the case where arcs are stored as individual objects. To follow an arc a from node x (already fetched) to node y, there are four possible cases regarding the number of additional pages fetched, as described in Table 5.2 column 1 where pages involved are denoted by P_u, P_v and P_w. Columns 2 and 3 of the table show the paging cost and the MPHF index cost for each such case. If the probability Q_i of each case is known for a given stored graph, the average cost C_1 of following an arc is

$$C_1 = 2Q_2 + 3Q_3 + 4Q_4.$$

Q_i's are easily computable from the stored graph. A good partitioning tends to increase the value of Q_1 and to reduce Q_2, Q_3 and Q_4.

To follow a path of length l, the average total paging cost can be computed as

$$C_l = C_1 l + 1,$$

assuming the initial node x requires 1 page I/O.

The above formulas yield an estimated averaged over arcs of all types. For paths involving arcs of a particular type combination, C_1's can be computed for arcs of different type. Denote C_1^t as the average cost of following an arc of type t, then the cost of obtaining a path of length l with arcs of type $t_1, t_2, \ldots t_l$ is

$$C_l = \sum_{i=1}^{l} C_1^{t_i}.$$

Another type of access initiating from a node x is to retrieve all the d destination nodes of x. The average cost of the operation is

$$C_d = dC_1.$$

5.4.2 Arcs Stored as Adjacency Lists

Assuming that each adjacency list can be stored with its nodes, without separation to multiple pages, and the pair of page number and index position of the adjacent node is stored in the adjacency list, then the estimation of C_1, C_l and C_d becomes very easy.

To follow an arc a from node x to node y, there are two possible cases regarding the number of pages fetched: x and y on the same page, or x and y on two different pages. Let the probability Q_1 and Q_2 of the two cases be known. The average cost C_1 of following an arc is

$$C_1 = Q_2.$$

To follow a path of length l, the paging cost can be computed as

$$C_l = lC_1 + 1.$$

Finally, the cost of fetching all adjacent nodes is

$$C_d = dC_1.$$

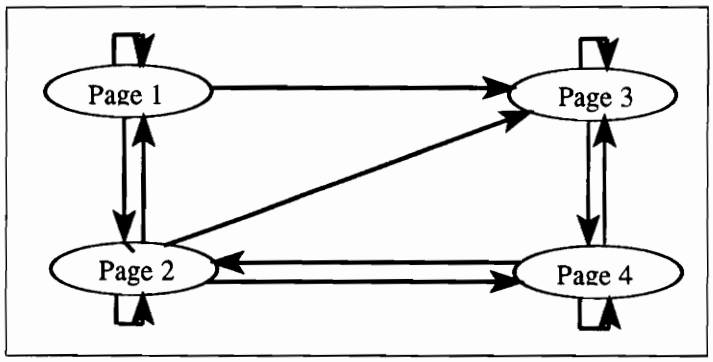


Figure 5.10: An Example Page Graph

5.4.3 An Analytical Cost Model

The previous section shows how to calculate paging costs for a given layout in LEND. If certain conditions are relaxed, it is possible to form an analytical cost model.

Consider G_p the page connection graph in a loaded graph G, where $N(G_p)$ is the page set and $E(G_p)$ is the page connection set. $|N(G_p)| = m$. In G_p, an arc $\in E(G_p)$ connects page x to page y if a node object stored in x connects to some node object on page y. Note that loops are allowed in G_p. Due to the clustering, adjacent nodes tend to be stored on the same pages. Figure 5.10 depicts the G_p for the sample G shown in Figure 5.5.

Let $N(G)$ denote the set of nodes in the graph G. For any layout plan L, a total function $f : N(G) \rightarrow N(G_p)$ exists. f maps a node $n \in N(G)$ to its page $f(n) \in N(G_p)$.

Suppose two nodes n_i and n_j in G are connected. n_i and n_j may be on the same or a different page according to the plan L. Let the probabilities of these two events be $p(n_i, n_j)$ and $q(n_i, n_j)$. In a layout plan L that treats all node pairs equally such as the SFM algorithm, all $p(n_i, n_j)$'s are equal and so are the $q(n_i, n_j)$'s, for all node pairs. To facilitate discussion, "(n_i, n_j)" is dropped, and the value of two probabilities are denoted as $p(n_i, n_j) = p$ and $q(n_i, n_j) = q$. The relationship $p + q = 1$ holds.

Consider a path X in G of length l. X can be denoted as $X = n_1, a_1, n_2, a_2, \ldots, a_l, n_{l+1}$, where n_i is the node and a_j is the arc for $i \in [1, \ldots, l+1]$ and $j \in [1, \ldots, l]$. The page access sequence P_X of X is $P_X = f(n_1), f(n_2), \ldots, f(n_{l+1})$, assuming storage plan 2 is used.

Let the random variable Z count the number of pages visited for path X. The probability $P(Z = z)$ is

$$P(Z = z) = \binom{l}{z-1} q^{z-l} p^{l-(z-1)}.$$

The mean of Z is

$$\begin{aligned} E(Z) &= \sum_{z=1}^{l+1} z p(z) \\ &= \sum_{z=1}^{l+1} z \binom{l}{z-1} q^{z-l} p^{l-(z-1)} \\ &= \sum_{x=0}^{l} (x+1) \binom{l}{x} q^x p^{l-x} \\ &= lq + 1. \end{aligned}$$

Assuming that path traversal is conducted randomly and the mapping of the partitioned graph to disk pages is indiscriminate, then the page visiting sequence is random in nature. According to [CD85] and [NFS91], allocating one page buffer should be a good strategy.

The expected cost $E(d)$ for finding all adjacent nodes of a given node n_i can also be calculated. Suppose n_i has d adjacent nodes. n_j can be stored on either the same page where x resides or on any of $m-1$ other pages. Given n_i, $q' = q/(m-1)$ is the probability that n_j is stored on a page distinct from the one where w is stored.

$1-(1-q')^d$ equals the probability of any one of these $m-1$ pages containing at least one adjacent node. The expected number of pages containing adjacent nodes is $(m-1)(1-(1-q')^d)$. Including also the page where n_i is stored, the expected page access cost is

$$E(d) = 1 + (m-1)(1-(1-q')^d).$$

5.5 Application to MeSH

5.5.1 MeSH Data Description

MeSH (Medical Subject Headings) is the National Library of Medicine's controlled vocabulary used for indexing, cataloguing and retrieving information from MEDLARS, a medical

bibliographic database. Both MeSH and MEDLARS are discussed in detail by Humphrey in [HM86]. In the following, we give a brief description of MeSH based on [HM86], [FKCF91] and [Kou91].

Organized as a polyhierarchical subject thesaurus, MeSH contains about 1500 main headings, 78,000 entry terms referring to the main headings, 50,000 names of chemical substances mapped to main headings, 700 qualifier terms, and 4,500 MeSH tree path names. The main heading terms are authoritative forms of descriptors used for indexing, together with other descriptive and control information about them. Entry terms include synonyms of headings and non-synonyms that do not merit the status of a MeSH heading. The chemical substance name terms control the mappings from the description of substance names to main headings and CAS registry numbers. The qualifier terms provide the valid modifiers of the main headings. MeSH tree path names are the tree numbers of the main headings.

Equivalence, hierarchical and associative relationships exist between terms in MeSH. An equivalence relation, relating the supplied term with its preferred term, is illustrated by the many-to-one correspondence between the entry terms and the main headings. The hierarchical relation is illustrated by the Broader main heading term connection. Finally, associative relations capture all other relations that are neither equivalence nor hierarchical. They can be between entities and their properties, operations and their agents or instruments, and actions and the product of the actions. In MeSH, examples of the associative relations are the one to many connections from chemical names to their sources and to their pharmacological actions.

5.5.2 MeSH Data Preprocessing

In the initial stage of the INCARD project (see [FKCF91]), the MeSH data was converted and organized into a format suitable for data analysis and loading into LEND. A detailed description is given in [Kou91]. Later, the duplicated fields and files were eliminated. The hierarchical structure encoded in the tree number collection was created and the redundant

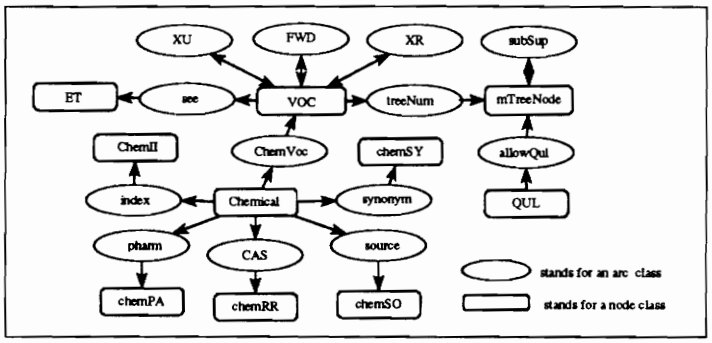


Figure 5.11: MeSH Thesaurus Graph View

tree number collection removed. Tables 5.3 and 5.4 list respectively the resulting terms and relations of MeSH. Figure 5.11 shows the graph representation of these terms and relations from the perspective of a complete MeSH graph. As promised by the object-oriented modeling technique, these terms and relations can be directly mapped to node objects and arc objects in LEND.

MeSH node and arcs sets are preprocessed to facilitate the application of the SFM algorithm. Specifically, the actual node cell contents each easily taking hundreds of bytes are irrelevant to the partitioning process in that knowing node sizes suffices to develop a layout, and the node object identifiers in the form of pairs of class and instance identification are more difficult to work with in the algorithm than pure integer identifications ranging over the number of nodes. Software converters were developed for both inconveniences. The first computes integer identifications for all MeSH objects using a MPHF, while the second replaces the object contents with their sizes. The arc collections are processed after the source and sink node identifications are replaced with their integer counterparts. The MPHF is kept for materializing node content strings back when the partitioning is done.

It is worthwhile to point out that identification replacement is almost a linear process as MPHF computing is almost linear and each object passes through the MPHF once to acquire its integer. Afterwards, the integer and byte count pair are saved as the normalization of the object. Note the byte count has included the actual object size as well as extra control bytes used by LEND. Arc collections are processed in a similar fashion. The time complexity is

Table 5.3: MeSH Term Summary

Term Name	Mnemonic	Fields	
Main Heading	VOC	1. Annotation	2. Allowable Topical Qualifier
		3. CAS Registry Number	4. Country of Publication
		5. Data Major Desc. Established	6. Data Minor Desc. Established
		7. Date of Entry	8. Date Withdrawn
		9. Descriptor Class	10. Descriptor Entry Version
		11. Descriptor Mapped-To	12. Descriptor Type
		13. History Note	14. Major Revision Date
		15. Mesh Heading	16. MeSH Scope Note
		17. Online Note	18. Public MeSH Note
		19. Running Head	20. Unique Identifier
Qualifier	QUL	1. Annotation	2. Date of Entry
		3. Date Qualifier Established	4. Geographic Area Code
		5. History Note	6. Language
		7. Language Usage	8. Major Revision Date
		9. MeSH Scope Note	10. Note
		11. Online Note	12. Qualifier Entry Version
		13. Qualifier Usage	14. Qualifier Type
		15.Subheading	14. Topical Qualifier Abb.
Chemical Name	Chemical	1. CAS Registry/EC Number	2. CAS Type 1 Name
		3. Date of Entry	4. Frequency
		5. Major Revision Date	6. Name of Substance
		7. Note	8. Unique Identifier
Chemical Indexing Info.	ChemII	1. Term	
Pharmacological Action	ChemPA	1. Term	
Related Regsistry No.	ChemRR	1. Term	
Chemical Source	ChemSO	1. Term	
Chemical Synonym	ChemSY	1. Term	
Entry Term	ET	1. Term	
MeSH Tree Node	mtreeNode	1. Term	

Table 5.4: MeSH Relation Summary

Relation Name	Mnemonic	Node Names	
See Under	XU	1. VOC	2. VOC
See Related	XR	1. VOC	2. VOC
Forward See Related	FWD	1. VOC	2. VOC
Hierarchical	subSup	1. mtreeNode	2. mtreeNode
Qualification	allowQul	1. QUL	2. mtreeNode
Vocabulary Tree Number	treeNum	1. VOC	2. mtreeNode
Vocabulary and Entry Term	see	1. VOC	2. ET
Vocabulary and Chemical Term	ChemVoc	1. Chemical	2. VOC
Chemical Indexing	index	1. Chemical	2. ChemII
Pharmacological Action	pharm	1. Chemical	2. ChemPA
Chemical Term CAS Number	CAS	1. Chemical	2. ChemRR
Chemical Term Source	source	1. Chemical	2. ChemSO
Chemical Term Synonym	synonym	1. Chemical	2. ChemSY

also linear.

Table 5.5 summarizes the characteristics of the MeSH graph. It gives the number of nodes, arcs, and the average and maximal degree in bipartite subgraphs induced by arc classes (listed in the 1st column). The last row in the table characterizes the whole MeSH graph. It can be seen from the table that there are two arc concentrations in subgraphs **treeNum** and **subSup** with average degrees at 2.436 and 10.99, respectively. The rest of the bipartite subgraphs have average degrees less than 2. The lowest average degree is for subgraph **XR**. Nodes with maximal degree appear in subgraph **ChemVoc**.

Table 5.6 gives further details on the connectivity for the subgraphs described in Table 5.5. Column 1 lists the names of arc-induced subgraphs. Column 2 and 3 respectively give the number of cyclic and acyclic components in each subgraph. Column 4 and 5 further list the number of nodes in the largest and second largest component in each subgraph. Column 6 is the number of isolated nodes with respect to each subgraph. Column 7 and 8 give the maximal and average size (in bytes) of nodes in each subgraph, respectively.

In general, the MeSH graph is of low density, with a moderate number of nodes and arcs. Since the average degree is low, it is expected that a simple algorithm will perform a

Table 5.5: Summary of the Characteristics of the MeSH Graph (Part A)

Name	Nodes	Arcs	Avg. Degree	Max Degree
XU	15843	2761	0.348	19
XR	15843	1575	0.198	19
FWD	15843	1494	0.189	18
allowQul	5415	2057	0.76	77
treeNum	20541	25022	2.436	134
subSup	4698	25829	10.99	36
see	75316	62951	1.672	86
ChemVoc	70257	30691	0.87	12596
index	56469	12479	0.442	548
pharm	55070	16135	0.586	1140
CAS	71887	17553	0.488	38
source	118583	72646	1.225	31
synonym	122633	68244	1.112	49
MeSH.ALL	287717	339751	2.362	12597

Table 5.6: Summary of the Characteristics of the MeSH Graph (Part B)

Name	Components		Nodes in Compt.		Isolated	Max	Avg.
	Cyclic	Acyclic	1st Lg.	2nd Lg.	Nodes	Size	Size
XU	1	1361	28	21	11721	1595	435.48
XR	37	560	54	38	13729	1595	435.48
FWD	79	510	36	32	13881	1595	435.48
allowQul	1	5240	175	1	5240	1182	355
treeNum	2	830	18456	17	122	1531	315
subSup	1	2	78	14	4596	11	10.91
see	602	13948	3391	496	2591	1531	97
ChemVoc	24	43458	12597	12153	43426	1531	187
index	10	1015	9626	43	43828	748	157.30
pharm	3	311	13733	48	40312	748	160.72
CAS	22	8192	38	36	46149	748	132.00
source	218	150	135	36	1097	748	96.40
synonym	218	150	135	36	1097	748	96.40
MeSH.ALL	103	18442	211008	67	864	1595	80.86

Table 5.7: Performance of SFM Algorithm on MeSH Graph

Path1			Path2		
Before	After	Ratio	Before	After	Ratio
132172	23160	82.50%	132172	23160	82.50%
52638	16339	68.90%	75140	6041	91.90%
23433	1962	91.60%	38016	485	98.70%
11184	2014	81.90%	18194	122	99.30%
5105	1533	69.90%	8336	19	99.80%
2622	689	73.70%	3703	5	99.90%
835	509	39.00%	1589	0	100%
279	5	98.20%	683	0	100%
114	2	98.30%	322	1	99.70%
49	0	100%	134	0	100%

decent partitioning job.

Partition Results

Table 5.7 shows the performance of the SFM algorithm on the MeSH graph. Since repeated bisection is applied, a binary tree can be perceived where a node stands for a graph of interest and its two child nodes the partitioned subgraphs of the parent graph. In Table 5.7, the partition situation along two arbitrarily selected paths going from the root to the leaves in such a binary tree are shown. The "Before" column lists cross arcs before the split and the "After" column gives the number after the split. The "Ratio" column calculates the percentage reduction in cross arcs, using the formula (`Before` – `After`)/`Before`.

Table 5.8 summarizes the connectivity of partitioned MeSH with respect to arc types. Here only the results of the storage plan where arcs are stored as individual objects are reported. There are 6 columns in the table. The first gives the arc class names. The remaining 5 columns contain information about the connectivity of interior or cross arcs. Column 2 lists, for each arc type, the number of cross arcs that cross at least one page pair. Column 3 lists the number of such page pairs crossed by these arcs. Column 4 gives the number of cross page arcs, that each crosses one page pair. Column 5 lists the number of

Table 5.8: Summary of MeSH Page Connections

Arc Type	Cross Page			Intra Page	No. Pages Touched
	Multiple	PagePairs (Multiple)	Single		
XU	22	11	1769	970	3490
XR	52	26	928	595	3490
FWD	88	43	800	606	3490
allowQUL	1591	129	48	418	981
treeNum	7267	3004	16287	1468	3512
subSup	18151	1556	2516	5162	1628
see	1813	607	1260	59878	816
ChemVoc	27026	5297	2088	1577	3546
index	6881	570	7501	3510	4236
pharm	4019	1526	10110	2006	3927
CAS	173	57	194	17180	3927
source	1099	343	4679	66868	3739
synonym	1099	197	568	67106	3940

Table 5.9: Packing Results

Before Packing		After Packing	
Pages	Utilization	Pages	Utilization
5047	71.4%	4334	83.2%

interior arcs. The last column gives the total number of pages on which arcs of each type are stored.

Table 5.9 reports space utilization before and after the packing process. Again, only the results of the storage plan where arcs are stored as individual objects are shown. In the table, it is assumed that an arc takes 18 bytes to store. As can be seen, the packing improves the utilization about 11.8%.

5.6 Performance Measurements

5.6.1 Testing Environment

The testing environment is a NeXTstation with a 68040 CPU (25 MHz) running NeXT operating system version 2.0. In the following, we abbreviate the operating system as NOS. The workstation has 20 MB of main memory, a 400 MB internal and a 1200 MB external disk. The external disk is directly mounted on the NOS file system. As a result, all objects are stored on files managed by the Mach file system.

Table 5.10 compares the elapsed time (in microseconds) regarding C I/O library buffered `fseek()`, `fread()` and NOS cached `seek()`, `read()` function calls. For C I/O buffering, two cases are distinguished: the use of the default C I/O library buffer and the LEND assigned buffer (through routine `setbuf()`). The purpose of the comparison is to test the performance of the cached and the C I/O library buffered reads. It is also beneficial to know the actual data transfer rate of the external disk, to be used in the experiment.

In the test, strings with sizes ranging from 512 to 8192 bytes are read in from a 35 MB file, using the three methods. The timings are averaged over 5000 reads starting at randomly selected positions (multiples of 512). As can be seen from the table, LEND buffered C I/O reads use less time than NOS cached reads. It also outperforms C I/O reads. 8192 seems a size that all three methods can perform equally well. As a result, the Unix file storage member function `readString()` is set to use C library I/O reading with self-assigned buffers. The page size is set to 8192 bytes.

Since the cache inside to the file system will impact the performance, two sets of testing condition are considered: "cold" start and "warm" start. By the "cold" start we mean the content of the NOS system cache is not useful to the retrieval. The "Cold" start condition is achieved through reading in 20 MB of garbage data by a separate program. The garbage data will fill the operating system buffer cache. After that reading operation, LEND is started. The "warm" start is one that initiates LEND immediately after a "cold" started test run finishes.

Table 5.10: Comparison of Timings of Seek and Read Routines

Bytes Read	C Buffered		LEND Buffered		NOS Cached	
	fseek()	fread()	fseek()	fread()	seek()	read()
512	131	27872	127	15895	82	16953
1024	141	27468	127	15913	83	17182
2048	133	26396	135	16183	84	17363
4096	138	23579	134	16814	89	17861
8192	139	18348	127	17735	99	18457
16384	139	28273	131	17937	90	29904

The testing data is the MeSH thesaurus, loaded in the unclustered and clustered forms. The former just uses a separate file for all objects of a class. The latter mixes all node and arc objects into a single file, using the SFM algorithm. Arcs are stored individually in both cases. Indices are stored in separate files in both cases too.

5.6.2 Node Retrieval Experiment

Table 5.11 and 5.12 report the node retrieval timing on four different classes of node objects.

Table 5.11 lists average time (in microseconds) taken by four preparation tasks before fetching 2,000 node objects. The first task is to obtain the class object which occupies a couple of bytes and is inexpensive to obtain. The second task is to get hold of the member class representative. This representative is used to act as the query object and to restore the desired object from a byte string. Again, it is cost-effective to acquire the representative. The third task is to initialize the representative with the query (in ASCII form). The representative becomes a query object afterwards. The last task is to find the location of the desired object through a MPHF index. The hash value is computed without visiting the external disk. The object location indexed by the value is then fetched from the disk. Due to this, indexing cost is about an order of magnitude higher than the other three tasks.

Table 5.12 lists average time (in milliseconds) taken to retrieve objects (2,000 of them) once their locations are known. Timings for both the cold and the warm start cases are given. It can be seen that clustered storage can speed up the object retrieval, except for

Table 5.11: Average Preparation Time for Node Retrieval

MeSH Form	Class Name	Get Class	Get Class Representative	Form Query	Search Index	
					Cold	Warm
Unclustered	VOC	113	45	162	3103	4978
	ET	119	44	170	4016	3603
	QUL	104	32	110	1013	1020
	Chemical	164	45	165	4067	3010
Clustered	VOC	123	47	176	2987	2808
	ET	110	45	174	4509	3949
	QUL	107	33	110	995	991
	Chemical	174	46	171	4452	3931

Table 5.12: Average Node Retrieval Time

Class Name	Unclustered		Clustered	
	Cold	Warm	Cold	Warm
VOC	27.5	4.9	11.1	3.7
ET	7.8	3.6	11.8	3.3
QUL	3.6	2.7	3.6	1.4
Chemical	18	4.7	13.5	3.5

the "cold" run to retrieve ET objects. The gain of the clustered method for the warm start condition is also marginal with respect to ET objects. There is a tie in "cold" run times for QUL objects. For all other cases, the difference of average "warm" run timings of the two storage methods is at least 1 millisecond.

We observe that the object size and NOS file system cache play an important role in performance. For the clustered method, we consider the graph density, the quality of the SFM algorithm results and the LEND internal LRU buffer policy as three additional factors. Since little is known about the file cache in NOS, the effects of these factors and their combinations are very difficult to determine. Nevertheless, though the clustered method is intended for faster path retrieval, it is useful to have preliminary evidence that it may help with the node object retrieval.

5.6.3 Path Retrieval Experiment

Table 5.13 reports the results of five path retrieval tests on the MeSH graph. Each test uses a distinct path query form (identified in column 1) which specifies the labels on the paths to be retrieved. The notation of path specification shown in the table is composed of a sequence of pairs. Each pair is made up of an arc label (the arc class name) and a direction indicator, which can be either plus(+) or minus(-). A plus means following the directed arc, while a minus means following the inverse of the directed arc. These paths start from 200 randomly selected source nodes. Queries Q_2, Q_3 and Q_4 are against the MeSH thesaurus hierarchy. Each query starts at the different level of the hierarchy, where the root of the hierarchy is numbered level 1.

Column 2 lists the total number of sink nodes found for each query specification. Column 3 indicates how many sink nodes are adjacent to each source node considered, on average. Columns 4 and 5 are statistics about MeSH loaded in the unclustered fashion. Columns 6 and 7 are statistics about MeSH loaded in the clustered fashion. Columns 4 and 6 show the "cold" start timing, while columns 5 and 7 show the "warm" start timing. All are in milliseconds. The indexing costs are not included as identical indexing mechanisms are used.

In general, the clustered method outperforms the unclustered one (on average) in all path retrieval tests for both the "cold" and "warm" start cases. This is true for queries (e.g., Q_1) that generate a few sink nodes, and for ones (e.g., Q_3, Q_4) that generate a large number of sink nodes. This suggests that the SFM algorithm can generate good partitioning for the MeSH data. Additional experiments can be run on this and other collections to further demonstrate the utility of the approach.

5.7 Summary

In this chapter, a simplified algorithm partitioning large and relatively sparse graphs and placing partitions of graphs on pages is described. The approach tries to cluster adjacent

Table 5.13: Timing Results on Path Retrieval Test

Query	Total Sink Nodes	Avg. Sinks per Source	Unclustered		Clustered	
			Cold Time	Warm Time	Cold Time	Warm Time
Q_1	247	1.23	7.6	2.37	5.9	1.8
Q_2	29997	149.9	121.6	137.0	81.4	74.1
Q_3	26552	132.8	107.9	107.5	73.9	71.3
Q_4	150612	753.1	804.8	804.9	643.6	646.3
Q_5	6059	30.3	35.8	25.3	20.3	10.4

Note:
Q_1 = (source-);
Q_2 = (treeNum+)(treeNum+), source node at the 1st level of the thesaurus hierarchy;
Q_3 = (treeNum+)(treeNum+), source node at the 2nd level of the thesaurus hierarchy;
Q_4 = (treeNum+)(treeNum+)(treeNum+), source node at the 3rd level of the thesaurus hierarchy;
Q_5 = (synonym-)(ChemVoc+)(see+);
Each query is run against 200 source nodes.

nodes (objects) onto the same page of a storage disk.

A set of data structures is then described which fully takes advantage of the graph partitioning. Coupled with MPHF indexing, the data structures make the random path access efficient. This claim is verified in a preliminary fashion by using MeSH in a series of experiments. The measurements on performance tests for node and path access on MeSH are included.

Chapter 6

Conclusions

6.1 Contributions

In this dissertation, we have described an object-oriented model (the LEND model). The purpose of the model is to provide a foundation for integrated storage and access of data based on a variety of IR models. Typical kinds of data considered include documents, concepts, thesauri, and hypertexts. The model has new features related to graph data, compared with existing object-oriented models. The graph-related features lead to natural expression of IR data in the model, especially for the thesauri encoded in semantic network forms and hypertexts. A query language which can simplify the formulation of graph-oriented queries is also presented.

The model has led to development of a prototype object-oriented database system to support IR applications. It is thus shown that object-oriented database techniques can help confirm the main hypothesis of this dissertation: integration of data storage and access for IR is possible using object-oriented techniques. Additional benefits include encapsulated data representation and easy extension of systems with new data types.

In this dissertation, we also developed data structures and algorithms to improve the efficiency of accessing graph data. The class of minimal perfect hashing algorithms proposed can in expected linear time find minimal perfect hash functions (MPHFs). MPHFs require

only one probe to fetch a node or a set of nodes from disk. Hence they are the optimal solution to the single-key retrieval problem. The hash table is 100% utilized and the function specification space approaches the theoretical lower bound. The elegance of these algorithms lies in the transformation of the problem to a workable domain and the application of various searching heuristics. To the best of the author's knowledge, these algorithms are the best so far when compared with those that can be found in the literature. They have wide applicability, including for compiler design, sparse array compaction and database systems.

The object placement problem is also studied in this dissertation in the context of clustering nodes of a graph for efficient random traversal operations. The graph studied (MeSH) is large, static, and relatively sparse. The approach applies clustering techniques to the entire graph. A suitable algorithm for such graphs is identified. The quality of the resulting partition seems quite satisfactory with respect to the traversal operation. It is conjectured that the approach will be applicable to other graphs with similar characteristics.

6.2 Future Research

This dissertation suggests several specific areas for future research: the LEND model area, the LEND system area, the MPHF algorithms area and the graph storage area.

6.2.1 LEND Model Research

The LEND model has extended the "classical" features (i.e., sets, lists, tuples) found in typical object-oriented database systems with the graph constructs. This new feature is powerful enough to handle applications originated from SNePS, hypertext and several important IR models.

Conceptualizing data as graphs has been a tradition in data model design for capturing important semantics. The Entity-Relationship Model [Che76] can be viewed as one such model. The LEND model puts no restriction on the content of nodes, as long as they are node objects. The LEND model does not put limits on general arc objects either.

Constraints on both nodes and arcs of particular kinds are supposed to be specified in their specialized classes. An interesting research topic, therefore, would be to see how well the LEND model can be applied to model typical database, information retrieval and knowledge based systems applications.

The pattern language is relatively declarative and set-based. Operators of the language have corresponding member functions defined on objects in the model. The declarative and procedural aspects of member functions are thus reflected in these operations. Making a SQL or SQL 3 dialect of the pattern language would be interesting and valuable.

6.2.2 LEND System Research

The class representative vector in the object layer plays an important role regarding object delivery performance. The key technique is the MPHF indexing and self duplication of objects. As a MPHF theoretically guarantees optimal access, the speed of identification of a particular class representative is assured. However, it is still interesting to compare the search performance of MPHF and some other algorithms (i.e., the binary searching) on memory resident data. A further question is whether h_{10} and h_{20} of the faster MPHF algorithm can be simplified. The simplification of the two functions should speed up the process of mapping a key to its hash value.

Object duplication is relatively expensive, particular for large objects. It is difficult for the object layer to know how the object asked will be used. If the usage can be known in advance, then the layer can initiate duplication action only for the components that will be used. This idea seems at odds with the encapsulation principle with respect to the object layer, and requires an object representation to be exposed to the object layer. However, by adding the component selection argument to the member function `copy`, we can realize the partial copy without sacrificing the benefit of encapsulation. A similar argument can be made regarding the member function `from_string` so that the object is partially initialized. An alternative and less attractive approach would be to have fixed representations for well-known constructs in the LEND model (i.e., set, list and tuple).

Some later versions of existing object-oriented database systems such as [BOS91] adopt an idea that entirely avoids copy. These systems put disk pages containing objects directly in the user's space. It is conceivable that this practice can get rid of expensive object copy operations at the risk of reducing the degree of page protection. The LEND system can certainly evolve toward this direction. Specifically, the LEND objects can have two representations: one that is self contained and independent of any pages and the other partial and containing pointers to pages.

Another important line of research regarding LEND is to make it a distributed system.

6.2.3 MPHF Algorithm Research

The MPHF algorithms presented in Chapter 4 are probabilistic. They use heuristics and are able to find MPHFs with small specification space in expected linear time. It is not known whether there exist polynomial algorithms that can produce the same results as the probabilistic algorithms. Putting this into a more general perspective, we assume b bits are given as the search space. A MPHF is sought for a key set of size n. It is interesting to know the time complexity of the problem of finding a MPHF. As a comparison, Fredman, Komlós and Szemerédi in [FKS84] have shown that their method can find a PHF in $O(n^3 \log u)$ time, where u is the size of the key universe. However, the PHF space is about $n + o(n)$ computer words.

Variations of the MPHF problem studied in the dissertation exist. The most practical is one where keys are of variable length and the keys are to be stored consecutively in the hash table. The problem has a nice OPMPHF solution which takes at least $O(n \log n)$ bits. It is interesting to know whether a MPHF solution exists that requires less than $O(n \log n)$ bits, for example $O(n)$ bits.

It would also be interesting to verify the variant version of the faster MPHF algorithm, and to compare its running time and MPHF space with those of the faster MPHF algorithm.

The Bin hashing algorithm also needs more experimental study. In particular, it is interesting to know the impact of the combination of the sorting strategy (Sort 1 or Sort

2) with the searching strategy (First fit or Best fit) on the running time and the size of the function space.

6.2.4 Graph Storage Research

The SFM algorithm has been proved useful for the MeSH thesaurus which is large, and relatively sparse. However, looking at the results, it can be seen that the partition quality varies with respect to the density of connections. Large variations of density can be expected to exist in other realistic data. The summary table in Johnson *et. al.* ([JAMS89], pp. 888, Table XIII) reports that the coalesced KL dominates both the KL and the Annealing algorithm for small average degree (2.5) and relatively large (1,000 nodes) graphs, while the Annealing algorithm can outperform the KL and the coalesced KL for large average degree (20.0) graphs (1,000 nodes). Given an array of different algorithms suitable for graphs of distinct densities, it is interesting to know the partitioning performance if they are separately applied to different portions of a graph. The idea is similar to that of expert systems, where a group of experts are working cooperatively, and each expert makes unique contributions to solving a problem. Here the problem is graph partitioning and experts are different algorithms each suitable for graphs of a certain density.

Mixed storage of the graph and its support indices is another appealing problem. Recall that graph partitioning generates a layout plan where the space utilization is seldom 100%. Recall also that the indices which likewise take space are necessary to identify nodes to start a traversal. It is beneficial to pack the page sets with the indices for a compact storage. The idea is sketched in the following paragraph.

Remember that the indices based on MPHFs map keys to a hash value (an integer) and then to a list. The elements of the list are locations of the objects that somehow match the key. Note that all hash values of a MPHF form an integer set. In the actual implementation, these hash values represent indices of an array which contain pointers to those location lists. Based on this configuration, packing of location lists and the hash value arrays are feasible. To pack a location list, we just find a page with sufficient empty space. A clever strategy is

to find a page that best fit the list. The quality of fit can be measured based on the number of locations in the page that are contained in the list.

REFERENCES

[AHU85] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addision-Wesley, Reading, Mass., 1985.

[AJ90] Rakesh Agrawal and H. V. Jagadish. Hybrid transitive closure algorithms. In *Proceedings of the 16th Very Large Database Conference*, pages 326–334, Brisbane, Australia, 1990.

[ASH84] M. Ai-Suwaiyel and E. Horowitz. Algorithms for trie compaction. *ACM Transactions on Database Systems*, 9:243–263, June 1984.

[Aus60] T.L. Austin. The enumeration of point labeled chromatic graphs and trees. *Canadian Journal of Mathematics*, 12:535–545, 1960.

[AV79] D. Angluin and L. Valiant. Fast probabilistic algorithms for Hamiltonian circuits and matchings. *Journal of Computer and System Sciences*, 19:155–193, 1979.

[BKKG88] Jay Banerjee, Won Kim, Sung-Jo Kim, and Jorge F Garza. Clustering a DAG for CAD databases. *IEEE Transactions on Software Engineering*, 14:1684–1699, November 1988.

[BM72] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[BM76] J. A. Bondy and U.S.R. Murty. *Graph Theory with Applications.* The Macmillan Press Ltd, London, 1976.

[BMPA88] D.A. Bell, E.J. Mcerlean, P.M.Stewart, and W. Arbuckle. Clustering related tuples in databases. *The Computer Journal*, 31:253–257, 1988.

[BOS91] Paul Butterworth, Allen Otis, and Jacob Stein. The GEMSTONE object database management system. *Communications of the ACM*, 34:64–77, October 1991.

[Boy77] R.R. Boyer. A fast string searching algorithm. *Communications of the ACM*, 20:762–772, 1977.

REFERENCES

[Bri59] De La Briandais. File searching using variable length keys. In *Proceedings of the Western Joint Computer Conference*, pages 295–298, 1959.

[BT90] M.D. Brain and A.L. Tharp. Perfect hashing using sparse matrix packing. *Information Systems*, 15:281–290, 1990.

[Cat91] E. G. G. Cattell. *Object Data Management.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1991.

[CD85] Hong-Tai Chou and David J. DeWitt. An evaluation of buffer management strategies for relational database systems conference. In *Proceedings of Very Large Database*, pages 127–141, Stockholm, 1985.

[CDLR90] Sophie Cluet, Claude Delobel, Christophe Lécluse, and Philippe Richard. Reloop, an algebra based query language for an object-oriented database system. In Won Kim, Jean-Marie Nicolas, and Shojiro Nishio, editors, *Deductive and Object-Oriented Databases*, pages 313–332. North-Holland, 1990.

[CH91] Jia Bing R. Cheng and A. R. Hurson. Effective clustering of complex objects in object-oriented databases. *SIGMOD Record*, 20:22–31, June 1991.

[Cha84] C. C. Chang. The study of an ordered minimal perfect hashing scheme. *Communications of the ACM*, 27:384–387, 1984.

[Cha86] C. C. Chang. Letter oriented reciprocal hashing scheme. *Information Science*, 38:243–255, 1986.

[Che76] P.P.-S. Chen. The entity-relationship model. *ACM Transactions on Database Systems*, 1:1–36, 1976.

[CHK85] G.V. Cormack, R.N.S. Horspool, and M. Kaiserswerth. Practical perfect hashing. *The Computer Journal*, 28:54–58, 1985.

[Cic80] R.J. Cichelli. Minimal perfect hash functions made simple. *Communications of the ACM*, 23:17–19, 1980.

[CKB83] N. Cercone, M. Krause, and J Boates. Minimal and almost minimal perfect hash function search with application to natural language lexicon design. *Computers and Mathematics with Applications*, 9:215–231, 1983.

[Cod70] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13:377–389, 1970.

[CP85] W. Bruce Croft and Thomas J. Parenty. A comparison of a network structure and a database system used for document retrieval. *Information Systems*, 10:377–390, 1985.

REFERENCES

[CW79] J.L. Carter and M.N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[Deu91] O. Deux. The O_2 system. *Communications of the ACM*, 34:34–48, October 1991.

[ED88] R.J. Enbody and H.C. Du. Dynamic hashing schemes. *ACM Computing Surveys*, 20:85–113, 1988.

[ER66] P. Erdos and A. Renyi. On the existence of a factor of degree one of connected random graphs. *Acta Math. Acad. Sci. Hung.*, 17:359–379, 1966.

[FCDH90] Edward A. Fox, QiFan Chen, Amjad M. Daoud, and Lenwood S. Heath. Ordering preserving minimal perfect hash functions and information retrieval. In Lean-Luc Vidick, editor, *the 13th International Conference on Research and Development in Information Retrieval*, pages 279–311, Brussels, Belgium, September 1990. ACM.

[FCDH91] Edward A. Fox, QiFan Chen, Amjad M. Daoud, and Lenwood S. Heath. Ordering preserving minimal perfect hash functions and information retrieval. *ACM Transactions on Information Systems*, 9:281–308, July 1991.

[FCHD89] Edward A. Fox, QiFan Chen, Lenwood S. Heath, and Sanjeev Datta. A more cost effective algorithm for finding perfect hash functions. In *Proceedings of 1989 ACM 17th Annual Computer Science Conference*, pages 114–122, Louisville, Kentucky, February 1989. ACM.

[FCHD90] Edward A. Fox, QiFan Chen, Lenwood S. Heath, and Amjad M. Daoud. A minimal perfect hash functions finding algorithm for large databases. Technical Report TR-90-41, Department of Computer Science, Virginia Polytechnic Institute & State University, 1990.

[Fel68] W. Feller. *An Introduction to Probability Theory and its Applications, Vol 1.* John Wiley and Sons, New York, 1968.

[FHCD92] Edward A. Fox, Lenwood S. Heath, QiFan Chen, and Amjad M. Daoud. Practical perfect hash function finding algorithms for large databases. *Communications of the ACM*, 35:105–121, January 1992.

[FKCF91] Edward A. Fox, M.P. Koushik, QiFan Chen, and R. K. France. Integrated access to a large medical literature database. Technical Report TR-91-15, Department of Computer Science, Virginia Polytechnic Institute & State University, Blacksburg, VA 24060, 1991.

[FKS84] M.L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Communications of the ACM*, 31:548–544, 1984.

REFERENCES

[FM82] C.M. Fiduccia and R.M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the ACM IEEE 19th Design Automation Conference*, pages 175–181, Caesars Palace, Las Vegas, Nevada, June 1982.

[FN87] E.A. Fox and J.T. Nutter. Building a large thesaurus for information retrieval. Proposal submitted to NSF, 1987.

[FNA+88] Edward A. Fox, J. Terry Nutter, Thomas Ahlswede, Matha Evens, and Judith Markowitz. Building a large thesaurus for information retrieval. In *Proceedings Second Conference on Applied Natural Language Processing*, pages 101–108, Austin, TX, February 1988.

[Fox80] Edward A. Fox. Lexical relations: enhancing effectiveness of information retrieval systems. *ACM SIGIR Forum*, 15:5–36, 1980.

[Fre60] E. Fredkin. Trie memory. *Communications of the ACM*, 3:490–499, September 1960.

[GB83] M.K. Goldberg and M. Burstein. Heuristic improvement technique for bisection of VLSI networks. In *Proceedings of IEEE International Conference on Computer Design*, pages 122–125, Port Chester, New York, 1983.

[GH90] Olivier Goldschmidt and Dorit S. Hochbaum. A fast perfect-matching algorithm in random graphs. *SIAM J. on Discrete Mathematics*, 3:48–57, February 1990.

[GL88] G.L. Gonnet and P. Larson. External hashing with limited internal storage. *Communications of the ACM*, 35:161–184, 1988.

[Han79] P. Hanks. *Collins English Dictionary.* William Collins Sons & Co., London, 1979.

[HK87] Richard Hull and Roger King. Semantic database modeling: survey, applications, and research issues. *ACM Computing Surveys*, 19:201–260, 1987.

[HM86] Susan M. Humphreys and Nancy E. Miller. The NLM indexing aid project. In *Proceedings on the 49th Annual Meeting of the American Society for Information Science*, pages 106–112, Chicago, IL, September 1986.

[HS78] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, 1978.

[HS90] Frank Halasz and Mayer Schwartz. The Dexter hypertext reference model. In *Hypertext Standardization Workshop*, National Institute of Standards and Technology (NIST), 1990.

[Jae81] G. Jaeschke. Reciprocal hashing—a method for generating minimal perfect hash functions. *Communications of the ACM*, 24:829–833, 1981.

REFERENCES

[JAMS89] David Johnson, Celilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization of simulated annealing: An experimental evaluation; Part I, graph partitioning. *Operations Research*, 37:865–891, November-December 1989.

[JDU+74] D.S. Johnson, A. Demers, J.D. Ullman, M.R. Garey, and R.L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM J. Comput.*, 3:299–300, December 1974.

[KGS91] Jeffrey E. Kotteman, Michael D. Gordan, and Jack W. Stott. A storage and access manager for ill-structured data. *Communications of the ACM*, 34:94–103, 1991.

[Kim90] Won Kim. Object-oriented databases: Definition and research directions. *IEEE Transactions on Knowledge and Data Engineering*, 2:327–341, September 1990.

[KL70] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, pages 291–307, February 1970.

[KL89] Won Kim and Frederick H. Lochovsky, editors. *Object-Oriented Concepts, Databases, and Applications.* ACM Press, 1989.

[Knu73a] D.E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley Publishing Company, Reading, MA, 1973.

[Knu73b] D.E. Knuth. *The Art of Computer Programming*, volume 4. Addison-Wesley Publishing Company, Reading, MA, 1973.

[Kou91] P. Koushik. MeSH data description. Internal Document, Department of Computer Science, Virginia Polytechnic Institute & State University, February 1991.

[Kri84] B. Krishanamurthy. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Transactions on Computers*, C-33:438–446, May 1984.

[Lav91] Joseph W. Lavinus. Efficient linearization of a graph-structured database. Thesis proposal, Department of Computer Science, Virginia Polytechnic Institute & State University, October 1991.

[LLOW91] Charles Lamb, Cordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34:50–63, 1991.

[LRV90] Christophe Lécluse, Philippe Richard, and Fernando Velez. O_2, an object-oriented data model. In Stanley B. Zdonik and David Maier, editors, *Readings in Object-Oriented Database Systems*, pages 227–241. Morgan Kaufmann Publishers, Inc, San Mateo, CA, 1990.

[LW89] Carl E. Langenhop and William E. Wright. A model of the dynamic behavior of B-trees. *Acta Informatica*, 27:41–59, 1989.

REFERENCES

[MA87] I.A. Macleod and Reuber A.R. The array model: a conceptual modeling approach to document retrieval. *Journal of the American Society for Information Science*, 38:162–170, 1987.

[Mai83] H.G. Mairson. The program complexity of searching a table. In *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science*, pages 43–47, 1983.

[Meh82] K. Mehlhorn. On the program size of perfect and universal hash functions. In *Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science*, pages 243–263, 1982.

[Mor85] Ernesto J.M. Morgado. *Semantic networks as abstract data types.* PhD thesis, Department of Computer Science, University of New York at Buffalo, 1985.

[Mos90] J. Eliot B. Moss. Design of the Mneme persistent object store. *ACM Transactions on Information Systems*, 8:102–139, April 1990.

[Mot89] Rajeev Motwani. Expanding graphs and the average-case analysis of algorithms for matchings and related problems. In *Proceedings of the 21th Annual Symposium on Theory of Computing*, pages 550–561, Seattle, Washington, May 1989.

[NFS91] Raymond T. Ng, Christos Faloutsos, and Timos Sellis. Flexible buffer allocation based on marginal gains. In *SIGMOD Record*, pages 387–396, Denver, Colorado, May 1991.

[Ore89] Tim Oren. Report on the Dexter workshop hypertext model. Workshop report, 1989.

[Pal85] E.M. Palmer. *Graphical Evolution: An Introduction to the Theory of Random Graphs.* John Wiley & Sons, New York, 1985.

[Par85] Edgar M. Parmer. *Graphical Evolution.* John Wiley & Sons, New York, 1985.

[Pea90] P.K. Pearson. Fast hashing of variable-length text strings. *Communications of the ACM*, 33:677–680, 1990.

[PM88a] S.K. Park and K.W. Miller. Random number generators: good ones are hard to find. *Communications of the ACM*, 31:1192–1201, 1988.

[PM88b] J. Peckham and F. Maryanski. Semantic data models. *ACM Computing Surveys*, 3:153–19, 1988.

[RJ76] S.E. Robertson and K.S. Jones. Relevance weighting of search terms. *Journal of the American Society for Information Science*, 27:129–146, 1976.

REFERENCES

[RL89] M.V. Ramakrishna and P. Larson. File organization using composite perfect hashing. *ACM Transactions on Database Systems*, 14:231–263, 1989.

[RW86] V.V. Raghavan and S.K.M. Wong. A critical analysis of vector space model for information retrieval. *Journal of the American Society for Information Science*, 37:279–287, September 1986.

[Sag84] T.J. Sager. A new method for generating minimal perfect hashing functions. Technical Report CSc-84-15, Department of Computer Science, University of Missouri-Rolla, Missouri, 1984.

[Sag85] T.J. Sager. A polynomial time generator for minimal perfect hash functions. *Communications of the ACM*, 28:523–532, 1985.

[San89] Laura A. Sanchis. Multiple-way network partitioning. *IEEE Transactions on Computers*, 38:62–81, 1989.

[Sed88] R. Sedgewick. *Algorithms.* Addison-Wesley Publishing Company, Reading, MA, 1988.

[SFW83] Gerard Salton, Edward. A. Fox, and Harry Wu. Extended boolean information retrieval. *Communications of the ACM*, 26:1022–1036, December 1983.

[Sha79] Stuart C. Shapiro. The SNePS semantic network processing system. In *Associative Networks - The Representation and Use of Knowledge by Computers*, pages 179–203. Academisc Press, New York, 1979.

[Sha89] Stuart C. Shapiro. *SNePS-2 User's Manual.* Department of Computer Science, State University of New York at Buffalo, 1989.

[SK91] Michael Stonebraker and Greg Kemnitz. The POSTGRES next-generation database management system. *Communications of the ACM*, 34:78–92, October 1991.

[SM83] G. Salton and M.J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill Book Company, 1983.

[SM90] Khushro Shahookar and Pinaki Mazumder. A genetic approach to standard cell placement using meta-genetic parameter optimization. *IEEE Transactions on Computer-Aided Design*, 9:500–511, May 1990.

[SÖ90] Dave D. Straube and M. Tamer Özsu. Queries and query processing in object-oriented database systems. *ACM Transactions on Information Systems*, 8:387–430, 1990.

[Spr77] R. Sprugnoli. Perfect hashing functions: a single probe retrieving method for static sets. *Communications of the ACM*, 20:841–850, November 1977.

[SS89] J.P. Schmidt and A. Siegel. On aspects of universality and performance for closed hashing. In *Proceedings of the 21st ACM Symposium on Theory of Computing*, pages 355–366, 1989.

[SSV87] C. Sechen and A. Sangiovanni-Vincentelli. The TimberWolf placement and routing package. *IEEE J. Solid-State Circuits*, SC-20:510–522, April 1987.

[Str86] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[SWC75] G. Salton, A. Wong, and Yang C.S. A vector space model for automatic indexing. *Communications of the ACM*, 18:613–620, 1975.

[TC91] Howard Turtle and W. Bruce Croft. Evaluation of an inference network-based retrieval model. *ACM Transactions on Information Systems*, 9:187–222, July 1991.

[TL82] D.C. Tsichritzis and F.H. Lochovsky, editors. *Data Models.* Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1982.

[Tom89] F. W. M. Tompa. A data model for flexible hypertext database systems. *ACM Transactions on Information Systems*, 7:85–100, January 1989.

[Tur91] Howard Robert Turtle. *Inference Networks for document retrieval.* PhD thesis, University of Massachusetts, 1991.

[TY79] Robert E. Tarjan and Andrew Chi-Chih Yao. Storing a sparse table. *Communications of the ACM*, 22:606–611, November 1979.

[Wal80] David W. Walkup. Matchings in random regular bipartite digraphs. *Discrete Mathematics*, 31:59–64, 1980.

[Wan89] Yair Wand. A proposal for a formal model of objects. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 537–559. Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[WZRR87] S.K.M. Wong, W. Ziarko, V.V. Raghavan, and Wong R.C.N. On modeling of information retrieval concepts in vector spaces. *ACM Transactions on Database Systems*, 12:299–321, 1987.

[ZM90] Stanley B. Zdonik and David Maier, editors. *Readings in Object-Oriented Database Systems.* Morgan Kaufmann, 1990.

VITA

QiFan Chen was born 31 November 1959 in Shanghai, China. He received his Bachelor of Science from Shanghai JiaoTong University in July of 1982. Immediately afterwards, he joined the Microcomputer Research Institute of the same school and began his graduate program. In March 1985, he received his Masters degree. From March to August 1985, he worked in the Microcomputer Research Institute gaining valuable experience in system engineering and management. In October 1985, he started his Ph.D program with the Department of Computer Science, VaTech, and finished the required coursework in 1987. From then on, he has been fully participated in the research and development of efficient data structures and algorithms for data access, prototyping an object-oriented database system for information systems applications. This dissertation is the summary of the work fulfilled.