TRIE METHODS FOR STRUCTURED DATA ON SECONDARY STORAGE

by
XIAOYAN ZHAO

School of Computer Science
McGill University
Montréal, Québec
Canada

October 2000

A DISSERTATION
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
OF MCGILL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY

Abstract

This thesis presents trie organizations for one-dimensional and multidimensional structured data on secondary storage. The new trie structures have several distinctive features: (1) they provide significant storage compression by sharing common paths near the root; (2) they are partitioned into pages and are suitable for secondary storage; (3) they are capable of dynamic insertions and deletions of records; (4) they support efficient multidimensional variable-resolution queries by storing the most significant bits near the root.

We apply the trie structures to indexing, storing and querying structured data on secondary storage. We are interested in the storage compactness, the I/O efficiency, the order-preserving properties, the general orthogonal range queries and the exact match queries for very large files and databases. We also apply the trie structures to relational joins (set operations).

We compare trie structures to various data structures on secondary storage: multipaging and grid files in the direct access method category, R-trees/R*-trees and X-trees in the logarithmic access cost category, as well as some representative join algorithms for performing join operations. Our results show that range queries by trie method are superior to these competitors in search cost when queries return more than a few records and are competitive to direct access methods for exact match queries. Furthermore, as the trie structure compresses data, it is the winner in terms of storage compared to all other methods mentioned above.

We also present a new tidy function for order-preserving key-to-address transformation. Our tidy function is easy to construct and cheaper in access time and storage cost compared to its closest competitor.

Résumé

Cette thèse prèsente des structures de trie pour des données unidimensionnelles et multidimensionnelles sur la mémoire secondaire. Les nouvelles structures de trie ont plusieurs dispositifs distincts: (1) elles fournissent la compression significative de données en partageant les voies d'accès communes près de la racine de disque; (2) elles sont divisées en pages et conviennent pour la mémoire secondaire; (3) elles permettent des mises en place et des suppressions dynamiques des enregistrements; (4) elles supportent des requêtes multidimensionnelles efficaces de résolution variable en enregistrant les bits les plus significatifs près de la racine.

Nous avons appliqués les structures de trie à l'indexation, l'enregistrement et la sélection des données structurées sur la mémoire secondaire. Nous sommes intéressés à la compacticité de mémoire, l'efficacité de E/S, les propriétés de conserver l'ordre, les requêtes orthogonales générales et les requêtes exactes pour les fichiers et les bases de données très grands. Nous avons utilisées également les structures de trie à l'apparenté de joint (opérations de pair).

Nous avons comparés des structures de trie aux autres diverses structures de données sur la mémoire secondaire: multipaging et grille classé dans la catégorie de méthode acces directe, le R-arbres /R*-arbres et les X-arbres dans la catégorie logarithmique de coût d'accès, ainsi que des algorithmes représentatifs pour exécuter des opérations de liens. Nos résultats prouvent que les requêtes d'intervalle par la méthode de trie sont supérieures à tous les ses concurrents sur le coût de recherche quand des requêtes retournant plus que seulement quelques enregistrements et sont concurrentielles aux méthodes d'accès direct pour des requêtes de recherches exactes. De plus, car la structure de trie comprime des données, elle est gagnante en termes de mémoire comparant à toutes autres méthodes mentionnées ci-dessus.

Nous présentons aussi une nouvelle fonction ("tidy function") pour des transformations clé-à-adressons avec l'ordre-préservé. Notre fonction "tidy" est facile à construire et peu coûteuse en temps d'accès et coût d'entreposage comparatovement à ses plus proches compétiteurs.

Acknowledgements

First and foremost, I would like to express my gratitude to my supervisor, Professor Tim Merrett, whose support and encouragement were indispensable throughout my doctoral program. He contributed a great deal of his time, effort and thought to the work presented in this dissertation; Professor Merrett has shown dedication to his students and his profession. During the years of my study in the program, I also received his constant financial support, without which it would be impossible for me to complete this program.

I am grateful to the School of Computer Science and IRIS for their financial support, and to my thesis committee members. Thanks must also go to Ms. Franca Cianci, Ms. Vicki Keirl, Ms. Teresa De Angelis and all the secretaries for their patience and readiness to provide administrative help, as well as all systems staff for their technical assistance.

I wish to thank all my friends during my years at McGill and Montréal for the joy and fun we shared. Special mention should be made of Mengxuan Zhuang, Jian Wang, Qin Huang, Nan Yang, Helen Qiao, Wei Gu, Xiaochen Zhang, Xinming Tian, Yanmei Zhang, and Song Hu.

Special thanks to Dr. Bill Dykshoorn, who did a great job of proofreading to get the writing into shape.

Thanks must also go to my dear parents and my brother for their love and constant support.

Finally, I would like to send a special note of appreciation to my husband, Ping Zhang, for his constant understanding and support, and for the love and joy we share.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Space and speed are two major considerations for storage and retrieval in large database systems on secondary storage. Furthermore, data types and distributions vary from application to application. There are special data such as text, spatial (vector) and image (pixel) data versus structured data such as payrolls and inventories; one dimensional versus multiple dimensional data; order-preserved data versus random-ordered data; uniformly versus non-uniformly distributed data; etc. Query selectivities performed on data also vary from one application to another, ranging from only one data item (record) to a significant percentage of source data files. Query operations can be unary or binary as well.

Among the various existing data structures for large database systems, a data structure that is good at structured data may become inefficient if it is applied to spatial or text data. A data structure that behaves quite well for uniform data may become inefficient for pathological data distributions. Data structures for organizing one dimensional keys may not work for keys with multi-dimensional attributes. Furthermore, data structures that support low selectivity queries may not be efficient in performing high selectivity queries.

Hashing in general is good at single retrieval only. B-trees [BM72] on secondary storage have logarithmic behavior. K-D-B-trees [Rob81] extend B-trees for multidimensional data. Their costs of accessing secondary storage depend on the height of trees, and thus the fan-out, as well as the data set size. The storage efficiency of grid files [NHS84], a multikey direct access method, is reduced by poorly distributed data. The access or storage efficiency of multipaging [MO81a, MO82] also decreases for pathological data distributions. Bang-files [Fre87] and the Interpolation Based Grid Files [OM92] use logarithmic accessing cost to compensate for the storage overhead of grid files for nonuniform data. R-trees [Gut84] and their variants [BKSS90, SR87] for spatial data contain overlapped hyper-rectangular regions which may lead to less efficient searches for some queries than others. In addition, all the above structures do not generally support text indexing and searching.

In summary, a data structure that is good at dealing with one kind of data in one particular area for one type of query is likely to have bad performance for another purpose. Is there a simple but powerful dynamic structure for large-scale databases that is efficient for text, spatial and structured data, in terms of both space and speed; regardless of the distribution, the dimensionality, the operations, and the query selectivity?

On the other hand, digital trees [Knu73], or *tries* [Bri59, Fre60], are tree structures that store data along paths rather than at nodes, which is what a tree structure usually does. They have many desirable characteristics. Among them, the following three are the most important.

1. They store data according to resolution, with the most important bits stored near the root. Thus, we call tries *variable resolution structures* or *zoom tries*. Such a zooming property of tries may speed up queries at different resolutions by starting with an approximation and refining it only when there is uncertainty. Tries have been applied to various spatial queries and map retrieving. However, it should be noted that the data need not to be spatial.
2. Tries compress data by requiring minimal storage overhead, only a couple of bits per node. Paths near the root are shared by many data. In text searching, such compression is important for indexing substrings in a large text, which requires at least a pointer to each character in the text.

3. Tries maintain order preservation in data, one of the fundamentals for efficient high selectivity queries.

Tries, simple yet powerful structures, have shown advantages in many application areas. In database systems they have achieved great performance in dealing with text data for which the trie compression makes substring indexing possible, which would otherwise be impractical. With spatial data, in addition to the storage advantage, tries provide a way to do variable resolution queries in sublinear time.

But are tries also strong at organizing structured data such as tables and relations? Are they efficient structures for both low selectivity (such as exact matches) and high selectivity (such as range queries)? Can general relational algebra, such as the join operations of relational databases be performed by tries efficiently, and how? If yes, what are the advantages of applying trie structures to general database systems compared with state-of-the-art data structures and algorithms? As far as we know, these remain open questions. This thesis attempts to answer these questions by extending trie methods to be dynamic and applying various unary and binary operations on structured data, and thus pursues the claim that tries offer the best general representations for large-scale databases.

1.2 Originality

To the best knowledge of the author, the originality of this work includes the following methods, algorithms, comparisons and corresponding experimental results:

- *DyOrTrie*, an extended pointerless bitstring representation for binary tries used for dynamic operations;
- A piece-wise, linear tidy function approximation method with minimum overflows, including construction and search algorithms for one-dimensional (1D) queries;
- Comparisons of tries versus the tidy functions for 1D queries;

- A new analysis of multidimensional data distribution for some direct access methods based on information theory;
- Experimental comparisons of tries versus various direct and logarithmic access methods for multidimensional unary queries, including exact match and range queries;
- Relational join algorithms by tries on structured data (binary operations on tries);
- Comparisons of trie join algorithms versus traditional and state-of-the-art join algorithms.

1.3 Glossary of Symbols

Here are symbols used through the thesis.

N:	number of keys or records in a file on secondary storage
n:	number of pages (blocks) in a file on secondary storage
P:	page capacity
k:	number of dimensions
p:	number of linear pieces for tidy functions
$D(x)$:	cumulative distribution function
B:	memory buffer size

1.4 Thesis Outline

The thesis is organized as follows. Chapter 1 presents motivation for the work and the problem domain, followed by a summary of new results.

Chapters 2 and 3 principally concentrate on a literature review of data structures and algorithms.

Chapter 2 concerns itself with trie structures. It reviews their properties, applications, representations and some refinements. The last section of the chapter presents an original dynamic trie structure. This dynamic trie is the underlying structure which is used throughout the thesis.

Conventional data structures and join algorithms are introduced in chapter 3. It reviews the data structures according to their dimensionality, from one-dimension (1D) to multidimensions. The 1D structures include hashing and order preserving key-to-address transformations. Multidimensional file structures are classified into direct access and logarithmic access files. A survey of join algorithm is also presented in this chapter.

The remaining chapters elucidate original work. They are organized in four chapters.

Chapter 4 introduces a heuristic piece-wise linear approximation tidy function on secondary storage. We show its superiority to its closest competitors – some order preserving hashing methods.

Chapters 5 to 7 present three trie applications.

Chapter 5 demonstrates tries for 1D queries, comparing with the piece-wise linear tidy function that we propose in the previous chapter. Detailed comparisons on storage and search cost are given.

Chapter 6 describes tries for multidimensional queries, comparing with grid files, multipaging, R*-trees and X-trees. Exact match and range query algorithms are provided, as well as detailed experimental comparison results and discussions on speed and space cost versus file size, record size, query selectivity, dimensionality, and distribution. The results indicate that tries are better than other data structures when files contain more than a few records and the query returns more than a few records.

Chapter 7 demonstrates how tries can be used for binary join operations. Natural join and union join algorithms are presented. When join attributes are organized by tries, we show their significant advantage over all other join algorithms based on both theoretical analysis and experimentations.

Chapter 8 summarizes the thesis and proposes some future research topics.

Chapter 2

Trie Structures

2.1 Trie Methods

The trie uses characters, or digital decomposition of a key, to direct the branching [Gon91]. The decision which way to follow during a search from an internal node at depth d is made according to the value of the dth position in the search key. For example, a trie for a key set of *table*, *space*, *speed*, *trie* and *text* is shown in Figure 2.1(a). The first letter splits the keys into two sets of subtries, s-keys and t-keys. The second letter splits the t-keys into three groups of ta-keys, te-keys and tr-keys and so on. When searching a word, say *trie*, the first letter t leads us to the right child of the root. The second letter r leads us to the rightmost descendant. Eventually, if a leaf node is reached, as in our case, a search returns successfully. Otherwise if a null link is reached, it means the word is not on the trie. Thus an unsuccessful search usually stops at an internal node.

The trie presented in Figure 2.1(a) is referred to as a *full trie* [CS77] or *pure trie* [Ore82b]. Note that there exist subtries leading to only one key (leaf node) in a full trie. Such subtries can be pruned and the resulting trie is called an *ordinary trie* (*radix search tree, pruned trie* [Knu73, CS77] or in most of the literature, simply *trie* [Gon91]). Figure 2.1(b) is an example of an ordinary trie. The truncated characters, *ce, ed, ble, xt* and *ie* can either be stored on leaf nodes or in a separate file

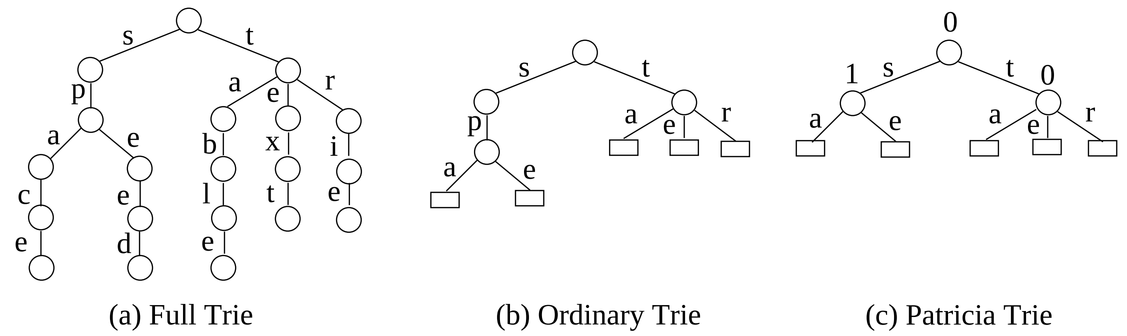


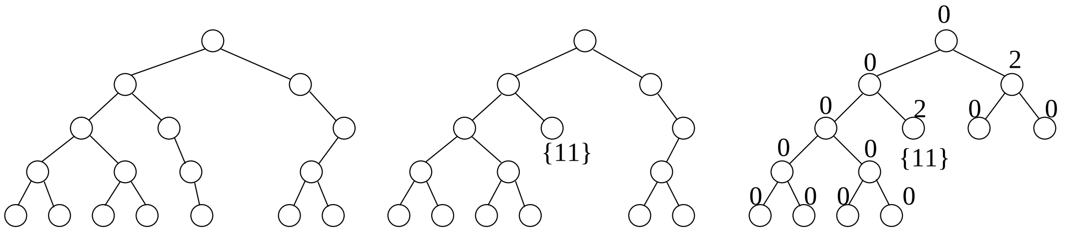
Figure 2.1: Trie Structures

pointed to by these leaf nodes.

There still exists a single descendant node, between link *s* and *p* for words *space* and *speed* in Figure 2.1(b), which does not branch a search to any new subtrie. Such node chains can be further eliminated if on the first branching descendant node a number is used to indicate its corresponding level, or the number of skips from its ancestor node. Figure 2.1(c) shows the resulting trie. Numbers on internal nodes indicate the number of skips from their parent nodes. Such tries, without single descendant nodes are called *Patricia tries* [Mor68, Knu73, MF85b, Gon91] (*P*ractical *A*lgorithm *T*o *R*etrieve *I*nformation *C*oded *I*n *A*lphanumeric). Patricia tries are especially capable of indexing very long, variable length and even unbounded key strings. Thus they are very useful in text searching (cf. section 2.3).

The tries in Figure 2.1 are n-ary tries. Tries can also be binary. A binary trie is a binary tree in which the branching decision on each node depends on the current bit of a binary search key: branching left if it is 0, else right. A binary trie can be formed on the binary string format of numerical or alphabetic keys. Figure 2.2 shows the binary full trie, ordinary trie and Patricia trie of the numerical key set {0, 1, 2, 3, 7, 12, 13}. The corresponding bitstring set is {0000, 0001, 0010, 0011, 0111, 1100, 1101}.

Tries were first developed by de la Briandais [Bri59] and E. H. Fredkin [Fre60]. The name trie comes from the word re*trie*val [Fre60]. They were used for prefix searching by Morrison [Mor68]. Intensive discussions about the structure can be found in Knuth [Knu73] and many other data structure books. Tries were associated



(a) Binary full trie (b) Binary ordinary trie (c) Binary partricia trie

11
11 01
11 01 10
11 11 01 11
00 00 00 00 00 00 00

(d) FuTrie representation

11
11 01
11 00{11} 10
11 11 11
00 00 00 00 00 00

(e) OrTrie representation

1{0}{}
1{0}{} 1{2}{10}
1{0}{} 0{2}{11} 0{0}{} 0{0}{}
1{0}{} 1{0}{}
0{0}{} 0{0}{} 0{0}{} 0{0}{}

(f) PaTrie representation

Figure 2.2: Binary Tries

with digital searches, and thus are also called digital trees [Knu73]. Since then, tries have been applied extensively to various text indexing and searching.

Several trie parameters are of great interest: trie *depth*, *height* and *size*. Trie depth is defined as the average path length from the trie root to its leaves. It represents the average cost of a successful search. Trie height is the longest path from the trie root to the leaves. It is indicative of the worst case search time. Ideally, it is advantageous to know depth distributions in order to understand the behavior of tries, such as how balanced/skewed the trie is. Trie depth/height has a rich research history since 1970s [Knu73, Dev82, Dev84, Pit85, Szp88, Szp90, Szp91, Jac91, Dev87, Szp92, Szp93, RJS93, CFV98, CFV99, KS00b, KS00a]. Trie size in storage consumption is as important a parameter as trie depth and height in measuring access time. It is the number of nodes in a trie. Trie size has been analyzed and explored in the literature [Knu73, Jac91, Szp90, Szp91, CFV98, CFV99]. A trie is called a *symmetric* trie if data stored on tries are uniformly distributed. Otherwise, it is an *asymmetric* trie. For asymmetric tries, the entropy determines depth distribution. The more asymmetric the symbol alphabet is, the more skewed a trie is. Some discussions of asymptotic behavior of asymmetric tries can be found in the literature [RJS93, FL94, KS00a, KS00b]. Since the late 1970s, prefix tries have also been applied to spatial data index schemes ranging from *quadtrees* * [Hun78, Dye82, Sam90], *octries* [Mea82], *k-d-ties* [Ore82b], *pr-ties* [Sam90], and *FuTries* [MS94, Sha94]. The 1980s saw various trie structures proposed for dynamic trie hashing. Other development and applications of tries include lexical analyzers and compilers, natural language analysis, data compression, pattern recognition, parallel searching, and even Internet IP routing. A brief trie history can be found in Appendix I.

The next two sections will focus more specifically on trie properties and applications.

2.2 Trie Properties

The simple but elegant trie structure has many attractive properties, and thus has been applied to various database and non-database applications.

- The compression of data by the overlap of paths near the root reduces space cost of the trie, and provides faster transfer of data from the secondary storage to the main memory.
- It stores the most significant bits first near the root. It allows queries to start at some approximation and perform refinements by reading lower levels of tries only if there is uncertainty.
- Tries are order preserving data structures which is essential to high selectivity queries such as range queries.
- *Prefix searching* looks for any word that matches a given prefix. Trie searching is, in fact, prefix searching, unlike hashing or any other normal tree search. Patricia tries are capable of indexing extremely long and *unbounded keys* and thus are extremely suitable for prefix searching.

*Unfortunately, the term quadtree is confusing as it has different meanings. In most cases, it refers to a trie structure and thus should be called *quadtrie*. In some other cases, it may also refer to a tree structure [FB74, FGPM93, FL94].

- The shape of the trie is uniquely determined by the data sets, and thus is not affected by poor data distributions.
- Tries can be interpreted as multiple key structures and hence are amendable to multidimensional space. A key with multiple attributes can first be interleaved bit by bit to an interleaved key. Then these interleaved keys can be used to construct a trie, as if they were in one dimension.
- Tries have short search time. Successful search cost is bounded by the length of the search key, regardless of the file size. Unsuccessful searches may cost less as they are likely to stop at internal nodes.
- Trie structures are flexible and can be combined with many other structures simply by applying the trie structure near the root and switching to these structures near the leaves. Further explanations are discussed in the next section.

2.3 Trie Applications

2.3.1 Prefix searching

Many applications require recognition of keywords from dictionaries and thus require efficient prefix searching. Traditional dictionary lookups, such as hashing and tree searching, do not support search keys to be prefixed or abbreviated and thus are inadequate. Trie structures, on the other hand, are ideal for indexing prefixes. Trie searching has been applied for data compression [BWC89, BK93], lexical analyzers and compilers [ASU86], pattern recognition [BS89, DTK91, ABV95], spelling checkers [LEMR89], natural language analysis [TITK88, Jon89], parallel searching [HCE91] and Internet routing [NK98].

2.3.2 Text Searching

A major problem for text indexing which is capable of accessing every substring of a large text is its size. Clearly, at least one pointer is needed for every character in the text and each pointer is at least $logN$ bits, where N is the number of substrings. The total index must be of size $N \log N$ bits or $\frac{1}{8} N \log N$ bytes. For a text of size 2^{27} characters or substrings such as the *New Oxford English Dictionary (OED)*, pointers to the text already require $3.4N$ bytes. Trie compression makes text indexes possible. Along with its capability to index very long and even unbounded strings and its fast access property, the trie is a suitable data structure for text indexing and searching.

- prefix search:

  Tries have been used for prefix searches by Morrison [Mor68] and exploited by Gonnet *et al.* [Gon88, GBY91, Tom92] as the basis for the retrieval methods used in the electronic version of the New OED, using PAT tries.

  Gonnet [Gon88] treats a text as a long single character string. A *sistring* is a semi-infinite suffix of a text. A trie of string suffixes is a *suffix trie* [Apo85, Szp92]. Every subtrie of a suffix trie has all the sistrings of the given prefix. Prefix searching in suffix tries consists of searching tries up to the point the prefix is exhausted, or when there are no more subtries. In either case the search cost is only bounded by the length of the prefix, independent of trie size. Suffix tries are efficient for prefix searching and longest repetition searching.

- longest repetition search:

  Longest repetition of a text is a match between two sistrings which has the most number of characters in the entire text. For PAT tries, it is the sistring pair with the highest depth. For a given text, it can be found during the construction of the trie. It can be applied to manipulations of general sequences of symbols [SK83], such as string editing, comparison, correction and collation of different versions of the same file. It is also applicable to genetic/biomedical sequences.

- range search:

  Suffix tries or PAT tries can do range search efficiently, searching for all the strings that are lexically between two given values, in the order of trie height [Gon91] and the size of the answer set.

- most frequent search:

  This search is to find the most frequently used string in the text. With suffix tries, it is equivalent to finding the largest subtrie whose search path begins with a space and ends with a second space [Gon91]. This type of search has great practical interest such as finding the most frequently used word in a text.

- regular expression search [BYG89]:

  Regular expression searching also has great practical interest. Tries have been also applied to regular expression searches of texts [BYG89, Gon91, Sha94].

- proximity search:

  This search gives strings that are at a fixed *distance* away from a given string. The distance of two strings can be defined as the number of differences (insertion, deletion, substitution and/or transportation) between the two given strings. Various data structures, algorithms and techniques have been developed and applied to solve this problem [Knu73, HD80, SK83, Kuk92, BYP92]. Again, tries are one of the structures that can be readily applied to it. In Gonnet's paper [GBY91], *PAT array*, a compact representation of the *PAT tree*, was used for proximity searching. Shang and Merrett [Sha94, SM96] apply *FuTrie*, a binary full trie structure, to proximity searching.

2.3.3 Spatial Data Representation

Spatial data are points, lines, etc. in multidimensional space. Prefix tries for spatial data are tries on interleaved numerical data, with the most significant bits stored close to the trie root. This variable-resolution structure allows some queries to look only part way down the trie to retrieve and search on approximations. The search choice on a node can be "accept" (all records in the subtrie are in the answer set), or "reject" (no record in the subtrie is in the answer set), or "explore", when there is uncertainty at this early stage. The search only goes down to subtries when there is uncertainty.

Tries have been applied to spatial data indexing schemes, ranging from *k-d-tries* [Ore82b], *octrees* [Mea82], *pr-tries* [Sam90], *zoom tries* [MS94] to various *quadtrees (quadtries)* [Hun78, Sam90]. Some of the other multi-dimensional structures such as *k-d-trees*, *K-D-B-trees*, *grid files*, *multipaging*, *R-trees* etc. will be discussed later in chapter 3.

K-d-tries are a generalization of 1D binary tries. They are named after k-d-trees [Ben75], because they use the same principle of interleaving coordinates. The advantage of k-d-tries over k-d-trees is that tries store data under variable resolutions, i.e., the most significant bits are stored near the root. This property, *zooming*, was exploited to display spatial data at any resolution using only one copy of the data and transferring from secondary storage only the amount of data needed for that display [MS94, Sha94].

2.3.4 Other Applications

Signal Processing and Telecommunications

One of the most important issues in signal processing is to estimate the output for a known input, i.e., a query from the input/output data seen to this point. A nonlinear adaptive estimation method that uses a k-d-trie was presented in [Iig95]. N records of k-dimensional input vectors and their corresponding scalar outputs are stored in the k-d-trie. These latest N input/output records are used to estimate the output of a given input (query point). A trie range search, with maximum distance from the query point in each dimension less than L, is performed. Then, a non-linear local model is applied to those records retrieved from the range query in order to obtain the estimate of the output. The method requires updating the trie as each new data point is available such that only the latest N data are maintained on the trie. The k-d-trie is chosen instead of a k-d-tree since it has superior performance to the latter in terms of the average time requirement for updating; it requires no rebalancing operations for insertions and deletions.

Data compression, message encoding/decoding techniques are widely used in telecommunications. Ziv-Lempel [ZL77, ZL78] coding is currently one of the most practical data compression schemes. It operates by replacing a substring of a text with a pointer to its previous occurrence in the input. Tries are one of the structures capable of longest string searching as mentioned above [BK93].

Message decoding and conflict resolution algorithms for broadcast communications can also be equivalent to trie search processes [Cap79, Ber84, MF85a].

Image processing

Trie hashing has been used as a dynamic method for similarity retrieval in pictorial database systems. It is claimed to have good performance in pictorial database management systems [CL93].

2.4 Trie Representations and Algorithms

2.4.1 Tabular Forms

Tries have been represented variously. A straightforward implementation is the table or matrix form [Fre60, Mor68, Knu73, RBK89]. A k-ary trie with S nodes and N keys is represented by a table of $k \times (S - N)$ entries, where k is the number of rows and $S - N$ the number of columns. In the table, each column represents an internal node of the trie, where each table entry contains a column number, a null pointer, or a pointer to a key (leaf node). The first column is the root. Table 2.1 gives the tabular format of the binary trie given in Figure 2.2(a). There are 20 table entries for $N = 7, k = 2$, and $S = 17$.

	0	1	2	3	4	5	6	7	8	9
0	1	2	9	0010		6	1100			0000
1	4	7	3	0011	5		1101	8	0111	0001

Table 2.1: Tabular Format of a Binary Trie

Given a key, the search is initiated by looking up the table starting at the first column, the root node. If following the column number of a link, an empty entry is found, the search returns unsuccessfully. A search is successful if a termination condition (a leaf node) is reached and matches with the search key. Dynamic insertion of a node simply amounts to adding a new column and putting a column number at the entry of its parent column. In fact, Table 2.1 is the result of dynamic insertions of the keys in the following order: {0010, 0011, 1100, 1101, 0111, 0000, 0001}. Deletions may leave blank columns in the table which may be filled by the last column. However, the parent node of the last column must be updated too. Merrett and Fayerman [MF85b] suggest locating the parent column by adding reverse pointers to each node. Shang [Sha94] suggests that it can be solved without adding extra storage overhead. Instead, simply search for a keyword down the subtrie rooted at the last column followed by another search of the keyword from the root.

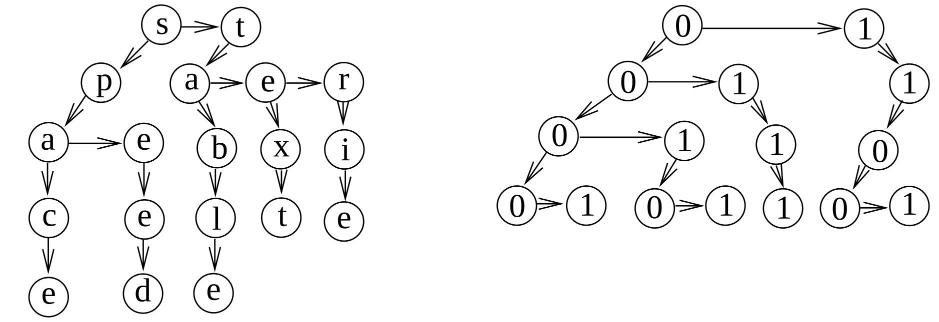
There are more subtle implementations of the tabular forms by either using three arrays or double arrays to minimize the storage overhead [TY79, Aoe89]. The idea is to compress the table into a 1D array with fewer entries by mapping from positions in the table to the array such that no two non-empty entries in the table are mapped to the same position in the array.

2.4.2 Linked Lists

Tabular representations are prohibitive when k is large for a k-ary trie when many of the entries in the table are empty. Dynamic structures, such as linked lists, are an alternative way to overcome the problem [ES63, Knu73, AHU83, Jon89, Dun91]. Figure 2.3 shows the corresponding linked list representations of the alphabetical and the numerical trie examples.

In the linked list representation, each node is a linked list of outgoing (right) links. The link contains a character and a pointer to the left-most child in the siblings of the child node (left link). This is in fact a *double chained tree* [ES63], or a binary tree [Knu73].

Searching on a node is done by comparison of a character (bit) of the key and the character (bit) on the node, following the outgoing (right) links until a match is found; then the pointer to the left-most child (left link) is taken to match the next



(a) Linkedlist for a n-ary trie (b) Linked list for a binary trie

Figure 2.3: Linked List Representations of Tries

character (bit). In the worst case, all outgoing links of a node have to be followed before a matched one is found.

Linked lists are general and highly flexible structures. Insertions and deletions are trivial with dynamic memory allocation techniques. Furthermore, it does not store null outgoing links and therefore, unlike the tabular representation, is indifferent to k, the degree of a node, in terms of storage space.

2.4.3 Other Representations

Compressed Tries

Compressed Tries or *C-tries* [Mal76] are a tree representation of tries for static data. Instead of using explicit pointers, a node of the compressed trie consists a bit array indicating the outgoing links and a counter of $log_k N$ bits indicating the number of links before the current node in the node level. Figure 2.4 shows the C-trie for the binary trie example. A node contains a bit array of size 2, indicating the set bits, and a counter of such bits in the level before the current node. Note that dotted lines indicating links on the figure do not exist explicitly. During the search, the address of a child node linked by the i^{th} set bit in the next level is the $i + counter^{th}$ node on the next level.

With a base address for each node level, the C-trie can be stored continuously

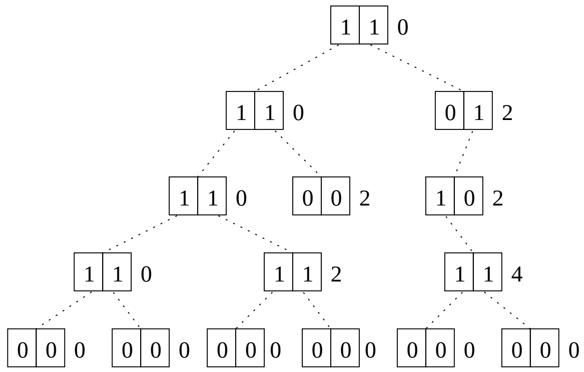


Figure 2.4: A Compressed Trie

on secondary storage level by level, and node by node within each level. It is a very compact representation for static data.

Bitstring Representations

Bitstring [Ore82a, Ore82b] extends the C-trie even further by only storing bit arrays. Finding the child link on the next level involves linear scanning on the next level. A bitstring representation of the binary trie example is given in Figure 2.5. The bitstring in curly brackets indicates the remaining bits of the leaf. But Orenstein argues that the number of bits scanned in each level can be reduced to an arbitrary constant by organizing bits into *blocks* depending on the block size.

```
11
11 01
11 00{11} 10
11 11 11
00 00 00 00 00 00 00
```

Figure 2.5: A Bitstring Representation of Trie

Both C-tries and bitstring representations are pointerless trie structures.

2.5 Trie Refinements

2.5.1 LC-tries

LC-trie [AN93] applies *level compression* to reduce further the height of the patricia trie. The basic idea is that if the i^{th} highest level of a trie is complete, but level $(i+1)$ is not, then the i^{th} highest levels are replaced by a single node of degree k^i (2^i for a binary trie). The replacement is applied top-down starting from the root. Figure 2.6 shows the LC-trie transfered from the patricia trie in Figure 2.2(c).

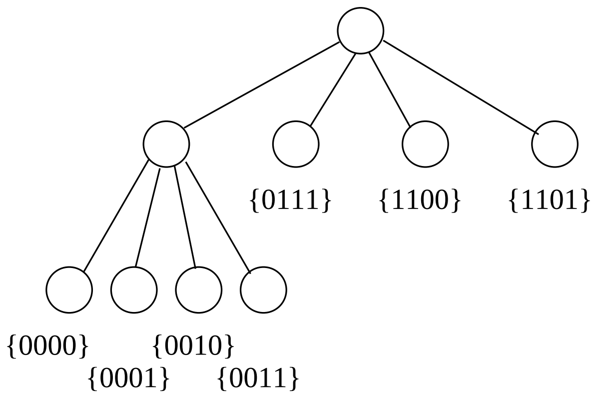


Figure 2.6: LC-trie

It is claimed by Andersson [AN93] that for random, independent data, the average depth of a LC-trie is reduced to $\Theta(\log^* N)$ from $O(\log N)$, where N is the number of keys.

2.5.2 Hybrid Tries and Trie Hashing

Because of the flexibility of trie structures, they are often combined with some other structures to obtain efficient behavior, i.e., applying trie structures near the root and switching to other data structures near the leaves. They are referred to as *hybrid tries*. One common combination is with external buckets, called *bucket tries* [Knu73]. Bucket tries are widely used as collision resolution strategy for *dynamic trie hashing* [ED80, Lit81, Lit85, LZL88, LRLH91].

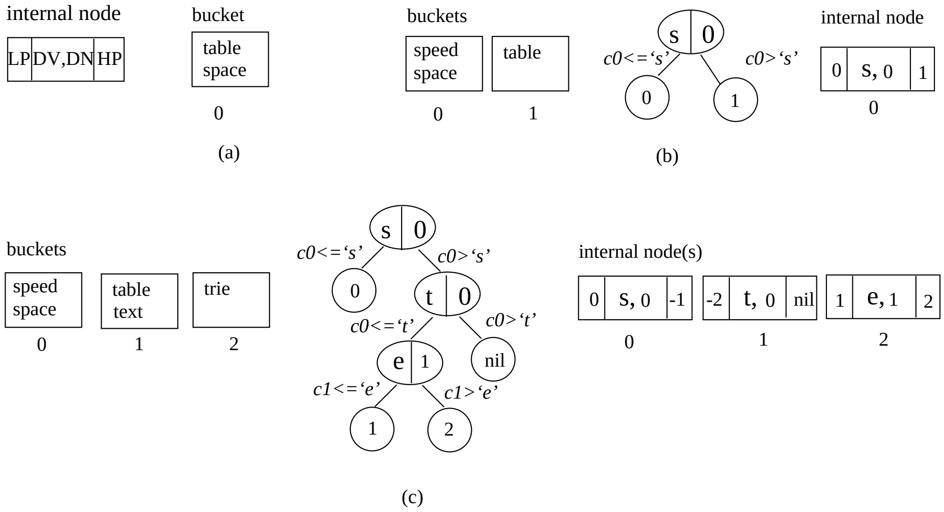


Figure 2.7: Construction of Bucket Tries for Trie Hashing

Suppose keys in Figure 2.1 are inserted in the following order: {*table, space, speed, trie, text*} and the capacity of a bucket is two items. Figure 2.7 shows the dynamic construction of the bucket trie for hashing. A trie node contains four fields: DV, DN, LP and HP, where DV is the value of the digit, DN the digit number of the key, and LP and UP are two lower and upper pointers, either to internal nodes or external buckets. The value of an internal pointer is a node address. The value of an external pointer is either the address of a bucket or null. In order to distinguish internal and external pointers, the value of an internal node is in fact the negative of the node address. In Figure 2.7(a), when the first two keys are inserted, they are stored in bucket 0. No internal nodes are needed. Figure 2.7(b) shows that when *speed* is inserted, bucket 0 contains three keys, and thus has to be split into two at the first digit *s*. Any key with the first digit greater than *s* is stored in bucket 1 and pointed to by the HP pointer; otherwise it is in bucket 0 and pointed to by the LP pointer. The branching information is stored in an internal node 0. Figure 2.7(c) shows the insertion of the last key. Bucket 1 containing *table, trie, text* is split into bucket 1 and 2 at the first two digits *te*. Correspondingly, two internal nodes are generated, with node 1 for the split at the first digit *t* and node 2 for the second digit *e*.

Trie hashing has been claimed to require one disk access when internal nodes of the trie can be held in RAM, and two accesses for very large files when the trie has to be on disk [Lit85, LZL88, LRLH91]. Furthermore, the file can be highly dynamic.

2.5.3 FuTrie, OrTrie and PaTrie on Secondary Storage

FuTrie, *OrTrie* and *PaTrie* are three pointerless trie structures. FuTries, OrTries and PaTries denote the binary full tries, binary ordinary tries and binary patricia tries [Sha94] respectively. The three organizations are extensions of Orenstein's pointerless bitstring representations on secondary storage. They use two bits for each node and tries are partitioned into pages, and thus are suitable for secondary storage.

FuTrie

FuTrie is a binary tree whose nodes do not store information and whose left links are labelled with '0's and right links with '1's. The i^{th} bit of a search key determines the link to be followed at level i of the trie; if it is '0', go left and otherwise right. Thus, each root-to-leaf path has a one-to-one correspondence to a key. The height of the trie is bounded by the length of the keys.

Now, we move on to FuTrie representations. Two bits are sufficient to represent a FuTrie node; 11 if the node has two descendants, 10 if it has only a left descendant, 01 if only a right, and 00 for a leaf. The FuTrie for the binary trie example is shown in Figure 2.2(d). This is exactly Orenstein's bitstring representation. Table 2.2 shows the definition of a FuTrie structure.

typedef enum	$\left\{\begin{array}{ll} 11 & \swarrow\searrow \\ 10 & \swarrow \\ 01 & \searrow \\ 00 & \bullet \end{array}\right\}$	*TrieNode*;
typedef struct	{ TrieNode trie_node[]; }	*TrieLevel*;
typedef struct	{ TrieLevel trie_level[];}	*FuTrie*;

Table 2.2: Data Structure for FuTrie

OrTrie

An OrTrie is a pruned FuTrie in which subtries containing only one leaf are pruned. Figure 2.2(e) shows the OrTrie transformed from Figure 2.2(d). Bits in curly brackets are the path-to-leaf suffices that have been truncated. It can also be a pointer to the corresponding key or record stored in an external file. The OrTrie structure is defined in table 2.3.

typedef enum	$\left\{\begin{array}{ll} 11 & \swarrow\searrow \\ 10 & \swarrow \\ 01 & \searrow \\ 00\left\{\begin{array}{l}\{suffix\}\{other\_attributes\} \\ or\{pointer\}\end{array}\right\} & \bullet \end{array}\right\}$	*TrieNode*;
typedef struct	{ TrieNode trie_node[]; }	*TrieLevel*;
typedef struct	{ TrieLevel trie_level[];}	*OrTrie*;

Table 2.3: Data Structure for OrTrie

PaTrie

PaTries are used to represent binary patricia tries. Figure 2.2(f) shows the PaTrie representing the patricia trie in Figure 2.2(c). As there are no single descendant nodes in a binary patricia trie, a node on a PaTrie can be represented by one bit, 1 for an internal node and 0 for a leaf node. For an internal node, the number of skips and the corresponding substring that has been skipped need only be attached. For a leaf node, either a pointer to the record or suffix of the key and other attributes of the record have to be stored. Table 2.4 shows such a PaTrie structure.

typedef enum	$\left\{\begin{array}{ll} 1\{\#skips\}\{substring\} & \swarrow\searrow \\ 0\left\{\begin{array}{l}\{length\}\{suffix\} \\ or\{pointer\}\end{array}\right\} & \bullet \end{array}\right\}$	*TrieNode*;
typedef struct	{ TrieNode trie_node[]; }	*TrieLevel*;
typedef struct	{ TrieLevel trie_level[];}	*PaTrie*;

Table 2.4: Data Structure for PaTrie

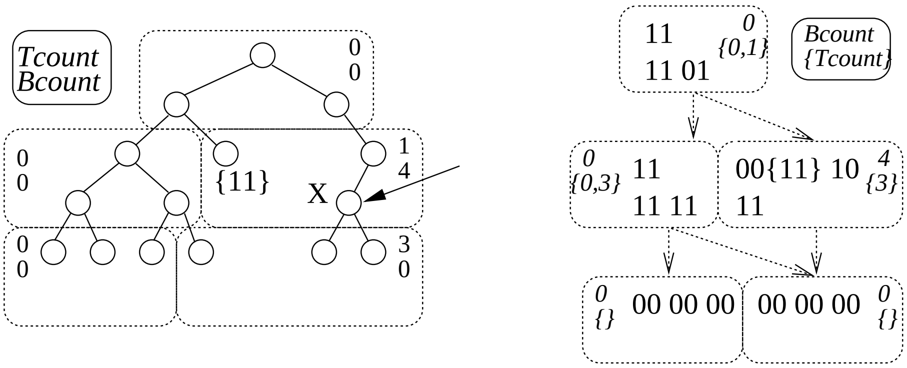
Paged Tries

As it stands, like the C-trie and the bitstring representation, FuTrie, OrTrie and PaTrie structures require a sequential search on each trie level, destroying the logarithmic search cost and the variable resolution advantage tries provide. However, in a paged structure for secondary storage, this can be fixed.

The paged structure partitions a trie into layers (page levels) of l node levels each, and then cuts each layer vertically into pages of subtries. Within a page, descendant nodes of each layer are either entirely on or entirely off the page, i.e., links can only cross the horizontal boundaries of layers, not the vertical boundaries of pages. The resulting *paged trie* reads one page per layer from secondary storage during the search, and restricts sequential search within pages only.

Figure 2.8(a) shows the paged OrTrie with $l = 3$ and a page capacity of three nodes. A page contains two counters to avoid the sequential search and redeem the trie search. *Tcount* enumerates the number of links entering the page level from the above, up to but not including the current page. *Bcount* does the same for links leaving the bottom of the page level. The two counters can be used to find the page where the left descendant of the node “X” locates without a sequential scan of pages in the next page level. The Bcount of the page with node “X” is 4, which means 4 links have already descended from earlier left pages in the page level. As the left descendant of “X” is the first link in the page, so it is the 5^{th} leaving the current page level. Thus in the next page level, we must look for a page with Tcount the greatest integer less than (or equal to) 5. The candidate pages on the page level below are Tcount= 0 and 3, and thus we choose 3. Thus the left descendant of “X” is located on the second page in the next page level.

Thus far, when checking Tcounts at the next page level, we still do a sequential scanning. However, it can be avoided simply by moving Tcounts of each page level up into lists of Tcounts in the parent pages above. Then we can calculate directly from the current page which page to follow on the next level. As shown in Figure 2.8(b), each page contains a Bcount and a list of Tcounts of child pages. Dashed lines pointing to pages are implicit in the paged trie structure.



(a) Paged OrTrie (b) Paged OrTrie Structure

Figure 2.8: Paged Tries

The paged OrTrie structure is given in Table 2.5.

typedef struct	{	int	Tcount; }	
		OrTriePage	*page; }	*LinkTo*;
typedef struct	{	int	Bcount;	
		LinkTo	linkto[];	
		OrTrie	bitstrings; }	*OrTriePage*;
typedef struct	{	*OrTriePage*	trie_page[];}	*PagedOrTrie*;

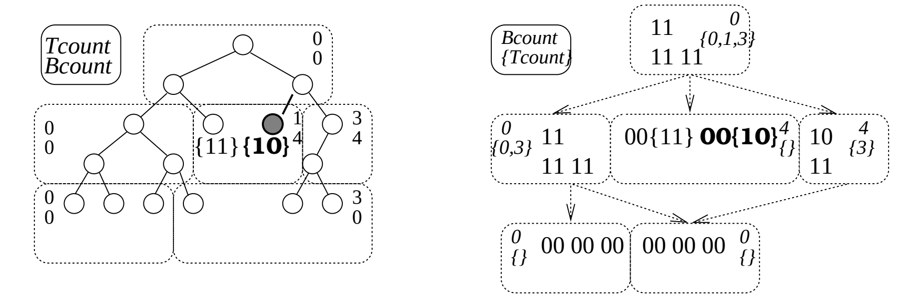
Table 2.5: Data Structure for Paged OrTrie

2.6 DyOrTrie, a Refinement of the OrTrie for Dynamic Data

This section describes new work, although it appears in a review of trie structures. It is an improvement over the existing paged trie structures, OrTries, for dynamic data insertions and deletions.

Insertion is straightforward for the paged OrTrie. An insertion of key *1010* to the trie presented in Figure 2.8 is given in Figure 2.9.

The insertion follows a search of the key. As the first bit is 1, the search goes



(a) Paged OrTrie: after inserting 1010 (b) Paged OrTrie Structure after inserting 1010

Figure 2.9: Insertion Key *1010* to Paged Trie

to the right descendant of the root. The second bit is 0 but the current node does not have a left descendant, and thus a new node (highlighted), the third node in the third node level, is inserted into the second page of the second page level. The corresponding parent node at the root page is thus updated from *01* to *11*. Following this, the page holding the new node is split into two as it exceeds the page capacity. The *Bcount* and *Tcount* lists of pages to the right of the split page in the same page level need to be updated accordingly. Similarly for the pages in parent page levels if the split propagates to the parent page levels.

From the example, we notice that the change caused by the insertion/deletion of a key is usually localized, i.e., updating a node at the root page from 01 to 11 and inserting a new node 00{10} into the second page on the second page level. But as pages in a page level are organized sequentially, top and bottom counters stored at the current page and pages following need to be updated, although usually only by a shift of a constant. But it means a sequential scanning and updating of those pages, which is prohibitively expensive on secondary storage. Thus, the paged structures are good at batched insertion of ordered keys [Sha94] due to the sequential page organization within each page level. To avoid this, we modify the paged trie structure so that all counters are stored separately with pointers pointing to their corresponding bitstring pages. With the assumption that these counters can be stored in RAM, the new paged trie structure *DyOrTrie*, given in table 2.6, is capable of efficient dynamic insertions and deletions.

typedef struct	{	int	Tcount; }	
		OrTrie	*page; }	*LinkTo*;
typedef struct	{	int	Bcount;	
		OrTrie	*page;	
		LinkTo	linkto[];}	*OrTrieLink*;
typedef struct	{	*OrTrie*	page[];	
		OrTrieLink	link[]; }	*DyPagedOrTrie*;

Table 2.6: Data Structure for Dynamic Paged OrTrie

Insertion of a record starts with a search of the record. The search may result in one of the following situations:

- A leaf node is found, and the remaining bits of the node match with that of the search key. This is a duplicate key and the insertion algorithm returns.
- The search stops at an internal node, because it can not find the branch according to the search key. In this situation, update the current node from 01 or 10 to 11 and add a corresponding descendant leaf node at the child level.
- The search stops at a leaf node, but the remaining bits of the node do not match with that of the key. In this case, the original leaf node that was truncated has to be extended to the level at which the bit of the original key differs from that of the new key.

The update in the second scenario is rather local. Updating the node from 01 or 10 to 11 costs no extra storage space. But adding a corresponding descendant leaf node to the child level may occasionally cause the page where the child level is located to exceed page limits, and thus a page split is required. But the split does not in any situation propagate to upper levels. On the other hand, the extension of links in the third scenario may cause the pages on the path from the node to the leaves exceed page limits and require splitting. Furthermore, the split may propagate from the page holding the two leaves to the page where the node extension happens.

Figure 2.10 shows the pseudo-code for key insertions. Like B-trees, the cost is bounded by the trie height due to the occasional splitting of trie pages which may

```
Boolean TrieInsertion(key)
{
  //search the key and find the node that is either a leaf node or
  //a node that the search has to stop because of a mismatch:
  node= search_and_find_node(key);
  if (node -> is_leaf() && node-> rest_bits_match_with( key)) {
     //the key is already on the trie:
     return (false);
  }
  if (node-> is_leaf() && !node-> rest_bits_match_with( key)) {
     //expend the node until the two keys differ (return the new leaf):
     node= node-> expend(key);
  }
  else {  //node is not a leaf, at which the node is updated to 11 and
          //a new leaf node is created at the next level:
     new_leaf_node= node-> update_node_value();
     node= new_leaf_node;
  }

  //split the page if necessary:
  page= node-> current_page();
  while ( page != NULL && page-> size() > page_capacity) {
    if (page -> parent_page != NULL) { // not root page
      split_page ( page);
      //update page links and counters in the parent page level:
      page= page-> parent_page();
      page -> update_page_links_and_counters();
    }
    else   //this is root page, cannot be split:
      return (false);
  }
}

void split_page( OrTrie *page)
{
  //linear scan of the page and find i'th subtrie at which
  //subtries 1,...,i consumes <= page_capacity while
  //subtries 1,...,i,i+1 consumes > page_capacity:
  i= find_subtrie_in_page_by_linear_scan ( page);
  //move subtries i+1, ... from page to a newly constructed page, newpage:
  OrTrie *newpage = new OrTrie(page, i+1);
  //update page links and counters of page and newpage:
  page->update_page_links_and_counters();
  newpage-> update_page_links_and_counters();
}
```

Figure 2.10: Paged Trie Insertion Algorithm

propagate up to the root page.

Now we discuss the splitting algorithm and the page utilization after splitting. When a trie page exceeds the page capacity during the insertion, the page is forced to be broken into two pages. Ideally, we would like the two pages to be equally full. Due to the fact that trie pages do not have branches to neighboring pages in the same page level, the issue becomes dividing the page containing a forest of subtries into two groups of subtries of approximately equal size. As these subtries are ordered, the splitting is to find the i^{th} subtrie in a total of m subtries such that subtries 1, 2, ..., and i consume no more than half of the page capacity, but if i+1 is included, they exceed half of the page capacity. Thus, subtries 1,2,...,i remain in the original page and subtries i+1,...,m move to the newly generated page. This can be done simply by a linear scan of the page in RAM.

The least page occupation after splitting occurs when the splitting boundary is set at subtrie $i+1$ which is a complete trie. More over, the least page utilization value is a function of the page capacity P and the number of node levels in a page, l. A complete subtrie of l levels has $2^l - 1$ nodes and each node takes 2 bits, i.e., approximately 2^{l+1} bits or 2^{l-2} bytes for the subtrie. So the least page utilization rate is correspondingly $1/2 - 2^{l-2}/P$. Thus, the larger the page capacity P is, the higher the least page occupation is. On the other hand, the more node levels there are, the lower the least page occupation can be. For instance, if the page capacity is 4096 bytes and there are 10 node levels in a page level, then the least page occupation rate is $1/2 - 2^{10-2}/4096 = 0.4375$.

A deletion also starts with a search of the key. There are three different situations as follows.

- The search stops at an internal node, the key is not found and the deletion stops there.

- The search stops at a leaf node, but the remaining bits of the node do not match with the that of the key; and thus the search fails as well as the deletion operation.

- The search stops at a leaf node, and the remaining bits of the node match with that of the key. The key is found and the leaf node is removed. Then, the parent node is updated. If the sibling node of the deleted leaf node is also a leaf node, the branch to the sibling node can be truncated and the parent node becomes a new leaf node. The truncation operation may propagate up to the root. Due to the truncation of the path, the page occupation decreases. If both the page and one of its neighboring page consume space less than a threshold, a merging of the two pages can be performed in order to improve the storage utilization. Like the page splitting in the insertion operation, this can also propagate to the root page level.

Figure 2.11 shows the pseudo-code for a key deletion. The cost is bounded by the trie height.

We claim that with the current RAM capacity, the dynamic paged trie structure is suitable for practical database sizes of the order of gigabytes (billions of records). For example, with a page capacity of 4096 bytes, it only requires a RAM size of 4 megabytes to build a dynamic trie holding up to 2^{32} (4-billion) records of data. If this calculation is altered for 2^{36} records of data, roughly 64 megabytes are required. If a record consumes 4 bytes, 2^{32} and 2^{36} records are 16 and 256 gigabytes of data respectively. The calculation is as follows.

Bottom/top counters and page pointers are stored in RAM. For simplicity, we choose to give examples with complete tries. A complete trie with 33 node levels, assuming page capacity is 4096 bytes and node levels in a page is 10, can hold 2^{32} (4-billion) records. A page of capacity 4096 (2^{12}) bytes can hold 2^{14} trie nodes, as each node consumes 2 bits. A complete subtrie of 10 levels has $2^{10} - 1 \approx 2^{10}$ trie nodes. Thus a page can hold up to $2^{14}/2^{10} = 16$ complete tries of 10 node levels.

We now calculate RAM space required by page levels.

The root page level only has one root page. On the bottom of the page, there are 2^{10} outgoing links to the next level and each page on the next level can hold at most 16 incoming links, as a page can hold only 16 complete tries of 10 node levels. The 2^{10} outgoing links has to go to $2^{10}/16 = 2^6$ different pages on the second level. So the

```
Boolean TrieDeletion(key)
{
    node = search_and_find_node( key);
    if ( !node-> is_leaf() || !node-> rest_bits_match_with(key)) {
       //the key is not found, cannot delete!
       return (false);
    }

    //The key is found:
    parent_node = node-> parent;
    parent_node -> remove_child (node);
    node= parent_node;
    while ( node-> has_only_one_child_node_which_is_a_leaf() ) {
      // update node and truncate the branch if possible:
      node= node-> parent;
      node-> value = 00; // remove child and set node to be a leaf
      if (node -> page() != node-> parent-> page()) {
        //the node is in the first node level of a page:
        page= node-> page();
        //merge_candidate() returns the neighbor page less full:
        neighbor_page= merge_candidate( page);

        if ( page-> size() + neighbor_page-> size() < Threshold)
          // do a merge with the neighbor page having less page utilization:
          merge( page, neighbor_page);

        //update page links/counters in the page level and the level above:
        page-> update_page_links_and_counters();
        node-> parent ->page()->  update_page_links_and_counters();
      }
    }
    return (true);
}

OrTrie *merge_candidate( OrTrie *page)
{
   if (size(page-> left_neighbor()) <= size(page-> right_neighbor()))
     return page-> left_neighbor;
   else
     return page-> left_neighbor;
}
```

Figure 2.11: Paged Trie Deletion Algorithm

fanout of the root page is 2^6 pages. It is the number of top counters of the root page.

The RAM space used by the root level is that of the top counters plus the page pointers to the next level of pages. A counter takes 4 bytes. So does a page pointer. So the root page level consumes $2^6(4+4) = 2^9$ bytes of RAM space.

On the second page level, there are 2^6 pages. Now we calculate the fanout of each page. The outgoing links at the bottom of this page level is $16 \times 2^{10} = 2^{14}$ per page since each page holds 16 complete subtries of 10 levels. A page in the next page level can hold as much as 16 complete tries. Thus the fanout of the second page level is $2^{14}/16 = 2^{10}$. Thus there are 2^{10} page pointers to the next page level per page. For each page, there is a bottom counter, which takes 4 bytes, and 2^{10} top counters and page pointers to the third page level. All together, a page uses $4 + 2^{10}(4+4)$ bytes of RAM space. In total, the second page level takes $2^6 \times (4 + 2^{10} \times (4+4)) \approx 2^{19}$ bytes of RAM space.

The third level is the second last level. There are 2^{20} incoming links from the previous level and each page in the level can hold 16 complete subtries of 10 levels, i.e., 16 nodes per page. Thus the number of pages in this level is $2^{20}/16 = 2^{16}$. The outgoing links at the bottom of this page level is $16 \times 2^{10} = 2^{14}$ per page, since each page holds 16 complete subtries of 10 levels. The page fanout in this page level is different from that of the previous one due to the fact that there are only three node levels in the next page level (the last page level). This is because the number of total node levels is 33. In the last page level, each subtrie only contains three levels of nodes and consumes only $(2^3 - 1) \times 2$ bits, i.e., approximately 2 bytes. A page of 4096 bytes can hold $2^{12}/2 = 2^{11}$ complete subtries of three levels. So the fanout of a page in the third level is $2^{14}/2^{11} = 4$. In total, the RAM space consumed by the third level is therefore $2^{16}(4 + 2^3 \times (4+4)) = 2^{22}$ bytes.

The last page level contains no outgoing links and the fanout of a page is zero. Thus no RAM space is required.

Summing up the RAM space used by all page levels, it is approximately $2^{22} = 4$ megabytes for a complete trie of 33 levels holding 4 billions of records. Note that it is principally the second last page level which consumes the most RAM space.

Now consider a complete trie of 37 levels containing 2^{64} records, only the second last level consumes more RAM space than in the last example of 2^{32} records. Pages in the last page level contain 7 node levels consuming $(2^7-1)\times 2$ bits, i.e., approximately 2^8 bits or 32 bytes per subtrie. Thus a page holds $2^{12}/32 = 2^7$ complete subtries. It makes the fanout of a page in the second last page level $16 \times 2^{10}/2^7 = 2^7$. So the RAM space consumed by the page level is $2^{16}(4 + 2^7(4 + 4)) \approx 2^{26}$ or 64 MB. This also represents the total approximate amount of memory consumed by all page levels.

The above are two examples of complete tries. For general tries, the RAM space is also mostly consumed by the second last page level. Due to "thinner" subtries, a page is able to hold more subtries than complete subtries. This allows the page level to have fewer pages than that of complete tries. On the other hand, a general trie holding the same number of records contains more node levels than a complete trie. This would make subtries in the last page level likely to have more node levels, which is a factor in reducing the number of subtries a page can hold in that level. As a consequence, it may increase the page fanout in the second last page level. We know that the RAM space consumed is roughly $\#pages \times (4 + fanout \times (4 + 4))$ in the second last page level. Since a general trie would have fewer pages but higher fanout, it depends on which one of the above two factors outweighs the other to determine whether it consumes more or less RAM space than a complete trie.

Chapter 3

Related Work

3.1 One-dimensional File Structures

3.1.1 Hash Functions

Hashing is a direct access method which locates records with given key by a *key-to-address transformation* function. The expected time to retrieve a key among N keys is effectively a constant, though the worst case can be proportional to N. Collision resolution strategies are needed to deal with imperfections in the key-to-address transformation.

Hashing works even better for files on secondary storage [Knu73] as many records are allowed to be stored at the same address. Collision handling is simpler than in RAM. On disks, file I/O are in units of pages (also called blocks) in order to take the advantage of high data transfer speed relative to block access time, with one page storing tens or hundreds of records. If more records are mapped to a page than its capacity, the extra records are overflow which must be stored somewhere else. This is the reason for extra accesses which may cause the worst case to be expensive, and increase the expected cost as well.

Perfect hashing [Spr77] is a hash function which yields no collision, thus the search cost is a constant even in the worst case. Perfect hashing may involve a certain amount of wasted space due to empty address space to which no keys are mapped. If a perfect hashing can reduce all the possible address space to the size of the presented key/record set, then it is called a *minimum perfect hashing*. Various algorithms with different time complexities have been presented for constructing (minimal) perfect hash functions. They fall into several general categories: number theoretical methods, segmentation techniques, algorithms based on search space reduction and algorithms based on sparse matrix packing [MWHC96]. They are claimed to be constructed in $O(N)$ expected time [CHK85, FHCD92, MWHC96], where N is the number of keys, though it usually requires many passes of the data set and would be prohibitively expensive for large amount of data on secondary storage. Another issue of (minimal) perfect hash functions is that they require auxiliary storage space [FHCD92, MWHC96] which is proportional to N [CHK85, FHCD92], or even more expensive ($N \log N$ [MWHC96]).

Hashing in general is only efficient for low *selectivity* queries such as exact match queries, where the selectivity is defined as the ratio of records retrieved by a query to the total number of records presented. It is not suitable for high selectivity queries such as sequential and range queries. This is mainly because hashing does not organize keys in order.

3.1.2 Tidy Functions

Ordered key-to-address transformation functions are called *tidy functions* [Mer83]*. Tidy functions support efficient high selectivity sequential accesses and range queries.

Tidy functions are order-preserving direct access functions [Mer83]. Like perfect hashing, the tidy function reduces the space of all possible values of search keys to a storage space just containing records actually present. But unlike hash functions, tidy functions preserve the order of keys. If $D(x)$ is a *cumulative distribution function* on the search key, and is defined as the following:

$$D(x) \equiv probability(key \leq x) \tag{3.1}$$

*In *Roger's Thesaurus of English Words and Phrases* (1936), “tidy” falls under the category “Reduction to Order”. Key-to-address transformations are intended to *reduce* the key space to a much smaller address space, and tidy functions intend to preserve order as well.

then the tidy function is

$$t \equiv \lceil n\ D(x) \rceil \tag{3.2}$$

where n is the number of pages on secondary storage. However, storing t to its full resolution means that we have to store index data for every block (data page). But when files are truly large, this cannot be assumed to be stored in RAM. We need to assume that the tidy function is stored on secondary storage too. Therefore, in order to get the page address of the keys, extra probes to the tidy function on secondary storage are required. This makes the tidy function less efficient both in access time and in storage.

Other approaches to tidy functions include the following.

Sorenson et al. [DST75, STD78] were interested in removing distribution dependence from hash functions and introduced $D(x)$ for this process. They describe four key-to-address functions based on this tidy function.

Gonnet et al. [GRG80] assume that a $D(x)$ is known analytically and propose a tidy function which has a search cost of $O(\log \log N)$ probes. Garg and Gotlieb [GG86] investigated ways to break down $D(x)$ into pieces small enough to contain uniform distributions, but give no performance results.

In some of the literature, tidy functions are referred to as *order preserving hash functions*. If a minimum perfect hash function also preserves the key order, then it is an *Order Preserving Minimum Perfect Hash Function* (OPMPHF). Clearly, it is a type of tidy function. The OPMPHF of Fox et al. [FCD91] used two mappings of a hash function to produce a bipartite graph which is then straightened out with the aid of a second hash function, which results in an order-preserving minimum perfect hash function. This method requires at least three accesses to auxiliary tables. Since the bipartite graph so formed is not always a tree, extra probes are required, the average cost of which works out to be 3.25 accesses. The construction consists of three steps, “mapping”, “ordering”, and “searching”. The first step requires the equivalent of at least 6.8 passes of the file containing keys, and the third is still more expensive. Details and more related works on OPMPHF can be found in further references [FHCD92, CHM92, GSB94].

More discussion on tidy functions will be given in Chapter 4 and 5.

3.2 Multikey File Structures

Multikey data structures permit access to data on several fields together or independently, based on one index structure. Multidimensional tree structures such as *k-d-trees* [Ben75] generalize the binary tree to multiple dimensions by cycling, from one level in the tree to the next, through the attributes to be used as composite keys. Like binary trees, k-d-trees can be unbalanced and thus may degenerate, resulting in $O(N)$ access time instead of $O(\log N)$, where N is the total number of keys. *K-D-B-trees* [Rob81] combine k-d-trees with *B-trees* [BM72]. Like B-trees, they preserve the height balance and have a guaranteed logarithmic retrieval performance. However, at that time, two direct access methods — *multipaging* [MO81a, MO82] and *grid files* [NHS84] were proposed as alternative direct access multikey file structures. In the next two subsections, we review direct and logarithmic multikey file structures accordingly.

3.2.1 Direct Access Methods

Grid files

In grid files, k-dimensional (kD) data space is partitioned by repeated bisections of the data space in each dimension into orthogonal grids. The grid can be defined by a kD array called *scales*. A boundary of a scale is a (k-1)D hyperplane which partitions the data space into two disjoint parts. Based on the scales, the *grid directory*, a kD array, can be built. An element of the grid directory is a *grid cell*. The grid cell has a pointer to a data page containing all data records that lie in the grid cell. There is a many-to-one correspondence between a grid cell and a data page on secondary storage. The region of grid cells pointing to the same data page is thus called a page region. Page regions are in shape of hyper-rectangles so that data can be clustered for efficient range queries.

Figure 3.1 is an example of grid directories in 2D. There are 7 pairs of numeric 2D data: {(7,15), (255,0), (0, 64), (0, 127), (32, 128), (64, 128), (96, 192)}, assuming the page capacity is two records. The example shows that the grid directory is obtained by repeated bisections of data space in each dimension in turn. Grid cells holding "-1" point to null data pages. Please note that page regions, grid cells with same page numbers, are all rectangular.

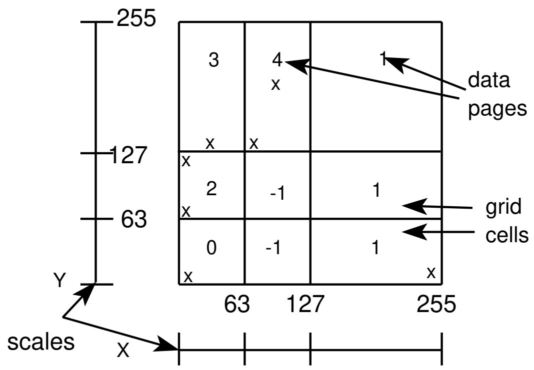


Figure 3.1: Grid Directory in Two Dimensions

Since the grid directory can be large and therefore cannot be assumed to be stored in RAM, the search cost for a data item is precisely two disk accesses: one to the grid directory, the page that contains the right search key address, and the other to the corresponding page address to retrieve the record.

The grid directory is stored as an extra index, with its size depending on the size of the source file. According to Regnier [Reg85], the growth of the index is superlinear to the file size for uniformed distributions. Moreover, the constraints on grid partitions prevent grid files from fine tuning. As a result, the size of the grid directory is also sensitive to data distribution. In fact, grid directories can grow extremely large under poor distributions. Heavily, nonuniformed data make some regions of the data space require more partitions than the rest. Yet these partitions are performed not only on the local regions but also across the whole data space. Thus, this can cause the increment of scales, grid cells and the size of the grid directories.

Bang files [Fre87] and *nested interpolation-based grid files* [OM92] were aimed at improving the growing directory problem against poor distributions. But their improvements on index sizes are based on tree structured directories, resulting in logarithmic search cost to the size of a problem, which differs from the direct access methods ($O(1)$).

Multipaging

Multipaging was invented even before grid files. Like grid files, the kD data space is partitioned rectilinearly by (k-1)D hyperplane parallel to all axes except the one which it intersects orthogonally. The partitions impose a grid of hyper-rectangles in the kD space. Every hyper-rectangle can be addressed by k 1D arrays called scales, just as in grid files. The main design difference is that there is a one-to-one correspondence between a hyper-rectangle and a data page. In addition, a scale of a dimension is obtained based on partitions on the specific set of data distributed over that field; each range of values of an attribute field is partitioned into m intervals such that there are approximately the same number of records located on each interval.

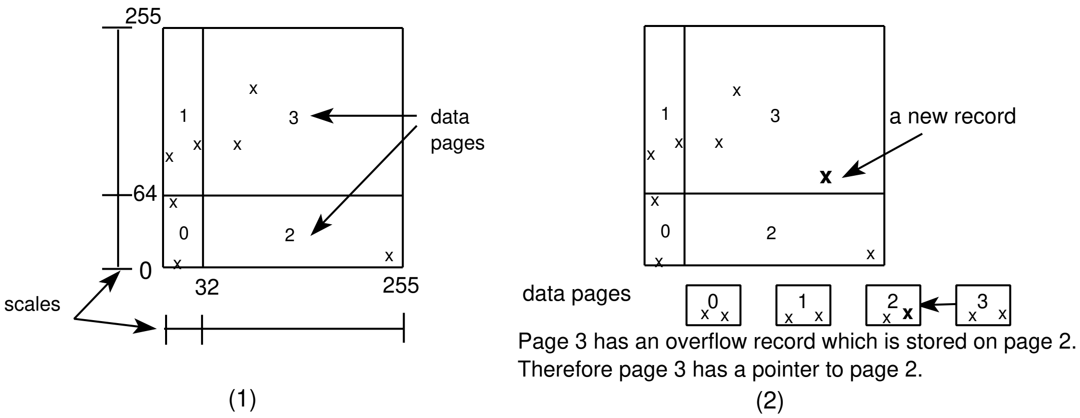


Figure 3.2: Multipaging in Two Dimensions

Figure 3.2(1) shows an example of a 2×2 multipage space, with the same data as given in the grid file example (7 pairs: $\{(7,15), (255,0), (0, 64), (0, 127), (32, 128), (64, 128), (96, 192)\}$). The numbering of data pages are shown for reference only. It is in the column-major order for the 2D array.

If data is poorly distributed, it is likely to have overflow records on some pages.

Overflow records of a page can be stored on a chain of pages starting with the most empty page, or they can be chained onto separate overflow pages. For example, figure 3.2(2) shows that when record (200,100) is inserted into page 3, it makes the page overflow. With the first collision resolution strategy, a pointer on page 3 is used to point to page 2 where the record is actually stored.

In the absence of knowledge of the shape of the data space, it is reasonable to assume that it is hypercubic, with axes of equal length, $n^{1/k}$, where n is the number of total pages. Then scales in kD are of size $kn^{1/k}$, and can easily be fit in RAM — there is almost no storage overhead. Thus ideally, multipaging can retrieve any record for an exact match in exactly one disk access. If there are overflow chains, more accesses might be needed. Poorly distributed data may cause many overflow records and long overflow chains, and thus increase the search cost. Distributions of a diagonal line and a circle in 2D are two examples of pathological distributions for multipaging.

One way to reduce the length of overflow chains is to reduce the *load factor*, the ratio of occupied space to available space, i.e., multipaging allows storage space to be traded for access cost.

There are versions of multipaging both for static and dynamic data [MO81b, MO82]. The former uses an algorithm to analyze the data in $O(kn \log n)$ time, and determines the storage utilization and expected number of probes possible for a multipaged file. The latter solves the problem of representing dynamic multidimensional arrays, and controls either the storage utilization or the expected number of probes.

3.2.2 Logarithmic Access Methods

R-tree Family

R-trees were first proposed by Guttman [Gut84] as a direct extension of B-trees for spatial data in multidimensions. Like B-trees, R-trees are height-balanced trees. A leaf node of the R-tree is of form (oid, R), where oid is an object identifier in the database, and R is a *minimum bounding rectangle* (MBR) approximation of the data object. R is of the form: $(b_0, b_1, ..., b_{k-1})$, where b_i represents the i^{th} coordinate pair of the lower-left and upper-right corners of a kD hyper-rectangle b. An internal node of the R-tree consists of $(ptrChild, R)$ pairs, where $ptrChild$ is a pointer to a child node in the next node level, and rectangle R is the MBR of all rectangles on the child node.

If M is the maximum number of entries that can fit in a node, and $m \leq M/2$ is a parameter specifying the minimal number of entries in a node, then the R-tree satisfies the following conditions:

1. The root node has at least two children unless it is a leaf.

2. Every node contains between m and M entries unless it is the root.

3. All leaves appear on the same level†.

4. For every entry in a leaf node, *oid* represents the smallest rectangle that spatially contains the kD spatial data object.

5. For every entry in a non-leaf node, R is the smallest rectangle that spatially contains all rectangles in the child node.

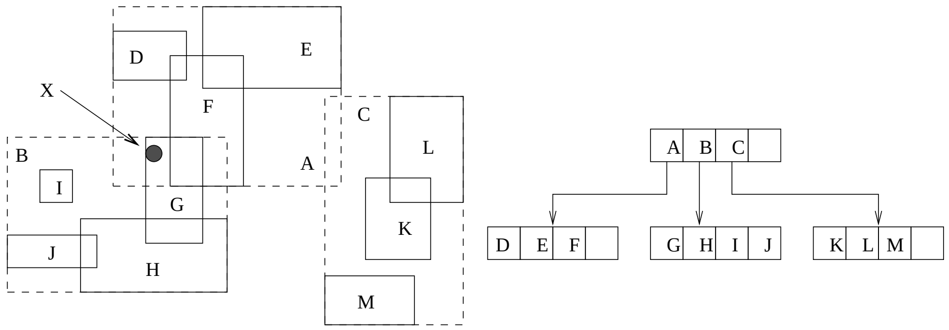


Figure 3.3: Rectangles Organized to Form an R-tree Structure

Figure 3.3 shows the structure of a R-tree and the containment and overlapping relationships among its rectangles. The R-tree families have many members, including *packed R-trees* [RL85] for static data, *R+ trees* [SR87] with guaranteed disjointness of nodes, *R*-trees* [BKSS90] with a complex but effective node splitting

†The first three properties are the same as a B-tree, except $m = \lceil M/2 \rceil$ for the B-tree.

algorithm for insertions, and *Hilbert R-trees* [KF94] that are an improved R-tree variance using fractals. There are also models developed to give analytical estimations on R-trees/R*-tree performance [TS96].

In the R-tree family, the R*-tree is a representative with the most efficiency. The construction algorithm is certainly more complex than that of the R-tree, but is still considered affordable. Thus it is the most widely applied R-tree variance. Its main improvement over the R-tree is its splitting algorithm used for insertions. The heuristics of the R*-tree combines the optimization (minimization) of the area of the MBR, the margin of the MBR and the overlap of enclosing rectangles. On the other hand, the original R-tree only minimizes the area of the enclosing rectangles.

The R-tree family is convenient for representing point and spatial objects and their embedded relationships. Like B-trees, they are height balanced trees. This limits the worst case performance in insertions and deletions to $(O\log(N))$, where N is the number of data objects. On the other hand, keeping trees balanced becomes one of the major tasks for R-trees as well as for B-trees and K-D-B-trees. Moreover, it is common that rectangles on R-trees overlap. Overlaps in the directory directly affect the query performance since it means multiple paths need to be searched. Thus worst searching cost cannot be guaranteed by $O(\log N)$, and minimizing overlaps of rectangles is of primary importance in R-tree construction. For example, data X in figure 3.3 locates in the overlap area of MBR A and B. Searching for X thus requires visiting two child nodes of the root out of three.

In a word, two major problems exist for R-tree family: how to keep it balanced, and how to minimize the overlap areas of rectangles. Tries, on the other hand, are free from both issues by nature.

X-trees for high dimensions

R*-trees may deteriorate rapidly when going to higher dimensions, as large overlap in directories may increase rapidly with the growing dimensionality of data. X-trees [BKK96] are invented as a hybrid of R*-trees and linear array-like directories, called “supernodes” (with extended variable node size), for higher dimensions in order to reduce the overlap problem in R*-trees.

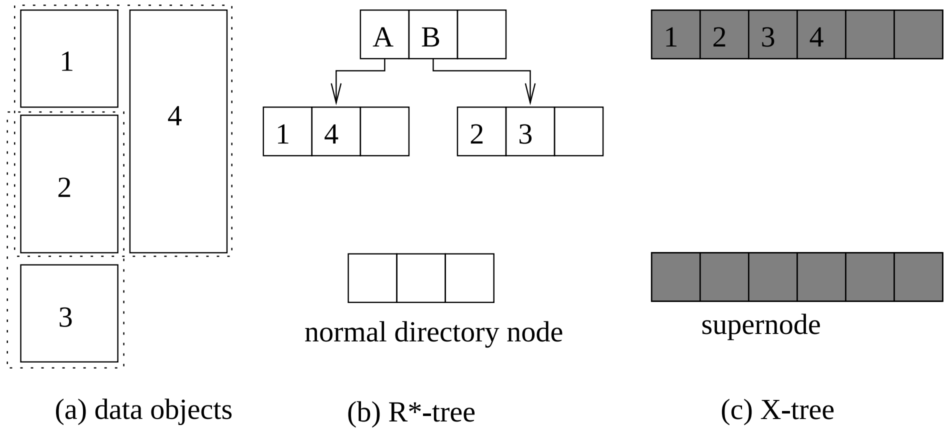


Figure 3.4: Example of X-tree Structure

Figure 3.4(c) shows a simple example of the X-tree structure for data objects shown in figure 3.4(a). For comparison, the corresponding R-tree is also given in figure 3.4(b). It is a 2D example containing 4 data objects. These objects have size 1, 2, 1 and 3 units respectively, and they are inserted according to their numeric order. We assume the maximal overlap allowed in the directory is 1 unit for X-trees, the maximal directory capacity M is 3 members, and the minimal capacity m is 2 for both the X-tree and the R*-tree. For the R*-tree, when data object 4 is inserted after object 1, 2 and 3, a split of the root containing the first three objects has to be performed since the capacity of a directory node is 3. The split should partition the four objects into two groups, each holding two objects. Note that the overlap area of MBR A and B on the root of the resulting R*-tree is 2 units.

If a node splitting using the R*-tree algorithm would make the overlap exceed a predetermined threshold value, the X-tree insertion method tries a so called "overlap-minimum-split" algorithm without considering the balance of the number of members in each group. If the second try fails due to a minimal load factor requirement on every node, a supernode is generated instead of a normal node split. The rationale is to improve the search by replacing part of the expensive hierarchical search (due to overlaps) by a cheaper linear search within the supernode. In our example, when inserting object 4, the R*-tree split algorithm is tried first. Since the overlap of A

and B is 2, greater than the threshold value of 1 unit, it then tries the "overlap-minimum-split" which generates two groups – object 1,2,3 in group one and object 4 in group two. It is indeed an overlap free split. Unfortunately, it does not meet the minimal node utilization requirement ($m = 2$), and thus fails the second attempt for a splitting. As a result, a supernode of capacity $2 \times M = 6$ is generated instead, holding all four objects. Figure 3.4(c) shows the resulting X-tree containing only a supernode. Searching an object in rectangle 2 by the X-tree becomes a linear search of the supernode in figure 3.4(c) instead of a traversal of all the R*-tree nodes in 3.4(b).

Obviously, a problem with the X-tree is that in very high dimensions, the construction algorithm may fail to generate a hybrid hierarchy structure at all; instead, the whole tree deteriorates into one single supernode. As a result, the search becomes a linear search of the directory. Although this linear search might still be more efficient than tree searches by the R*-tree, it is likely not the most efficient method available. In addition, the construction cost of the X-tree is much higher than that of the R*-tree.

There are many other structures for high dimensional data. Among them are the TV-tree [LJF94], the hB-tree [Lom90], the SS-tree [WJ96] and the SR-tree [KS97a], etc.

The TV-tree improves the performance of the R*-tree for higher dimensional data by employing the reduction of dimensionality and the shift of active dimensions. But a restriction is that it requires an ordering on dimensions based on importance, and the dimensionality is reduced by activating only a few of the more important dimensions for indexing.

hB-trees [Lom90] are based on K-D-B-trees and the idea of "holey-bricks", bricks in which subregions have been extracted. The internode search and growth processes of hB-trees are precisely analogous to the corresponding B-tree processes. The intranode processes use k-d trees as the structure within nodes. Node splitting results in a k-d tree split which produces nodes no longer represented by brick-like regions in k-space, but rather as holey bricks. hB-trees have been invented to obtain high storage utilizations of the directory.

The SS-tree and SR-tree are two index structures especially designed for high-dimensional nearest-neighbour queries.

Unlike tries that give data compression, all the above index structures require extra space for directories. The utilization (load factor) of directory pages and data pages is normally between 1/2 and 2/3.

K-D-B-trees

K-D-B-trees [Rob81] extend B-trees to multidimensional space. The data space is recursively partitioned into hyper-rectangular subspaces, each of which corresponds to a page. The k-coordinates of each point in a data page correspond to the k key attributes of a tuple in a page. It is not specified how subspaces are to be represented in the index pages. A splitting of both data and directory pages can be performed on an arbitrarily chosen point along an arbitrarily chosen partition axis. The method in general has to cascade a split downwards through every lower level of the tree to the leaves when a directory page is split. This is a severe drawback, since the insertion costs and the minimal occupancy of the resulting pages have no guarantees.

A later attempt by Freeston [Fre95] is also a generalization of B-trees to k dimensions. The new BV-tree is designed to have a guaranteed minimal page occupancy of 1/3. The trade-off is that it is no longer a height-balanced tree.

3.3 Join Algorithms

3.3.1 General Review

Joins are one of the fundamental database operations. Since the invention of the relational data model in the early 1970's [Cod70], many join processing techniques have been proposed and investigated. Basically, there are three classes of join algorithms according to implementations: nested-loops joins, sort-merge joins and hash joins. Early database query processing schemes were based on either nested loop (nested block scans) joins or sort merge joins. Hash based joins were proposed later to improve the join performance. Mishra et al. [ME92] have written an overview of many of the three classes of join algorithms. There had been a long debate on the issue of the best join schemes (sort merge joins or hash joins) until recent comprehensive studies showed that there exist dualities between the two [GLS94]. Current commercial databases support both of them.

There are many ways to classify join algorithms. They could be grouped by whether the algorithm uses special indexes such as join indexes [Val87, BM90], or by different join operators. The join operator can be *equijoins* (including natural joins), and *non-equijoins*, including band joins [DNS91, LT95], division joins, etc. Join algorithms that are efficient for one type of join predicates may not be efficient for another.

Meanwhile, there has been a lot of effort placed in the development of efficient join algorithms for parallel and distributed database systems [ME92, CY96, HCY97]. Efficient spatial join algorithms have become a new focus for database researchers. According to whether indexed structures are used or not, spatial join algorithms can be classified as joins based on special data structures, such as R*-tree joins [BKS93, HJR97] and seeded trees [LR94], and non-index based join algorithms, such as size separation spatial joins [KS97b], spatial hash joins [LR96], partition based spatial-merge joins [PD96], etc.

Nevertheless, natural join operations are one of the most critical and fundamental operations for efficient query processing. Many such joins algorithms have been proposed since the 70s. The relatively recent algorithms among them include *Joins by Fragmentation*(JF) [Sac86], *Distributive Join*(DJ) [NP91], *Stack Oriented Filter Technique* (SOFT) *joins* [SM94] and *Bucket Skip Merge Join* (BSMJ) algorithms [KR96]. These natural join algorithms apply one or more of the following strategies:

- partitioning (see next section);
- avoiding duplicate page accesses;
- skipping page accesses whenever possible;
- using special (extra) indexes.

Overall, there is no single algorithm that is the best in all cases; a proper algorithm should be chosen based on the characteristic of an application. Furthermore, to the best of our knowledge, there has been no work on join algorithms that takes the advantage of structures that directly organize the data, i.e., without using extra indexes, for better join performance, and yet achieving spatial compression at the same time.

3.3.2 Some Representative Join Algorithms

We consider joins of large relations that cannot be held in RAM. First, we will review several join algorithms under the assumption that values of join attributes are unique. In the last subsection, we will discuss the situation when duplicate join attributes are present.

3.3.3 Sort-Merge Join (SMJ)

The standard sort-merge join is executed in two steps. In the first step, the two relations R and S are sorted according to their join attributes. This can be done using a *(B-1)-way* disk merge-sort algorithm [BE77]. Since the number of passes of the relation during the sorting is $\lceil \log_{(B-1)} N \rceil$ (see Section 1.3 for symbols), the I/O cost of the merge-sort is

$$2N_R\lceil \log_{B-1} N_R \rceil + 2N_S\lceil \log_{B-1} N_S \rceil$$

where N_R, N_S is the number of records in R and S. The second step is the merge-scan, during which both relations are scanned in the order of the join attributes, and tuples satisfying the join condition are merged to form the output relation. The I/O cost of the merge-scan is exactly $N_R + N_S$ if the join attributes in R or S have no duplicates. Otherwise, the cost may not be linear [BE77, Sac86]. See more discussion in Subsection 3.3.8.

The sort-merge join algorithm has found to be the best choice if there is no indexes available on the join attributes, if not much is known about the selectivity, and there is no basis for choosing particular algorithms [BE77, Su88, ME92].

3.3.4 Stack Oriented Filter Technique (SOFT)

Among the recent techniques of join operations, partitioning[‡] has been found to not only ease parallelization but also improve the overall efficiency. The basic idea is to partition the input relations R and S into m disjoint sub-relations R_i and S_i, where R_i and S_j are disjoint if $i \neq j, i = 1, ..., m$, such that

$$R \bowtie S = \sum_{i=1,...,m} R_i \bowtie S_i$$

SOFT, hashed loop joins[DG85], JF and DJ are some of the join implementations of the technique.

In SOFT, joining sets are repeatedly divided by a maximum of five statistically independent hash functions until a partition of both joining sets is found to have an identical join attribute. We will not discuss SOFT further as it uses stacks during the partitioning process and assumes that data sets can fit in RAM, while our interest, as mentioned at the beginning, is in large data sets stored on secondary storage.

3.3.5 Join by Fragment (JF)

As a representative partition based hash join algorithm, JF applies relatively similar ideas to SOFT. First, it partitions the joining datasets recursively into disjoint subsets by hash functions, with the use of B blocks of memory buffer. The difference is that the partition continues until any subset from the smaller relation can fit into a memory buffer of size $B - 1$. In this way the JF algorithm partitions the data sets without sorting them. Then it performs the join by a merge-scan of subset pairs.

Figure 3.5(a) gives relations R and S. Figure 3.5(b) shows the merge-scan phase by JF, assuming the block capacity is 4 data items. In the figure, sets R and S have been partitioned into fragments by a hash function before the merge-scan phase. The hash function used during the partition is k mod 2, i.e., the first fragments for R and S hold odd values, and the second fragments hold even values.

[‡] In the literature, it is inaccurately referred to as “divide-and-conquer”. Strictly speaking, divide-and-conquer is a strategy for reducing asymptotic complexity, such as sort from $O(N^2)$ to $O(N \log N)$.

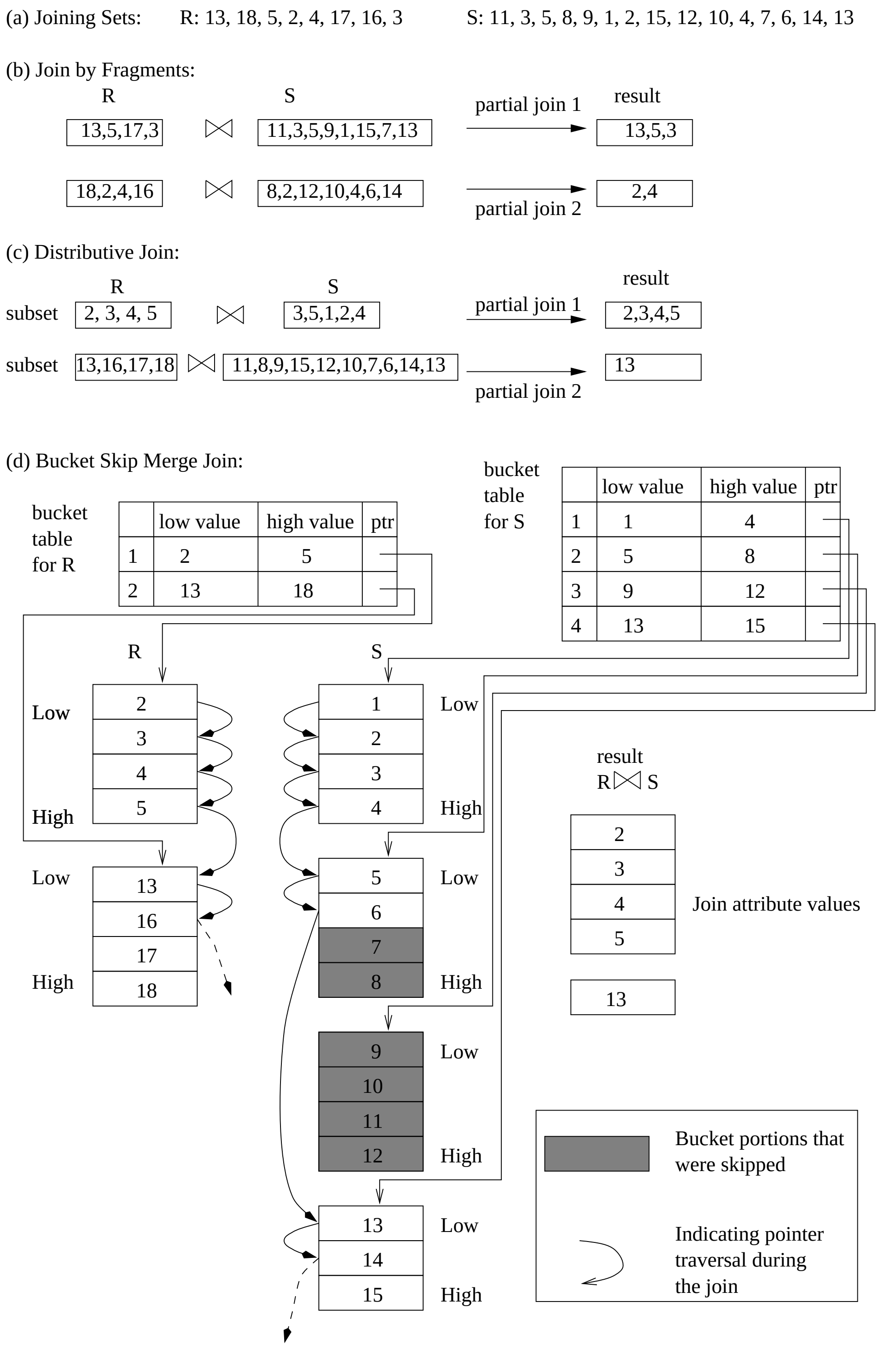


Figure 3.5: Three Join Methods for Data Set R and S

The I/O cost of JF contains two parts, the partitioning cost and the merge-scan cost. With the use of B blocks in memory, JF applies the hash function ($k \bmod (B-1)$) to partition R and S into $B-1$ sub-groups. Subsequently, it divides the subsets recursively until one of the resulting subset R_i and S_i has no more than $B-1$ blocks. In case of uniform distributions, if the number of partitions is k_R for relation R, then $N_R/(B-1)^{k_R} \leq B-1$. Therefore

$$k_R = \lceil \log_{B-1} N_R \rceil - 1$$

Each application of the hash partition on R has an I/O cost of

$$N_R + (B-1)\lceil \frac{N_R}{(B-1)} \rceil \approx 2N_R$$

As a result, the cost of completely partitioning R or S is

$$2N_i(\lceil \log_{B-1} N_i \rceil - 1)$$

where $i = R$ or S. Note that this is lower than the sorting cost by SMJ. In case of no duplicate join attribute values in R or S, the merge-scan phase is just $N_R + N_S$ (same as SMJ) and the total cost of the JF is therefore

$$2N_R(\lceil \log_{B-1} N_R \rceil) + 2N_S(\lceil \log_{B-1} N_R \rceil) - N_R - N_S$$

Like other hash partition join techniques, the JF algorithm is particularly suitable when indexes for joining attributes do not exist and the data is unsorted.

3.3.6 Distributive Join (DJ)

Although sharing the same "divide and conquer" idea as JF, the approach by DJ is different. DJ belongs to the "sort-merge" category. But it improves the standard sort-merge algorithm by avoiding completely sorting both input data sets on the join attributes. Instead, it only sorts the one with the smaller amount of data completely and partitions it into subrelations. The size of the subrelation is less than that of the available RAM buffer. The boundary values of these subrelations are stored in a table called the *distributive table*. Then the large relation is partitioned and partially sorted using the distributive table. The DJ algorithm, therefore saves part of the sorting cost. There are three steps in the algorithm:

1. completely sort the smaller data set into ordered subsets $(R_1, R_2, ..., R_m)$ with each subset size no more than $B-1$, and produce a distributive table containing boundary values of the join attributes of the subsets;
2. distribute the larger data set into subsets using the values in the distributive table as the boundaries of its subrelations using a partial distributive sort;
3. perform the join between the sorted and the distributed sets.

Figure 3.5(c) shows the merge-join phase of sets R and S given in figure 3.5(a). Before the merge-join, set R has been sorted into two sets, while set S has only been sorted partially.

The cost of DJ consists three parts: the cost of complete sorting of the smaller relation R, $(2N_R\lceil\log_{B-1} N_R\rceil)$, the cost of distributing the larger relation S into $\lceil N_R/(B-1)\rceil$ subsets, and the cost of merge-join between R_i (sorted set) and S_i (distributed set), where $i = 1, ..., m$. The last part is $(N_R + N_S)$ in the absence of duplicate join values. Using a $B-1$ way distribution algorithm on secondary storage, the number of passes to distribute S into m subrelations is $log_{B-1}\lceil N_R/(B-1)\rceil$, which is strictly smaller than the number of passes to sort S completely by SMJ in most cases [NP91]. Therefore at least one pass of the large relation can be saved in comparison to SMJ.

3.3.7 Bucket Skip Merge Join (BSMJ)

The Bucket Skip Merge Join (BSMJ) improves SMJ by creating and maintaining some extra storage, bucket tables on secondary storage, to hold upper and lower values of join attributes for each data bucket (page) as well as pointers to corresponding data buckets. The bucket table entries are first accessed during the join processing in order to avoid data bucket fetches and attribute comparisons in RAM whenever possible.

BSMJ has its great advantage over all hash joins including JF and DJ in the situation that the input sets have been sorted and indexed.

Figure 3.5(d) shows bucket tables, data buckets, file pointer movements of the bucket skip merge join on R and S in figure 3.5(a) and the result of the join. The shaded area is the bucket portion that has been skipped during the join.

BSMJ focuses on sorted relations. By maintaining extra storage for bucket tables, the cost of merge-join can be less than one pass because of the skip factor, when there is no duplicate values.

3.3.8 Duplicate join-attribute values

In the presence of duplicate values, the cost of all above join algorithms may increase.

When DJ algorithms are applied to duplicate values, the cost to distribute the larger relation increases in order to meet the restriction that all tuples having the same value of join attributes must belong to the same subrelation. This suggests there may be more subrelations in the presence of duplications. But it has been proven that with duplicates, the extra cost of distributing the larger relation can be no more than one pass than without duplicates [NP91].

The cost of both partitioning and/or sorting is independent of data distribution and duplication for SMJ, JF and BSMJ algorithms. It is only the final join phase that is affected by the duplicate values. If some of the duplicate join values exceed the capacity of the available buffer in RAM, all SMJ, DJ, JF and BSMJ algorithms are forced to apply nested block scans, and the cost of merging can become quadratic.

3.4 Summary

The chapter gives a literature review on 1D to multi-dimensional data structures, direct access method to tree structures, as well as a review on join algorithms. In conclusion, there does not exist a single approach as a solution for all the problems posed by one dimensional to multidimensional data, low to high selectivity, special to structured data, efficient in both space and time and distribution independence.

Chapter 4

Tidy Functions

4.1 Piece-wise Linear Tidy Functions

Let us look at the tidy function definition from section 3.1.2:

$$t \equiv \lceil n\ D(x) \rceil \tag{4.1}$$

where x is a key and n is the number of pages on secondary storage. If t has to be stored in its full resolution, it may exceed the RAM capacity for large files. Indeed, it is valuable to make the assumption that RAM capacity is $O(1)$ while file size is $O(n)$. Tidy functions stored on secondary storage will not only increase the storage cost but also the retrieval time. A solution for this is a piece-wise linear interpolation of $nD(x)$, which results in a so called *piece-wise linear tidy function.*

Assume that the number of linear pieces, p, is given, the tidy function can be represented by $2p$ numbers with $2(p-1)$ pairs of x and $nD(x)$, the range of keys and the total number of pages n. This tidy function is certainly a bigger structure and more complex to calculate than a simple hash function, such as $x \bmod n$. But it is small and quick compared to secondary storage which it is addressing. Here, $p \ll n$ is bounded by RAM capacity, and has a data distribution dependency as well.

We assume that the file is loaded onto pages in key order just like a sequential file. The load factor can be as high as one, i.e., our tidy function is indeed an order preserving minimal hashing function. But it need not to be one; there can be empty space in our file. In this case, we simply have a different distribution function to approximate, and the tidy function is an order preserving hashing.

Overflows

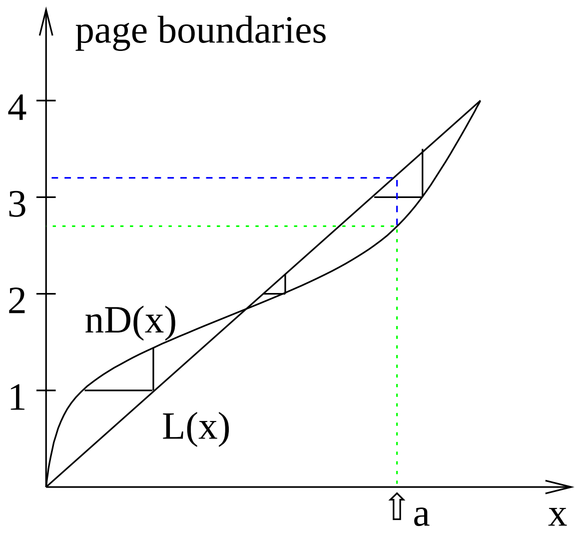


Figure 4.1: A Distribution Function and a 1-piece Line Tidy Function

Figure 4.1 shows a distribution function $nD(x)$ and a one-piece linear approximation tidy function $L(x)$. Three horizontal lines are drawn at page boundaries from $nD(x)$ to $L(x)$. We shall show that these lines represent the overflow keys that the tidy function $L(x)$ gives the wrong page address. For example, key a maps to page 3 according to tidy function $nD(x)$, but according to the approximate line $L(x)$, it should be on page 4. As a matter of fact, $L(x)$ maps any key on the three horizontal lines between $nD(x)$ and $L(x)$ at page boundary 1, 2 and 3 to a wrong page address. Thus these horizontal lines at page boundaries represents all possible records which will cause the search to make at least one extra probe. The sum of the length of these lines gives the total number of overflowing records. For continuous key space this holds strictly, but we still get the right proportion if the key space is discrete. The above discussion counts both successful and unsuccessful searches. More often, successful searches are more useful and one should only apply records present on those horizontal lines between the tidy function and the linear pieces at page boundaries.

Now we show that the *area* between $nD(x)$ and $L(x)$ gives approximately the total number of overflows. This is because the horizontal lines drawn at page boundaries are equally spaced vertically, by a distance of a page, h. Each section of the area between $L(x)$ and $nD(x)$ is approximately a parallelogram of height h and length b_i, the length of the horizontal line. We have

$$area \approx \sum_i b_i h \propto \sum_i b_i$$

Clearly, the *area* is proportional to the total length of horizontal lines drawn at page boundaries which represent the total overflow. In other words, the area between the tidy function $nD(x)$ and the approximation linear pieces $L(x)$ gives the total number of extra probes.

When overflows occur, different relative positions of $L(x)$ and $nD(x)$ give different extra probings. If $L(x)$ is above $nD(x)$, the linear probing is downwards from the target page of the tidy function to the page it truly presents. The linear probing is upwards when $L(x)$ is below $nD(x)$. Any page that contains an intersection of $nD(x)$ with $L(x)$, with $nD(x)$ starting above and crossing downwards under $L(x)$, will have no overflows. On the other hand, any page with an opposite intersection will overflow in both directions and thus needs both downwards and upwards linear probings.

Obviously, the goal of the piece-wise linear tidy function approximation is to find a set of p points on $nD(x)$ with minimal overflows, i.e., the minimal sum of the length of horizontal lines, or the minimal area between $nD(x)$ and $L(x)$.

Analytical Example

We start to minimize the overflow with a simple analytical example $D(x) = x^2$, with $x \in [0, 1)$, by two linear pieces. Assume that the two pieces meet at (z, z^2). Then $L(x)$ contains two pieces: $L_1(x) = zx$ and $L_2(x) = (1 + z)x - z$. The area between the two curves are

$$\int_0^z (zx - x^2)dx + \int_z^1 ((1 + z)x - z - x^2)dx = \frac{1}{2}(z^2 - z + \frac{1}{3}) \tag{4.2}$$

After minimizing the above, we get

$$\frac{d}{dz}\frac{1}{2}(z^2 - z + \frac{1}{3}) = z - \frac{1}{2} = 0 => z = 1/2$$

So the optimal approximation by two linear pieces is to partition the range of search keys at the middle. Similarly, we can minimize the overflow with three pieces of $L(x)$ for $D(x) = x^2$ at points $(z_1, {z_1}^2)$ and $(z_2, {z_2}^2)$. The calculation gives $z_1 = \frac{1}{3}$ and $z_2 = \frac{2}{3}$.

4.2 Heuristic Construction Algorithms with Minimal Overflow

In practice, distribution functions are not analytical. For distributions of interest, we find that we can turn to dynamic programming to get the optimal partitions with minimal overflow.

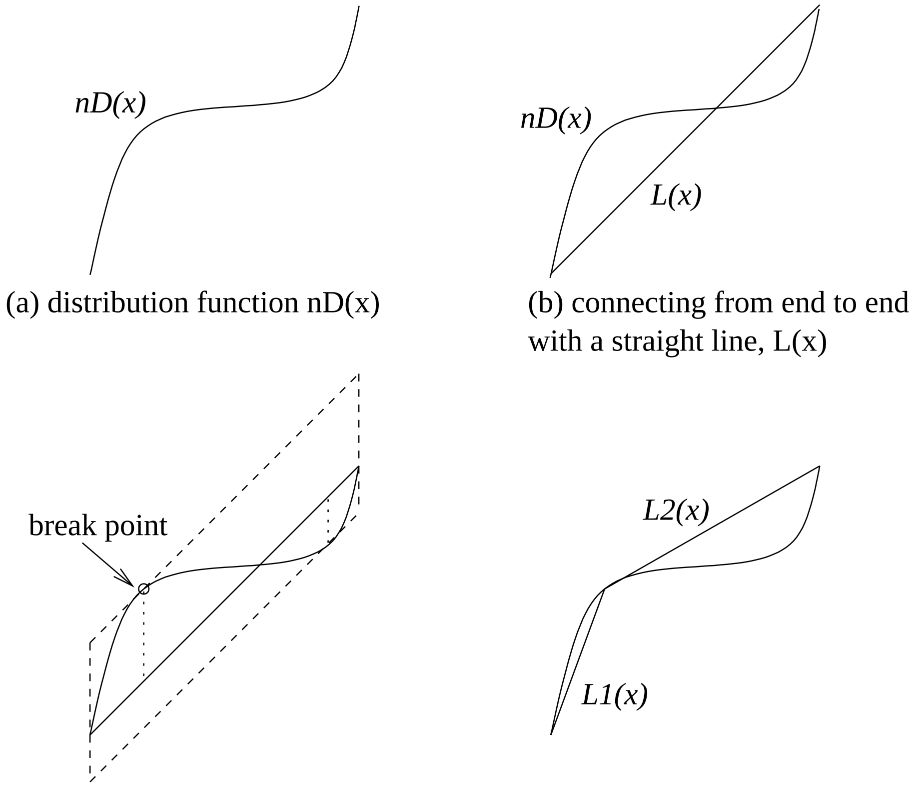
Let $m(n, p)$ denote the minimal overflow cost if we use p linear pieces to approximate the distribution function $nD(x)$, and $d(i, j)$ be the overflow cost of the line piece if we connect point i to j with a straight line. Then $m(n, p)$ can be recursively defined as follows:

$$m(n, p) = min_{p<i<n}\{m(i, p-1) + d(i, n)\}$$

where n is the number of pages, $0 < i < n$ and p is the number of linear pieces. A brute-force method to calculate $m(n, p)$ costs $O(n^p)$. Dynamic programming solves it by calculating and storing every possible value of m in a $n \times p$ table, and d in a $n \times n$ table, and therefore avoids recomputation of every entry in these tables. A bottom-up calculation of the table elements is used, from $m(1, 1), m(2, 1), m(2, 2), m(3, 1), m(3, 2)$, ..., $m(i, j)$, ..., to $m(n, p)$. It turns out that the cost of this dynamic programming is $O(pn^2)$. This is too many disk accesses to be acceptable. Instead, we propose a less expensive heuristic construction algorithm which can also make the optimal use of p linear pieces by applying them to the line segments with maximal overflow.

With the new method, systematically, we look for straight segments of the distribution, and to fit the longest and straightest of these segments using as many of the p pieces as we can; i.e., we look for sections with zero curvature.

We do this by bounding $nD(x)$ with a sequence of parallelograms, iteratively splitting the most expensive one until there are p parallelograms. The first steps are shown in Figure 4.2.



(a) distribution function nD(x)

(b) connecting from end to end with a straight line, L(x)

(c) forming a bounding parallelogram with breakpoint at the maximum vertical distance from nD(x) to L(x)

(d) forming two linear pieces L1 and L2

Figure 4.2: Forming a Bounding Parallelogram in Tidy Function Construction

We start by connecting $nD(x)$ from end to end with a straight line (Figure 4.2(b)). Then we look for the point on $nD(x)$ which has the maximal vertical distance from the line (Figure 4.2(c)); this maximum vertical distance corresponds to the longest overflow chain from the page which the straight line indicates the record at that point is on, to the page it is really on, according to $nD(x)$. We call the point "breakpoint", for it is chosen as the endpoint of two new straight lines (Figure 4.2(d)). Considering each of these lines as an approximate tidy function, we calculate the overflow cost for each. The segment with the larger cost is the next candidate for splitting. For the candidate segment, we repeat step (c) and (d) in Figure 4.2. Thus, we again find the point of $nD(x)$ in this segment that is the maximum vertical distance from the line, and make it the endpoint of two new straight lines. As an example, these first few steps are shown in Figure 4.3.

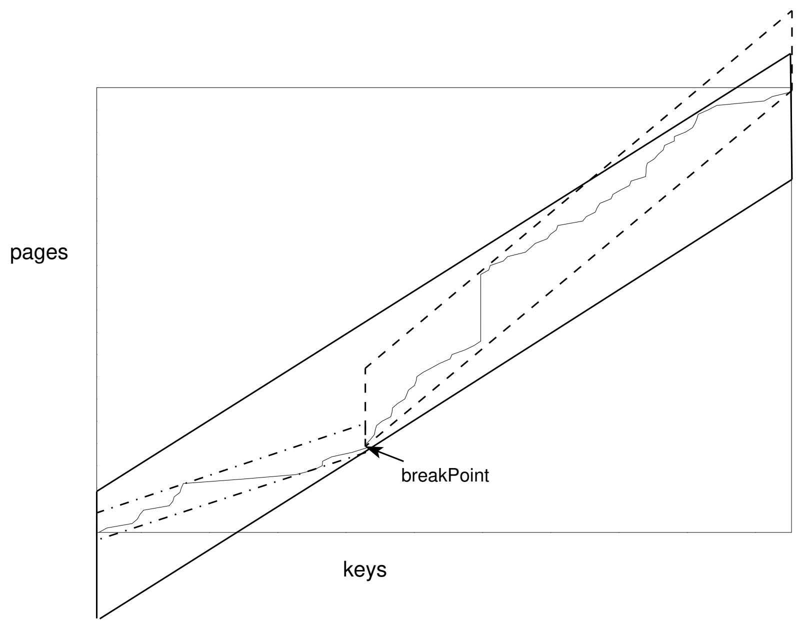


Figure 4.3: Finding p Segments of Zero Curvature: First Steps

In the end, we have a fit to the original distribution, such as shown in Figure 4.4. The example shows a fit of 10 pieces to an original data distribution $100D(x)$ given in Figure 4.3. The numbers from 1 to 9 on the curve correspond to the iteration at which the point is selected as the "breakpoint".

The parallelograms that we show in the figures serve to indicate the furthest vertical point of the distributions. Clearly, the skinnier the parallelogram is, the more nearly the part of the distribution function it bounds has zero curvature, and the longer they are, the more useful a linear approximation is to the segment. The heuristic construction algorithm is outlined in Figure 4.5, where Q serves as a queue

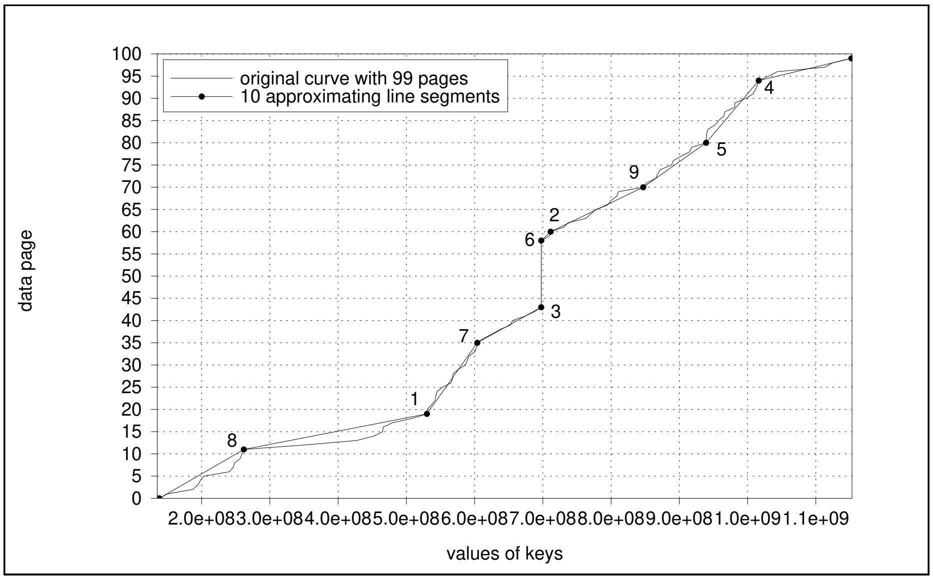


Figure 4.4: A Tidy Function with 10 Linear Pieces

of candidates for splitting, ordered by decreasing overflow cost.

Note that building our tidy function requires an initial pass of the whole file to load the data and to extract only those keys marking the boundary of pages. Thereafter, we work with the set of n keys, which we call a *pagekey file*.

The worst case cost of the above construction algorithm happens when the i^{th} breakpoint has to be found among $n-i+1$ page keys, and the search has to look over $n-i+1$ page keys. It happens when the $p-1$ breakpoints are where the first (or last) $p-1$ page keys are located. Therefore, the total cost is

$$\sum_{i=1}^{p-1}(n-i+1) = (p-1)(n+1) - 0.5p(p-1) = O(pn)$$

This corresponds to about p passes of the page keys.

The lowest construction cost happens when the data distribution is uniform and the breakpoint is always the k^{th} point of $2k$ points in any linear piece. Thus finding the i^{th} breakpoint means checking through $n/\lfloor\log_2(i+1)\rfloor$ page keys in the range. Therefore the best construction cost is

$$\sum_{i=1}^{p-1} n/\lfloor\log_2(i+1)\rfloor \approx n\log_2 p = O(n\log_2 p)$$

```
typedef struct {                         /* Definition of a Segment */
  POINT *staPoint, *endPoint, *brkPoint; /* POINT is a pair:(k, nD(k))*/
  float  overflow;
} Segment;

typedef struct {                         /* Definition of a Segment Set */
  Segment *set;
  int      num_elements;
} LineSegmentSet;

/* num:  is the number of line segment pieces
 * p0, p1: two end points of the distribution function
 * Q: a queue of candidates for splitting, ordered by decreasing overflow cost
 */
void tidyConstruction( int num, POINT *p0, Point *p1, LineSegmentSet Q)
{
  Q= {};
  aSegment= processOneSegment(p0, p1);
  insertNewElement(Q, aSegment);
  for ( i= 0; i< num-1; i++) {
    aSegment = removeFirstElement(Q); /* take 1st member of Q for processing */
    segment1= processOneSegment(aSegment.staPoint, aSegment.brkPoint);
    segment2= processOneSegment(aSegment.brkPoint, aSegment.endPoint);
    insertNewElement(Q, segment1); /* insert a member to Q such that the   */
    insertNewElement(Q, segment2); /* overflow cost is in decreasing order */
  }
}

Segment *processOneSegment(POINT *p0, POINT *p1)
/* find the breakpoint and calculate the overflow cost of the segment*/
{
  Segment *aSegment = new Segment(p0, p1);
  maxYDist = 0;
  totalYDist = 0;

  for POINT *p= p0 to p1 {
    curYDist = verticalDistance(p, Line(p0, p1));
    totalYDist = totalYDist + curYDist;
    if (curYDist > maxYDist) {
       brkPoint = p;  /* brkPoint is the one with max distance to nD(x) */
       maxYDist = curYDist;
    }
  }
  aSegment->brkPoint= brkPoint;
  aSegment->totalOverflow= totalYDist;
  /* overflow is proportional to the sum of vertical distance to nD(x) */
  return aSegment;
}
```

Figure 4.5: Tidy Function Construction Algorithm

This amounts to about $\log_2 p$ passes of the page keys. For example, if $p = 25,000$, the least tidy function construction cost is about 15 passes of the page keys.

4.3 Search Algorithms

Searching the data using the tidy function consists of two steps. First, calculate the page address where we expect to find the data using the tidy function stored in RAM, as we would use a hash function. We call this page address the *home page*. Second, like "collision resolution" in hashing, search the file for the page that really contains the data. The second step is similar to linear probing in hashing. The difference, however, is that the probing can be in either direction: either downwards or upwards from the home page as discussed in section 4.1. If the linear approximate line is *above* $nD(x)$, i.e., the data on the home page have larger value than what we are searching for, the search will continue downwards. If it is *below*, the actual data will be above the home page, and the lookup will be upwards.

In either case, if the data being hunted is not present in the file (unsuccessful search), the search must proceed in the appropriate direction(s) until the data values are above and/or below the one that we are seeking and a failure is discovered.

Our experiments show that in a large number of situations, all data that is expected to lie in a given home page may in fact be found in a range of pages considerably above or below the home page for a small number of linear pieces, p, and irregular distributions. The above search algorithm can be improved by eliminating the linear probing through the pages containing no data which are searched for by storing a pointer on each page pointing to the nearest page that actually holds relevant data. While the overhead is negligible, our experiments show an order of magnitude searching performance improvement for very large files as a result of storing these pointers.

Furthermore, sometimes even the range of pages holding the actual data maybe large, and a linear search through the pages after following the pointer is still quite expensive. We can improve the search performance further if a second pointer on each page, pointing to the furthest in the range, is used. With the two pointers on each page, we can do a binary search between these two pointers.

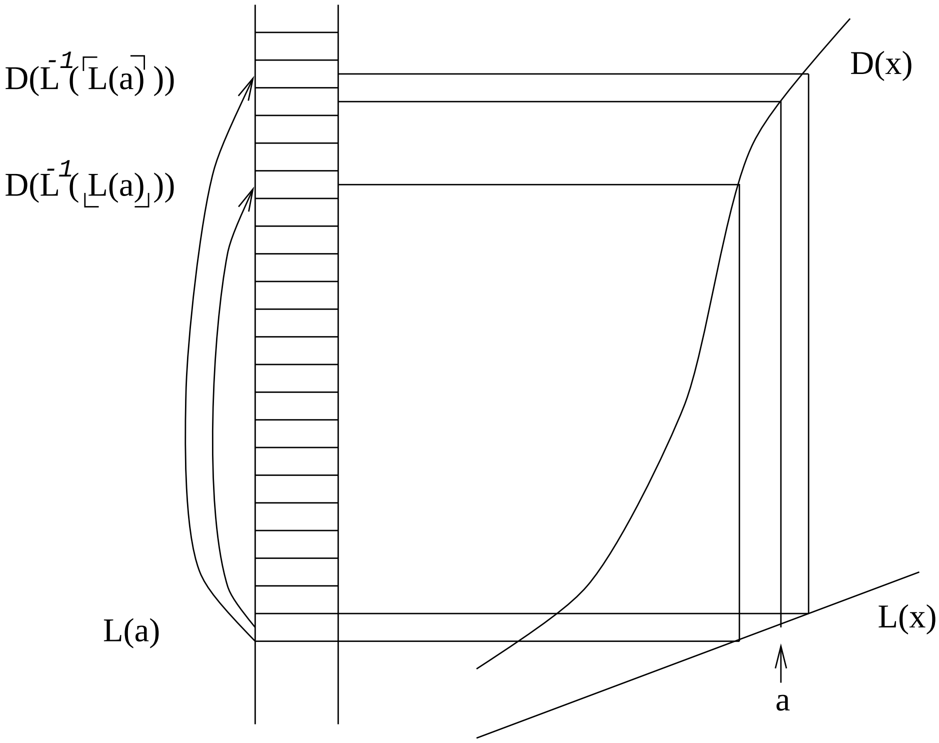


Figure 4.6: Searching and "Collision Resolution" for Tidy Function

Figure 4.6 shows the idea of these pointers by an example. It shows a key a, mapped by the linear approximation to page $L(a)$. The bottom of the page is denoted $\lfloor L(a) \rfloor$, and the top is $\lceil L(a) \rceil$. Data a is actually stored on page $nD(a)$, in a 5-page range starting sixteen pages above the home page $L(a)$. The bottom page of the range is page $nD(L^{-1}(\lfloor (a) \rfloor))$, and the top of the range is page $nD(L^{-1}(\lceil (a) \rceil))$. The two pointers in the home page to the bottom and top pages are shown in the figure. It is clear from the example that steep portions of $nD(x)$ are responsible for large ranges, and thus large overflows. This happens where the search keys are clustered closely together and there are not enough linear pieces to closely approximate these clusters.

4.4 Experimental Results

Our experiments were conducted on two sets of data. The first set was generated in the following way to produce cumulative distributions similar to what are shown in Figures 4.3 and 4.4. n random numbers were generated in g groups. Each group is based on a random (0 – 2^{15}) multiple of 2^{15}, and contains a random number of elements, each of which is generated randomly by $mod\, 2^{27}$. After sorting, these numbers produce values of keys located at the top of each of n pages, and give a cumulative distribution to work with. Each of these cumulative distributions has g major steps, with minor fluctuations. However, when g gets larger than several hundred, and $n >\approx 10^4$, the big steps tend to disappear and the resulting distribution becomes too easy to approximate well. So we modify the method by inserting additional random elements within the steps in order to increase the vertical scale of the distributions, such as those shown in Figures 4.3 and 4.4. By doing so, we increase the numbers of extra probes in searching.

The second set of data were real phone data from ProCD, a business phone book [Pro96], which contains more than 15 million American business phone numbers. It was used to verify the results we obtained from synthetic data. We choose the phone data because the cumulative distribution has big vertical steps, which challenges our tidy functions.

We will compare our experimental results with Fox's order-preserving minimal perfect hash method, the closest competitor, in construction time, storage consumption, and search time.

4.4.1 Construction

After the initial pass of the whole file to extract the pagekey file of size n, the cost of finding p linear pieces to approximate $nD(x)$ using our method depends on p. Figure 4.7 shows the number of passes we must make of the pagekey file in order to construct p linear pieces. Since the data describing these p pieces must be contained in RAM, we may build up to only so many pieces. In Figure 4.7, we stop at $p = 25,000$, and for these many pieces, the cost is 17.5 passes of the pagekey file.

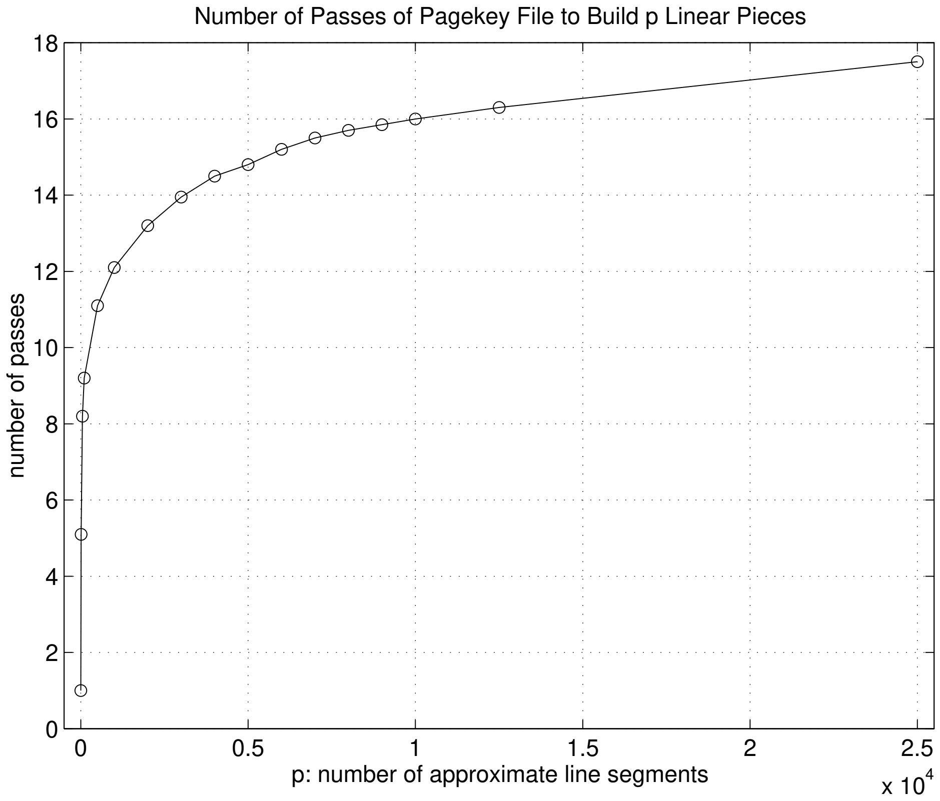


Figure 4.7: Number of Passes of Pagekey File to Build p Linear Pieces

To compare this construction with Fox's method, we make a conservative assumption that each page only holds 10 records, i.e., $N/n = 10$. This is a minimal number. We also assume that these records consist of nothing but keys. So 17.5 passes of the pagekey file need the same amount of I/O as 1.75 passes of the whole file. This is much less expensive compared with 6.8 passes for the *first* cheap stage of Fox's construction method. We have easily won by an order of magnitude as compared to Fox in the above estimate.

We can do another comparison on construction time from a later paper by Fox et al.[FHCD92] on minimal perfect hash functions which are not order-preserving. The construction cost here is lower than the order-preserving hash function by inspecting the two algorithms. In their paper, they constructed 3.9 million keys into a hash function in 33000 seconds on their fastest machine. If we suppose, reasonably, these keys and their associated records are stored on 400,000 pages, and that each key consumes 10 bytes, the pagekey file would be 4MB in size. We now estimate how much it would take to construct the tidy function on a slow disk with 2μsecond transfer time per byte. One pass of the file would cost about 8 seconds, and 17.5 passes would cost 140 seconds. On a shared disk, we would have to access each block separately. Suppose an average disk seek time and rotation time is 20ms, and a block size for the pagekey file is 1KB, then 4MB page keys would be on 4000 blocks with loading each block costing 20ms+ 1000 $\times 2\mu s = 22ms$ or 88 seconds per pass. This translates to 1540 seconds for 17.5 passes, or 21 times faster than that of Fox's method.

By both comparisons, our tidy function construction method is more than one order of magnitude faster than that of Fox et al. Moreover, our method is truly linear, and not almost linear as for the other authors.

4.4.2 Storage

Our method is a minimal perfect hashing, requiring no empty locations in the data file. The *load factor*, α, the ratio of occupied space to available space, is 1.

The storage overhead of our method is negligible. It stores only a p-piece set of linear approximations which by definition can fit into RAM, and two pointers per data page used to set up the binary search for overflows.

Although Fox et al. claim a minimal perfect hash function ($\alpha = 1$), their method must store an auxiliary table of size $1.26N$ pointers, or 1.26 pointers per record.

4.4.3 Searching

Figure 4.8 shows our experimental results on searching with the synthetic randomly generated cumulative distributions and 15 million American business phone numbers (the crosses on the figure). The cost of searching is a function of file size and p, the number of approximating linear pieces. We have measured up to 10^7 pages which may correspond to at least a 10 GB file. With this size, our expected number of probes

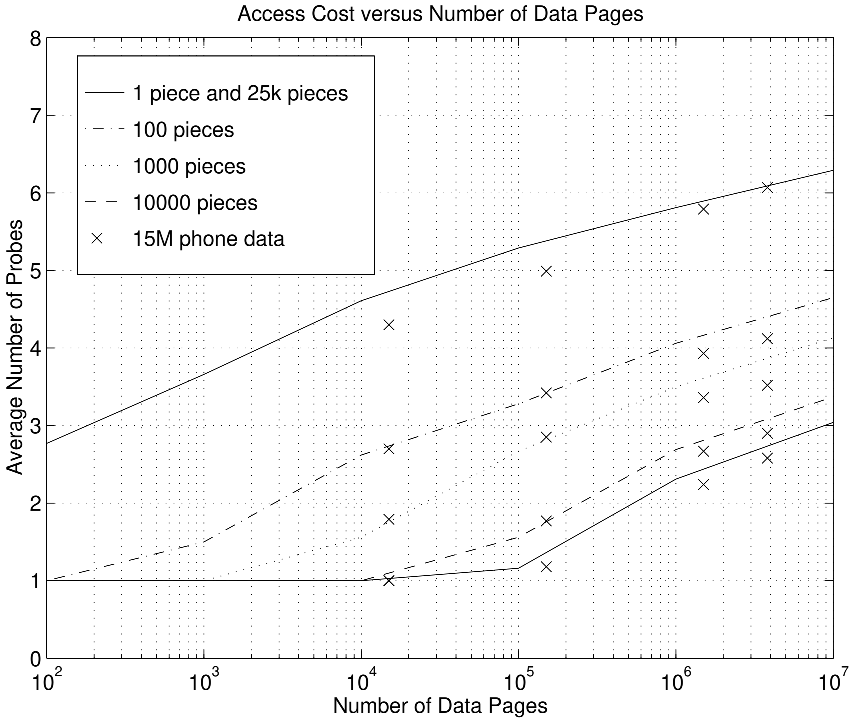


Figure 4.8: Average Probes per Search versus File Size

to search the file, with 25,000 linear pieces, just gets up to 3. This is still better than the 3.25 of Fox et al. Since Fox's search cost is independent of file size, they would eventually improve our result for large file size. However, to construct a file of 10^7 pages, or 10^8 keys, by their method would require at least 10^6 seconds, or two weeks.

In Figure 4.8, we also give the search cost tested on the 15 million phone data. The result matches with what we get from our synthetic data.

It is clear from Figure 4.8 that for file sizes massively larger than the allowed number of linear pieces, the logarithmic behavior of the binary search takes over from the direct access behavior of the tidy function. This leads to the natural question of whether the B-tree should in fact be used. It is indeed true that the search cost by B-tree can be competitive to tidy functions when the first two levels are stored in RAM. However, the construction cost of B-tree is $O(n \log n)$ while our method is much cheaper — it is $O(n)$.

4.5 Summary

In this chapter, we have introduced a class of order-preserving key-to-address transformation functions (tidy functions) that can be constructed in linear time and are significantly faster than the time for the closest competition. Our method requires no storage overhead while the earlier methods need linear index space on disks. Our method is simple in conception and the algorithms are straightforward to implement.

Chapter 5

Tries for One Dimensional Queries

A major weakness of the linear heuristic tidy function we proposed in the last chapter is that it is intended for static datasets. In this chapter, we propose tries as an alternative structure to order-preserving key-to-address transformation functions. The gain from this is three-fold: dynamic, compactness and speed.

5.1 Tries as Tidy Functions

A 1D trie on the search keys can be interpreted as if it is an order-preserving key-to-address transformation function, even though the former is not direct access method as the latter. The remaining fields of a data record are stored either on a trie leaf, or a separate file pointed to by a pointer on the leaf node. The method exploits the variable-resolution capacity of tries, their order-preserving properties, and profits from substantial compression achieved using indexed keys.

Construction of the trie consists of two steps: *a)* convert keys to their binary representations, and since these keys are in order, so are the binary representations of these keys, *b)* perform a batched process of merging trie nodes to construct DyOrTries, similar to OrTrie construction proposed by Shang [Sha94]. Trie paging can be done during the process by creating a trie page and writing it to the disk whenever trie nodes in the page layer exceeds a given page capacity. The construction algorithm reads the ordered keys once and writes the trie once. Thus, the cost is linear to the number of keys and file size.

Using the insertion and deletion algorithms of the DyOrTrie structure proposed in section 2.6, our trie is capable of inserting and deleting keys at a cost of $O(\log N)$ page accesses, where N is the number of keys. Nevertheless, the linear construction algorithm is much cheaper than building the trie by dynamic insertions.

To search a data item with a given key, first, we convert the key to its binary representation, and then look for it on the trie starting at the root. At any trie node, if the current bit of the binary key is "0", the search goes on to the left branch, and to the right branch if "1". Whenever a leaf node is reached, a record with the given key is found by following the pointer to the remaining data attributes in the file. Therefore, the cost of successful searching is h page accesses, where h is the height of the trie, assuming the root page of the trie can be stored in RAM. A search is unsuccessful if it has to stop at an internal node of the trie at page level i. The cost in this case is simply i-1 disk accesses.

5.2 Experimental Comparisons with Tidy Functions

In order to compare with tidy functions, we use the same two groups of data, as in the last chapter (unless mentioned otherwise): one is the synthetic nonuniform data, and the other 15 million American business phone numbers.

5.2.1 Storage

The trie compresses data and can achieve a significant compression rate due to the overlap of paths near the root. As the file gets larger, this compression effect becomes more pronounced. For files consisting of uniformly distributed keys only, Figure 5.1 shows that the data compression increases to over 90% as the file gets large. Obviously, this is superior to the tidy function which needs a small mount of space overhead to store information about the p linear segments, and two pointers per page used to set up the binary search.

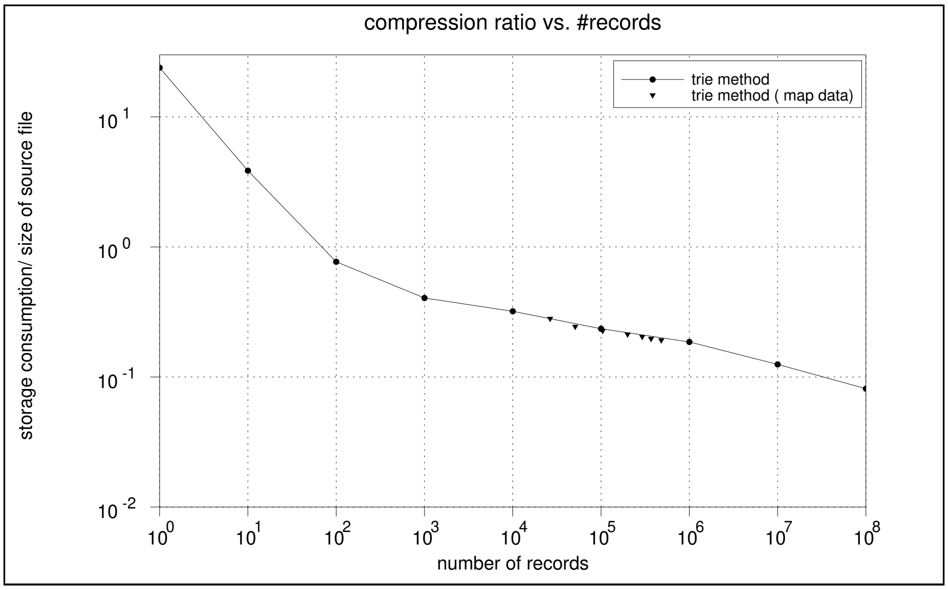


Figure 5.1: Trie Compression vs. File Size

5.2.2 Searching

Figure 5.2 shows the cost of searching by trie method on nonuniform data sets. This is compared with the tidy functions of 25,000 linear segments given in Figure 4.8 *. The trie page capacity is fixed at 4096 bytes. This measure also assumes that the first two trie page levels are stored in RAM. This is based on the assumption that the RAM buffer size can store the information of 25,000 linear segments. From the figure, we see that for large files (10^7 to 10^8 records), the trie method gives 2 page accesses, which is superior to the performance of tidy functions. For small files, the trie is at least as good as the tidy function method with 25,000 linear segments. When the file is small enough and the trie is stored in RAM, one access to the data page is the only cost of searching, this happens when the number of records does not exceed 10^5.

The figure also shows the results from the 15 million entries from phone book data. It verifies and matches with the results obtained from the synthetic data sets.

*Here file size is given in number of records instead of number of pages. We assume that a page holds 10 records.

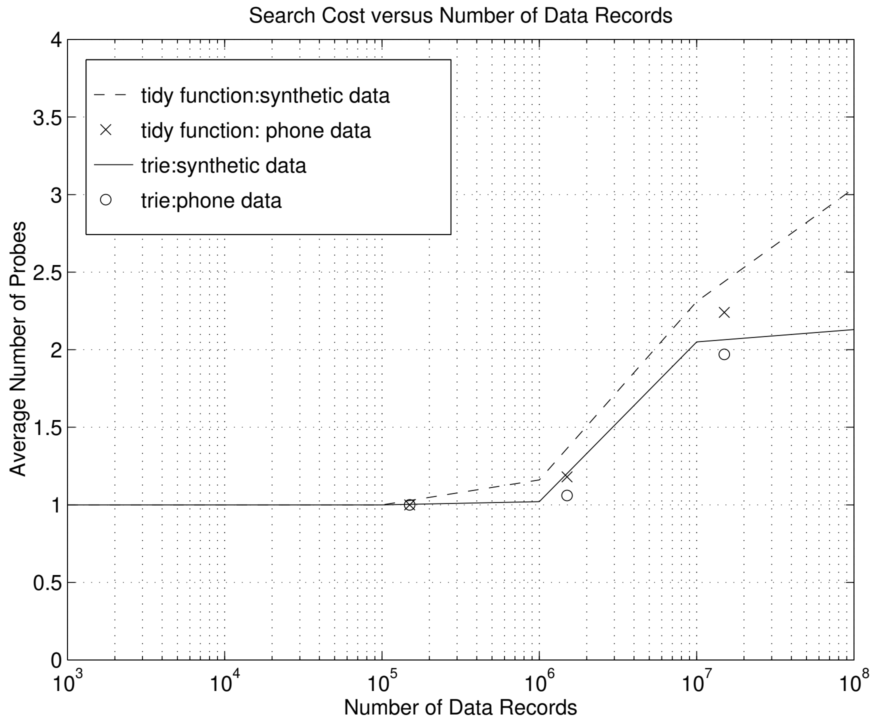


Figure 5.2: Average Number of Probes per Search vs. Number of Data Records

5.3 Summary

In this chapter, we have applied the 1D trie method as an alternative to the order-preserving key-to-address transformation function. The 1-d-trie achieves compatible searching performance to that of the tidy function. The gain is the storage compression the trie achieves, as well as its capability of dynamic insertions and deletions at a cost of no more than $O(\log N)$ page accesses per key.

A B-tree with a fanout of 100 and its first two levels stored in RAM is competitive in its search cost. We assume that the page capacity is 4096 bytes and a page holds 100 records on average. For example, with 10^8 keys, a B-tree file has four levels and a total of approximately 10^6 pages (one root page on the first level, 10^2 pages on the second, 10^4 pages on the third, and 10^6 pages on the fourth (leaf) level). The cost for searching a record is two accesses, which is quite compatible with the trie method. However, in terms of space cost, there is no way that it can be deemed compatible with the storage compression of the trie structure.

Chapter 6

Tries for Multidimensional Queries

In this chapter, we shall extend tries for multidimensional queries, including exact match and orthogonal range queries.

We shall compare our method with that of existing representative multidimensional methods. These include direct access methods such as multipaging and grid files, as well as well-applied multidimensional logarithmic methods such as R*-trees and their variances X-trees for high dimensional data.

6.1 Variable Resolution Queries

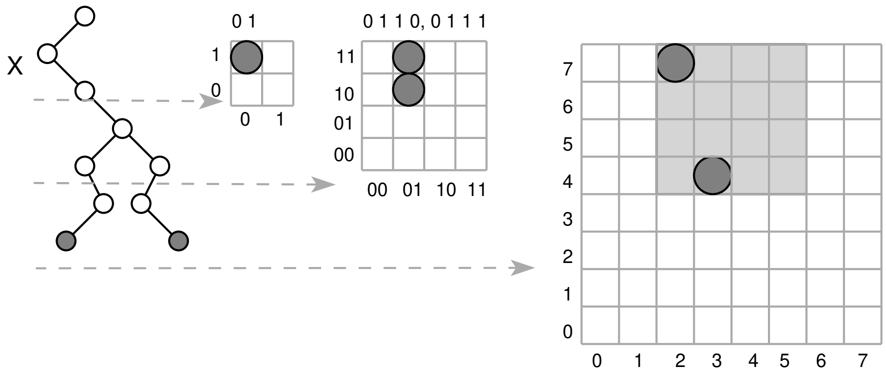


Figure 6.1: Variable Resolution in Two Dimensions

A k-d-trie for multidimensional data can be interpreted as a variable-resolution file structure. For example, Figure 6.1 shows the variable-resolution trie for two numerical point data (3,4) and (2,7). Note that the two points are interleaved into two bit strings 011010 and 011101 before inserting them in the trie. We consider two bits (two trie node levels) at a time. At the first resolution level, both strings appear as 01, representing the upper left corner of a 2×2 space shown in the leftmost square. They are not distinguishable at this time. When we move on to the next level of resolution to include the next most significant two bits, this gives two strings, 0110 and 0111, shown in the middle 4×4 square. Finally, at the full resolution level, we have the full strings in the rightmost 8×8 square.

This is how spatial queries of Merrett and Shang[MS94] display at various resolutions using one copy of data. They are not limited to two dimensions. In k dimensions, the queries consider k bits at a time. They are not limited to point data either. Edges/rectangles in k dimensions become points in $2k$ dimensions; triangles in k dimensions become points in $3k$ dimensions.

In this section, we go on to consider general queries based on variable-resolution views of tries. The data need not be spatial, but more importantly they are multidimensional and we must consider the multidimensional interpretation of data when processing queries.

6.1.1 Exact Match Queries

This section is a description of algorithms on exact match queries in multidimensions. These are prior work done by others [MO81a, Sha94, NHS84].

Exact Match by Tries

Exact match queries are one of the most frequently used queries. The process of the exact match query by trie is a straightforward 1D trie search once the multidimensional search key is interleaved into a binary string. The 1D exact match search by (full) trie method has been introduced in section 2.1.

For an exact match query using an ordinary trie, the algorithm differs only at

```
Boolean TrieExactMatchQuery( DATA key)
{
   String skey;
   TrieNode *node;

   skey= interleave(key);
   node= root();
   return (doExactMatch(node, skey));
}

Boolean doExactMatch(TrieNode node, String skey)
{
   if (node== ‘00’) { //leaf node
     //compare the suffix string stored at the leaf to that of skey:
     if (strcmp( node-> suffix(), suffix(skey, node-> level())) == 0)
       return (true); // a match
     return(false);   // a mismatch
   }
   else {//node is NOT a leaf
     if (node && current_bit(skey, node-> level())) {//a match
       node= node-> child( skey);
       return( doExactMatch(node, ++skey));
     }
     else //mismatch
       return(false);
   }
}
```

Figure 6.2: Exact Match Queries by DyOrTrie

the leaf node: a comparison of the truncated suffix at the leaf node to the remaining bits of the interleaved key has to be done. The multidimensional exact match query algorithm by DyOrTrie is given in Figure 6.2. Note that we assume the paging mechanism on secondary storage runs in the background.

Clearly, the cost of exact match queries depends on the search key length, and not directly on the size of the trie. However, the upper bound of the search cost is the height of the trie, which is a logarithmic function of the file size.

Exact Match by Multipaging

Exact match queries by multipaging can be done in two steps. First, search the scales in RAM and calculate the corresponding page index. Second, load the page into RAM and perform the search. If there are overflow records stored on other pages, these pages in the overflow chain have to be loaded and searched. So the cost of exact match by multipaging is one disk access ideally, and the length of the overflow chain plus one if there are overflows.

Exact Match by Grid file

Grid files share the first step in exact match queries by multipaging to search scales and retrieve the index. But to obtain the page address, it requires one access to the directory on secondary storage. Then the corresponding data page is loaded and examined. Thus the cost using grid files is exactly two disk page accesses.

6.1.2 Orthogonal Range Query

Orthogonal Range Queries by Tries

An orthogonal range query is a range query for multikey files, involving ranges of several fields: all records with every attribute value of a search key in a given range. For orthogonal range queries, we must interpret the trie multidimensionally. Figure 6.3 shows a 2D trie, with each node schematically indicating the region it represents (shaded area). Labels at each node present the paths from the root. Thus there is a

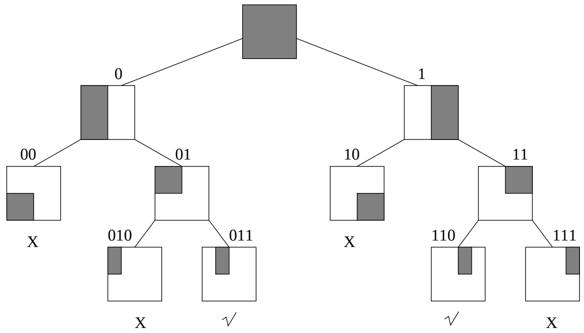


Figure 6.3: Range Query of [2,6)x[4,8) in an 8x8 Space

one-to-one correspondence between a node, determined by a path from the root, and an area it represents. An orthogonal range query involves the searching of a set of nodes whose areas overlap the given ranges. Consider the 8×8 space of Figure 6.1. The range query is from 2 to 5 in the first dimension and from 4 to 7 in the second dimension, i.e., $[2, 6) \times [4, 8)$, the shaded area in the right most, 8×8 square of figure 6.1. The range query is processed on a trie node recursively as follows:

1. If the current node does not overlap with the query ranges, it is *rejected*, along with the subtrie rooted at the node.
2. If it is entirely contained in the query, it is *accepted*, along with the subtrie rooted at the node.
3. Otherwise overlap occurs, and the search *continues* on descendents of the node recursively.

Therefore, the range query in the given example *continues* on the root and its two descendents. It *rejects* at node 00 and *continues* at 01. When processing the descendents of 01, it *rejects* 010 and *accepts* 011. Once a node is *accepted*, all the leaves that descend from it are also accepted. The same search considerations apply to node 1 as well. The crosses and checks show where the search halts in rejection or acceptance respectively.

The orthogonal range query algorithm by DyOrTrie structure is outlined in Figure 6.4. Again, we assume that the paging mechanism is running in the background. The ranges are given by the multidimensional lower and upper corner coordinates.

```
void TrieRangeQuery( DATA lower, DATA upper)
{
   TrieNode *node;
   node= root();
   doRangeQuery(node, lower, upper);
}

void doRangeQuery( TrieNode *node, DATA lower, DATA upper)
{
   if ( node-> value() == '00') { // leaf node
     if (overlap( node-> suffix(), lower, upper))
       outputSubtrieRootedAt( node); //accept the leaf node
     return;
   }
   else { //internal node:
     if (!overlap( node->area(),  lower, upper))
       return; //rejecting
     else
     if (contained( node->area(), lower, upper)) {
       outputSubtrieRootedAt( node); //accept leaves in the subtrie
       return;
     }
     else { //overlaps: continuing on descendents
       doRangeQuery(node-> left(), lower, upper);
       doRangeQuery(node-> right(), lower, upper);
     }
   }
}
```

Figure 6.4: Orthogonal Range Query using DyOrTrie

The variable-resolution property of the trie enables queries like exact match and range queries to eliminate impossible subtries at an early stage near the root and only continue to process on refined data by increasing the resolution until only relevant data remain. Thus, the cost of orthogonal range queries is sublinear to the size of the trie. Along with the fact that tries are compressed in storage, they have a strong potential in achieving significant performance improvement over the existing query methods by other data structures.

In order to do experimental comparisons with existing methods, we give a brief description of orthogonal range range queries by grid files and multipaging as follows. Prior work done by others exists for both.

Orthogonal Range Queries by Grid Files

The orthogonal range query algorithm by grid files is shown in Figure 6.5. The ranges are given by the lower and upper corner coordinates.

```
void GridFileRangeQuery(DATA lower, DATA upper)
{
  int lowind[k], upind[k];
  DataPage *celllist;
  SearchScale(lower, upper, lowind, upind); //convert into index vectors;
  GenCellList(lowind, upind, cellist); //generate a list of page addresses;
  DoRangeQuery(lower, upper, cellist);      //pages are looked up;
}
```

Figure 6.5: Orthogonal Range Query using Grid File or Multipaging

First, using index scales stored in RAM, vectors of lower and upper corner coordinates of the range query are converted into index vectors, `lowind` and `upind`, by procedure the `SearchScale` (not shown). Then these index vectors are used by the grid directory to generate a list of page addresses in procedure the `GenCellList` (not shown). This requires `k` loops, from `lowind` to `upind` in each dimension, to convert all the page cells into a set of pointers to data pages. Duplicate pointers to the same data page are removed. The above process needs to access the grid directory usually on secondary storage. Finally, the list of data pages are accessed. The cost in terms of disk accesses is loosely bounded by

$$\prod_{i=1}^{k}(upind[i] - lowind[i] + 1) \tag{6.1}$$

and will be examined in the experimental section.

Orthogonal Range Queries by Multipaging

Range queries by multipaging share the first two stages of `SearchScale` and `GenCellList` as those of grid files. The only difference is that `GenCellList` can be performed in RAM and thus requires no time in our model, where only the number of disk accesses count. But the `DoRangeQuery` is more complex than that of grid files because a data page may have pointers to its overflow pages. The pseudo-code of `DoRangeQuery` for multipaging is given in Figure 6.6. The cost of multipaging when distributions are close to uniform is also bounded by Equation 6.1. However, for pathological distributions, overflowing pages could lead to a visit of the entire data file.

6.2 Experimental Comparisons with Multikey File Structures

In the following experiments, we address costs against four parameters that may affect performance. These four parameters are file size, distribution, dimensionality and query selectivity. Costs studied include the number of disk page accesses, access time and storage costs. The file structures we are going to compare to are all suitable for multidimensional data, including direct access methods (multipaging and grid files) and logarithmic file structures (R-trees/R*-trees and their derivative X-trees).

6.2.1 Costs

The number of page accesses to the secondary storage are a straightforward assessment of query costs. But in some situations, not all page accesses require the same amount of time. Counting numbers of page accesses becomes insufficient and may

```
void DoRangeQuery(DATA lower, DATA upper, DataPage *cellHead)
{
  DataPage *p= cellHead; //p is head of the linklist of pages
  while (p is not null) {
    DoRangeQueryOnAPage( lower, upper, p); //process one page at a time
    p = p-> next;  //p points to the next data page member of the linklist
  }
}

void DoRangeQueryOnAPage(DATA lower, DATA upper, DataPage *p)
{
  DataPage *q;
  //load data page p into RAM and do a sequential search on the page:
  sequentialSearch(lower, upper, p);

  //if p has an overflow page chain list, search them as well:
  q= p-> linkto;  //q point to the first member of the overflow page list
  while ( q is not null) {
    if (!q-> visited())        //if the page has not been processed
      DoRangeQueryOnAPage(q);  //do range query on the overflow page
    q= q-> linkto;             //goto the next page of the overflow list
  }
}
```

Figure 6.6: Page Searching in Orthogonal Range Query using Multipaging

even be misleading. In the case of X-trees, supernodes are groups of nodes which are accessed sequentially rather than directly. So for X-trees, we cite the access times. The times are not measured directly because of operating system interference as well as the existence of memory buffers and disk buffers. Instead, time is calculated using the parameters of our disk, a Seagate Hawk 2XL Ultra SCSI 3 with an average seek time and rotation time of 14.54 msec, and a read time (for 1KB of data) of 168μsec. For sequential reads, the random seek time is omitted.

Another cost to be considered is storage. Tries compress data, as opposed to all the other methods we have discussed in the context of this thesis. We will look at the trie compression rate in subsection 6.2.8. On the other hand, grid files are significantly dependent on the correlation among multidimensional distributed data, and which in turn affects the amount of storage for grid directories. Multipaging also degenerates for exact match queries when there is a strong correlation among the data. Section 6.2.7 investigates this based on new models in characterizing distributions.

6.2.2 Data File and Algorithm Implementation

Data used in the experiments are from several sources. We use uniform random synthetic data, synthetic data with nonuniform distributions, as well as data from contour maps [EMR] and the U.S. Census TIGER data.

All trie, multipaging and grid file algorithms are implemented by the author. The R*- and X-tree implementations are from the X-tree author Berchtold [BKK96]. These are the implementations they wrote, and were run on their machine as well. The block size of files on secondary storage was set at 4096 bytes for all tests unless specified otherwise.

In all cases, results shown are averages of 50 to 100 runs where applicable. All queries are on point data. Tries on shaped data are investigated and compared with an analytical model of R*-trees [TS96], which is beyond the scope of this chapter.

6.2.3 Parameters

There are four parameters that may affect the query cost: file size, dimensionality, query selectivity and distribution.

The cost of tries, R*- and X-trees are logarithmic to file sizes. Multipaging and grid files are direct access methods and thus they are independent of file size in general. Section 6.2.4 shows that direct access methods have greater speed in exact match queries. But for range queries, tries outperform all others.

The increase of dimensionality degenerates the R*-tree performance. One reason is due to the overlap of directories which increases rapidly with the growing dimensionality of the data. The X-tree is designed to improve the R*-tree at high dimensions using a hybrid organization of partial hierarchy and partial linear organizations. Tries do not have the overlapping problem as that of the R-tree variants. In section 6.2.6, we demonstrate that tries outperform R*- and X-tree for both exact and range queries in various dimensions.

Selectivity is defined as the ratio of records retrieved by a query to the total number of records in the file. A range query is an example of high selectivity in most cases. Section 6.2.5 will show that tries outperform all other data structures except at very low selectivities, such as an exact match query.

The distribution of data in multidimensions, i.e., the way in which data correlate across dimensions, significantly affects the performance of multipaging and grid files. Section 6.2.7 shows that tries are not affected entirely by poor distributions in terms of speed and storage costs.

6.2.4 Speed versus File Size

Figure 6.7, 6.8, and 6.9 show how the cost of exact match queries varies with the file size. Figure 6.7 shows the 2D result, and Figure 6.8 displays the high 16D result. Tries are compared with X-trees, and R*-trees in 16D. As X-trees contain supernodes which use subsequent sequential searches among its directory search, a complete analysis should also include time delay measures. Figure 6.9 shows the equivalent elapse time spent on disk accesses.

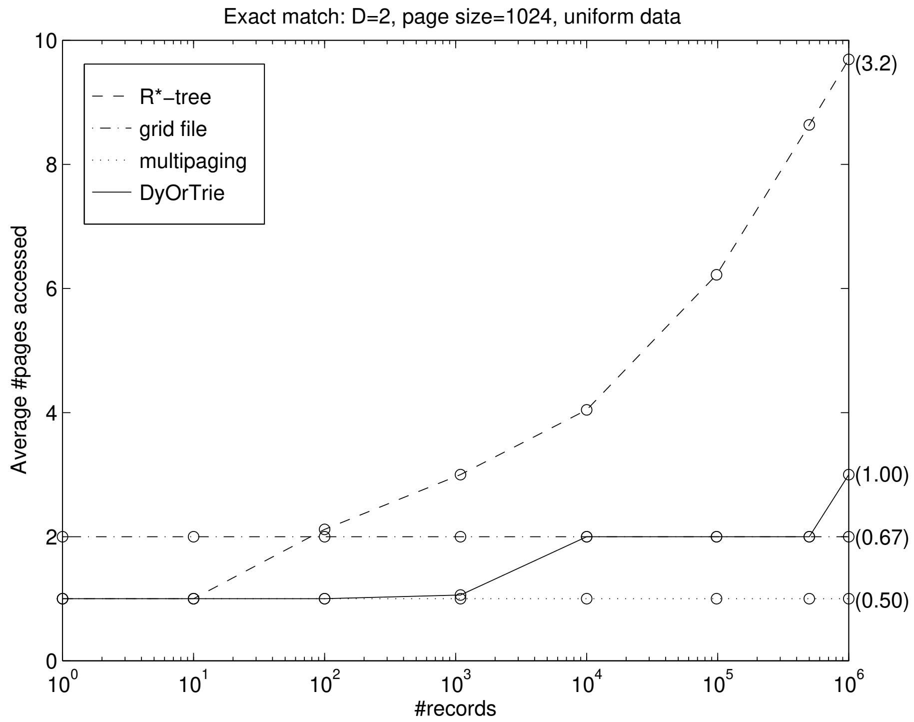


Figure 6.7: Page Accesses vs. File Size, 2D, Exact Match

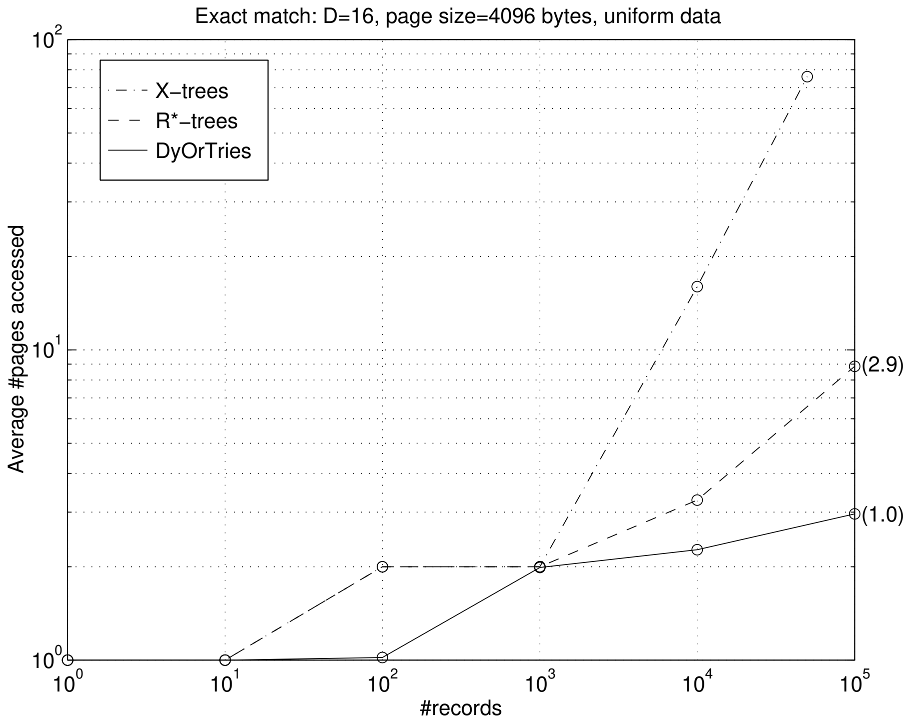


Figure 6.8: Page Accesses vs. File Size, 16D, Exact Match

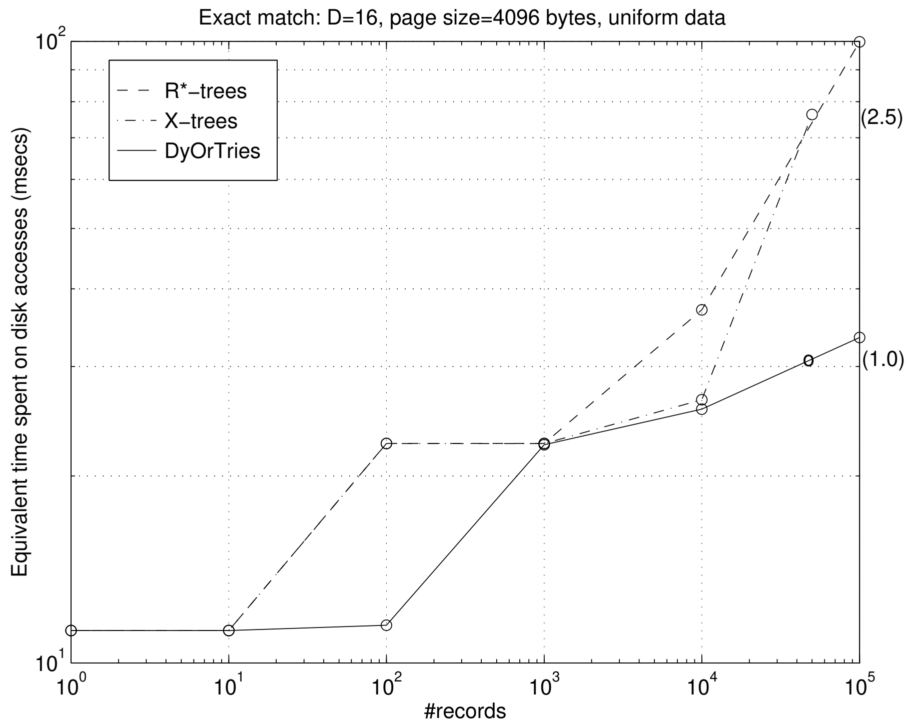


Figure 6.9: Equivalent Access Times vs. File Size, 16D, Exact Match

From these results, when the number of records exceeds a million, tries are slower than multipaging by a factor of 3 and grid files by 1.5, simply because tries are logarithmic and not direct access structures. However, comparing with tree-like structures, tries outperform R*-trees by factors of 3.2 at 10^6 records in 2D. In 16D, tries are 2.9 times more efficient in page accesses using 10^5 records, and outperform X-trees at 5×10^4 records by 2.5 when considering the total time spent on disk accesses.

When file sizes increase, the trend is that trie costs increase logarithmicly, however, tries supercede R*-trees and X-trees by greater degrees in exact match queries. This is because R*-trees and X-trees performance deteriorates when files are bigger in high dimensions. It is difficult for R*-trees to find overlap-free partitions when files are very big, and thus they have more overlapped directory pages to be searched. The X-tree tends to fail building hybrid hierarchy structures when there are an increasing number of records in a file. Instead, the whole structure generates more and larger supernodes, especially near the root. These supernodes near the root are more likely to be visited during a query. This results in a sequential search of most of the file instead of a tree search. In fact, it is exactly this reason why in Figure 6.8, when the number of records exceeds 10^3, R*-trees have an improved ability over X-trees in terms of the number of page accesses; R*-trees have overlapped directory pages to be visited, while X-trees have to search some supernodes* which further increase costs. On the other hand, in terms of time spent on these disk accesses, X-trees do sequential searches on those supernodes, which save considerable amount of access time to secondary storage. This is why in Figure 6.9, the time delay search costs of X-trees are lower or no worse than that of R*-trees when files are larger than 10^3 records in 16D.

For orthogonal range queries, Figures 6.10, 6.11 and 6.12 give the same set of comparisons at 0.2 selectivity for 2D and 16D respectively. Cost increases linearly, along with the number of records, briefly for all structures, both in 2D and 16D, when there are enough records. Tries are superior to all other structures. In 2D, tries outperform R*-trees by a factor up to 5.3, and improve performance over grid files and multipaging by a factor of 1.9 and 2.1 respectively at 10^6 records. In 16D , tries outdistance both R*-trees and X-trees up to a factor of 2.2, at 10^5 records.

The major reason for the enhanced performance of tries over other structures is on account to the storage compression they achieve. In fact, tries compress data while other methods add storage overhead which in summary accounts for the result. We look further at the compression rate of tries in section 6.2.8.

6.2.5 Speed versus Selectivity

In this section, we give two groups of figures for the query cost versus the selectivity for the contour and TIGER data, one in 2D and the other in 16D. The selectivity plays a principle role in the query cost for all data structures. The higher the selectivity is, the higher the query cost for any structure. Figures 6.13, 6.14 and 6.15 show that tries outperform all other methods, surpassing R*-trees, grid files and multipaging by factors 5.5, 3.1, and 2.1 respectively at 100% selectivity in 2D.

*One supernode contains a number of regular nodes, thus counts for more than a page.

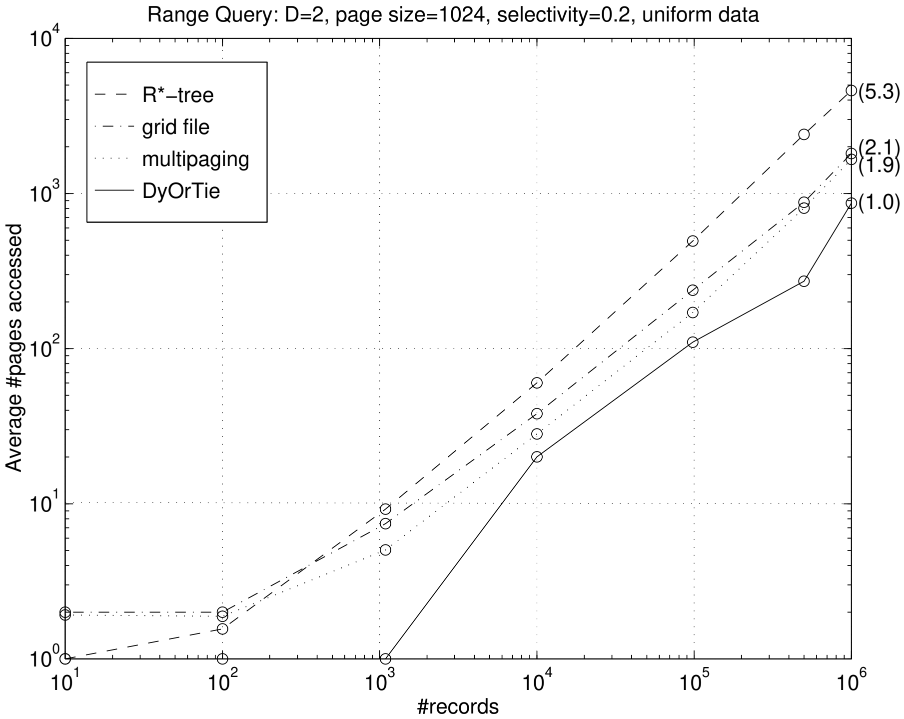


Figure 6.10: Page Accesses vs. File Size, 2D, Range Query

In Figure 6.13 for 2D, all curves are linear except multipaging. The cost of multipaging increases quickly at low-end selectivities. This is due to the fact that the experimental data used here are non-uniformly. A brief explanation follows. When the selectivity is low, the probability that pages in overflow chains are outside the query range is high, and consequently increases the search cost. When the selectivity increases, the probability that those pages holding overflow records are located outside the query range decreases.

In high dimensions, tries retain an advantage over both R*-trees and X-trees at any selectivity. This again is due to the compression property of tries. The factor of improvement now appears in the range 2.4 to 2.7 at 100% selectivity. Note that these curves are different from that in 2D — they are not linear. All costs increase rapidly at low selectivities, e.g., at 10% selectivity for contour map data, costs of the R*-tree and X-tree are about 30% of those at 100% selectivity, and those of the trie are approximately 80% of the 100% selectivity. This is the so-called “curse of

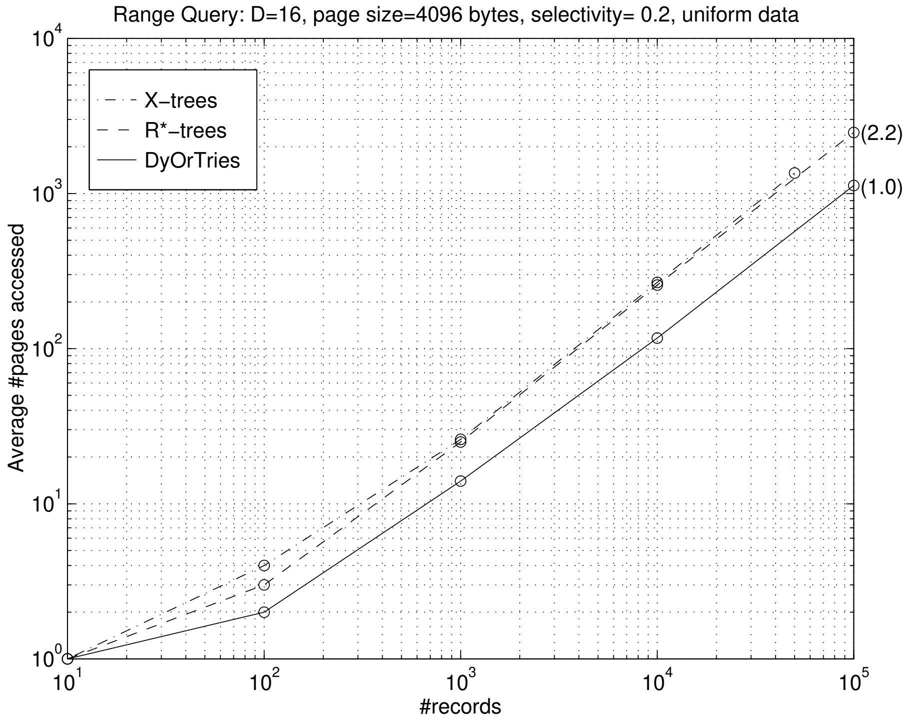


Figure 6.11: Page Accesses vs. File Size, 16D, Range Query

dimensionality". For simplicity, assume the query range is a hypercube. With the selectivity at 10%, the side length in a dimension is $0.1^{1/16} = 0.87$ in a data space of a 16D unit hypercube $[0, 1)^{16}$. This is larger than three-fourth of the space in that dimension. We know that tries always partition the space into two half subspaces. This means that at least the first $2 \times 16 + 1 = 33$ levels of the trie nodes need to be visited. A complete trie with 33 levels of nodes can hold up to $2^{32} = 4$ billion records. Thus, with 6×10^4 records in the contour map file, 33 levels of nodes certainly means most pages of the trie. For the R*-tree and the X-tree, the data space is split only once in a number of dimensions. It is not split at all in the remaining dimensions, and thus the bounding boxes of the pages include almost the whole extension of the data space in these dimensions. For example, if a 16D data space has been split exactly once in each dimension, it would require $2^{16} = 65536$ data pages, and we have only less than 2000 pages for both the R*-tree and X-tree. On the other hand, it is intuitively clear that a query with side length of 0.87, must intersect with every

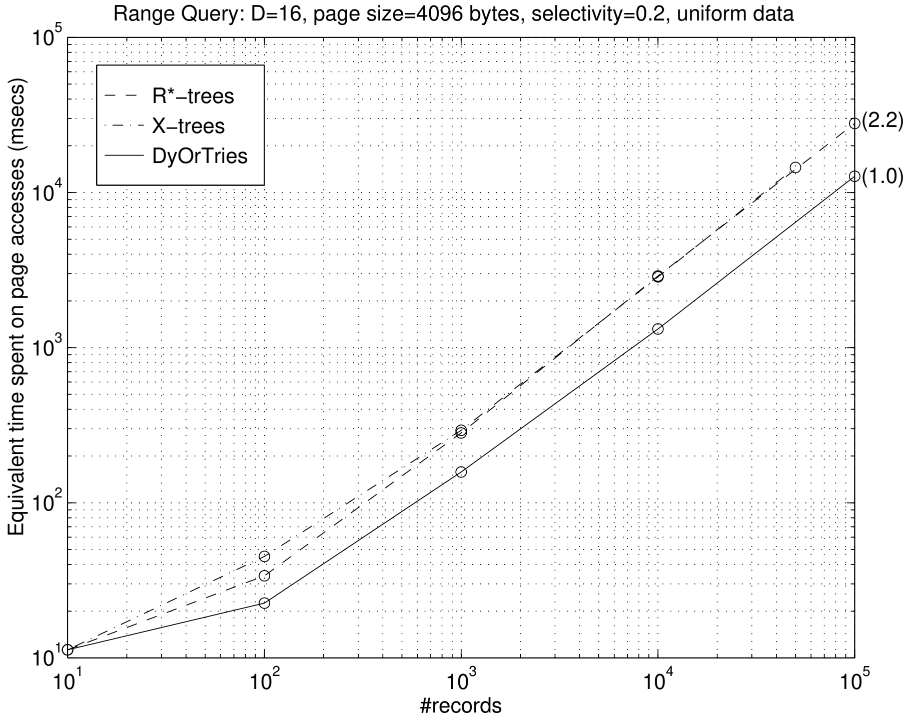


Figure 6.12: Equivalent Access Times vs. File Size, 16D, Range Query

bounding box having at least side length 0.13 in each dimension. Thus the access probability of pages in the R*-tree and the X-tree are high even when the selectivity is not particularly high; it is this which contributes to the high cost of the queries at 10% selectivity. A more accurate analysis for the curse of dimensionality on X-trees (and R*-trees) for range queries can be found in the literature [BBK98].

6.2.6 Speed versus Dimension

Poor R*-tree behavior at higher dimensions has provided a motivation for the introduction of X-trees. We move on to see how tries behave in higher dimensions. We make two sets of experiments on tries, R*-trees and X-trees, one for exact match query and the other for high selectivity range queries. In each set, we give both the average number of disk accesses and the equivalent time spent on these page accesses.

Figures 6.16 and 6.18 show that tries remain better than R-trees and X-trees by factors similar to the experimental results in the previous subsections for exact

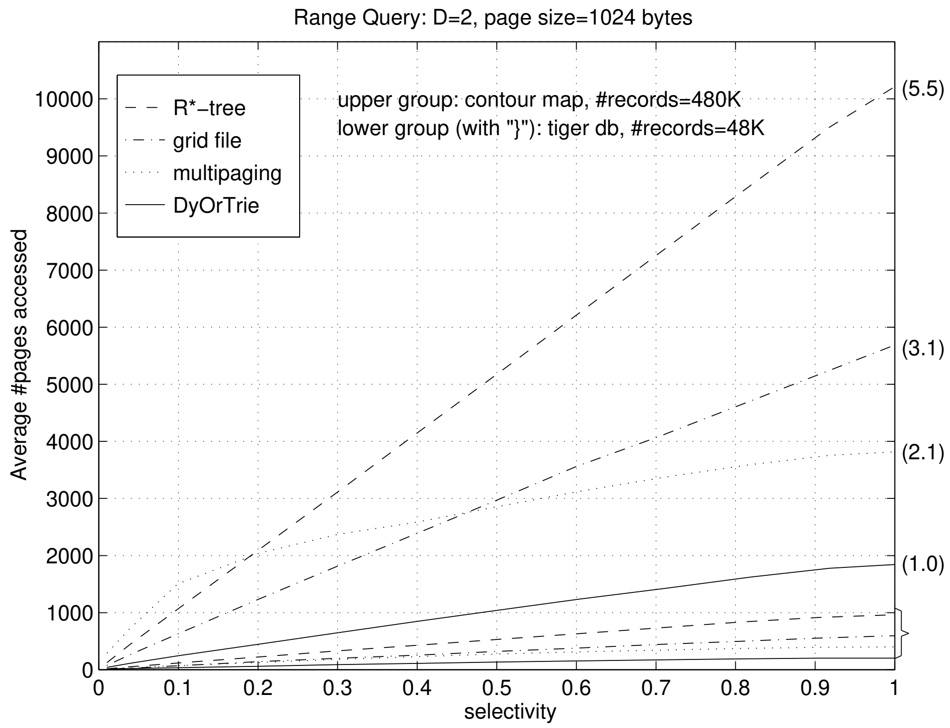


Figure 6.13: Page Accesses vs. Selectivity, 2D

matches and range queries with the selectivity fixed at 20%. Figures 6.17 and 6.19 again confirm superior trie performance when measuring times spent on disk accesses.

These experiments are done by fixing the file size at 6.4MB. The higher the dimensionality, the fewer the number of records actually stored. This explains why tries have slightly better performance for exact matches in higher dimensions. On the other hand, the key length increases with dimension. Moreover, for range queries with fixed selectivity, the higher dimensionality indicates that in each dimension the range to be searched within that attribute space becomes larger on average. This is the reason that costs for range queries by tries increase with dimensionality. However, tries remain cost effective over both R*-tress and X-trees due to their compactness.

The trie curves in Figure 6.18 and 6.19 have the same shape as those of R*/X-trees. All of them increase rapidly for low dimensionality, say less than 6 to 8 dimensions. The reason is quite similar to what is explained at the end of section 6.2.5, the so-called "curse of dimensionality". Assuming the query range is a hypercube, when

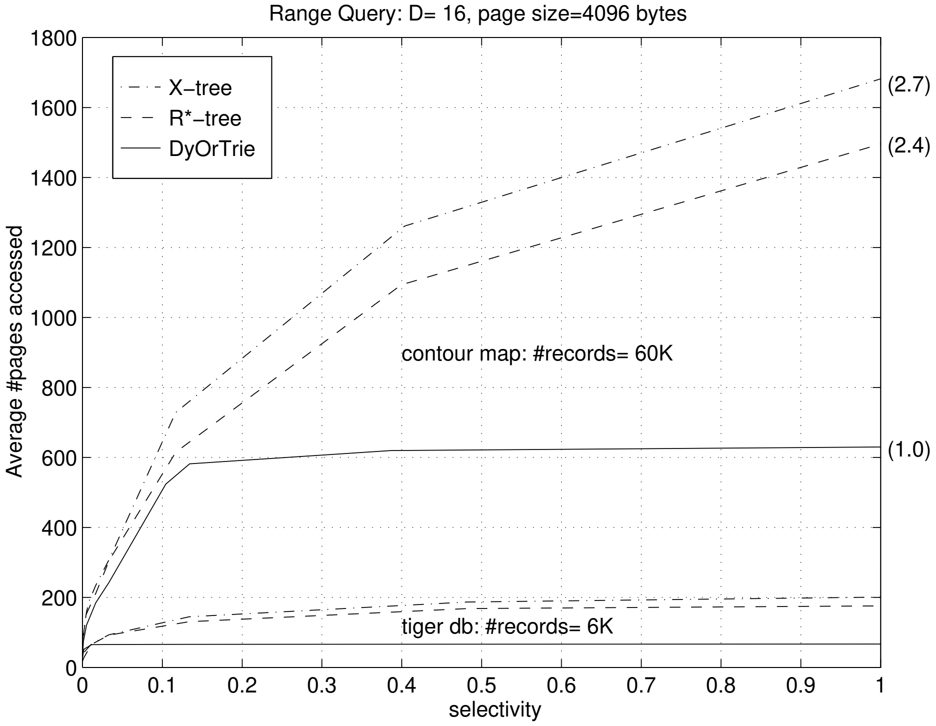


Figure 6.14: Page Accesses vs. Selectivity, 16D

the dimensionality increases, the query side length increases non-linearly. At 20% selectivity, the side length is 0.45 in 2D, 0.67 in 4D, 0.82 in 8D, and 0.90 in 16D. For X-trees (and R*-trees), the percentage of accessed pages quickly approaches the 100%-mark as the dimensionality reaches 10, and when database size and selectivity are fixed, according to Berchtold [BBK98]. This roughly explains why when dimensions exceed 8, the curves of the X- and R*-trees are almost flat — the queries retrieve almost all pages in the files while the database size (file size) is fixed.

6.2.7 Speed and Storage Cost versus Data Distribution

The distribution of data in the multidimensional space can seriously affect the performance of grid files and multipaging in storage space and search time respectively. Thus we must find ways to characterize and quantify these distributions.

The usual statistical measures of correlation are not suitable indicators of the distribution as they depend on the order of the data. Thus, attribute Y may be a

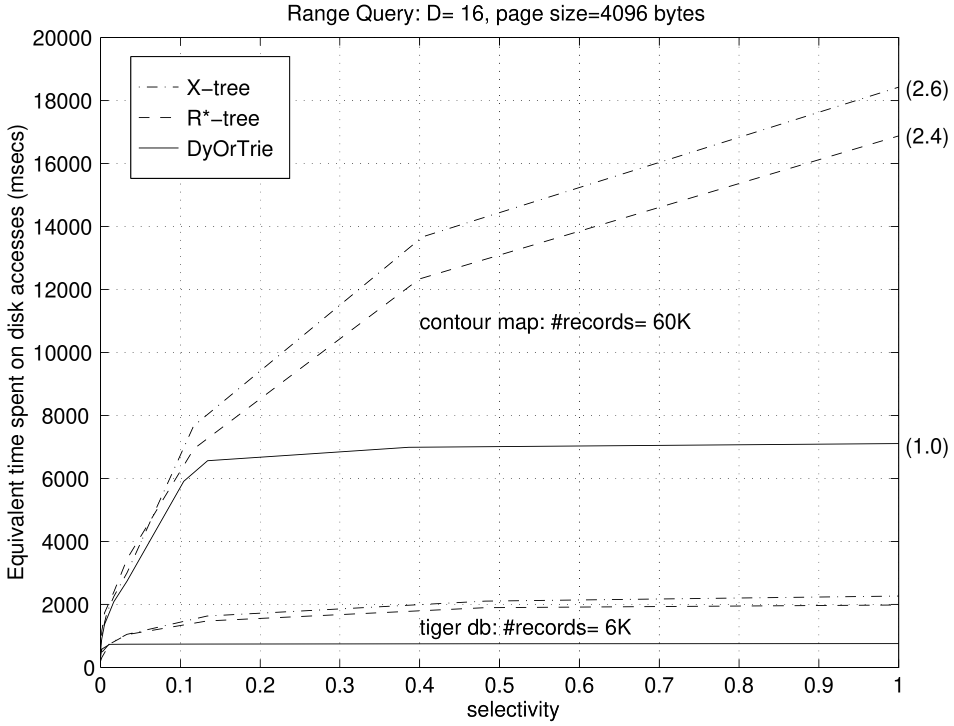


Figure 6.15: Equivalent Access Times vs. Selectivity, 16D

function of attribute X, $y = f(x)$, but the statistical correlation between X and Y will be quite different depending on whether f is a straight line or f has points which are distributed uniformly across the plane.

We turn to information theory and find that the information-theoretic correlation, which we will from now refer to as *correlation* for simplicity, provides an excellent measure for the behavior of multipaging. Similarly, the mutual information, which we will from now on refer to as *information*, is a good indicator for grid files performance.

For random variables $X, Y, ..., Z$ with probability $p(x_i, y_j, ..., z_k)$, and $X = x_i$, $Y = y_j$ ... $Z = z_k$, the information is

$$I_{XY...Z} = \sum_{ij...k} p(x_i, y_j, ..., z_k) \log p(x_i, y_j, ..., z_k)$$

The correlation is given by the expression

$$I_{XY...Z} - I_X - I_Y - ... - I_Z$$

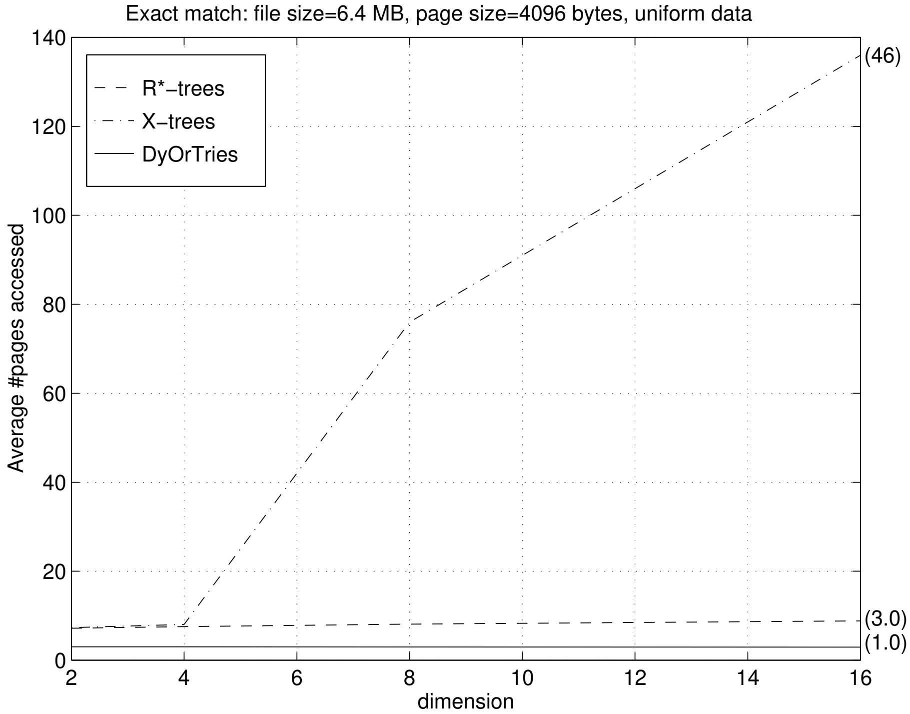


Figure 6.16: Exact Match: Page Accesses vs. Dimension

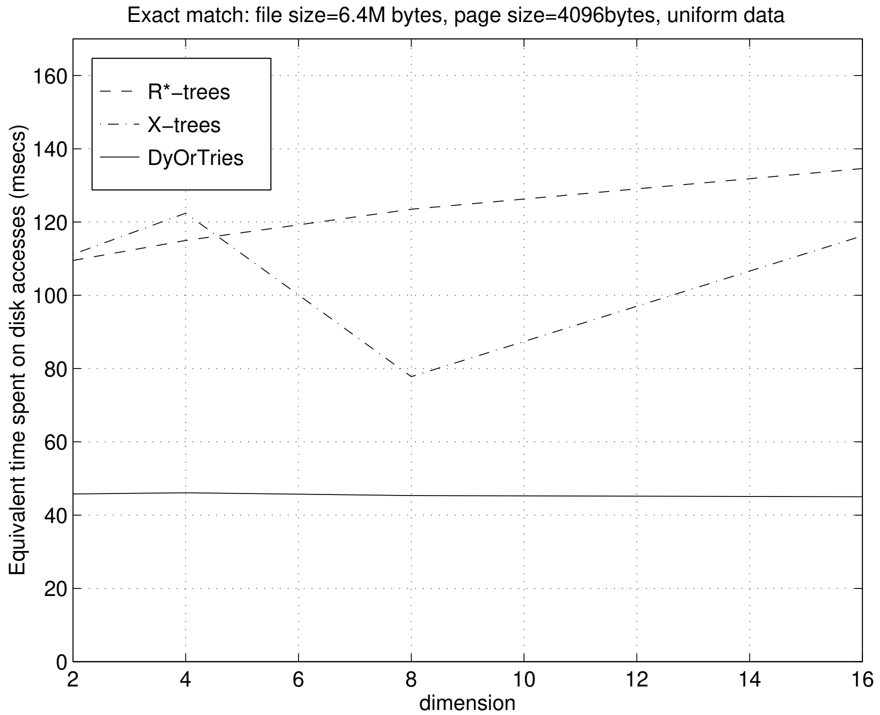


Figure 6.17: Exact Match: Access Time vs. Dimension

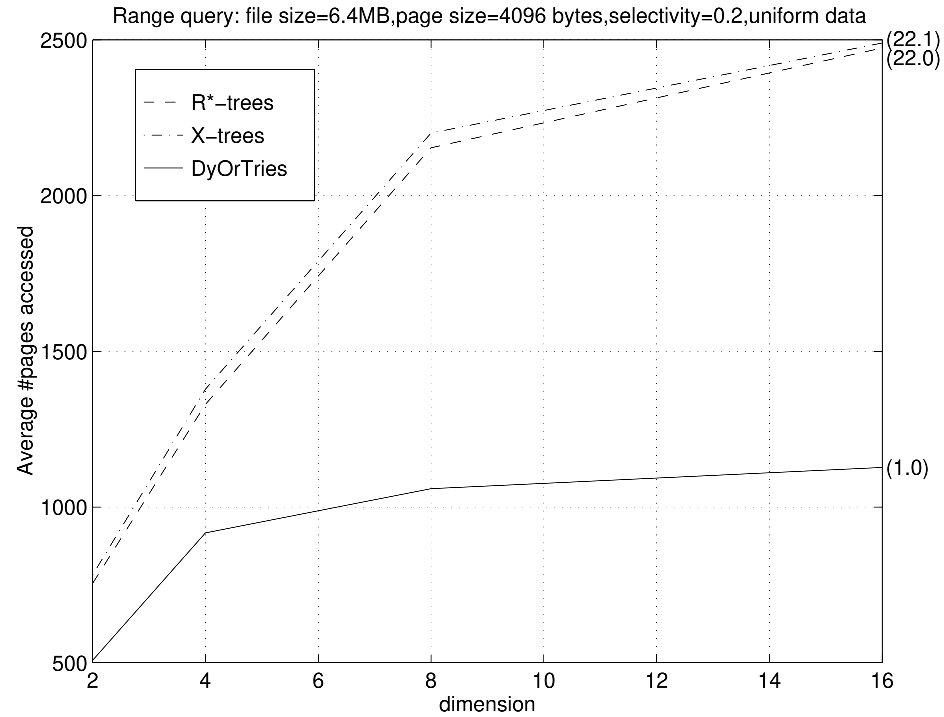


Figure 6.18: Range Query: Page Accesses vs. Dimension

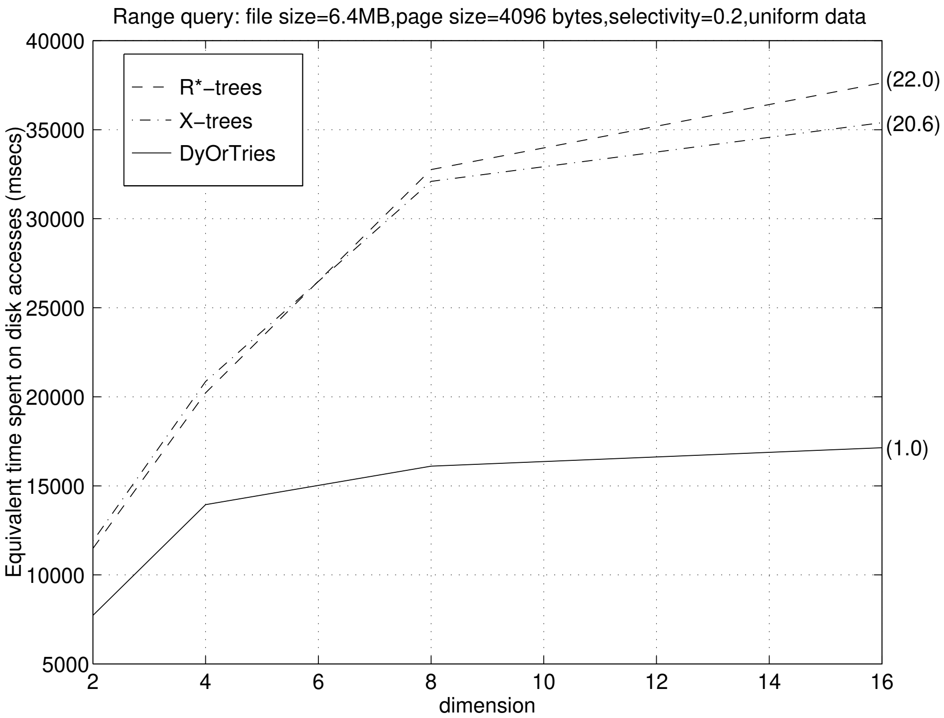


Figure 6.19: Range Query: Access Times vs. Dimension

The correlation vanishes only when the distribution of $X, Y, ..., Z$ is the product of the distributions of the individual fields. This is referred to as a *Cartesian product* distribution. The correlation reaches its maximum when every pair of variables forms a function. A Cartesian product distribution is ideal for multipaging. On the other hand, functional dependencies produce pathological distributions. An example of such is a circular distribution, which also has a high correlation. Therefore, we consider the correlation as a good predictor of multipaging performance for access times.

The information is minimal when the probabilities are all equal and the distribution is uniform†. The information is maximal when all probabilities but one are zero, and the distribution is sharp. This happens exactly when the grid file directory has to waste the most space. Thus we can use the information to predict grid file directory space costs.

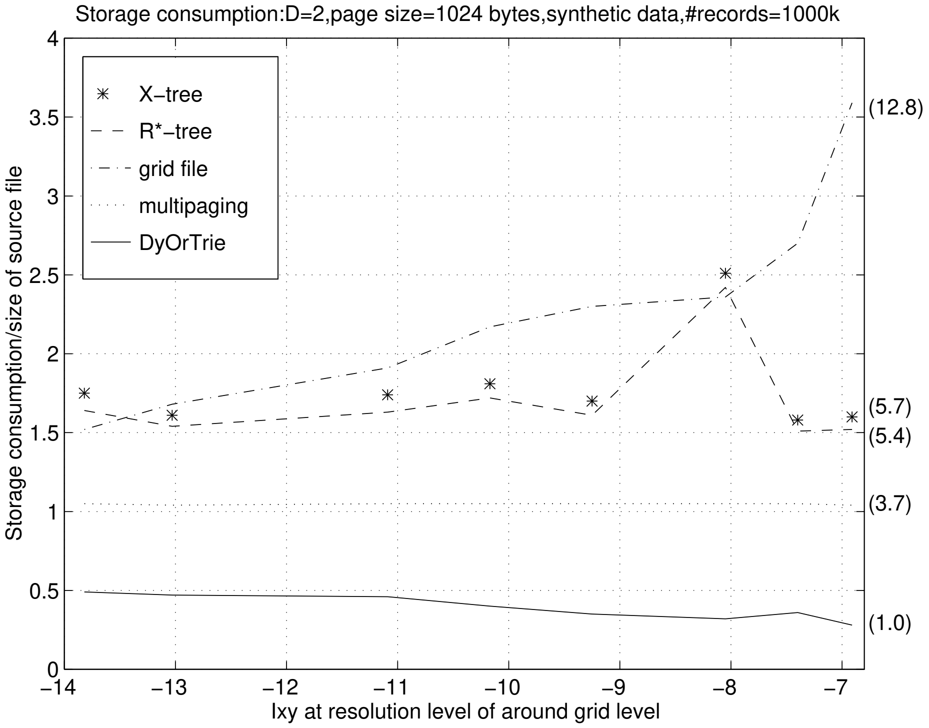


Figure 6.20: Storage Cost vs. Distribution

†Note that we are using the negative of what is normally called "information" and which is maximum for uniform distributions. This is to make the consistency that low values always correspond to good performance.

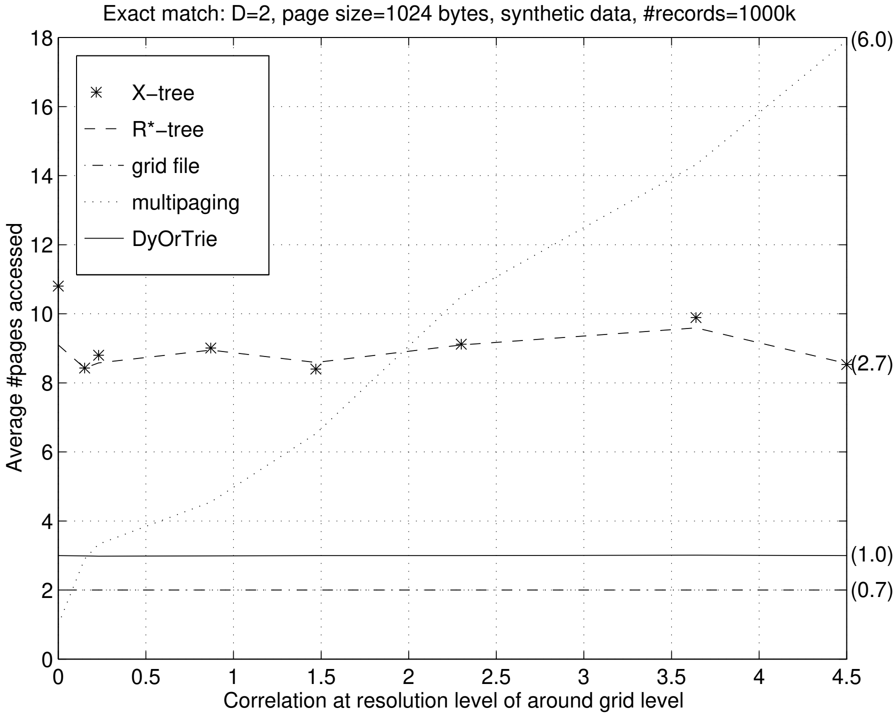


Figure 6.21: Exact Match: Access Cost vs. Distribution

It can be misleading if we measure information and correlation based on each individual record location. For instance, Y might be a function of X and the correlation is at a maximum, and thus the predicated performance of multipaging is very poor. But this function could be such that every page of a multipaging space is uniformly occupied, which results in optimal performance. As we count costs in terms of disk pages/blocks on secondary storage, we measure the probabilities for *page* locations, i.e., aggregated over pagefuls of records, in the following experiments.

To test the effects of data distribution, we use a 2D synthetic data with distributions ranging from Cartesian product to functional straight lines, each with 10^6 records and a page capacity of 1024 bytes.

A poor data distribution affects the storage of grid files, but does not bother tries or multipaging. Thus, we use information as a measure of the distribution in Figure 6.20 [‡]. Multipaging requires a small storage overhead. The trie gains

‡The correlations as defined are negative, and to help intuition, we double the negative.

substantial compression due to the overlap of paths near roots. The grid file starts with a 50% overhead, and it rises to 350% as the distribution quality declines. This is mainly due to the expansion of the grid directory. The R*- and X-trees have overheads that fall in the range of 130% to 300%. But apparently they do not depend on the information.

Figure 6.21 shows the effect of the data distribution on access costs. Multipaging is the only data structure which shows a dependency, and thus the correlation is used as a measure of the data distribution[§]. For a good distribution (correlation is 0), multipaging demonstrates the best performance at one access, followed by grid files and tries, at a constant two and three accesses respectively, at this particular number of keys.

R-trees and X-trees retain higher access costs than the other methods, but are unaffected by the data distribution in general. R-trees have higher access costs because of the overlaps of rectangles causing more pages than necessary to be searched. The costs of X-trees are very similar to R-trees since we count a super node access as equivalent to many regular page accesses, however, recall that the linear accesses can in fact reduce the costs in time to those less than that of R-trees.

The cost of multipaging degenerates quickly, while others are remain stable for different distributions.

6.2.8 Data Compression versus Storage Overhead

One of the main reasons that tries obtain an advantage over other methods for exact match and range queries rests on the fact that it compresses data. We define the *file compression rate* as the ratio of the size of the compressed file to the size of the original data file. Figure 5.1 in chapter 5 shows that the file compression rate becomes less than 0.1, i.e., over 90% of the file size has been reduced, for large files in 1D. In addition, the data compression also depends on the size of a data record (if we assume that every attribute of a record consumes the same space, then the size of a

[§]The absolute value ranges from 0 to a maximum, which depends on the size of the data set. The range is not significant.

data record is determined by its dimensionality as well). Figure 6.22 shows how the trie compression rate varies depending on the record size (dimensions). The curves correspond to file sizes of 1.28 and 6.4 MB of uniform data. From the Figures we see that an inverse relationship exists between record size and file compression rate. This follows, since if we fix the size of the data file, larger record size implies fewer records, and thus less common paths to be shared near the trie root.

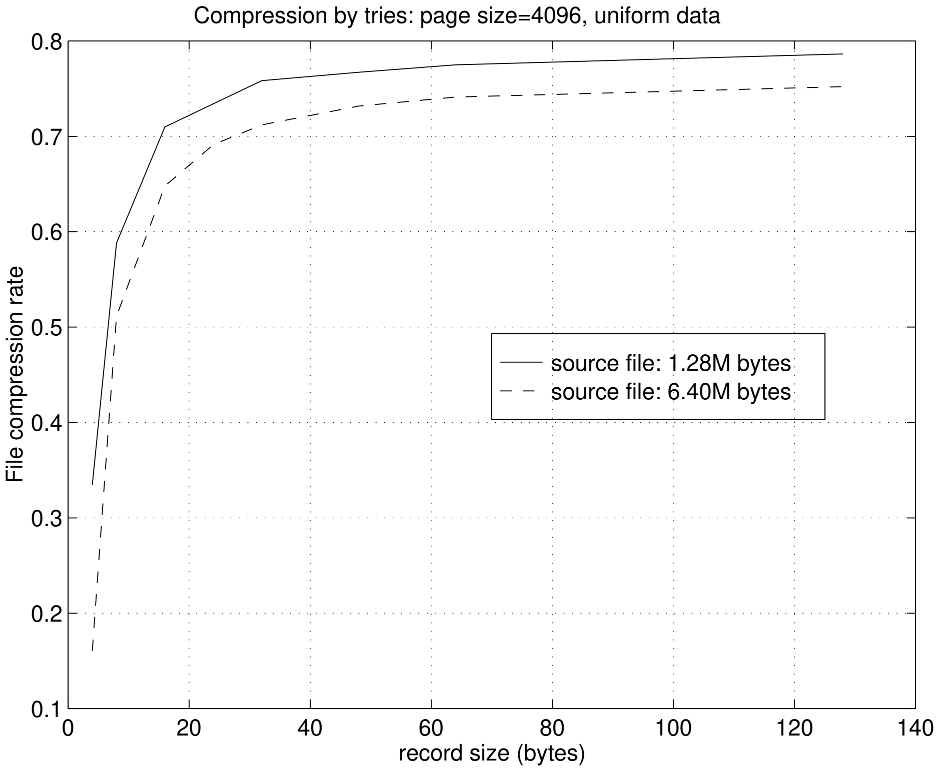


Figure 6.22: Trie Compression vs. Record Size (Dimension)

6.3 Summary

This chapter extends tries for general queries other than text searching and spatial data retrieval. We use tries for multidimensional exact match queries and orthogonal range queries. Orthogonal range queries are applicable to spatial data such as maps, but are strictly more general — the attributes queried need not be even numeric.

We address access cost and storage cost in terms of file size, record size, selectivity, dimensionality and distribution. Our experimental results show that tries are superior to all other structures (both direct access methods represented by multipaging and grid files, and tree structures including B-trees, R-trees/R*-trees and X-trees) in queries returning more than a few records. Moreover, tries are competitive with direct access method in exact match queries.

Unlike multipaging which can deteriorate on search cost with pathological distributions, and grid files which can waste storage space on grid directories for bad distributions, tries are unaffected by the data distribution.

For high dimensionality, tries are still up to 2 and 3 times better than R*- and X-trees which were invented for high dimensional spatial data.

Tries always compress data, resulting in savings for storage cost. This is in contrast to grid files which consume large amounts of space for uneven distributions, and R*-trees and X-trees that use at least twice as much space as the sources. However, it is noted that multipaging does indeed have small space overhead as in the case of tries.

In fact, trie compression is one of the two major reasons that tries achieve better speed performance for queries returning more than a few records. This combines with the variable-resolution property, enable tries to be a superior method for multidimensional general data. Our task is to extend them to more general queries and operations. In the next chapter, we will apply tries to relational join operations.

Chapter 7

Relational Joins by Tries

We have surveyed join algorithms in Section 3.3. Overall, to the best of our knowledge, there has been no work on join algorithms that takes advantage of existing data structures, i.e., no extra indexes, for better join performance, and yet achieving spatial compression at the same time.

In this chapter, we are working on data sets organized by tries, not relations directly. With the built-in properties of order preservation on keys, storage compression and variable-resolution, tries have benefits not only in indexing large text data and spatial data retrieval, but also in general database queries such as exact matches and orthogonal range queries. In this chapter we extend them for even broader use to join processing.

We first assume that the join attributes do not have duplicates, i.e., they are keys. In this situation, joins are in fact set operations. Our inputs are two tries built on the join attributes and the join result is a new trie. Based on the unique characteristics of the data structure, there are a number of advantages that joins by tries can achieve.

1. The variable-resolution structure of the trie supports less than one pass search cost in terms of the trie size. This happens in practice by making decisions at low resolutions near the root as to whether a particular subtrie needs to be visited or not for various unary queries and binary joins.
2. The worst case cost of the trie join (TJ) algorithm happens when ranges of join

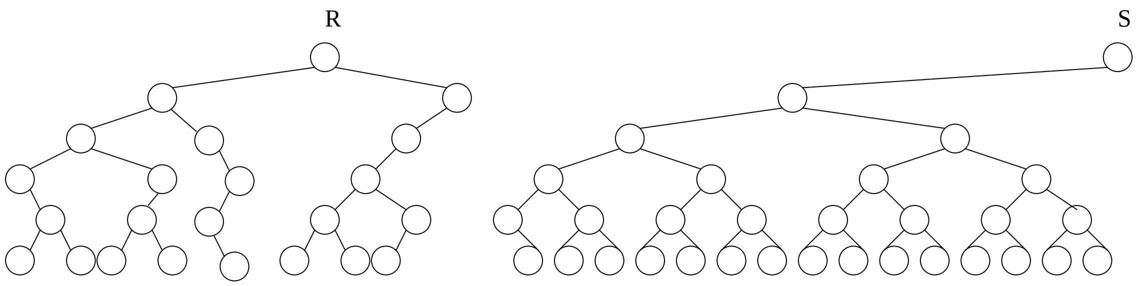
attribute values are totally overlapped, and that the output values organized by the trie are as large as the input tries. In this case, complete traversals of both input tries have to be done. But due to the compression tries achieve, the cost is still less than one pass of the original data sets without trie compressions. Therefore the worst case join cost is linear with a leading coefficient which is less unity. Note that we exclude the cost of the input trie construction and sorting.

3. Hash join methods do not preserve order, which is indeed maintained by tries. This property allows tries to be easily extended for efficient non-equijoins. Among them, union (*union*), symmetric difference (*xor*), and difference join (*minus*) operators will be considered in further discussions.

4. When there exist indexes on joining attributes, which is the situation in very large databases, the queries can make use of these indexes to achieve better performance. Our input tries have built-in indexes on joining attributes.

7.1 Join Algorithms by Tries

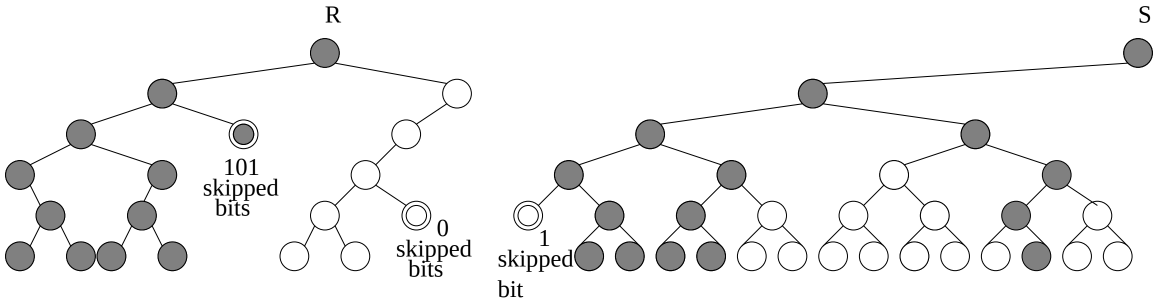
Binary tries for data sets R and S in section 3.3.2 Figure 3.5(a) are given in Figure 7.1(a). The two sets in binary format are [00010, 00011, 00100, 00101, 01101, 10000, 10001, 10010], and $[00001, 00010, ..., 01111]$ respectively. Figure 7.1(b) shows their corresponding OrTries. Our join algorithm performs synchronous, depth-first, post-order traversals of the tries. First, root nodes of R and S are visited, and the depth-first traversals lead to the first left leaf node of R and the second left leaf node of S shown in Figure 7.1(b), with paths *10, 11, 11, 01, 11*. The two nodes are compared by an *and* operator, *00 and 00*, resulting in *00*, a new leaf node. Then, the join visits the right siblings and performs *and* operation on them, which results in a leaf node *00* as well. Next, the post-order traversal returns to the parent node *11*. As it has both left and right children, the value *11* is made final. The post-order traversal returns back another level, to node *01*, and it is made final as well, as it only has a right child node *11*. The post-order traversals of both tries continue. When the second nodes



11
11 10
11 01 10
01 10 10 11
11 11 01 11 10
00 00 00 00 00 00 00 00

10
11
11 11
11 11 11 11
01 11 11 11 11 11 11 11
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

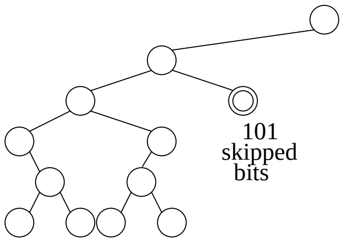
(a) (Full) Tries for Set R and S



11
11 10
11 00{101} 10
01 10 11
11 11 11 00{0}
00 00 00 00 00 00

10
11
11 11
11 11 11 11
00{1} 11 11 11 11 11 11 11
00 00 00 00 00 00 00 00 00 00 00 00 00 00

(b) OrTries for Set R and S (Shaded nodes are the common nodes visited during the join process.)



10
11
11 00{101}
01 10
11 11
00 00 00 00

(c) Resulting Trie after Natural Join of R and S

Figure 7.1: Joining Data Set R and S by Tries

on the third levels, *00*{*010*} and *11*, are reached, with the former a leaf node and the latter an internal node, and since the path *010* can also lead to a leaf on S, a match is found and the join results in a leaf node *00*{*010*}. Otherwise, the resulting node is empty, and we denote it by ϕ. If this happened, when returning to the parent node, the corresponding offspring, right child in this case, should have been truncated; the parent node should be changed to *10* from *11*. When the traversals of R and S return to the roots, the output node *10* is made final, and the join process is complete. The shaded nodes in Figure 7.1(b) give the common paths that are traversed by the join process. Other nodes have been skipped during the process. The output trie is given in Figure 7.1(c).

Now we summarize the trie join algorithm from the above example; it is a synchronous, depth-first post-order traversal of the two tries of input data sets. The traversal starts with the roots of the two tries, and moves down levels in tandem until there is no match possible in the subtries, or a leaf node is reached in one of them. At each step, the two nodes from two tries are compared. If the result is a match, the corresponding subtries are then visited, resulting in depth-first traversals. Otherwise, when returning to its parent node, this offspring should be removed in the output, and the subtries rooted at these two nodes need no further visits. When two nodes are first visited during the traversals, the logical *and* operation is performed on the 2-bit nodes. If the result is not *00* for internal nodes, it is a match. However, when any leaf node *00*{*remaining bits*} is involved, if the other node in comparison is also a leaf node and their remaining bits are a match, or if it is an internal node and a path from the node to a leaf representing the same *remaining bits*, then it is still considered as a match. In both cases, *00*{*remaining bits*} is the result of the join. Otherwise, the result is a mismatch and the path from its ancestor node must be truncated.

Table 7.1 is a binary table showing the result of the logical *and* operator on two nodes from the input tries, when they are *first* visited during the traversals. The first column and the first row give node values from the two input tries respectively. The result is *Rb*, an abbreviation for *remaining bits*, if the two inputs match, else empty, denoted by ϕ. When the result is empty, it means that the link must be removed from its parent node when the post-order traversal returns to the parent level. As a result,

	$00\{rb\}$	01	10	11
$00\{rb\}$	$00\{rb\}$ or ϕ	$00\{rb\}$ or ϕ	$00\{rb\}$ or ϕ	$00\{rb\}$ or ϕ
01	$00\{rb\}$ or ϕ	01	ϕ	01
10	$00\{rb\}$ or ϕ	ϕ	10	10
11	$00\{rb\}$ or ϕ	01	10	11

Table 7.1: *And* Operation of Two nodes in Natural Joins by Tries

Table 7.1 is modified to Table 7.2 to include all possible output values when the node is *revisited* after both its children have been visited in the post-order traversal.

	$00\{rb\}$	01	10	11
$00\{rb\}$	$00\{rb\}$ or ϕ	$00\{rb\}$ or ϕ	$00\{rb\}$ or ϕ	$00\{rb\}$ or ϕ
01	$00\{rb\}$ or ϕ	01 or ϕ	ϕ	01 or ϕ
10	$00\{rb\}$ or ϕ	ϕ	10 or ϕ	10 or ϕ
11	$00\{rb\}$ or ϕ	01 or ϕ	ϕ or 10	01, 10, 11 or ϕ

Table 7.2: Matching Two Nodes in Natural Joins by Tries

With B blocks of memory buffer, larger than three trie heights in pages, the paths from root to leaf during traversals on both input and output tries can always stay in RAM. Therefore, the cost of the join in terms of disk accesses consists only of the cost of traversals of both input tries, and thus is always no more than one pass of the input tries. Figure 7.2 gives the detailed natural join algorithm by tries (TJ).

As tries maintain key order, the TJ algorithm can be extended for efficient union joins (*or*), symmetric difference (*xor*) and difference (*minus*) operations as well. Table 7.3 shows the possible union join results on two input trie nodes. When no leaf nodes are involved, the union of two nodes is the simple logical *or* operation on two nodes. Otherwise, the result depends on what the remaining bits stored on a leaf node are. For example, *00*$\{$*0xxx*$\}$ *union 00*$\{$*1yyy*$\}$ results in *11* (and *00*$\{$*xxx*$\}$ *00*$\{$*yyy*$\}$ at its child level). Tables 7.4 and 7.5 give the *xor* and *minus* operations on two input nodes respectively.

Similarly to the natural join algorithm given in Figure 7.2, the union, symmetric difference and difference join operations on tries can be obtained. The union join

```
BOOL NaturalJoin(TrieNode *t1, TrieNode *t2)
{
  if( is_leaf(t1) or is_leaf(t2)) {
    if(NaturalJoinOneLeaf (t1, t2)) {
      write_a_leaf_node(t1); //output the leaf node to the resulting trie;
      return(TRUE);
    }
    else
      return(FALSE);
  }

  node= 0;  // set two bits for the node to be 00
  if (both t1 and t2 have left branches) {
    if( NaturalJoin (t1-> left, t2-> left))
       node += 2;      // set two bits for the node to be ‘1X’
  }
  if (both t1 and t2 have right branches)
    if (NaturalJoin (t1-> right, t2-> right))
       node += 1;      // set two bits for the node to be ‘X1’
  }
  if (node == 0)       // no match
    return( FALSE);

  write_a_node( node); //output the node to the resulting trie;
  return (TRUE);
}

BOOL NaturalJoinOneLeaf(TrieNode *t1, TrieNode *t2)
{
  if (is_leaf(t1) and is_leaf(t2))
    return (NaturalJoinLeaves( t1, t2));

  if (is_leaf(t2))
    return (NaturalJoinOneLeaf( t1, t2)); //t1 is the leaf node

  //there is only one path from node t2 that might match with t1
  if ( cur_bit(t1) matches with node t2 ) {
    if ( is_leaf_level(t2) )
      return (TRUE);
    else
      return( NaturalJoinOneLeaf( t1, t2-> left));
  }
  return (FALSE);
}

BOOL NaturalJoinLeaves( TrieNode *t1, TrieNode *t2)
{
  if (rest_bits_match( t1, t2))
    return (TRUE);
  return (FALSE);
}
```

Figure 7.2: Natural Join Algorithm by Tries

	$00\{rb\}$	01	10	11
$00\{rb\}$	$00\{rb\}$, 01, 10 or 11	01 or 11	10 or 11	11
01	01 or 11	01	11	11
10	10 or 11	11	10	11
11	11	11	11	11

Table 7.3: Matching Two Nodes in Union Join of Tries

	$00\{rb\}$	01	10	11
$00\{rb\}$	ϕ, 01, 10 or 11	ϕ, 01 or 11	ϕ, 10 or 11	01, 10 or 11
01	ϕ, 01 or 11	ϕ or 01	11	10 or 11
10	ϕ, 10 or 11	11	ϕ or 10	01 or 11
11	01,10 or 11	10 or 11	01 or 11	ϕ, 01, 10 or 11

Table 7.4: Matching Two Nodes in Symmetric Difference Join of Tries

	$00\{rb\}$	01	10	11
$00\{rb\}$	ϕ or $00\{rb\}$	ϕ or $00\{rb\}$	ϕ or $00\{rb\}$	ϕ or $00\{rb\}$
01	00 or 01	ϕ or 01	01	ϕ or 01
10	00 or 10	10	ϕ or 10	ϕ or 10
11	01, 10 or 11	10 or 11	01 or 11	ϕ, 01, 10 or 11

Table 7.5: Matching Two Nodes in Difference Join of Tries

algorithm is simpler than the other joins in the sense that there is no need to backtrack and truncate paths from leaves to the root. The point is that tries are capable of various binary join operations. In the following discussion, we focus on natural join as a representative of various join types which can be performed efficiently on trie structures.

The above discussion on trie joins assumes that join attributes have no duplicate values. In the presence of duplicates, there are two different situations:

1. There exist non-join attributes, which are not indexed by the tries. This leads to a scenario where trie leaf nodes contain pointers to more than one record stored on a separate file (or records share the same indexed attributes but have different remaining attributes stored on a trie leaf).

2. There exist non-join attributes which are indexed on k-d-tries. This implies that some trie node levels are non-join attribute bits and should be skipped during the join.

In the first case, the join algorithm remains the same except at the end of the join when leaves are reached; either pointers are used to point to all remaining non-join attributes corresponding to many records on files, or these records have to be stored on the leaf node themselves. We now partition the cost of joins into two parts: one for reading the input tries, and the other for writing the result to disk. Clearly, unlike other join algorithms, the complexity of the first part of the TJ algorithm does not increase, even though the size of the result increases. However, the cost of writing the output trie to disk increases.

In the second case, the TJ algorithm performs only on node levels representing the join attributes and skips others. The trie join becomes a many-forest join . The same is true for all other join algorithms. If duplicates can not be held within the memory buffer, the cost of the TJ algorithm will be quadratic.

7.2 Comparisons of TJ with Existing Join Algorithms

Some of the representative and state-of-the-art join algorithms include the sort-merge join (SMJ), the distributive join (DJ), the join by fragment algorithm and the bucket skip merge join (BSMJ). However, since the TJ algorithms use the built-in indexes on joining attributes, we assume the following for data sets in order to make fair comparisons.

1. For SMJ, both input sets are ordered according to join attributes. That is, only the merge phrase has to be carried out. Therefore we call it merge join (MJ) from now on.
2. For DJ, both input data sets are already in order, and the distribution table has been built.
3. For BSMJ, input sets are in order and the bucket tables have been built up.

It is pointed out and proven by Negri [NP91] that when input files are sorted according to join attributes, the above join methods have the following relations:

$$MJ > DJ > JF$$

where '>' means more efficient than. Therefore it is true that TJ is a more efficient join algorithm than MJ, DJ, JF and BSMJ if only we can show that TJ is more efficient than both MJ and BSMJ for data sets ordered on join attributes. Thus, our comparisons become focussed on TJ, MJ and BSMJ.

7.2.1 Best and Worst Case Analysis of TJ, MJ and BSMJ Algorithms

The efficiency of a join algorithm is best characterized by the amount of disk accesses required for the join processing. In this subsection, we analyze the best and worst case in terms of disk accesses for TJ, MJ and BSMJ.

The best case for the TJ scheme occurs when there is no match at the top trie levels, especially if the root pages do not match. In this instance, the total disk access cost of TJ is two pages. However, MJ and BSMJ may still need to load all the pages for a possible match as ranges for the two datasets may fully overlap with each other. Therefore, this may still be the worst case for the MJ and BSMJ schemes.

The best case for MJ and BSMJ occurs when all values of the join attributes of one set is less than (or greater than) that of the other set. In this case, MJ will examine all the pages of one dataset and only the first page of the other, i.e., $N_S + 1$ blocks, as the cursor moves only in one key set. BSMJ only needs to visit the bucket tables, and no pages in the data set are visited. In fact, one bucket table page of set R and all bucket table pages of set S, which correspond to $N_S b/p$ pages. Assume a file with 10^6 data items of 8 bytes each, page size $p = 4096$ bytes and bucket table entry size $b = 20$ bytes. Then $N_S = 8 \times 10^6/4096 = 1953$, and the bucket table size is $1953 \times 20/4096 \approx 10$ pages. The best case for BSMJ is to visit only the bucket table for set S, plus the first bucket table page for set R. Thus the total cost is about 11 accesses. The upper bound for the cost of TJ in this situation is $Height(trie_R) + Height(trie_S)$, where the height of the trie in pages is a logarithmic function of the size of the data set. For a file with several million data items, the trie height can be as low as 3 pages. Thus a typical upper bound of TJ cost is about $2 \times 3 = 6$ pages.

The worst case of the TJ algorithm occurs when all pages have to be loaded from the disk. Of course, this happens when all the join attributes overlap. The same situation also coincides with the worst case of the MJ and BSMJ schemes when MJ needs to visit all data pages, and BSMJ not only has to visit all data pages but also the entire set of bucket tables.

Table 7.6 summarizes the best and the worst case cost for each join algorithm. The best case speed-up of TJ can be orders of magnitudes improvement over MJ and BSMJ. It shows that even in the worst case, TJ outperforms MJ and BSMJ by a factor of $1/r$ (assume $r = r_R = r_S$), where r is the compression ratio of the trie organization. This compression ratio is a function of the file size and the size of data items, and it can be any value between 0 and 1 for data sets containing more than

Case	Join Algorithm	Disk Access Cost
Best case	BSMJ	$N_S \frac{b}{p} + 1$
	MJ	$N_S + 1$
	TJ	2
Worst case	BSMJ	$(1 + \frac{b}{p})(N_R + N_S)$
	MJ	$(N_R + N_S)$
	TJ	$r_R N_R + r_S N_S$

Table 7.6: Best and Worst Case Cost Summary for Join Methods

a few hundred records. This means a speed-up by TJ over MJ and BSMJ is no less than one in the worst case.

In the case of duplicate join attribute values, nested block scan have to be applied for all these join algorithms including the TJ scheme. The worst case cost is quadratic to the input relation sizes. But since tries organize data with considerable compression, the gain by a TJ scheme over other methods increases. Assume the cost of MJ by nested block scan is $C\ N^2$, where C is a constant, and N is the input relation size. If we assume the compression rate by trie organization is r, where $0 < r < 1$, then the size of the input tries is rN and the cost by TJ is no more than $C(rN)^2 = r^2CN^2$. Hence, the TJ scheme still outperforms its competitors, and its cost is no more than r^2 of that of MJ with nested block scan. The more tries compress data, the smaller r is resulting in a more efficient TJ.

7.2.2 Experimental Comparisons

In this subsection we give and compare experimental results for TJ, MJ and BSMJ. Two important parameters for the tests are the *join selectivity*, defined as the ratio of the overlapped range to the total input data range of an attribute, and the dataset size. Thus we have made two sets of experiments; one is disk access versus the join selectivity, and the other is disk access with respect to input file size.

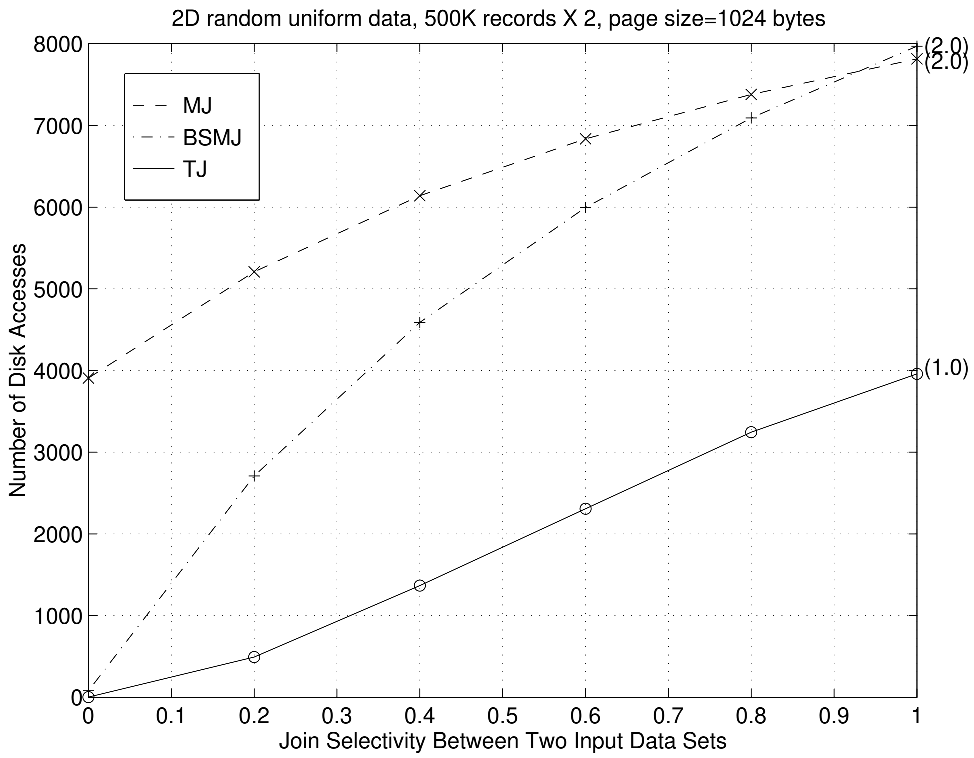
All test data sets are 2D random data uniformly distributed between $[0, 1)^2$. If the join selectivity is P, then the two joins sets are in the range of $[0, (1+P)/2)^2$ and $[(1-P)/2, 1)^2$ respectively. For example, if P= 0.4, the left set range is $[0, 0.7)^2$ and the right set range is $[0.3, 1)^2$.

Figure 7.3(a) shows the cost of disk accesses with respect to the join selectivity. The dataset size is 500k records (4M bytes) for both input sets. TJ has the fewest disk accesses, while BSMJ and MJ vie in their role as the most expensive. The starting cost of MJ is the highest, since it has to examine at least one of the two input data files, even when the files have no intersection at all. The starting cost of BSMJ is not as cost as MJ because it includes only the visits to bucket tables when the join selectivity is zero. TJ is the best among the three at $P = 0$, as it avoids visiting all the remaining files by making decisions at the root page levels. It can be observed that all three schemes are affected by the join selectivity because when the selectivity grows, more data items are involved in the join and the corresponding disk accesses increase.

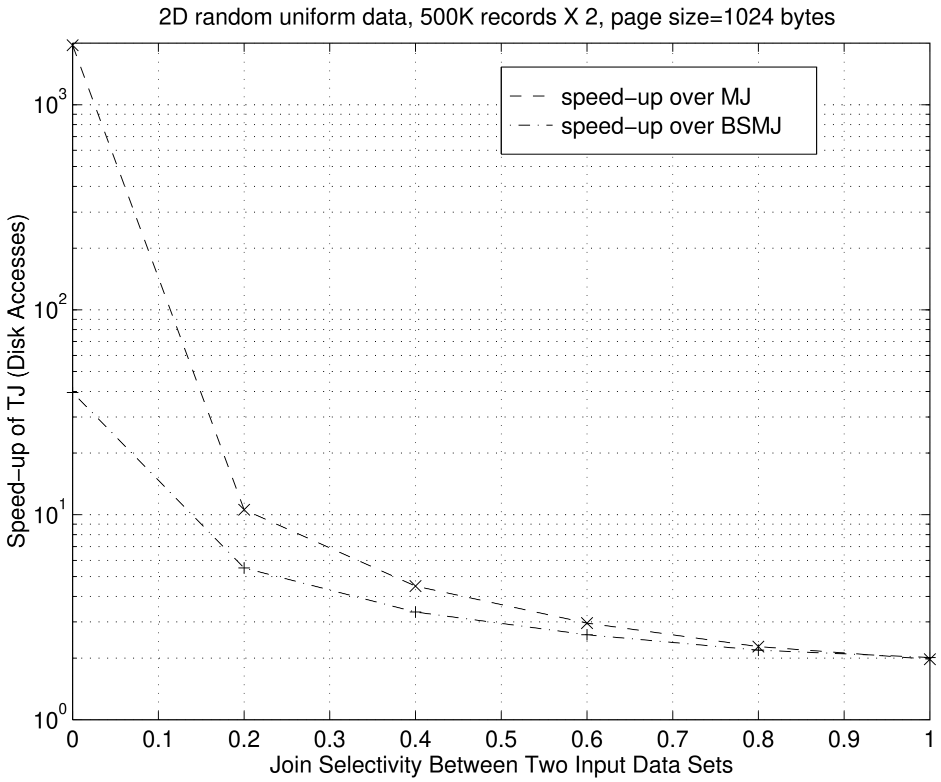
Figure 7.3(b) shows the speed-up of TJ over MJ and BSMJ. The lower the join selectivity, the more TJ improves over the other two methods. Even when the selectivity is one, i.e., the two joining sets are fully overlapped, TJ has a marked improvement over both MJ and BSMJ by a factor of 2. BSMJ is better than MJ when join selectivity is less than one. However, this is reversed when selectivity is one because of the extra accesses to the bucket tables by BSMJ.

The added advantages that TJ exhibits in terms of disk access can be explained by the following points. Firstly, the compression property of tries makes the input data sets organized by tries consume less space. Secondly, the variable-resolution structure of tries enable TJ to avoid some accesses to lower level (high resolution) pages by making decisions at low resolution levels near the root. When the join selectivity is low, the variable-resolution property is the primary reason for the enhanced performance over the other two join algorithms. When the join selectivity is high, the compression property becomes the major factor for the improvement.

TJ scales very well with respect to the size of the input sets; the scalability of the algorithm is compared in Figure 7.4. The join selectivity is fixed at 0.4 for these tests. Figure 7.4(a) shows the cost of joins in terms of disk accesses when the file size varies from 100 records to 10^6 records. The abscissa represents the file size as indicated by the number of records in each input data file. Both the abscissa and ordinate are

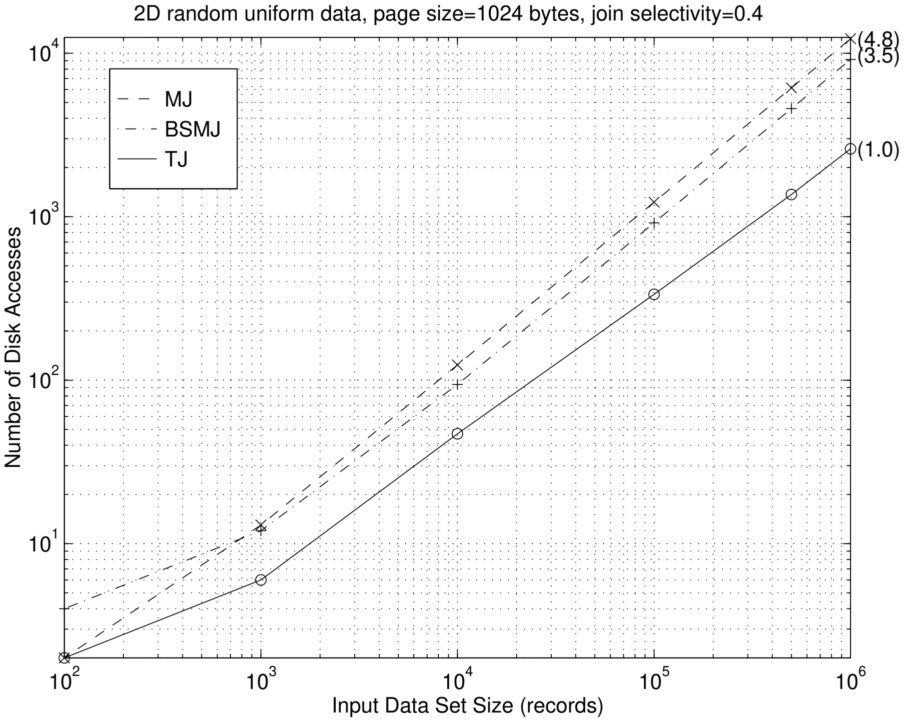


(a) Cost in terms of disk accesses

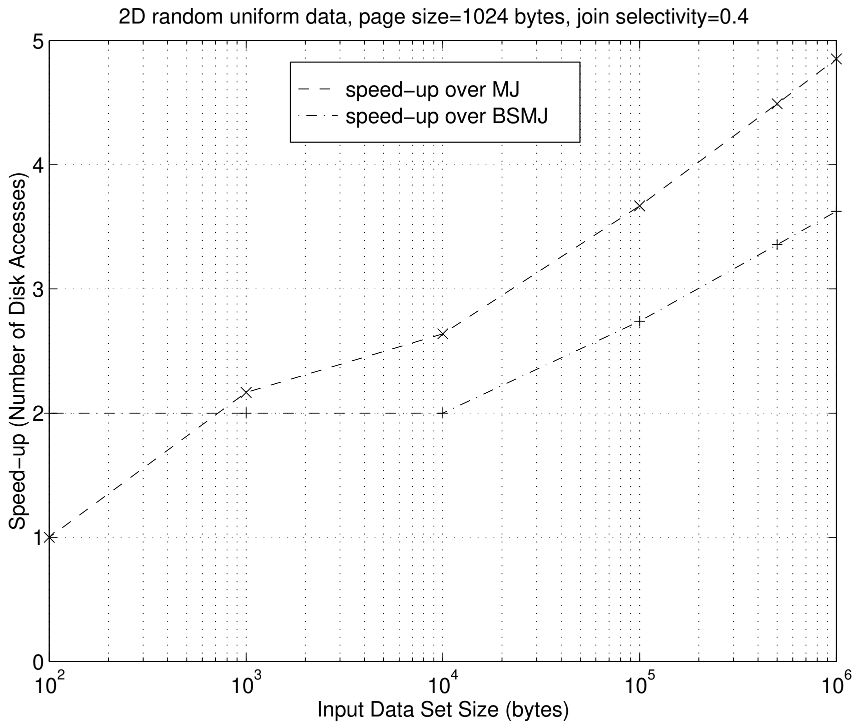


(b) Speed-up (disk accesses) of TJ

Figure 7.3: Disk Accesses versus Join Selectivity



(a) Cost in terms of disk accesses



(b) Speed-up (disk accesses) of TJ

Figure 7.4: Disk Accesses versus File Size

in logarithmic measures. Even though costs for all three algorithms are linear with respect to the input size, TJ not only has the lowest cost but also the smallest slope. This results from tries compression as the sizes of the data sets increase. On the right of Figure 7.4(b), we clearly see the relative speed-up of tries over the other two methods. The improvement is between 2.2 and 4.8 times for MJ, and between 2.0 and 3.5 times for BSMJ when files contain from a 10^3 to 10^6 records. Both graphs in Figure 7.4 clearly show that the TJ method scales very well with input size.

7.3 Discussions and Conclusions

Join processing, a costly operation in relational database systems, critically determines the performance of information retrievals and database queries. The cost of join operations on large input data files largely consists of the disk accesses. Although it is a relatively mature field in database technology, the TJ algorithm we present here is a new method that achieves significant performance improvement over the traditional join schemes and the recent BSMJ scheme, and is therefore more efficient than the representative join algorithms JF and DJ when input data sets are already sorted according to the join attributes. Like BSMJ scheme, it improves over SMJ by avoiding accesses to data items and reducing page accesses whenever possible. But unlike BSMJ which maintains extra bucket tables to achieve the bucket skips, TJ achieves the improvement without using extra indexes. It simply applies the variable-resolution structure of tries on the join attributes to avoid unnecessary block/page accesses at an early stage near the trie root. Instead of using extra storage, TJ takes the advantage of the compression property of tries to achieve storage compression as well as speed improvement.

The major contributions of the chapter are as follows:

- TJ algorithm and its extension from natural joins to union joins as one aspect in the process of proposing tries as efficient data structures on secondary storage for specialized and structured data [Zha96];
- Analysis of the best and the worst case performance of TJ algorithms as well

as competitive join algorithms MJ and BSMJ;

- Detailed performance tests which show the superior performance improvement of TJ over BSMJ and MJ, and thus show the superiority over the improved hash join method JF and the improved merge join method DJ when input sets are ordered.

So far we have assumed that the join attributes of the input data files are keys stored on trie structures. If the input data files are not organized by tries or the join attributes are not stored as keys on trie structures, we need to build tries on these join attributes before performing the join. The construction of a trie from an unordered input data set, like the construction of any other index structure, has a cost of $N\log(N)$, where N is the number of keys. Thus, it remains an open research problem whether in this case joins by tries would still outperform existing join algorithms. Nevertheless, detailed TJ algorithms and experimental results need to be explored in the situation when some, but not all attributes indexed by k-d-tries, are involved in the join process.

Chapter 8

Conclusion

8.1 Contributions

Trie Organization

Based on the pointerless trie structures *FuTrie*, *OrTrie* and *PaTrie*, we have proposed *DyOrTries*. They represent an improvement in that they separate trie nodes stored on pages from the page headers and counters which are used to preserve tree searches instead of linear searches within a page level, and thus are capable of dynamic insertions and deletions of records. We have focussed on this particular methodology to make tries dynamic in order to lead to our own work and further claims.

Order-preserving Key-to-Address Transformation Function

We have proposed a class of order-preserving key-to-address transformation functions which we call tidy functions. These heuristic piece-wise linear tidy functions have no space overhead and can be constructed in linear time. They are competitive to B-trees and the closest minimum order-preserving hashing method in search performance for files up to more than 10GB (10^7 pages with page capacity 1KB) in size, while requiring no extra storage. Our results show that for files with about 10^7 pages, the method requires around three disk accesses to search a given key. It is simple in concept, and the construction and searching algorithms are straightforward.

However, we have applied also 1D tries as the order-preserving key-to-address transformation function. The search performance by the trie method benefits from the fact that it stores keys with the most important bits first, near the root. This variable-resolution structure permits the trie to avoid unnecessary searches into some subtries at the early stage and only explore nodes where there is a possible match. From our experimental results, 1D tries are superior in search performance to the tidy function we proposed; indeed with ordered keys, the construction algorithm needs only to pass the source file once, which is less costly than the construction of linear heuristic tidy functions. Other advantages include reduced storage space requirement as a result of trie compression, and support for dynamic insertions and deletions of records.

Variable Resolution Queries

In this work, we extend tries to multidimensional structured data on secondary storage other than text and spatial data. To supplement this effort, we propose exact match and the general orthogonal range query algorithms.

We also have done extensive experimental comparisons with representative multidimensional data structures in two categories: direct access methods including grid files and multipaging, and logarithmic methods including R*-trees and X-trees. The access and storage costs are addressed based on four categories: the query selectivity, data distribution, the dimensionality and the file size (record size). The access cost is measured both with the number of disk accesses, and the equivalent time spent on the disk accesses based on disk parameters.

Our experimental results show that tries have the best query performance among all the above competitors for queries returning more than a few records. Tries are also competitive in exact match queries with the two direct access methods (multipaging and grid files).

Tries are not affected by poor data distributions both in searching and storage cost; this is in contrast to the access cost of multipaging which increases rapidly for pathological distributions and the grid file degeneration as the directory size grows rapidly. To aid in our analysis, we have proposed two new analysis methods to characterize and quantify the distribution for grid files and multipaging.

As we carry on the comparison to R*-trees and X-trees in high dimensions, tries are able to show their reliability by displaying improved performance by factors by up to 2 or 3 times depending on the query selectivity.

Tries are also superior to all other methods in terms of storage cost. These other methods either require a small amount of additional space (multipaging), or significant amounts of directory (index) space (grid files, R*-trees and X-trees) on secondary storage. Tries compress data due to sharing common paths near the root. The file compression ratio of tries ranges between zero and one, depending on the file size and the record size. In general, trie compression increases as the file size grows larger. An inverse relationship exists as the record size becomes larger.

Trie Joins

We apply tries to binary operations by presenting new (natural) join algorithms by tries. The new algorithm takes two input tries and outputs a new trie by joining common keys. It achieves significant performance improvement in comparison to the bucket skip merge join, the best among competitors to trie joins, including the sort-merge join, the distributive join, and the join-by-fragment algorithms. Our scheme applies the order-preserving and variable-resolution trie structures on join keys to avoid unnecessary page accessing at an early stage near the root. Instead of requiring extra indexes, it takes advantage of trie compression in order to obtain significant speed improvement. Our natural join (set merging) algorithm of tries can easily be extended to union joins, difference joins and exclusive or operations on two tries.

Moreover, the join attributes need not to be keys. If the input trie indexes attributes contain attributes that are not exclusively join attributes, the algorithm can be easily extended by skipping the levels of trie nodes containing non-join attributes and performs the join operations only on node levels of the join attributes.

By extending tries, traditionally applied to text and spatial data, to general 1D or multidimensional data on secondary storage to perform exact match queries, orthogonal range queries, as well as binary operations represented by join operations on tries, we have explicitly demonstrated the following:

- the variable-resolution trie structure achieves efficient query performance;
- the trie compression not only saves storage cost, but also makes high selectivity queries and operations on tries affordable;
- the trie method is unaffected by data distributions; and
- the query performance by tries is better than the representative tree structures, such as R*- and X-trees, designed for multiple and high dimensions.

8.2 Future Research

Although trie construction methods are much more efficient when all the data is inserted at once rather than by single record insertion, we have discussed and modified the trie organization on secondary storage so that it is is capable of inserting and deleting records at run time. However, we have not discussed how the storage overhead would increase as a trade-off for supporting these dynamic operations.

Several new data structures and methodologies are invented each year; to keep track of them and continually compare with tries becomes a challenging, and even an endless task!

We need to extend the trie method to broader queries other than the exact match queries and range queries. Will tries still be more efficient or at least competitive for other queries, such as the nearest neighbor queries of recent specialized data structures, such as SS-trees and SR-trees, which are especially designed for such queries?

We have investigated principally natural joins of tries with the attributes organized by the input tries. This raises the question of the whole family of various joins. What if the attributes indexed on one trie are not in the same order as the attributes on the other input trie? What if there are more attributes on the input tries than the common join attributes? what if the join attributes are not ordered by the trie at all? Can we still benefit from trie joins? Algorithms in the above situations need to be invented, implemented and tested.

In a word, this work is a modest, albeit important, attempt at presenting the trie method, a simple but powerful and efficient method, as a key to general purpose queries and operations on multidimensional data. There still remains a lot of unplotted ground for further research and study.

Bibliography

[ABV95] Walid G. Aref, Daniel Barbará, and Padmavathi Vallabhaneni. The handwritten trie: Indexing electronic ink. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 151–162, San Jose, California, may 1995.

[AHU83] A.B. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, MA, 1983.

[AN93] A. Andersson and S. Nilsson. Efficient implementation of suffix trees. *Software Practice and Experience*, 25(2):129–41, Febrary 1993.

[Aoe89] J.I. Aoe. An efficient digital search algorithm by using a double-array structure. *IEEE Transactions on Software Engineering*, 15(9):1066–77, 1989.

[Apo85] A. Apostolico. The myriad virtues of suffix trees. In *Combinatorial Algorithms on Words*, pages 85–96. Springer-Verlag, 1985.

[ASU86] A.B. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[BBK98] Stefan Berchtold, Christian Bohm, and Hans-Peter Kriegel. The pyramid-technique: Towards breaking the curse of dimensionality. In *Proceedings of the ACM SIGMOD Annual Conference*, pages 142–53, 1998.

[BE77] E. W. Blasgen and K. P. Eswaran. Storage and access in a relational database. *IBM System Journal*, 16(4):361–77, 1977.

[Ben75] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–17, September 1975.

[Ber84] T. Berger. Poisson multiple access for packet broadcast channels. *IEEE Transactions on Information Theory*, IT-30:745–51, 1984.

[BK93] T. Bell and D. Kulp. Longest-match string searching for Ziv-Lempel compression. *Software Practice and Experience*, 23(7):757–71, December 1993.

[BKK96] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-tree: An index structure for high-dimensional data. In *Proceedings of 22th International Conference on VLDB*, pages 28–39, 1996.

[BKS93] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using R-trees. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 237–246, Washington, D.C., may 1993.

[BKSS90] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The R* - tree: an efficient and robust access method for points and rectangles. In *Proceedings of the SIGMOD Conference*, pages 322–31, May 1990.

[BM72] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 13:173–89, 1972.

[BM90] Jose A. Blakeley and Nancy L. Martin. Join index, materialized view, and hybrid-hash join: A performance analysis. In *Proceedings, Sixth International Conference on Data Engineering: February 5–9, 1990, Los Angeles Airport Hilton and Towers, Los Angeles, California, USA*, pages

256–263, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1990. IEEE Computer Society Press.

[Bri59] R. De La Briandais. File searching using variable length keys. In *Proceedings of the Western Joint Computer Conference*, volume 15, pages 295–8, IRE, New York, 1959. Spartan Books, New York.

[BS89] R.M. Bozinovic and S.N. Srihari. Off-line cursive script word recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(1):68–83, January 1989.

[BWC89] T. Bell, I.H. Witten, and J.G. Cleary. Modeling for text compression. *ACM Computing Surveys*, 21(4):557–91, December 1989.

[BYG89] R.A. Baeza-Yates and G.H. Gonnet. Efficient text searching of regular expressions. In *Proceedings of 16th International Colloquium on Automata, Languages and Programming*, LNCS 372, pages 46–62, Stresa, Italy, July 1989. Springer-Verlag.

[BYP92] R.A. Baeza-Yates and C.H. Perleberg. Fast and practical approximate string matching. In *Proceedings of 3rd Annual Symposium on Combinatorial Pattern Matching*, LNCS 644, pages 185–92, Tucson, Arizona, April 1992. Springer-Verlag.

[Cap79] J.I. Capetanakis. Tree algorithms for packet broadcast channels. *IEEE Transactions on Information Theory*, IT-25(5):505–15, 1979.

[CFV98] J. Clément, P. Flajolet, and B. Vallée. The analysis of hybrid trie structures. In *Proceedings of the Ninth ACM-SIAM Symposium on Discrete Algorithms*, San Francisco, CA, January 1998.

[CFV99] J. Clément, P. Flajolet, and B. Vallée. Dynamical sources in information theory: A general analysis of trie structures. Technical Report 3645, Institut National De Recherche en Informatique et en Automatique, 1999.

[CHK85] G. V. Cormack, R. N. S. Horspool, and M. Kaiserswerth. Practical perfect hashing. *Computer Journal*, 28(1):54–58, 1985.

[CHM92] Z. J. Czech, G. Havas, and B. S. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, 43(5):257–64, Oct 1992.

[CL93] C. C. Chang and J. Liang. Dynamic pictorial databases design for similarity retrieval. *Information Science*, 87(1-3):29–46, Nov. 1993.

[CM95] Richard H. Connelly and F. Lockwood Morris. A generalization of the trie data structure. *Mathematical Structures in Computer Science*, 5(3):381–418, September 1995.

[Cod70] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–87, June 1970.

[CS77] D. Comer and R. Sethi. The complexity of trie index construction. *Journal of the ACM*, 24(3):428–40, July 1977.

[CY96] Soon M. Chung and Jaerheen Yang. Parallel distributive join algorithm for cube-connected multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(2):127–137, February 1996.

[Dev82] L. Devroye. A note on the average depth of tries. *Computing*, 28:367–71, 1982.

[Dev84] L. Devroye. A probabilistic analysis of the height of tries and of the complexity of triesort. *Acta Informatica*, 21:229–37, 1984.

[Dev87] L. Devroye. Branching processes in the analysis of the heights of trees. *Acta Informatica*, 24:277–98, 1987.

[DG85] D. J. DeWitt and R. H. Gerber. Multiprocessor hash-based join algorithms. In A. Pirotte and Y. Vassiliou, editors, *Very Large Data Bases: Stockholm, 1985/11th International Conference on Very Large*

Data Bases, Stockholm, August 21–23, 1985, page 151, Los Altos, CA 94022, USA, 1985. Morgan Kaufmann Publishers.

[DNS91] D. DeWitt, J. Naughton, and D. Schneider. An evaluation of non-equijoin algorithms. In *vldb*, pages 443–452, 1991.

[DST75] R. F. Deutscher, P. G. Sorenson, and J. P. Tremblay. Distribution-dependent hashing functions and their characteristics. In *Proceedings of the International Conference on the Management of Data*, pages 224–36, San Jose, CA, May 1975.

[DTK91] A.C. Downton, R.W.S. Tregidgo, and E. Kabir. Recognition and verification of handwritten and hand printed British postal addresses. *International Journal of Pattern Recognition and Artificial Intelligence*, 5(1-2):265–91, 1991.

[Dun91] J.A. Dundas. Implementing dynamic minimal-prefix tries. *Software Practice and Experience*, 21(20):1027–40, October 1991.

[Dye82] C.R. Dyer. The space efficiency of quadtrees. *Computer Graphics and Image Processing*, 19(4):335–48, August 1982.

[ED80] R. J. Enbody and H. C. Du. Dynamic hashing schemes. *ACM Computing Surveys*, 20:85–113, 1980.

[EMR] Canada Energy, Mines, and Resources. Memphremagog: 1:50,000, 31h1.

[ES63] Jr. E.H. Sussenguth. Use of tree structures for processing files. *Communications of the ACM*, 6(5):272–9, 1963.

[FB74] R.A. Finkel and J.L. Bentley. Quad trees – a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[FCD91] Edward A. Fox, Qi Fan Chen, and Amjad M. Daoud. Order-preserving minimal perfect hash function. *ACM Transactions on Information Systems*, 9(3):281–308, 7 1991.

[FG89] E.R. Fiala and D.H. Greene. Data compression with finite windows. *Communications of the ACM*, 32(4):490–505, 1989.

[FGPM93] P. Flajolet, G. Gonnet, C. Puech, and Robson J. M. Analytic variations on quadtrees. *Algorithmica*, 10(7):473–500, December 1993.

[FHCD92] Edward A. Fox, Lenwood S. Heath, Qi Fan Chen, and Amjad M. Daoud. Practical minimal perfect hash functions for large databases. *Communications of the ACM*, 35(1):105–121, 1 1992.

[FL94] P. Flajolet and T. Lafforgue. Search costs in quadtrees and singularity pertubation asymptotics. *Discrete and Computational Geometry*, 12(4):151–175, 1994.

[Fre60] E.H. Fredkin. Trie memory. *Communications of the ACM*, 3:490–500, September 1960.

[Fre87] M. Freeston. The BANG file: A new kind of grid file. In *Proceedings of the SIGMOD Conference*, pages 260–69, San Francisco, May 1987.

[Fre95] M. Freeston. A general solution of the n-dimensional B-tree problem. In *Proceedings of the SIGMOD Conference*, pages 80–91, San Jose, California, May 1995.

[GBY91] G.H. Gonnet and R.A. Baeza-Yates. Lexicographical indices for text: Inverted files vs. pat trees. Technical Report OED-91-01, Centre for the New OED., University of Waterloo, 1991.

[GG86] Anil. K. Garg and C. C. Gotlieb. Order-presearving key transformations. *ACM Transactions on Database Systems*, 11(2):213–234, 1986.

[GLS94] G. Graefe, A. Linville, and L. D. Shapiro. Sort versus hash revisited. *ACM Transactions on Knowledge and Data Engineering*, 6(6):934–44, December 1994.

[Gon88] G.H. Gonnet. Efficient searching of text and pictures. Technical Report OED-88-02, Centre for the New OED., University of Waterloo, 1988.

[Gon91] G.H. Gonnet. *Handbook of Algorithms and Data Structures*. Addison-Wesley, Reading, MA, 1991.

[GRG80] G. H. Gonnet, L. D. Rogers, and J. A. George. An algorithmic and complexity analysis of interpolation search. *Acta Informatica*, 13:39–52, 1980.

[GSB94] R. Gupta, S. A. Smolka, and S. Bhashar. On randomization in sequential and distributed algorithms. *ACM Computing Surveys*, 26(1):23–86, March 1994.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the SIGMOD Conference*, pages 45–57, Boston, June 1984.

[HCE91] J.J. Hardwicke, J.H. Connolly, and J. Edwards. Parallel access to an English dictionary. *Microprocessors and Microsystems*, 15(6):291–8, July 1991.

[HCY97] Hui-I Hsiao, Ming-Syan Chen, and P. S. Yu. Parallel execution of hash joins in parallel databases. *IEEE Transactions on Parallel and Distributed Systems*, 8(8):872–883, August 1997.

[HD80] P.A.V. Hall and G.R. Dowling. Approximate string matching. *Computing Surveys*, 12(4):381–402, December 1980.

[HJR97] Yun-Wu Huang, Ning Jing, and Elke A. Rundensteiner. Spatial joins using R-trees: Breadth-first traversal with global optimizations. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases*, pages 396–405, 1997.

[Hun78] G.M. Hunter. *Efficient Computation and Data Structure for Graphics.* PhD dissertation, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ, 1978.

[Iig95] Y. Iiguni. Nonlinear signal processing based upon a reconfigurable knowledge database. *Electronics and computations in Japan, Part III: Fundamental Electronic Science*, 78(8):22–30, August 1995.

[Jac91] P. Jacquet. Analysis of digital tries with markovian dependency. *IEEE Transactions on Information Theory*, IT-37(5):1407–75, September 1991.

[Jon89] L.P. Jones. Portrep: A portable repeated string finder. *Software Practice and Experience*, 19(1):63–77, January 1989.

[KF94] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proceedings of 20th International Conference on VLDB*, pages 500–9, September 1994.

[Knu68] D.E. Knuth. *Information Structures*, volume 1 of *The Art of Computer Programming.* Addison-Wesley, Reading, MA, 1968.

[Knu73] D.E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley, Reading, MA, 1973.

[KR96] M. Kamath and K. Ramamritham. Bucket skip merge join: A scalable algorithm for join processing in very large databases using indexes. Technical Report UM-CS-1996-020, University of Massachusetts, Amherst, Computer Science, March, 1996.

[KS97a] Norio Katayama and Shin'ichi Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proceedings of the SIGMOD Conference*, pages 369–79, Arizona, May 1997.

[KS97b] Nick Koudas and Kenneth C. Sevcik. Size separation spatial join. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 26,2 of *SIGMOD Record*, pages 324–335, New York, May13–15 1997. ACM Press.

[KS00a] C. Knessl and W. Szpankowski. Asymptotic behavior of the height in a digital search tree and the longest phrase of the Lempel-Ziv scheme. In *SIAM-ACM Symposium on Discrete Algorithms*, pages 187–196, San Franscisco, January 2000.

[KS00b] C. Knessl and W. Szpankowski. Limit laws for heights in generalized tries and patricia tries. In *LATIN'2000*, Punta del Este, Uruguay, January 2000.

[Kuk92] K. Kukich. Techniques for automatically correcting words in text. *Computing Surveys*, 24(4):377–439, December 1992.

[LEMR89] Y.H. Lee, M. Evens, J.A. Michael, and A.A. Rovick. Spelling correction for an intelligent tutoring system. In *Proceedings of Computing in the 90's. The First Great Lakes Computer Science Conference*, pages 77–83, Kalamazoo, MI, October 1989.

[Lit80] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proceedings of 6th International Conference on VLDB*, pages 212–223, Montreal, October 1980.

[Lit81] W. Litwin. Trie hashing. In *Proceedings of ACM SIGMOD 81*, pages 19–29, April 1981.

[Lit85] W. Litwin. Trie hashing: Further properties and performances. In *Proceedings of the International Conference on Foundations of Data Organization and Algorithms*, 1985.

[LJF94] K.-I. Lin, H.V. Jagadish, and C. Faloutsos. The TV-tree: An index structure for high-dimensional data. *The VLDB Journal*, 5(4):517–42, 1994.

[Lom90] David B. Lomet. The hB-tree: A multiattribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, 15(4):625–658, December 1990.

[LR94] Ming-Ling Lo and C. V. Ravishankar. Spatial joins using seeded trees. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(2):209–220, June 1994.

[LR96] Ming-Ling Lo and Chinya V. Ravishankar. Spatial hash-joins. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 247–258, Montreal, Quebec, Canada, June 1996.

[LRLH91] W. Litwin, N. Roussopoulos, G. Levy, and W. Hong. Trie hashing with controlled load. *IEEE Transactions on Software Engineering*, 17(7):678–91, 1991.

[LT95] Hongjun Lu and Kian-Lee Tan. On sort-merge algorithm for band joins. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):508–510, June 1995.

[LZL88] W. Litwin, D. Zegour, and G. Levy. Multilevel trie hashing. In *Advances in Database Technology, EDBT'88: International conference on Extending Database Technology*, Venice, Italy, 1988. Berlin, New York: Springer-Verlag.

[Mal76] K. Maly. Compressed tries. *Communications of the ACM*, 19(7):409–15, 1976.

[ME92] Priti Mishra and Margaret H. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, March 1992.

[Mea82] D. Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2):129–47, June 1982.

[Mer83] T. H. Merrett. *Relational Information Systems*. Reston, 1983.

[MF85a] P. Mathys and P. Flajolet. Q-ary collision resolution algorithms in random access system with free and blocked channel access. *IEEE Transactions on Information Theory*, IT-31(2):217–43, 1985.

[MF85b] T. H. Merrett and B. Fayerman. Dynamic Patricia. In *Proceedings of the International Conference on Fundations of Data Organization*, pages 13–20, Kyoto, Japan, May 1985.

[MIA94] Katsushi Morimoto, Hirokazu Iriguchi, and Jun-Ichi Aoe. A method of compressing trie structures. *Software-Practice and Experience*, 24(3):265–288, March 1994.

[MO81a] T. H. Merrett and E. Otoo. Multidimensional paging for associative searching. Technical Report SOCS-81.18, School of Computer Science, McGill University, May 1981.

[MO81b] T. H. Merrett and E. J. Otoo. Dynamic multipaging: a storage structure for large shared data banks. Technical Report SOCS-81-26, School of Computer Science, McGill University, Montreal, Quebec, Canada, 1981.

[MO82] T. H. Merrett and E. J. Otoo. Dynamic multipaging: A storage structure for large shared data banks. In *Improving Database Usability and Responsiveness*, pages 237–54. Academic Press, New York, 1982.

[Mor68] D.R. Morrison. Patricia – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 14(4):514–34, October 1968.

[MS94] T. H. Merrett and H. Shang. Zoom tries: A file structure to support spatial zooming. In *Sixth International Symposium on Spatial Data Handling*, pages 792–804, Edinburgh, 1994.

[MSZ96] T.H. Merrett, H. Shang, and X. Zhao. Database structures, based on tries, for text, spatial and general data. In *Proceedings of International Symposium on Cooperative Database Systems for Advanced Applications*, pages 316–24, Kyoto, Japan, Dec 1996.

[MWHC96] B. S. Majewski, N. C. Wormald, G. Havas, and Z. J. Czech. A family of perfect hashing methods. *Computer Journal*, 39(6):547–54, Dec 1996.

[NHS84] J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, March 1984.

[NK98] Stefan Nilsson and Gunnar Karlsson. Internet programming fast IP routing with LC-tries: Achieving gbit/sec speed in software. *Dr. Dobb's Journal of Software Tools*, 23(8):70, 72–75, August 1998.

[NP91] M. Negri and G. Pelagatti. Distributive join: A new algorithm for joining relations. *ACM Transactions on Database Systems*, 16(4):655–669, December 1991.

[OM84] J.A. Orenstein and T.H. Merrett. A class of data structures for associative searching. In *Proceedings of Third ACM SIGACT–SIGMOD Symposium on Principles of Database Systems*, pages 181–90, Waterloo, April 1984.

[OM92] M.A. Ouksel and O. Mayer. A robust and efficient spatial data structure–the nested interpolation-based grid file. *Acta Informatica*, 29(4):335–373, 1992.

[Ore82a] J.A. Orenstein. *Algorithms and Data Structures for the Implementation of a Relational Database*. Technical report socs-82-17, School of Computer Science, McGill University, 1982.

[Ore82b] J.A. Orenstein. Multidimensional tries used for associative searching. *Information Processing Letters*, 14(14):150–6, June 1982.

[PD96] Jignesh M. Patel and David J. DeWitt. Partition based spatial-merge join. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 259–270, Montreal, Quebec, Canada, June 1996.

[Pit85] B. Pittel. Asymptotical growth of a class of random trees. *The Annals of Probability*, 13(12):414–27, 1985.

[Pro96] ProCD. *Business phone book*. ProCD Inc, 222 Rosewood Drive, Danvers, MA 01923-9893, 1996.

[RBK89] R. Ramesh, A.J.G. Babu, and J.P. Kincaid. Variable-depth trie index optimization: Theory and experimental results. *ACM Transactions on Database Systems*, 14(1):41–74, March 1989.

[Reg85] M. Regnier. Analysis of grid file algorithms. *BIT*, 25:335–357, 1985.

[Reg88] M. Regnier. Trie hashing analysis. In *Proceedings of Fourth International Conference on Data Engineering*, pages 377–81, Los Angeles, CA, February 1988.

[RJS93] B. Rais, P. Jaquet, and W. Szpankowski. A limiting distribution for the depth in patricia tries. *SIAM Journal on Discrete Mathematics*, 6:197–213, 1993.

[RL85] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. In *Proceedings of the SIGMOD Conference*, pages 17–26, May 1985.

[Rob81] J. T. Robinson. The K-D-B-tree: A search structure for large multidimensional dynamic indexes. *Proceedings ACM SIGMOD Conference on Management of Data*, June 1981.

[Sac86] G. M. Sacco. Fragmentation: A technique for efficient query processing. *ACM Transactions on Database Systems*, 11(2):113–133, June 1986.

[Sam90] H. Samet. *Applications of Spatial Data Structrues: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, Reading, Mass., 1990.

[Sha94] H. Shang. *Trie methods for text and spatial data on secondary storage.* PhD dissertation, School of Computer Science, McGill University, Montreal, Quebec, November 1994.

[SK83] D. Sankoff and J.B. Kruskal. *Time Warps, String Edits, and Macromolecules : the Theory and Practice of Sequence Comparison.* Addison-Wesley, Reading, Mass., 1983.

[SM94] D. K. Shin and A. C. Meltzer. A new join algorithm. *sigmod*, 23(4):13–18, December 1994.

[SM96] H. Shang and T.H. Merrett. Tries for approximate string matching. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):540–7, August 1996.

[Spr77] R. Sprugnoli. Perfect hash functions:a single probe retrieval method for static sets. *Communications of the ACM*, 20(11):841–50, 1977.

[SR87] T. Sellis and N. Roussopoulos. The R+ -tree: a dynamic index for multidimensional objects. In *Proceedings of 13th International Conference on VLDB*, pages 507–18, Brighton, UK, September 1987.

[STD78] P. G. Sorenson, J. P. Tremblay, and R.F. Deutscher. Key-to-address transformation techniques. *INFOR*, 16(1):1–34, 1978.

[Su88] S.Y.W. Su. *Database Computers: Principles, Architectures, and Techniques.* McGraw-Hill, New York, 1988.

[Szp88] W. Szpankowski. Some results on V-ary asymmetric tries. *Journal of Algorithms*, 9:224–44, 1988.

[Szp90] W. Szpankowski. Patricia tries revisited. *Communications of the ACM*, 37(4):691–711, October 1990.

[Szp91] W. Szpankowski. A characterization of digital search trees from the successful search viewpoint. *Theoretical Computer Science*, 85:117–34, 1991.

[Szp92] W. Szpankowski. Probabilistic analysis of generalized suffix trees. In Goos and Hartmanis [BYP92], pages 1–14.

[Szp93] W. Szpankowski. Asympototic properties of data compression and suffix trees. *IEEE Transactions on Information Theory*, IT-39, 1993.

[TITK88] T. Tokunaga, M. Iwayama, H. Tanaka, and T. Kamiwaki. Langlab: A natural language analysis system. In *Proceedings of the 12th International Conference on Computational Linguistics*, pages 655–60, Budapest, Hungary, August 1988.

[Tom92] F.W. Tompa. An overview of waterloo's database software for the OED. Technical Report OED-92-01, Centre for the New OED., University of Waterloo, 1992.

[TS96] Y. Theodoridis and T. Sellis. A model for the prediction of R-tree performance. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems - ACM PODS*, pages 161–71, Montreal, Canada, June 1996.

[TY79] R.E. Tarjan and A.C.C. Yao. Storing a sparse table. *Communications of the ACM*, 21(11):606–11, October 1979.

[Val87] P. Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2):218–46, June 1987.

[WJ96] D. A. White and R. Jain. Similarity indexing with the SS-tree. In *Proceedings of the 12th International Conference on Data Engineering*, pages 516–23, New Orleans, Feb 1996.

[Zha96] Xiaoyan Zhao. Tries for structured data on secondary storage. Ph.D. thesis proposal, January 1996.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–43, 1977.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–6, 1978.

Appendix I. Brief History of Trie Structures

Year(s)	Descriptions and Citations
59	First paper on trie structures [Bri59]
60	Trie memory (Re*trie*val) [Fre60]
68	Patricia tries for text and prefix searching [Mor68]
68,73	Pruned trie, digital trees, patricia trie [Knu68, Knu73]
76	Compressed trie/c-trie for static data [Mal76]
77	Complexity of trie constructions [CS77]
78	Quadtries [Hun78]
80,81	Trie hashing [ED80, Lit80, Lit81]
82	K-d-tries and bitstring representation of tries [Ore82b], octree [Mea82]
84	Z-order to organize k-dimensional data [OM84]
85	Suffix tries [Apo85], Trie hashing [Lit85]
86	Trie methods applied on lexical analyzers and compliers [ASU86]
88	Trie methods on natural language analysis [TITK88]
	Prefix text search, PAT tree [Gon88]
	More trie hashing [LZL88, Reg88]
89	Tries applied to data compression [BWC89, FG89]
	Pattern recognition [BS89],
	Natual language analysis [Jon89], and
	Spelling checker [LEMR89]
	Trie implementation [RBK89]
90	Pr-tries and quadtries for spatial data [Sam90]
91	Tries for pattern recognitions[DTK91],
	Parallel searching [HCE91]
	Trie implementations for minimal-prefix tries [Dun91]
	Patarray for text indexing [GBY91, Gon91]
92	Tries for prefix text search [Tom92]
	Suffix tries analysis[Szp92]
93	Suffix trie implementations[AN93]

Trie hashing for similarity retrieval in pictorial databases [CL93]

94 FuTrie for spatial data zooming [MS94]

PaTrie for text indexing and spatial data zooming [Sha94]

Compressing trie [MIA94]

95 Trie application on signal processing [Iig95]

Trie application on pattern recognition [ABV95]

Generalization of trie structures [CM95]

96 Tries for approximate string matching [SM96, MSZ96]

98 LC-tries on IP routing [NK98]