Efficient Storage Methods for a Literary Data Warehouse

by

Steven W. Keith

BScCS — University of New Brunswick Saint John

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of Computer Science

THE UNIVERSITY OF NEW BRUNSWICK

May 2006

Dedication

To my parents

Abstract

Computer-assisted reading and analysis of text has applications in the humanities and social sciences. Ever-larger electronic text archives have the advantage of allowing a more complete analysis but the disadvantage of forcing longer waits for results. This thesis addresses the issue of efficiently storing data in a literary data warehouse. The method in which the data is stored directly influences the ability to extract useful, analytical results from the data warehouse in a timely fashion. A variety of storage methods including mapped files, trees, hashing, and databases are evaluated to determine the most efficient method of storing data points in cubes in the data warehouse. Each storage method's ability to insert and retrieve data points as well as slice, dice, and roll-up a cube is evaluated. The amount of disk space required to store the cubes is also considered. Five test cubes of various sizes are used to determine which method being evaluated is most efficient. The results lead to various storage methods being efficient, depending on properties of the cube and the requirements of the user.

Table of Contents

Vita

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

The decrease in cost and forever increasing size of digital storage media has resulted in large quantities of data being accumulated on disk. This data, however, is generally not organized to be easily accessed and queried by general users. When this raw data can be searched for trends or patterns, the queries are generally computationally expensive. The reason for this expense is clearly the enormous amount of data that requires analysis.

It is well known that the primary expense when extracting useful information from a large data source is the number of disk accesses that are required when reading the data from a secondary, non-volatile memory device such as a hard disk. Even when the primary memory is large, it is only capable of accommodating a very small subset of the actual data. Thus, the accessing time required by the secondary storage medium can easily become the most expensive portion of a query.

Though storage becomes larger and more economical, the sizes of data sets are increasing at a similar rate. Moore's Law [37] states that the number of transistors that can fit in a given space have been doubling every two years. The performance of processors has increased exponentially because of the addition of transistors [18]. Despite hard disk capacities doubling every year, hard disk latency has remained fairly constant. In the past 10 years primary storage access times have decreased by a factor of 20 while hard disk access times have only decreased by a factor of 1.4 [10]. Gray [17] recognizes that the amount of data stored is doubling every year, growing at the same rate at which hard disk sizes are increasing. Assuming these trends continue, it is evident that efficient storage schemes will continue to be important when querying large data sources.

Numerous solutions to the problem of costly secondary storage accesses have been introduced. These solutions include sorting and indexing methods, which provide a means of extracting specific subsets of the data without accessing the entire data set. Efficient storage and indexing methods are paramount to reducing the amount of time required for the execution of queries.

One area in which there exists very large sets of unprocessed information is digital *corpora*[1]. A corpus is a collection of texts. A digital corpus, such

[1]Corpora: plural form of corpus

as Project Gutenberg [45], is a collection of electronic texts, also referred to as *eTexts*. These eTexts may be in any digital format from well structured XML to simple ASCII files. The eTexts available from Project Gutenberg are entirely unprocessed, not necessarily conforming to any defined structure.

1.2 Practical Applications

User-driven analytical tools are used in the humanities for *author attribution*, *lexical analysis*, and *stylometric analysis*. These tools include Signature [28], Word Cruncher [1], Word Smith Tools [48], and Intext [21].

Author attribution is determining the authorship of an anonymous piece of writing through various stylistic and statistical methods. Mendenhall was the pioneer of this area with his study of word lengths [33]. This field was made famous in the 1990's by Foster's work [11, 12] in attributing the authorship of *A Funeral Elegy* to Shakespeare. Foster was also responsible for determining the author of *Primary Colors* and has testified in a number of criminal court cases such as the trial of Theodore Kaczynski (The Unabomber). Li et al. have recently applied these methods to online discussions and instant messages [30]. Though automatic author attribution has been implemented with reasonable measures of success [2, 9, 50], the complexity of language and stylistic analysis, as well as the fact that language is forever evolving, makes it difficult to automate the process reliably.

Lexical analysis includes many measurements of vocabulary usage such as *Type-Token Ratio*[2], *Number of Different Words* and *Mean Word Frequency* [54]. These calculations may be applied to a single book, a collection of books by a single author or time period, or an entire collection of books.

Stylometric analysis not only considers the words in use but also accounts for other statistical elements of style such as word length, sentence length, use of punctuation and many other features. Analysis of this type tends to be used in assisting with author attribution as well as studying the development of an author over time [6].

Even analogies, and thus semantics, can be studied. Analogies of the form *A is to B as C is to D* [53] can be characterized by co-occurrences: two words connected by a *joining word* such as *has*, *on*, and *with* (64 joining words or phrases were initially proposed [53]). Pairs of words related by similar joining terms are deemed analogous.

1.3 Contributions

This thesis is a small part of a much larger open-source development initiative, project Lemur [29]. Lemur is a low-level library providing support for multidimensional databases and OLAP. This library is being used to manage a literary data warehouse containing information derived from the Project Gutenberg corpus.

The thesis involved the development, implementation, and evaluation of

[2]Type-Token ratio is the ratio of different words (types) to the total number of words (tokens)

a number of storage methods. Modifications were made to Lemur storage methods that predated the thesis. Furthermore, the Lemur implementations of the Linear Hashing with Reversed Bit Interleaving, Rotated B+ Tree, and PostgreSQL multidimensional storage structures were added as part of the thesis work. All of the experiments and evaluations of the various storage methods were developed and implemented for the thesis.

This thesis argues that well established Database Management Systems are not efficient storage methods for a literary data warehouse.

1.4 Organization

The remaining chapters of this thesis present the details of the exploration used in evaluating various storage methods. Chapter 2 provides the necessary background information required to understand the various storage methods, the operations these storage methods support, as well as the structure of the data that is to be stored. The implementation details of each storage method are presented in Chapter 3. Chapter 4 outlines the various experiments used in evaluating the strengths and weaknesses of each storage method. This chapter also presents the results of these experiments as well as some preliminary observations. The conditions under which each storage method would perform well are presented in Chapters 4 and 5. The final chapter concludes with some thoughts for future work that have stemmed from this research.

Chapter 2

Background

This chapter provides an introduction into the areas of data warehouses and OLAP. An example of data that will be stored in the literary data warehouse is presented as well as the operations that are supported by the Lemur library. Specific tree and hashing data structures are introduced. Finally, the chapter concludes with a discussion of the metrics that will be used to determine the efficiency of each storage method.

2.1 Data Warehouses

A *data warehouse* is a collection of data gathered from one or more sources, stored and structured to support analysis (see Figure 2.1). Generally, data warehouses do not store current data, as time is required for the data to be verified and transformed into a loadable format. Loading of data is generally done in bulk when the data warehouse is not being queried. Some of the *ACID* properties of transactional systems, namely Atomicity, Consistency, and Isolation [13], are unnecessary in a data warehouse environment as the data in the warehouse is in a fixed state while it is being queried. If the contents of the data warehouse is to be changed, the users can be prohibited from accessing the contents until the changes are complete. The fact that these ACID properties are not required is one of the main reasons that customized storage structures could outperform standard database management systems in the storage and querying of a data warehouse.

2.1.1 Description of Data

The data being stored in the literary data warehouse consists of *cubes*, sometimes referred to as *cuboids*. A cube can be considered equivalent to a multidimensional array, in the fact that it maps key k to a value v. The key k can have any number of dimensions d, where $d \geq 0$. Some examples of dimensions applicable to literary data are words, authors, books, etc.

Dimensions can also conform to specific hierarchies. For example, Figures 2.2 and 2.3 illustrate hierarchies of the word and book dimensions respectively. The data is stored in the data warehouse without making any unnecessary aggregations. For example, if publication dates were available by their exact year, the year would be stored instead of simply the decade or era. This leaves the most options available for summarization. Considering the previous example, it would be possible to group publications by

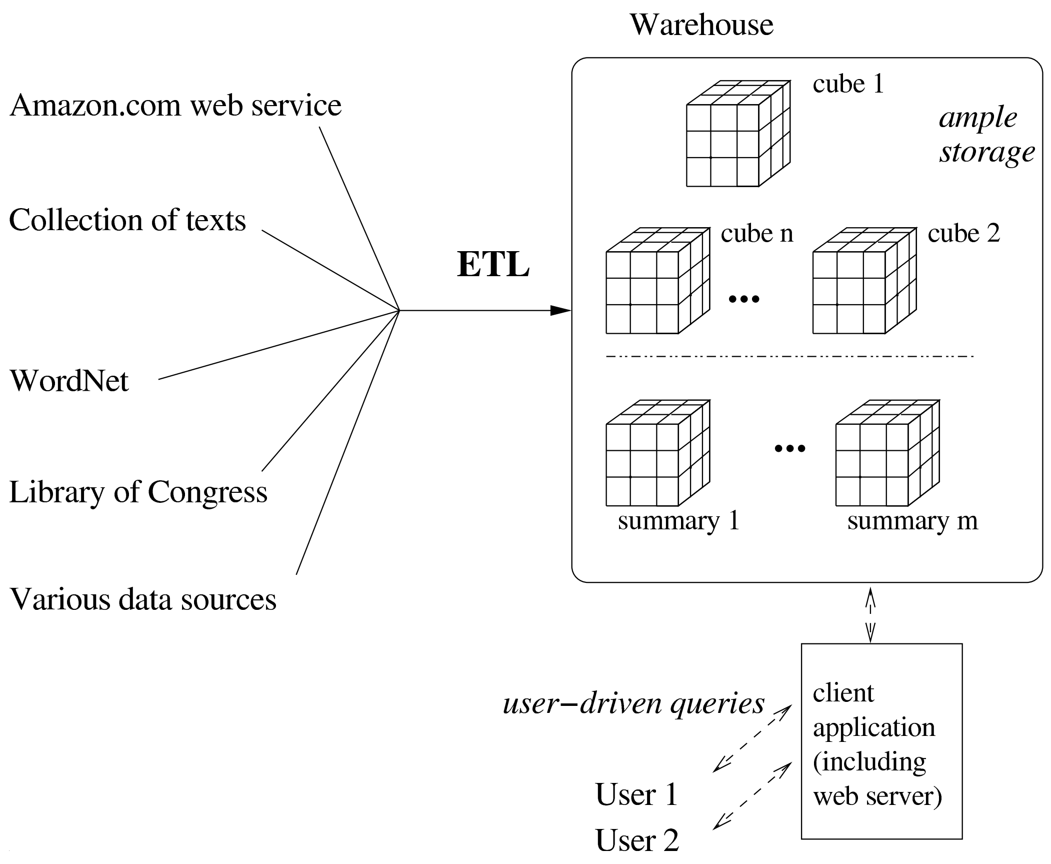


Figure 2.1: Data warehouse architecture

year, decade, century, or era if the most specific value for publication date is stored. The term *hypernyms* found in Figure 2.2 describes groupings of similarly categorized words using Wordnet [27]. For example, the word *monkey* is a hypernym of *mammal* and *mammal* is a hypernym of *living creature.*

Using these dimensions we can create a cube to study any of the previously mentioned application areas of literary analysis. For example, to study analogies a 4-d cube could be created with two *word* dimensions, a *joining*

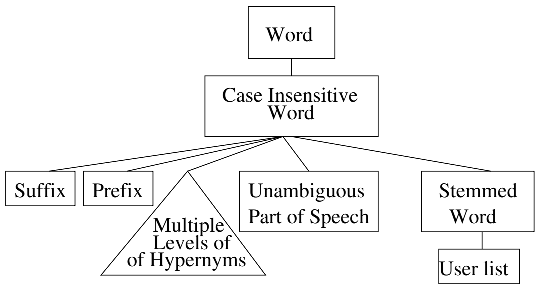


Figure 2.2: Word hierarchy

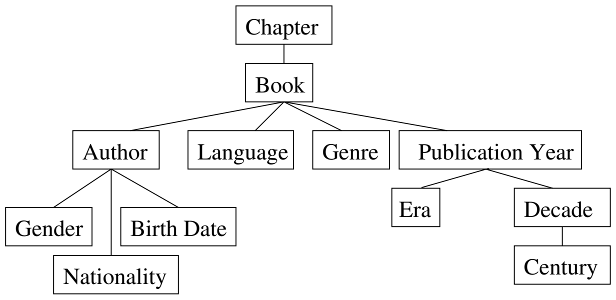
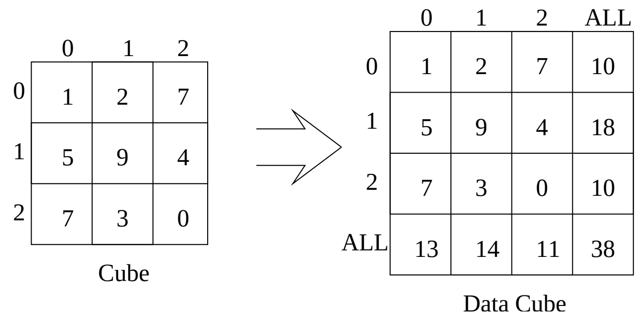


Figure 2.3: Book hierarchy

word dimension and a *book* dimension. This particular cube is henceforth referred to as an Analogy Cube.

2.2 Data Cubes

A data cube as defined by Gray [16] is "the N-dimensional generalization of simple aggregate functions." A data cube is created from a cube by including the entire unchanged cube, but increasing the range of each dimension by 1 to include the "ALL", or aggregation, item. For the purposes of this thesis, the only aggregation operation that is considered is sum. The possible aggregation operations include sum, count, average, maximum, minimum, etc. This restriction does not have large implications on the results as the majority of aggregation operations would take approximately equal time to compute. Figure 2.4 shows the creation of a data cube from a 2-d cube.



$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & 2 & 7 \\ 1 & 5 & 9 & 4 \\ 2 & 7 & 3 & 0 \end{array} \Rightarrow \begin{array}{c|cccc} & 0 & 1 & 2 & \text{ALL} \\ \hline 0 & 1 & 2 & 7 & 10 \\ 1 & 5 & 9 & 4 & 18 \\ 2 & 7 & 3 & 0 & 10 \\ \text{ALL} & 13 & 14 & 11 & 38 \end{array}$$

Figure 2.4: Data cube

2.2.1 Operations

Data cubes were designed to facilitate the computation of generalization query results in standard relational databases. They also provide support for multidimensional, generally hierarchical data. When these queries are expressed in *Structured Query Language* (SQL), they are generally expensive to compute by a relational database system (RDBMS). The data cube has popularized many of these queries such as *slice*, *dice*, and *roll-up* [44].

2.2.1.1 Slice

A slice is used to select a subset of a *d*-dimensional cube, leaving only those points with a selected value in a single dimension (see Figure 2.5). For example, the Analogy Cube can be sliced on the book *Alice in Wonderland*. Slicing completely removes the sliced dimension from the resultant cube leaving a $(d-1)$-dimensional cube. In the previous example the sliced Analogy Cube would result in a 3-dimensional cube in which all data points were from the book *Alice in Wonderland*.

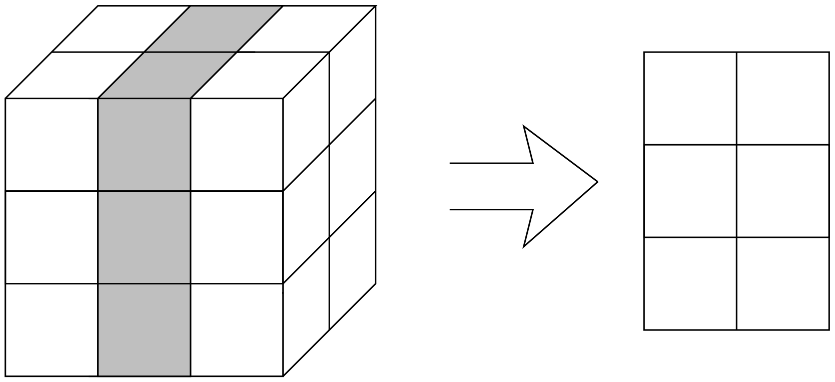


Figure 2.5: Slice operation

2.2.1.2 Dice

A dice is used to select a ranges of values in one or more dimensions of a cube, leaving only those points within the ranges (see Figure 2.6). For example, the Analogy Cube could be diced on the two word dimensions, selecting only those words beginning with vowels. The resultant cube could be used to analyse the analogous relationship of words beginning with vowels.

Dicing does not remove dimensions from the cube, only those values which are not within the specified ranges. Thus when a 4-dimensional cube is diced, the resultant cube will also be 4-dimensional.

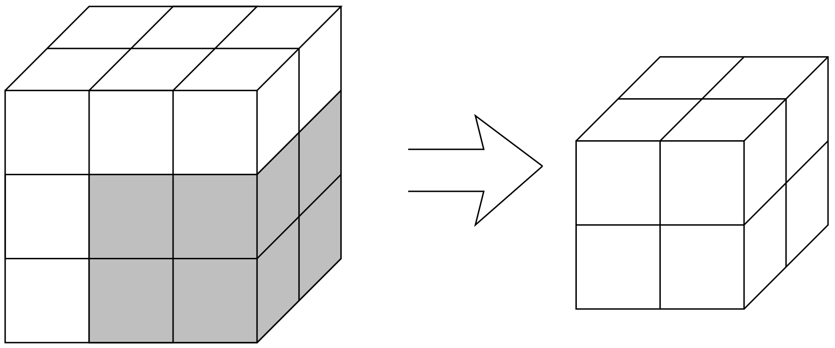


Figure 2.6: Dice operation

2.2.1.3 Roll-Up

The roll-up operation is used to summarize one or more dimensions of a cube completely (see Figure 2.7). The dimension or dimensions that are aggregated will no longer exist in the resultant cube. Roll-ups are used as a summarizing query. For example, rolling up on the joining word and book dimensions of the Analogy Cube results in a cube that contains the number of times two words appear close together.

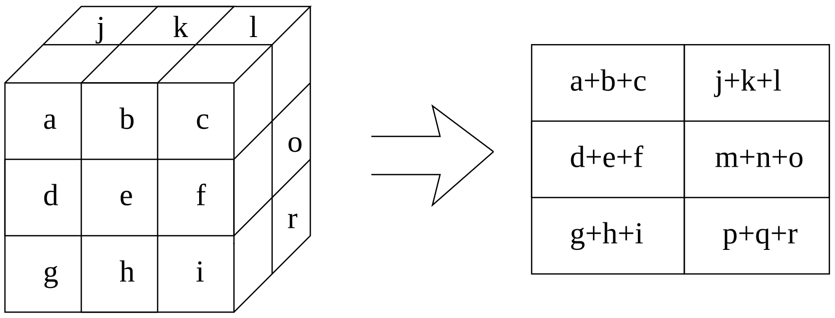
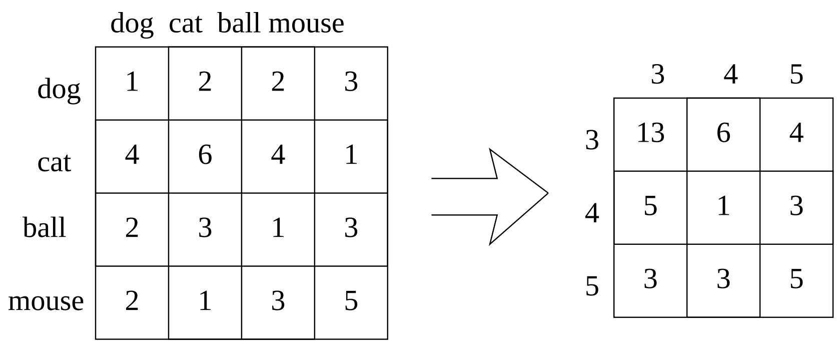


Figure 2.7: Roll-up operation

2.2.1.4 Partial Roll-Up

A partial roll-up is used to summarize one or more dimensions of a cube partially based on the hierarchical nature of those dimensions. For example, both words dimensions of the Analogy Cube could be partially rolled up based on their word lengths to determine the analogous relationship of words of different lengths (see Figure 2.8). Partial roll-ups generally decrease the size of each dimension, creating a smaller cube with the same number of dimensions as the original cube.



	dog	cat	ball	mouse
dog	1	2	2	3
cat	4	6	4	1
ball	2	3	1	3
mouse	2	1	3	5

	3	4	5
3	13	6	4
4	5	1	3
5	3	3	5

Figure 2.8: Partial roll-up on a 2-d cube. Each word dimension is rolled up based on its length

2.2.1.5 Drill Down

The drill down operation is the inverse of the roll up operation. Instead of aggregating data points together, the drill down operation divides them apart to view a more detailed distribution of the data points (see Figure 2.9). For example, the 2-dimensional cube resulting from the roll-up example could be drilled down to once again include the joining word dimension. The drill down operation results in a cube of finer granularity.

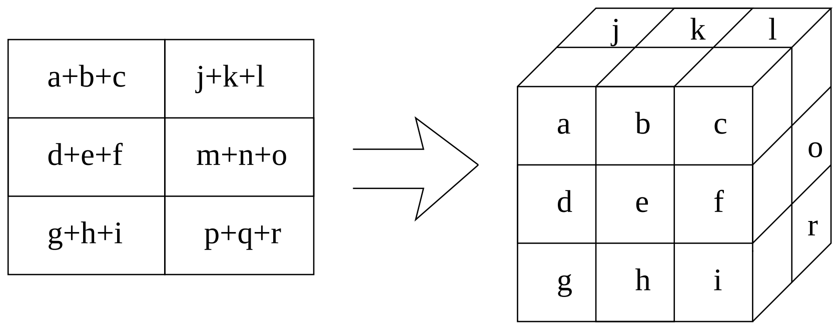


Figure 2.9: Drill down operation

2.3 On-Line Analytical Processing

A possible solution to improve query evaluation time is On-Line Analytical Processing (OLAP) [7]. OLAP is the storage of multidimensional, generally hierarchical, data providing near constant-time answers to queries. Many commercial Multidimensional Database Management Systems (MDBMS) are designed to support OLAP, such as Cognos [8] and Essbase [19]. Using an MDBMS to support analysis of a literary data warehouse would require completely re-engineering how queries are executed and evaluated, which is out of the scope of this thesis.

A frequently implemented improvement in query efficiency of OLAP systems is the pre-computation of commonly executed query results [15]. This avoids the re-computation of results and eliminates scanning the entire data set, saving many disk accesses. Complex queries can be solved by performing operations on the precomputed results in such a way as to achieve the solution without scanning the entire dataset. Determining which query results to pre-compute has been a well explored area of research [26, 52, 56]. The benefit of pre-computing query results can be illustrated by the following example. If a roll-up on the book dimension of an Analogy Cube was previously created, it would be easy to determine how many times the word "balloon" was joined to the word "string" by the word "with." Having pre-computed query results readily available would greatly reduce the amount of time required by many queries.

If we exclude Information Retrieval (IR) [32], no attempts have been made to apply OLAP to literary and linguistics applications. OLAP has been used in conjunction with text mining, however [51].

2.4 B+ Tree

The B+ Tree, initially proposed by Knuth [25], has been adopted as a frequently implemented secondary storage data structure. The B+ Tree storage mechanism is used in many open source and commercial databases management systems [35, 38, 42].

The B+ Tree (see Figure 2.10) is structured in two parts; the non-leaf nodes of the tree act as the index and the leaf nodes contain actual data. The non-leaf nodes of the tree are capable of containing n values and have up to $(n+1)$ pointers to nodes deeper in the tree. Each node in the B+ Tree, with the exception of the root, is required to be at least 50% full, though this percentage can be set higher. Parameter n is chosen to be large, so that a non-leaf node fills a disk block. Therefore, the height of a B+ Tree is small and searching it requires few disk accesses.

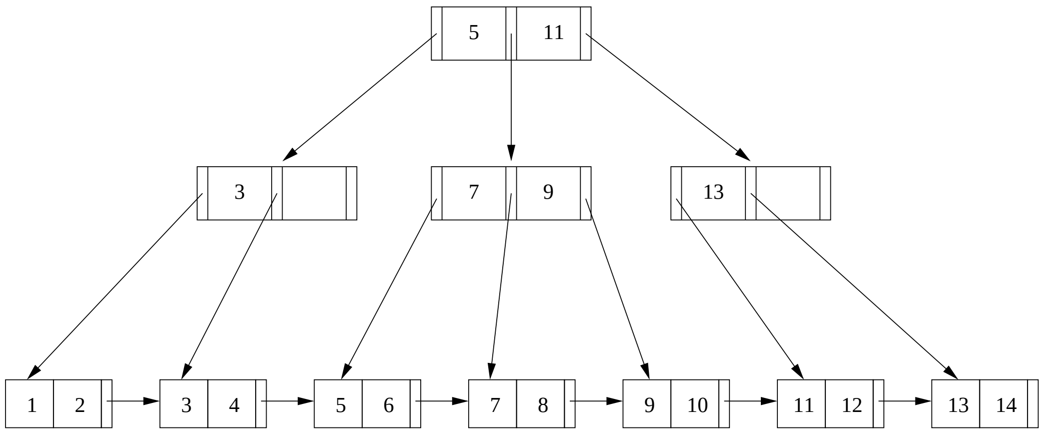


Figure 2.10: An example B+ Tree containing the values from 1 to 14 inclusive.

Searching the B+ Tree for a value, v_s, begins at the root and proceeds to a leaf node by following pointers in the non-leaf nodes. Which pointer to follow is determined by comparing values in the non-leaf nodes to v_s. Once v_s is reached in a leaf node, sequential scanning can be done on all values in the leaf nodes greater than v_s. This allows for efficient range selection and is used to improve the efficiency of some of the operations mentioned in Section 2.2.1 by minimizing the number of disk access required in scanning the data.

Inserting a value, v_i, into a B+ Tree is performed similarly to searching for a value. The leaf node in which v_i is to be inserted is determined by searching the B+ Tree for v_i. If the leaf node is not full, v_i is inserted into the leaf node, in the position that maintains the sorted order of the values in that node. If the leaf node is full, however, the leaf is split into two nodes, dividing the points evenly among the two leaf nodes and adjusting the index to accommodate the additional leaf. If the parent of the split node cannot accommodate the additional child, additional adjustments are required. However, this is infrequent.

The B+ Tree is a well-documented storage structure with further information available in *Database Management Systems* by Ramakrishnan and Gehrke [46, chapter 10].

2.5 Linear Hashing with Reversed Bit Interleaving

Linear Hashing, first introduced by Litwin [31] and described in detail in *Database Management Systems* by Ramakrishnan and Gehrke [46, chapter 11], is a method of storing n data points in a linear amount of space, requiring linear time to retrieve a single data point. The data points are stored in an array of buckets B of length $\text{length}(B)$, with each bucket, $b_i \in B, i \in [0, \text{length}(B)-1]$, being capable of storing a predetermined number of values, $\text{size}(B)$.

Each data point is identified by a unique key $k \in K$ and is assigned to a single bucket through the use of a hashing function. If the key is a scalar, that single value can be easily fed into the hashing function to determine in which bucket the data point is to be stored. The hashing function is dependent on the current length of B (see Figure 2.11).

The hashing function is defined as follows [1]

$$h(x) = \begin{cases} x \bmod 2^{\lceil \log \mathrm{length}(B) \rceil}, & x \bmod 2^{\lceil \log \mathrm{length}(B) \rceil} < \mathrm{length}(B) \\ x \bmod 2^{\lfloor \log \mathrm{length}(B) \rfloor}, & \text{otherwise} \end{cases} \tag{2.1}$$

as it provides a balanced division of the data points when the keys are evenly distributed throughout K.

Bucket 0	Bucket 1	Bucket 2
0000	0001	0010
0100	0011	0110
1000	0101	1010
1100	0111	1110

Figure 2.11: Since the length of B is 3, we can see both cases of the hashing function in this example. The value 2 (0010 in binary) hashes to bucket 2, since 2 is less than the length of B. The value 3 (0011 in binary) hashes to bucket 3, however since that bucket does not exist yet the second case of the hashing function applies and the value 3 is hashed to bucket 1 instead.

If a data point is assigned to a bucket B_i already containing size(B) elements, a collision occurs. An overflow bucket is created and is linked with

[1]The base of all logarithms, in this thesis, is 2.

B_i to store this data point. The total number of data points assigned to a bucket B_i (including the points in any overflow buckets associated with B_i) is denoted by elements(B_i).

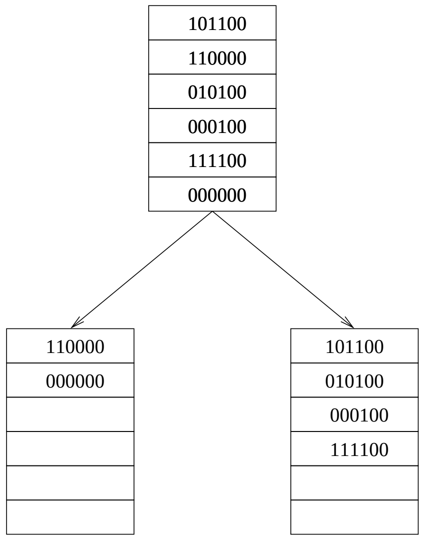


Figure 2.12: Bucket splitting example when splitting bucket B_0 into buckets B_0 and B_4. This split increases the length of B from 4 to 5.

As the buckets become saturated with points, each is split in turn to further divide the data points into a larger number of buckets. After every insertion, the usage of the data structure is determined, where usage is defined as

$$\text{usage} = \frac{\sum_i \text{elements}(B_i)}{\text{length}(B) \times \text{size}(B)} \tag{2.2}$$

Splitting occurs on a single bucket, B_s when the usage exceeds a predetermined splitting threshold, τ, where $0 < \tau \leq 1$. The variable s stores the index of the next bucket to be split, initially 0. After splitting B_s, s is increased by 1 or reset to 0 if $s = \lfloor \frac{\text{length}(B)}{2} \rfloor$.

When a split occurs on bucket B_s, a bucket is added to the end of B and the elements in B_s are re-hashed into either B_s or $B_{\text{length}(B)-1}$, depending on if a 0 or a 1 is in the $\lceil \log \text{length}(B) \rceil$ bit position of the value passed to the hashing function (see Figure 2.12).

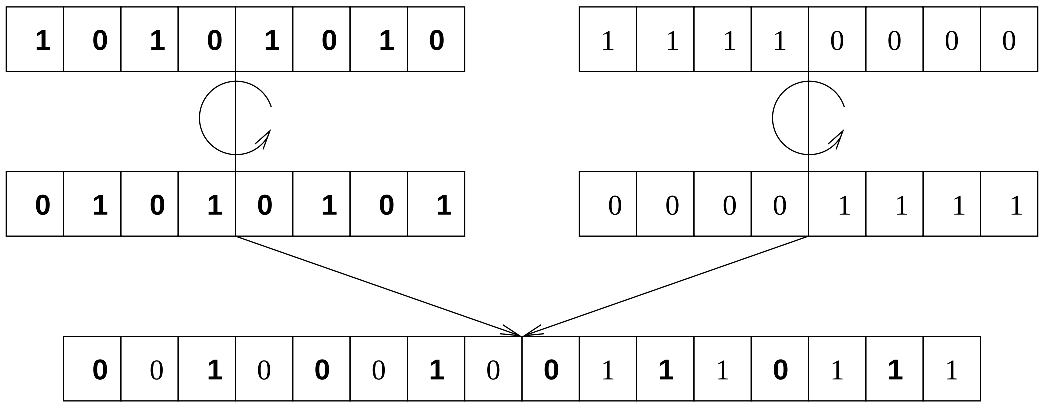


Figure 2.13: Reverse bit interleaving. First, each dimension's bit values are reversed followed by the interleaving process.

When each data point's key, its unique identifier, is multidimensional, the dimension values are bit-interleaved to give a single value that can be passed to the hashing function (see Figure 2.13). The disadvantage to hashing in this manner is that $h(x)$ will assign data points to buckets by considering the least significant bits of k. In this way the ordering of the keys is not preserved. Thus, when a range query is issued, all buckets must be scanned for keys that are within the range.

To preserve the order of the keys, one can interleave the reversed bits of each dimension before evaluating the hashing function. This works because

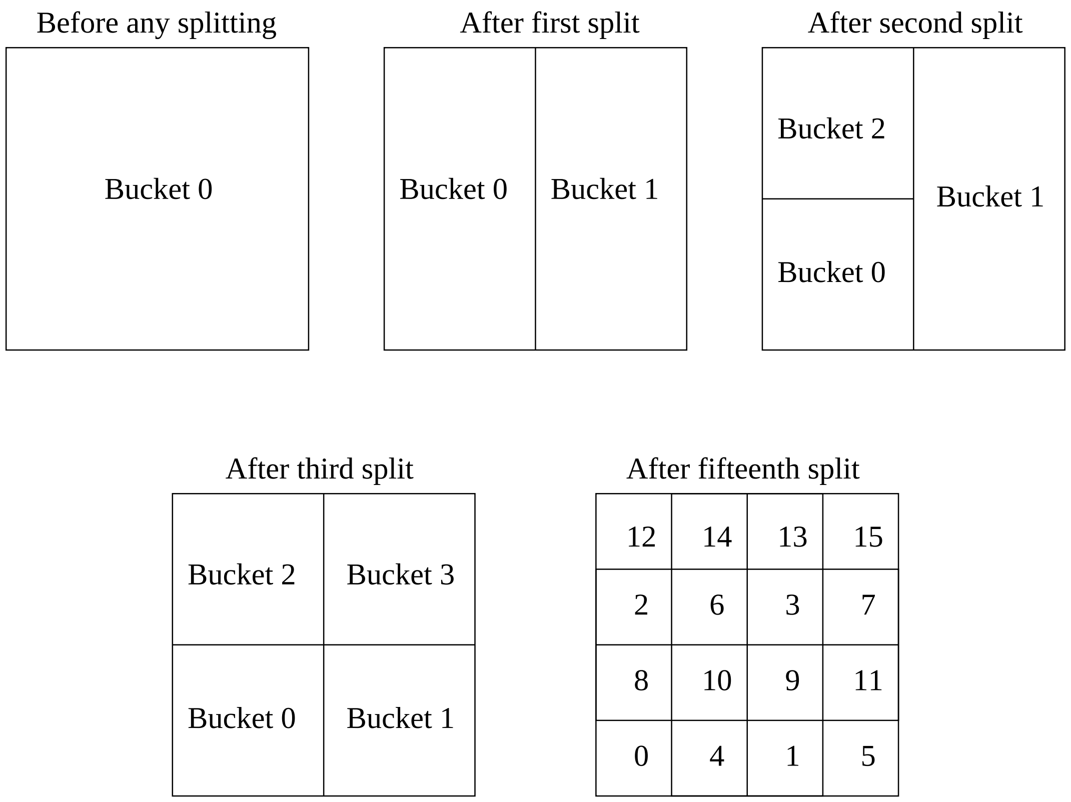


Figure 2.14: Spatial division of a two dimensional space using reversed bit interleaving

the most significant bits of the keys are used in determining the bucket for insertion [5, 39, 41]. An example of reversed bit interleaving is given in Figure 2.13. Reverse bit interleaving has the effect of equally splitting the space defined by each bucket (see Figure 2.14). As buckets are repeatedly split, they are divided along each of the possible dimensions in turn.

When a range query is issued many of the buckets may be ignored since it would be impossible for a key within the query range to be hashed to those buckets [40]. Thus only a subset of the buckets, $BQ \subseteq B$ require scanning for keys within the query range. For example in Figure 2.15, only buckets $BQ = \{0,\ 1,\ 2,\ 34,\ 6,\ 8,\ 9,\ 12\}$ require scanning.

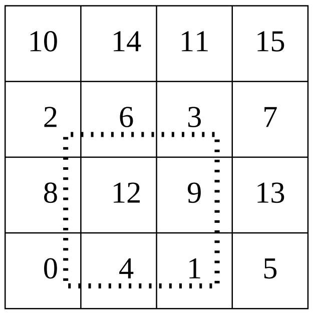


Figure 2.15: Example range query

Linear hashing with reversed bit interleaving (LHWRBI) has linear storage requirements as well as constant expected insertion and retrieval operation time. The expected complexity of range queries is proportional to length(BQ).

2.6 Efficiency Metrics

The efficiency of each storage method is evaluated by two main measures. The first is the speed at which storage methods perform operations (operation speed). These operations include simple operations such as inserting and retrieving a single data point as well as the more complicated data cube operations previously introduced in Section 2.2.1. Secondly, the size that the storage method occupies on disk is considered. Since disk space is finite, storage structures requiring less disk space are preferred.

These two measures are expected to be related. Increased operation speed is often achieved by additional indexing or a more complicated structuring of the data requiring more space on disk. For example a “Range Tree” is more efficient than a “k-d Tree” at evaluating range queries, however Range Trees have larger storage requirements than k-d Trees [3].

These metrics provide the basis by which all of the storage methods are compared. Through various experiments, the conditions under which each storage method shows promise can be derived.

Chapter 3

Implementations of Multidimensional Storage Methods

This chapter defines the specific implementation details of the five main storage methods being investigated. Each storage method is implemented in the object oriented language C++. This implementation is an excellent candidate for object oriented programming as there is a clear division of data (keys and values) as well as operations that can be applied to the data(`put()`, `get()`, `slice()`, `dice()`, and so on). There also exists a hierarchical nature to the implementation which can be exploited using an object oriented programming language.

The storage methods are chosen to examine the advantages and disadvantages of structuring the data using each of the three general storage schemes; direct mapping, tree, and hashing. These schemes are compared with a traditional relational database management system (RDBMS). Each method must be able to perform two basic storage operations: `put()` and `get()`, as well as three OLAP operations: `slice(), dice()`, and `rollup()`.

3.1 Multidimensional Cube

The multidimensional cube provides a basic foundation for all of the storage methods. In the object-oriented implementation environment of C++, the `MultidimensionalCube` class can be considered as the base class from which all storage methods inherit their OLAP functionality. Each storage method is required only to implement functionality for the two basic storage operations and an iterator to traverse through the data points that are stored. This library can, using those two operations and the iterator, perform `slice(), dice(), rollup()`, and `partial_rollup()` operations (see the class diagram [4] for `MultidimensionalCube` in Figure 3.1). Descriptions of the algorithms used to implement these operations follow. Each storage method may override the following algorithms if it is possible for that method to be implemented more efficiently, taking advantage of the properties of that storage structure.

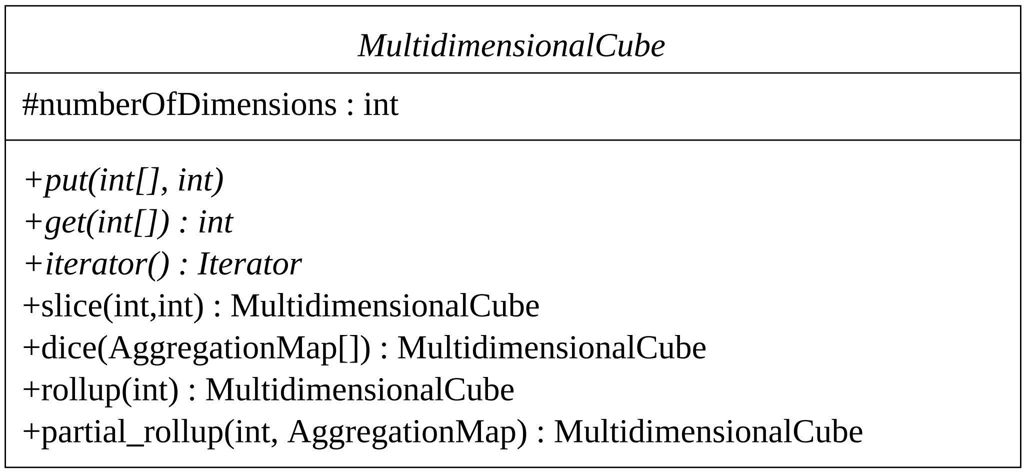

MultidimensionalCube
#numberOfDimensions : int
+put(int[], int)
+get(int[]) : int
+iterator() : Iterator
+slice(int,int) : MultidimensionalCube
+dice(AggregationMap[]) : MultidimensionalCube
+rollup(int) : MultidimensionalCube
+partial_rollup(int, AggregationMap) : MultidimensionalCube

Figure 3.1: Unified modelling language class diagram of `MultidimensionalCube`

3.1.1 Iterator

Iterators are used to move through some or all of the data points within a storage structure. They are initialized with a starting key, $s = [s_1, s_2, \ldots, s_d]$, and an ending key, $e = [e_1, e_2, \ldots, e_d]$. The iterator then moves successively to each of the keys, k, contained within the hyper-rectangle $[s, e)$[1]. The order of visitation depends on the specific implementation of each storage method's iterator.

Specialized iterators are, at times, more efficient. Iterating through specific subsets of the data points can be done more efficiently by exploiting the organization of the data. One example would be an iterator designed to slice the first dimension of a lexicographically sorted set of data points. In

[1] $[s, e) = [s_1, e_1) \times [s_2, e_2) \times \cdots \times [s_d, e_d)$

this case it would only be required to return a single contiguous range of the sorted set, eliminating the need of examining the keys of all data points.

3.1.2 Slice

The slice implementation follows directly from its description in Section 2.2.1.1 and can be described algorithmically as follows in Algorithm 1.

Algorithm 1 Algorithm to slice a multidimensional cube.

```
**INPUT:** a $d$-dimensional cube $C$, a dimension to slice $d_{\text{slice}}$, and a value to slice at $v_{\text{slice}}$
**OUTPUT:** a $(d-1)$-dimensional cube $C_{\text{slice}}$
Create a new cube $C_{\text{slice}}$ with $d-1$ dimensions
**for** $i \in \{1, \ldots, d_{\text{slice}}-1, d_{\text{slice}}+1, \ldots, d\}$ **do**
  $s_i \leftarrow 0$
  $e_i \leftarrow 1+$ maximum index value in dimension $i$
**end for**
$s_{d_{\text{slice}}} \leftarrow v_{\text{slice}}$
$e_{d_{\text{slice}}} \leftarrow v_{\text{slice}} + 1$
$iter \leftarrow$ a new iterator of $C$ with a range $[s, e)$
**while** $iter$ has next value **do**
  $k \leftarrow iter$'s current key with dimension $d_{\text{slice}}$ removed
  $v \leftarrow iter$'s current value
  put value $v$ into $C_{\text{slice}}$ at position $k$
**end while**
return cube $C_{\text{slice}}$
```

3.1.3 Dice

The dicing operation is used to select a subset of the cube by including single or multiple ranges in one or more dimensions (see Algorithm 2). This selection process is implemented using a specialized form of an *aggregation map*.

Aggregation maps are used to describe the hierarchy within a single dimension, allowing various items within a dimension to be grouped together. The dicing algorithm uses aggregation maps to select values from each dimension. Each selected dimension value v is contained within the map along with a singleton list representing the new index $l(v)$ of v in the resultant diced cube. Each singleton list is unique in the map since no two dimension values can be mapped to the same location in the diced cube. This insures that no aggregation is performed. Each dimension of the key would have its own aggregation map of unique singleton lists, allowing the dice to be selective in multiple dimensions simultaneously (see Figure 3.2).

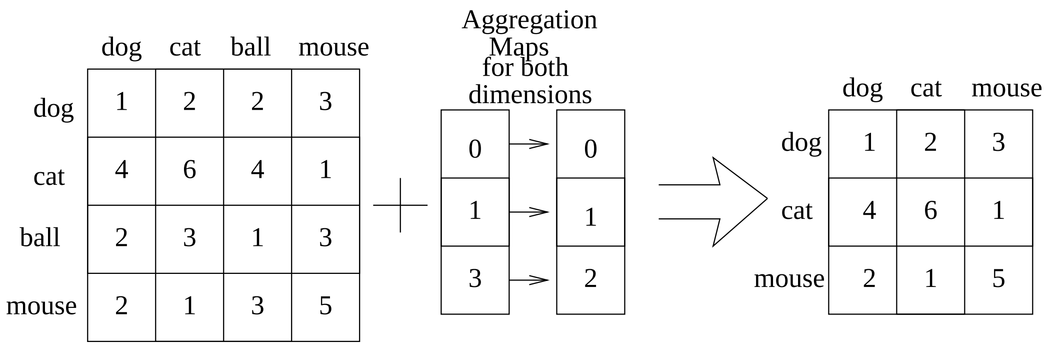


Figure 3.2: Dicing a 2-d cube

3.1.4 Roll-Up

The roll-up implementation follows directly from its previously discussed description in Section 2.2.1.3 and is described in Algorithm 3. It should be noted that only aggregation by sum is considered in this investigation. The

Algorithm 2 Algorithm to dice a multidimensional cube.

```
**INPUT:** a $d$-dimensional cube $C$, an array of aggregation maps $M$ where
$m_i \in M$
**OUTPUT:** a $d$-dimensional cube $C_{\text{dice}}$
Create a new cube $C_{\text{dice}}$ with $d$ dimensions
for $i \in \{1, \ldots, d\}$ do
    $s_i \leftarrow 0$
    $e_i \leftarrow 1+$ maximum index value in dimension $i$
end for
$iter \leftarrow$ a new iterator of $C$ with a range $[s$,$e)$
while $iter$ has next value do
    $k \leftarrow iter$'s current key
    if each dimension value in $k$ has a mapping defined for it in $M$ then
        $k_{\text{new}} \leftarrow k$ with all dimensions converted using the mappings defined
        in $M$
        $v \leftarrow iter$'s current value
        put value $v$ into $C_{\text{dice}}$ at position $k_{\text{new}}$
    end if
end while
return cube $C_{\text{dice}}$
```

following algorithms reflect this fact.

3.1.5 Partial Roll-Up

Partial roll-ups are much more difficult to implement than the general roll-up operation since partial roll-ups do not completely summarize a dimension, they simply roll-up a dimension's values based on the hierarchy of that dimension. An example of a partial roll-up would be aggregating a dimension of words based on their lengths. In this instance the values stored at multiple words with equal lengths would be aggregated together, resulting in a smaller cube with the same dimensionality.

Algorithm 3 Algorithm to roll-up a multidimensional cube.

```
**INPUT:** a $d$-dimensional cube $C$, a dimension to roll-up $d_{\text{roll-up}}$
**OUTPUT:** a $(d-1)$-dimensional cube $C_{\text{roll-up}}$
Create a new cube $C_{\text{roll-up}}$ with $d-1$ dimensions
for $i \in \{1, \ldots, d\}$ do
    $s_i \leftarrow 0$
    $e_i \leftarrow 1+$ maximum index value in dimension $i$
end for
$iter \leftarrow$ a new iterator of $C$ with a range $[s,e)$
while $iter$ has next value do
    $k \leftarrow iter$'s current key with dimension $d_{\text{roll-up}}$ removed
    $v_1 \leftarrow iter$'s current value
    $v_2 \leftarrow$ get value in $C_{\text{roll-up}}$ at position $k$
    put value $v_1 + v_2$ into $C_{\text{roll-up}}$ at position $k$
end while
return cube $C_{\text{roll-up}}$
```

To support a partial roll-up, a mapping of a dimension's previous values to their new values must be created. This is achieved using the aggregation map previously mentioned in Section 3.1.3.

3.2 Flat Binary File

Flat Binary File (FBF) is a storage mechanism identical to that used for a multidimensional array in a typical programming language. When the FBF is created it requires the specification of the cube's shape. The shape is used to determine the number of data points that can exist within the data cube. Space is allocated on the disk to allow for the storage of each possible data point that may occur in the cube.

Algorithm 4 Algorithm to partially roll-up a multidimensional cube.

```
INPUT: a $d$-dimensional cube $C$, a dimension to partially roll-up $d_{\text{partial}}$,
and an aggregation map $m$
OUTPUT: a $d$-dimensional cube $C_{\text{partial}}$
Create a new cube $C_{\text{partial}}$ with $d$ dimensions
for $i \in \{1, \ldots, d\}$ do
  $s_i \leftarrow 0$
  $e_i \leftarrow 1+$ maximum index value in dimension $i$
end for
$iter \leftarrow$ a new iterator of $C$ with a range $[s,e)$
while $iter$ has next value do
  $k \leftarrow iter$'s current key with dimension $d_{\text{partial}}$ converted using the map-
  ping $m$
  $v_1 \leftarrow iter$'s current value
  $v_2 \leftarrow$ get value in $C_{\text{partial}}$ at position $k$
  put value $v_1 + v_2$ into $C_{\text{partial}}$ at position $k$
end while
return cube $C_{\text{partial}}$
```

Since disk addressing is one dimensional, the multidimensional cube must be flattened to a single dimension. This is done using a *row major* scheme as can be seen in Figure 3.3. The data point's key is used to calculate that data point's unique offset into the file. Since the conversion from key to offset is a one-to-one function we are able to generate the key from the offset as well. Thus, it is unnecessary to store the key with the value, saving storage space.

The ordering of the data points in the file is lexicographical, thus data points with identical first-dimension key values are in close proximity on disk. Therefore, when slicing or dicing on the first dimension the number of disk accesses is minimized as the data points are contiguous on disk. It is expected that the slice and dice operations are fastest when applied to the first dimension.

Due to limitations of our particular libraries, our implementation is only capable of indexing into a file less than 1 GiB in size. It would be possible to expand this structure into separate files to overcome this obstacle.

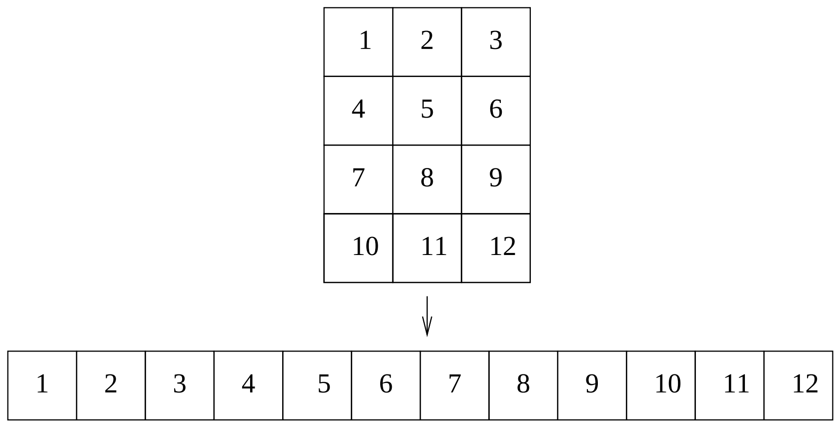


Figure 3.3: Flattening multidimensional data

3.3 B+ Tree

The Quick Database Manager (QDBM) [36] provides a library for managing databases. Specifically it includes the capability of storing data points in a B+ Tree on disk, allowing for the storage, retrieval, and sequential traversal of the data points. The QDBM library does not implement support for multidimensional cubes, therefore support has been added to extend the functionality of the library.

Any multidimensional keys are handled by converting each dimension's 4 byte integer value to bytes and concatenating the set of bytes to form a single key. This has the effect of gluing the dimensions together into a single dimension. The leaves of the B+ Tree hold the data points in lexicographically sorted order.

Lexicographical ordering provides special properties that can be exploited when slicing or dicing on the first dimension. Since the order of the sequential traversal is lexicographical, only a small subset of the data points need to be traversed to perform a slice or a dice. These two operations take into consideration this property, which should greatly improve their speed.

3.4 Rotated B+ Tree

The Rotated B+ Tree is an extension of the B+ Tree based on the work of Sarawagi and Stonebraker [47]. The basic idea is that we are able to perform operations faster on the cube by storing redundant information. When the speed of querying operations outweighs the need for minimal storage usage, this idea could prove useful.

A Rotated B+ Tree storing a k dimensional cube can be defined as a set of k B+ Trees. In each of the k B+ Trees, the key is rotated a different amount so that each dimension of the key is in the first dimension of one of the B+ Trees. Figure 3.4 shows an example of how the dimension values in a key would be rotated. In this way the advantages of slicing and dicing on the first dimension can be extended to every dimension.

The population of values into a Rotated B+ Tree can be done by two

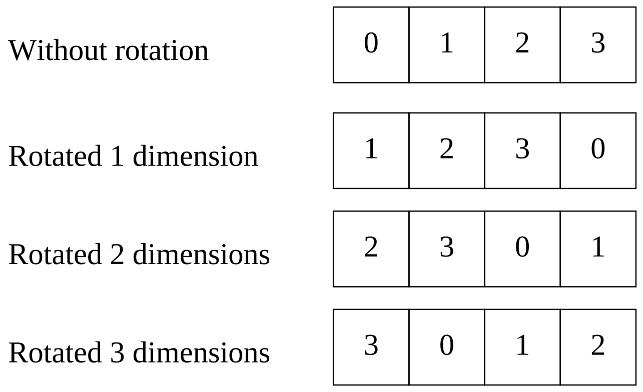


Figure 3.4: Rotating keys

methods. The obvious method is to insert the data points into each of the B+ Trees as the cube is built. In this way each of the B+ Trees in the Rotated B+ Tree are inserting their values in turn. The other method is to populate a single B+ Tree and then to add additional B+ Trees rotated on each dimension. These additional B+ Trees would be populated with the same data as the initial B+ Tree, only the keys would be rotated. The additional B+ Trees can be created eagerly at any point during or following the population of the unrotated B+ Tree, or lazily when a slice or dice is attempted on a dimension that has yet to be rotated to the first position in the key. The second method inserts the values into the additional B+ Trees in the order in which the iterator passes through the data points of the existing B+ Tree and not the original insertion order. This modified insertion order can be expected to affect the amount of time required to construct the B+ Tree.

When creating an additional B+ Tree, T, in the Rotated B+ Tree, there may exist a number of other B+ Trees already present in the Rotated B+ Tree and T could be created from any of them. The conditions used to select which existing B+ Tree is used to create T have been determined experimentally and are outlined in Section 4.4.3.1.

3.5 LHWRBI

The LHWRBI storage method discussed in Section 2.5 is implemented in two parts. First, the Bucket storage structure is created to store a fixed number, size(B), of data points. The data points are inserted contiguously into a file on disk. The Bucket is also able to create a chain of overflow Buckets in the event that more than size(B) data points are inserted into the Bucket. Second, the LHWRBI storage structure maintains the vector of primary Buckets (the overflow buckets are accessed through the primary Bucket). As data points are inserted, their keys are reverse bit interleaved and it is determined into which bucket, B_i, they are inserted. Unfortunately, in the current implementation only a certain number of bits from each dimension of the key can be used in the interleaving process, limiting the order preservation of the keys. If the data has d dimensions, only $\frac{30}{d}$ bits from each dimension can be used. The least significant bits are chosen as they are assumed to more uniformly divide the data (this assumption is later verified experimentally). A more uniform distribution of the data is achieved at the expense of disturbing the order preserving property of LHWRBI.

Slicing and dicing operations are implemented efficiently despite compromising the preservation of order. The subset of Buckets that may contain data points within the range is first determined. Only this subset of Buckets requires scanning for data points within the range of the slice or dice. The buckets that are excluded are those which, due to the level of hashing, could not possibly contain any data points within the range of the slice or dice. The number of buckets that can be excluded from the scan is reduced by only considering the $\frac{30}{d}$ least significant bits of each dimension. However, smaller ranges are unaffected by this restriction since the starting and ending points in the range are usually only different in the least significant bits. Thus, slices are not affected when considering only the $\frac{30}{d}$ least significant bits of each dimension. Dices selecting small ranges are similar.

3.6 Database Management System

Many OLAP engines are built using a RDBMS to store the data and perform queries [20, 34]. Multidimensional database management systems are also used in OLAP engines [19]. This thesis does not implement a MDBMS storage method, however, due to the lack of an available, competitive, open-source MDBMS implementation.

The implementation of the multidimensional OLAP operations previously mentioned in Section 3.1 is unnecessarily complicated, performing excessive `put()` and `get()` operations, and does not take advantage of the already well established RDBMS functionality. To provide a good basis of comparison, these operations are re-engineered to take advantage of the capabilities inherent to a RDBMS system.

The PostgreSQL [43] database management system, version 7.3.4, is used for the RDBMS storage implementation, however, many RDBMS would have the same structured query language (SQL) calls made to it when performing OLAP operations. The libpqxx library [14] is used to provide access to the database system from the C++ code. Libpqxx is chosen as the library as it interfaces with libpq, PostgreSQL's own, well established API to the database system.

A d-dimensional cube is represented in a database as a table with d attribute columns and a single measure column. A single table is used to store the cube instead of structuring the information using the star or snowflake schemas in the database. This is done for two main reasons. First, the other storage methods do not maintain this dimension information. Also, not storing the dimension information eliminates the overhead of performing joins when quering the cube.

The combination of attribute values in the database table must also be unique. When a `put()` operation is performed it must over-write any previously inserted entry at that location. To insure this outcome, every table has a *trigger* associated with it that calls a *function* to delete any previous entry in the table at the same location. Functions and triggers, though implemented in slightly different ways, are found in most RDBMS.

To insert many values in an efficient manner, a *tablewriter* object defined in the libpqxx library is used. This object allows for the direct insertion of numerous table rows into a database. Many values are inserted frequently when a new cube is created either initially or as a result of a slice, dice, or roll-up of a previously created cube.

The OLAP operations themselves over-ride the implementations defined for the multidimensional cube as many single `put()` and `get()` operations can be replaced by single, more complex SQL calls. The `slice()` and `rollup()` functions can be implemented as a single SQL call to select or aggregate as desired from other existing tables into a new table. The `dice()` function is somewhat more complex, requiring a single SQL call for each of the mappings defined by the set of aggregation maps. Multiple mappings cannot be easily grouped into a single SQL call since each mapping is independent of the others.

3.7 Validating the Implementations

To verify that the implementations of each storage method were correct, a program was written to validate each of the operations. The program created two cubes, one using the B+ Tree storage method and the other using the storage method being validated. The initial implementation of the B+ Tree had already been verified as part of the Lemur library, thus it was known to reliably produce correct results. Both cubes are loaded with the same data points, and both perform operations requiring calls to `put()`, `get()`, `slice()`, `dice()`, `rollup()` as well as the use of iterators. If both cubes achieve the same results, the storage method being tested is determined to be valid as well.

Chapter 4

Experiments

This chapter describes the experiments used to measure the relative efficiencies of each storage method. Additional experiments are included to evaluate certain specialized aspects of a single storage method. All of the experiments were conducted on a single machine using the Linux operating system, to insure consistent results.

The results generated by these experiments are used to establish a set of facts from which final conclusions are drawn. The results do not point to a single optimal storage structure. Therefore, the conditions under which each storage structure is most efficient are outlined.

4.1 Methodology

To compare the various storage methods that have been implemented, a variety of experiments were designed to evaluate the relative performance of each. The performance of each storage method was broken down to consider two main factors, size on disk and operation speed. The operations consisted of two standard storage operations, `put()` and `get()`, as well as the OLAP operations `slice()`, `dice()`, and `rollup()`.

The size of the cube was measured as the number of bytes being used in external memory for the storage structure. In some cases, such as FBF, not all of the space allocated to the storage structure on disk actually contains data. The keys for which there is no data point associated still have a position allocated on disk. Since this memory is, however, still required by the storage structure, it was included in the total size.

Time measurements were determined by the UNIX function `getrusage`, which returns the amount of user and system time that has been used by the program. These two values were summed together to provide the time measurement. Since the time values returned by `getrusage` do not include time spent waiting for I/O, an additional time measurement was made by calling `gettimeofday`, which returns the number of seconds and microseconds since the *Epoch* (00:00:00 UTC, January 1, 1970). In order for the `gettimeofday` timings to be meaningful, no unavoidable activities were concurrently being executed with the experiments. This minimized the amount of time that was spent executing other processes. The timings returned by `getrusage` are referred to as *CPU times* and those returned by `gettimeofday` as *wallclock times*. Each operation was be repeatedly executed until sufficient time had passed to insure that the error introduced by the slightly inaccurate total time was negligible.

The OLAP operations can be performed on any of the cube's dimensions, thus, when evaluating the `slice()`, `dice()`, and `rollup` operations all of the cube's dimensions are sliced, diced, and rolled up to provide a more meaningful evaluation. Since the performance of the OLAP operations on one dimension of a cube is independent of its performance on the other dimensions, we average the operation speed over different dimensions with the *geometric mean*. The geometric mean of a collection of n numbers is defined as the n^{th} root of the product of all n numbers in the collection.

4.2 Testing Platform

All of the results were obtained by creating various cubes and executing the operations on a single machine, Ennui, to ensure consistent results. Ennui was a Dell PowerEdge 6400 SMP with four Pentium III 700MHz Intel Xeon processors with 2MB cache. The fact that the testing platform was a multi-processor machine should not have affected the results as only a single processor was used by the implementation. Ennui had 2GB of RAM running at 133MHz and a SCSI hard disk formatted with the ext3 [55] file system. The hard disk spun at 10,000RPM and had an average seek time of 5.2ms [49]. Ennui ran the Red Hat operating system using version 2.4.20-30.9smp of the Linux kernel. To compile the implementations, `g++` (version 3.2.2) was used with optimization level 2.

4.3 Test Data Sources

The test data being used for these experiments consisted of five cubes that were derived from an Analogy Cube [23, 24]. Recall from Section 2.1.1 that an Analogy Cube is a four dimensional cube comprised of two word dimensions, a joining word dimension, and a book dimension. The Analogy Cube was loaded with actual literary data from a subset of eBooks found in the Project Gutenberg corpus. Each cube that was derived from the Analogy Cube was specifically chosen to test different aspects of the various storage methods, and these cubes are described as follows.

The first cube was derived from the Analogy Cube by rolling up the last three dimensions, leaving only a single word dimension. This resultant cube contains the number of times each word in that dimension appeared as the first word in an analogy. Since the original Analogy Cube was stored as a B+ Tree, the resultant `rollup()` inserted the values into the test cubes in lexicographically sorted order. The cube contained $20,177$ cells, one for each word, and had actual values in $16,758$[1] of those cells. The `slice()` and

[1]The remaining 3419 cells are undefined.

`rollup()` operations were not performed on this cube as the resultant cube would have had 0 dimensions and only a single value.

The second test case was derived from the Analogy Cube by rolling up the final book dimension, partially rolling up the two word dimensions on each word's final suffix, and partially rolling up the joining word dimension based on the length of the joining word. The resultant cube contained the analogous relationship of words with regard to their suffixes. This could be used to answer queries such as *Words ending in ly are most commonly joined to words ending in x by words of length 5* where x is unknown. The original analogy cube did not sort based on suffix, thus the inserts into the resultant cube were in unsorted order. The cube contained $18 \times 18 \times 6 = 1,944$ cells and had 761 actual values.

The third cube was derived from the Analogy Cube by rolling up the final book dimension and partially rolling up the first three dimensions on the length of the words. The resultant cube contained the analogous relationship between words based on their lengths. This could be used to answer questions such as *Words of length 4 are to words of length 5 as words of length 10 are to words of length x*, where x is unknown. The original cube was not sorted based on word length, therefore the `rollup()` inserted the values into the test cubes in an unsorted order. The cube contained $16 \times 16 \times 6 = 1,536$ cells and had 951 actual values.

The fourth test case was derived from the Analogy Cube by simply rolling up on the final book dimension. The resultant cube contained the information required to answer analogy questions such as *dog is to cat as train is to x*, where x is unknown. The `rollup()` operation again inserted values into the storage structures in lexicographical order. The cube contained $20,177 \times 20,177 \times 40 = 16,284,453,160$ cells and had $295,132$ actual values.

The final cube was derived from the Analogy Cube by rolling up the book dimension and partially rolling up the remaining dimensions five levels using WordNet hypernyms. The original Analogy Cube was not ordered based on WordNet groupings thus, the `rollup()` operation inserted the values in unsorted order. The cube defined $3,748 \times 3,748 \times 38 = 533,805,152$ possible cells and actually contained $506,908$ values.

These test cubes were initially created using the Lemur library. During their creation, the order the points were inserted into each test cube was recorded by logging the key and value for each `put()` operation that was performed. This allowed the experiments to re-create the test cubes, inserting the points in the same order. The results of the `put()` operation time was therefore, the average amount of time required to insert a single point into the cube. The `get()` operation time was calculated by averaging the retrieval time of the values at all keys that were logged during the original creation of the cube.

The logged keys were also used when performing the `slice()` and `dice()` operations. When slicing, a random key from the set of logged keys was selected and the `slice()` was performed on dimension i using the selected key's i^{th} dimension value. Similarly, when dicing, two random keys from the set of logged keys were selected and the `dice()` was performed on dimension i, selecting all values between they two keys' i^{th} dimension values.

4.4 Results

The following sections present the observed operation speeds of the `put()`, `get()`, `slice()`, `dice()`, and `rollup()` operations for each storage method. These results are then summarized to determine the relative efficiency of the storage methods. Finally, the amount of disk space required by each of the methods is presented.

4.4.1 Flat Binary File

The Flat Binary File storage method appears highly efficient in both operation speed (see Tables 4.1 and 4.2) and use of disk space (this is discussed later in Section 4.4.7) when the cubes are adequately dense and not excessively large. Disk space is wasted for those undefined areas in the multidimensional array. The amount of space wasted by undefined values is offset, however, since the keys of each data point need not be stored with their corresponding values. Our implementation of the flat binary file is unable to accommodate extremely large cubes, thus, test cubes 4 and 5 are exempted from the following experiments.

The `put()`, `get()`, and `slice()` operation times appear to be primarily computationally expensive, as their wallclock timings were approximately

Cube	N	put	get	slice	dice	roll-up
1	$16,758$	1.87×10^{-7}	1.69×10^{-7}	—	1.42×10^{-2}	—
2	761	2.90×10^{-7}	3.05×10^{-7}	4.51×10^{-4}	9.33×10^{-4}	1.45×10^{-3}
3	943	2.94×10^{-7}	2.71×10^{-7}	4.65×10^{-4}	1.13×10^{-3}	1.99×10^{-3}

Table 4.1: Flat Binary File operation speed (CPU seconds)

Cube	N	put	get	slice	dice	roll-up
1	$16,758$	1.87×10^{-7}	1.70×10^{-7}	—	5.26×10^{-2}	—
2	761	2.90×10^{-7}	2.83×10^{-7}	4.42×10^{-4}	1.46×10^{-2}	1.28×10^{-2}
3	943	2.90×10^{-7}	2.83×10^{-7}	4.95×10^{-4}	1.46×10^{-2}	1.29×10^{-2}

Table 4.2: Flat Binary File operation speed (wallclock seconds)

identical to their CPU timings[2]. This indicates that an insignificant amount of the operation time was the result of I/O operations. The `dice()` and `rollup()` operations, however, appear to be I/O intensive as their wallclock timings were considerably slower (89 percent of the total time was not counted as CPU time).

Slicing different dimensions does not appear to have considerable effect on the performance of the `slice()` operation (see Tables 4.3 and 4.4). Since the cubes are small, they fit nicely on very few disk pages. The nature of literary cubes appears to be such that a cube large enough to take advantage of disk locality would be rather sparse, making other structures more appealing since their sizes are dependant on the number of data points and not the shape of the cube (described in more detail later in Table 4.26).

Slicing the final dimension is more expensive than slicing the first two

[2]In some cases the wallclock timings were less than the CPU timings, indicating a slightly inaccurate measurement of time.

due to the shape of the cubes being sliced. Both cubes 2 and 3 have only 6 possible values in their final dimension. Slicing on the final dimension is more expensive for two main reasons. First, slicing on the final dimension forces the cube to iterate through more data points than a `slice()` on the either of the other two dimensions. For example, slicing on dimension 0 or 1 of cube 2 (which has shape $18 \times 18 \times 6$) would iterate though $18 \times 6 = 108$ data points while slicing on the final dimension would iterate through $18 \times 18 = 324$ data points. These additional iterations account for some of the additional time required to slice the final dimension of these cubes. Second, the cube being created to store the `slice` is larger when slicing the final dimension. The slicing times on the final dimension are not three times slower, since the resultant cube creation time, and initially navigating to the new cubes' location on disk, are also factors of the total time.

Cube	0	1	2
2	3.60×10^{-4}	4.00×10^{-4}	4.80×10^{-4}
3	4.40×10^{-4}	4.40×10^{-4}	5.20×10^{-4}

Table 4.3: FBF slicing times on different dimensions (CPU seconds)

Cube	0	1	2
2	4.78×10^{-4}	4.82×10^{-4}	5.72×10^{-4}
3	4.73×10^{-4}	4.77×10^{-4}	5.58×10^{-4}

Table 4.4: FBF slicing times on different dimensions (wallclock seconds)

When slicing a Flat Binary File, all elements that could possibly exist in the slice must be examined, and not only the points that do exist. In a large sparse cube this would would result in many unnecessary operations reducing the efficiency of the `slice()`, `dice()`, and `rollup()` operations.

4.4.2 B+ Tree

The B+ Tree storage method proves to be highly versatile, balancing all important aspects of operational efficiency (see Tables 4.5 and 4.6). The `put()` and `get()` operations are relatively fast, even when the number of items in the structure is large. As the number of data points being stored increases it is notable that the time required for insertions and retrievals does increase, because the path from root to leaf increases in length.

Cube	N	put	get	slice	dice	roll-up
1	$16,758$	3.27×10^{-6}	3.67×10^{-6}	—	9.40×10^{-2}	—
2	761	3.14×10^{-6}	2.74×10^{-6}	5.19×10^{-3}	1.08×10^{-2}	1.23×10^{-2}
3	951	3.46×10^{-6}	3.05×10^{-6}	4.58×10^{-3}	1.10×10^{-2}	1.67×10^{-2}
4	$295,132$	9.85×10^{-6}	8.60×10^{-6}	3.81×10^{-1}	4.73×10^{0}	8.75×10^{0}
5	$506,908$	3.32×10^{-5}	2.49×10^{-5}	1.26×10^{0}	7.82×10^{0}	1.14×10^{1}

Table 4.5: B+ Tree operation speed (CPU seconds)

Cube	N	put	get	slice	dice	roll-up
1	$16,758$	3.34×10^{-6}	3.75×10^{-6}	—	2.89×10^{-1}	—
2	761	3.20×10^{-6}	2.81×10^{-6}	8.05×10^{-2}	1.59×10^{-1}	1.65×10^{-1}
3	951	3.51×10^{-6}	3.14×10^{-6}	9.80×10^{-2}	2.01×10^{-1}	2.14×10^{-1}
4	$295,132$	9.79×10^{-6}	8.66×10^{-6}	9.60×10^{-1}	4.88×10^{0}	8.90×10^{0}
5	$506,908$	2.97×10^{-5}	2.73×10^{-5}	1.68×10^{0}	8.05×10^{0}	1.16×10^{1}

Table 4.6: B+ Tree operation speed (wallclock seconds)

For larger cubes, it is exceptionally fast to `slice()` on the first dimension (see Tables 4.7 and 4.8). This is due to the locality on disk of items whose first dimension key values are identical. For small cubes it appears as though slicing is comparably efficient for all dimensions since the size of the structure is small enough to require few disk accesses despite the data points' unsorted nature.

Cube	0	1	2
2	6.00×10^{-3}	8.00×10^{-3}	6.00×10^{-3}
3	4.00×10^{-3}	8.00×10^{-3}	5.00×10^{-3}
4	4.00×10^{-3}	3.20×10^{0}	3.49×10^{0}
5	7.60×10^{-2}	5.56×10^{0}	5.01×10^{0}

Table 4.7: B+ Tree slicing times on different dimensions (CPU seconds)

Cube	0	1	2
2	9.76×10^{-2}	9.40×10^{-2}	9.59×10^{-2}
3	8.69×10^{-2}	8.67×10^{-2}	8.55×10^{-2}
4	1.04×10^{-1}	3.29×10^{0}	3.60×10^{0}
5	4.72×10^{-1}	5.67×10^{0}	5.12×10^{0}

Table 4.8: B+ Tree slicing times on different dimensions (wallclock seconds)

4.4.3 Rotated B+ Tree

The Rotated B+ Tree has similar results to those of the B+ Tree, with the improvements and expenses associated with redundant storage. These results can be found in Tables 4.9 and 4.10.

The cost of the `put()` operation is more expensive when populating all possible rotated B+ Trees while data points are inserted, since the key must be rotated and the insertion is done into one B+ Tree for every dimension contained in the key. On the first test cube there is an approximately identical insertion time for the Rotated B+ Tree and the B+ Tree since its key is a single dimension. This results in only a single B+ Tree being present in the Rotated B+ Tree storage structure and, thus, only a single, unrotated `put()` is required. There is, however, overhead encountered in checking for additional rotations. This explains the extra time required when performing `put()` operations on the first test cube.

Cube	N	put	get	slice	dice	roll-up
1	$16,758$	4.17×10^{-6}	3.71×10^{-6}	—	9.80×10^{-2}	—
2	761	1.28×10^{-5}	2.71×10^{-6}	3.11×10^{-3}	1.17×10^{-2}	1.47×10^{-2}
3	951	1.36×10^{-5}	3.12×10^{-6}	2.29×10^{-3}	1.26×10^{-2}	1.69×10^{-2}
4	$295,132$	1.91×10^{-4}	9.64×10^{-6}	3.85×10^{-2}	5.20×10^{0}	9.69×10^{0}
5	$506,908$	4.48×10^{-4}	2.68×10^{-5}	2.35×10^{-1}	8.48×10^{0}	1.30×10^{1}

Table 4.9: Rotated B+ Tree operation speed (CPU seconds)

Cube	N	put	get	slice	dice	roll-up
1	$16,758$	4.27×10^{-6}	3.76×10^{-6}	—	2.57×10^{-1}	—
2	761	1.31×10^{-5}	2.84×10^{-6}	4.57×10^{-2}	1.18×10^{-1}	1.12×10^{-1}
3	951	1.39×10^{-5}	3.15×10^{-6}	5.12×10^{-2}	1.21×10^{-1}	1.21×10^{-1}
4	$295,132$	1.95×10^{-4}	9.65×10^{-6}	1.74×10^{-1}	5.31×10^{0}	9.82×10^{0}
5	$506,908$	4.38×10^{-4}	2.68×10^{-5}	4.07×10^{-1}	8.60×10^{0}	1.31×10^{1}

Table 4.10: Rotated B+ Tree operation speed (wallclock seconds)

Insertion times for larger cubes have the additional expense of numerous disk accesses. This is due to the fact that the various B+ Trees grow large enough that they become fragmented on disk requiring longer to access each of them. This is especially evident in the `put()` operation time for test cube 5 in Tables 4.9 and 4.10. The wallclock time is approximately 3 times greater than the CPU time, suggesting a large portion of the time is spent waiting for I/O operations to complete.

The `get()` operation is similar in its complexity and operation time to a `get()` in the B+ Tree since it is always possible to do a single `get()` from the unrotated B+ Tree.

Cube	0	1	2
2	3.00×10^{-3}	4.00×10^{-3}	4.00×10^{-3}
3	2.00×10^{-3}	3.00×10^{-3}	7.00×10^{-3}
4	4.00×10^{-3}	3.00×10^{-3}	1.20×10^{0}
5	8.50×10^{-2}	7.90×10^{-2}	2.05×10^{0}

Table 4.11: Rotated B+ Tree slicing times on different dimensions (CPU seconds)

Cube	0	1	2
2	5.74×10^{-2}	5.52×10^{-2}	5.61×10^{-2}
3	5.48×10^{-2}	5.58×10^{-2}	5.24×10^{-2}
4	2.48×10^{-1}	7.16×10^{-2}	1.28×10^{0}
5	4.05×10^{-1}	2.08×10^{-1}	2.33×10^{0}

Table 4.12: Rotated B+ Tree slicing times on different dimensions (wallclock seconds)

The `slice(), dice()`, and `rollup()` operations are executed to provide the query results as quickly as possible. For this reason, only the single unrotated B+ Tree is created in the Rotated B+ Tree structure when these operations are issued in the experiments. Additional rotated B+ Trees would only be necessary if the query was adequately complicated to require multiple nested operations For example, removing two dimensions of the cube by `slice()` operations on different dimensions. Since the queries require only a single operation, only the fastest method of returning results is considered.

Slicing a Rotated B+ Tree is fast because a B+ Tree can always be selected having the slicing dimension rotated to the first dimension. This takes advantage of disk locality and iterates through the least amount of the B+ Tree possible. The `slice()` operation remains slow on the last dimension of the test cubes 4 and 5 (see Tables 4.11 and 4.12) since those dimensions are small, having one of 6 possible values. Slicing in that case results in a larger cube than slicing on the larger dimensions. In these cases the slower insertion time and size of the `slice()` result offsets the gains made by having to scan only a subset of the entire B+ Tree.

The `dice()` operation speed results are surprisingly similar to those of the B+ Tree. It was expected that the gains made by dicing on the first dimension would outweigh the cost of unrotating the keys found within the dice, however, it appears as though the two costs were approximately equal.

The `rollup()` operation times are nearly identical to those of the B+ Tree because only a single unrotated B+ Tree, T, is created in the resultant Rotated B+ Tree. T is created from the unrotated B+ Tree in the original structure, thus the operations performed are almost identical to those that were performed by the B+ Tree storage method.

4.4.3.1 Lazily Rotating B+ Trees

When rotating from a previously established B+ Tree, it appears most efficient to create an additional B+ Tree from one in which each value of the key requires shifting a single position to the right (see Table 4.13). For example, a three dimensional cube having keys $k_{\text{old}} = [k_0, k_1, k_2]$ would rotate the key to $k_{\text{new}} = [k_2, k_1, k_0]$. This has the effect of inserting data points in a very

From\To	1	2	3	4	5
0	234.95	247.82	240.37	235.72	**41.69**
1	—	244.08	245.95	244.97	250.00
2	**45.02**	—	238.63	253.15	244.58
3	244.58	**42.10**	—	238.49	241.10
4	258.72	247.82	**42.07**	—	235.66
5	237.39	242.84	239.44	**42.85**	—

Table 4.13: This table shows the amount of time required (wallclock seconds) to create a new B+ Tree rotated from a previously created B+ Tree. Rotations are most efficient when rotating from a B+ Tree with dimension i as its first dimension to one with dimension $(i+1)\text{mod}(d)$ as the new B+ Tree's first dimension.

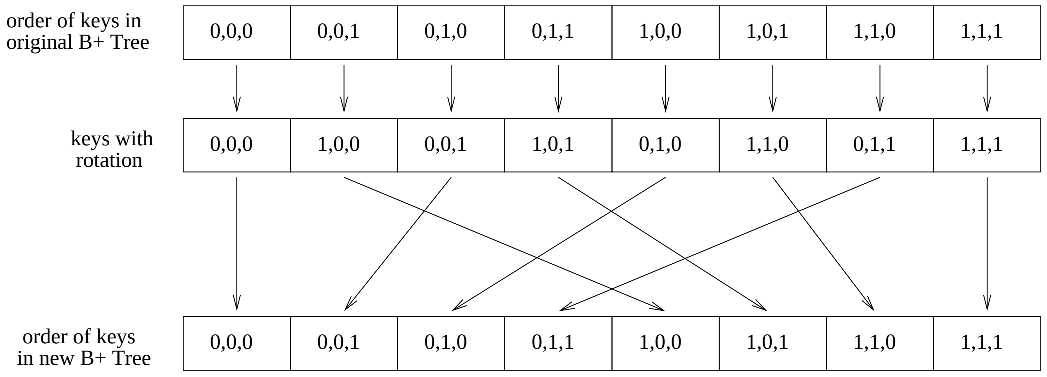


Figure 4.1: Rotating a new B+ Tree from an existing one

balanced manner, minimizing the amount of index restructuring that would generally be necessary (see Figure 4.1).

When a `slice()` is performed on a dimension, i, that has yet to be rotated to the first dimension of a B+ Tree in the Rotated B+ Tree structure, the additional B+ Tree can be lazily created before performing the `slice()`. This has the disadvantage of requiring additional time performing this initial `slice()` but has the advantage of decreasing the time required to preform any `slice()` on dimension i in the future.

In cases where there is a well defined *loading stage*[3] of the data warehouse, it may prove more efficient to create a single B+ Tree in the set of B+ Trees T_{created}, in the Rotated B+ Tree structure. From this initial, singleton set, T_{created}, the additional set of B+ Trees, $T_{\text{additional}}$, can be created by rotating a B+ Tree from T_{created}. As each B+ Tree from $T_{\text{additional}}$ is created, it is added to the set T_{created}. In this manner, the trees in $T_{\text{additional}}$ can be created in such an order as to always allow the B+ Tree's creation from a tree in T_{created} by shifting the dimensions of the key a single position to the right, yielding the most efficient creation time of the additional B+ Trees. For example, considering the case of Table 4.13 in which there are 6 dimensions, assume the initially unrotated B+ Tree was created first. It would then be most efficient to create the remaining five B+ Trees in the following order, where each number represents the first dimension of the key: 5, 4, 3, 2, 1. In other words, each tree is created from the tree that was itself created in the previous iteration.

4.4.4 LHWRBI

One of the main problems with the LHWRBI storage structure is the assumption that the data points will be at random throughout the range of all possible keys. This would allow each bucket to contain approximately

[3]The *loading stage* refers to the period of time used in creating all initial cubes before any queries are issued on the data in the data warehouse.

the same number of data points. Since our data points are more clustered, additional adjustments must be made to accommodate the data.

Figures 4.2 and 4.4 clearly indicates that the majority of data points are collected within a small subset of the buckets when the hashing function interleaves $\frac{30}{d}$ bits from each dimension of the key k. This is due to the fact that the most significant bits considered when hashing are constant in cubes when the range of a dimension is small. For example, the final dimension of the fifth test cube contains only 15 values. All of these values can be indexed using only 4 bits of the key. Since 10 bits from each dimension of this test case can be used in the hashing function, $2^{10} - 2^4 = 1,008$ buckets can potentially be empty (if enough splitting occurs).

To overcome this issue, fewer bits can be considered from each dimension, resulting in a better distribution of the data points across the set of buckets (see Figures 4.3 and 4.4). By using only 15 bits in the value passed to the hashing function, the data points may still be divided over $2^{15} = 32,768$ buckets, yet we will only use the least significant $\frac{15}{d}$ bits from each dimension. When considering the fifth test cube, this has a result of creating only $2^5 - 2^4 = 16$ buckets that can potentially be empty.

A better distribution of data points has positive effects on the performance of all operations (see Tables 4.14, 4.15, 4.16, and 4.17). The `put()` and `get()` operations are considerably faster since there are fewer points in each bucket. In the case of the `put()` operation, fewer overflow buckets would need to be traversed before the insertion could happen. In the case

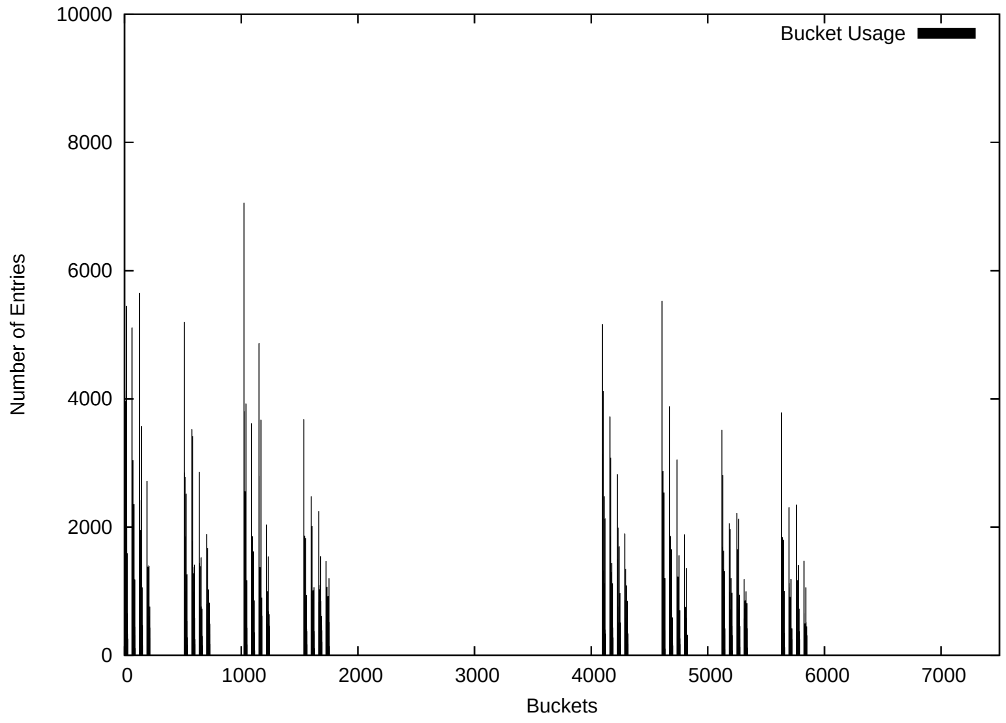


Figure 4.2: Bucket usage when interleaving 30 bits.

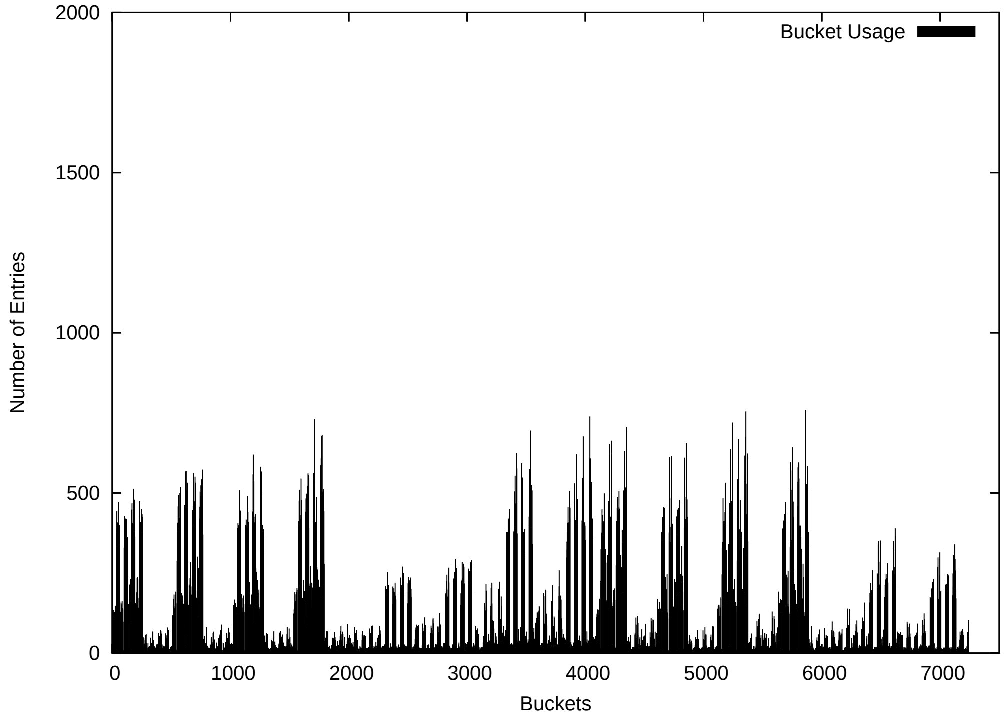


Figure 4.3: Bucket usage when interleaving 15 bits.

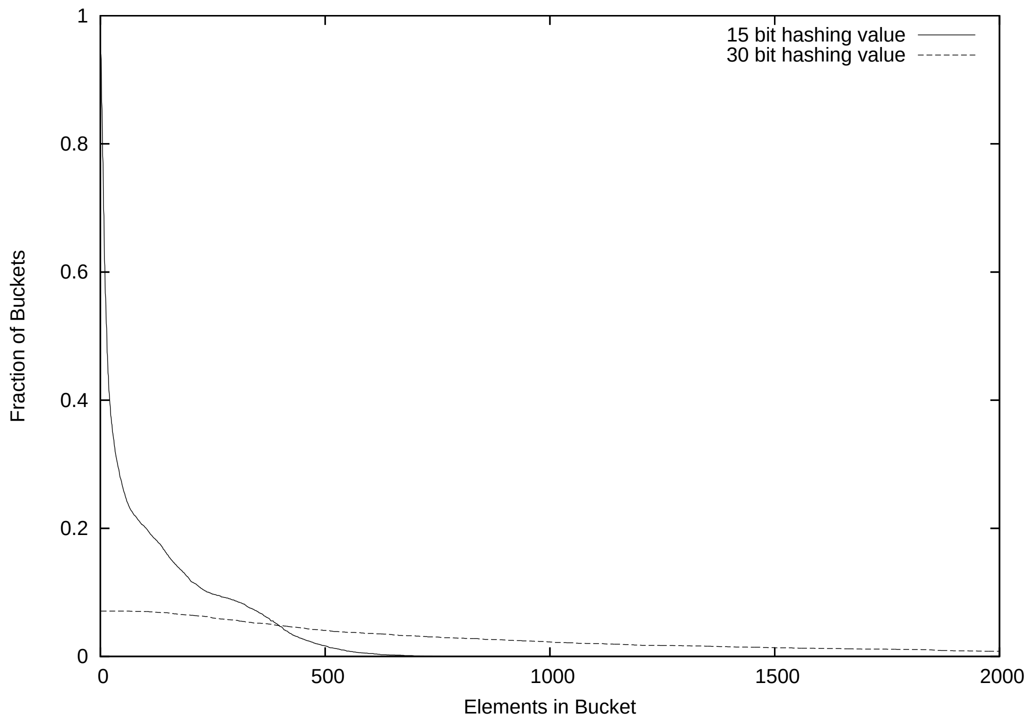


Figure 4.4: The cumulative distribution of points in buckets in the LHWRBI storage method. The two lines represent the fraction of buckets that contain at least x elements.

of the `get()` operation, fewer points in the bucket, as well as any overflow buckets, would need to be searched for a matching key. Reversed bit interleaving is also less complex when considering fewer bits as the interleaving process would consider fewer bits, resulting in fewer mathematical and bit shifting operations.

Cube	N	put	get	slice	dice	roll-up
1	16,758	1.70×10^{-3}	1.54×10^{-3}	—	1.71×10^{1}	—
2	761	1.79×10^{-5}	1.65×10^{-5}	8.67×10^{-3}	4.80×10^{-2}	4.17×10^{-2}
3	951	7.13×10^{-5}	6.79×10^{-5}	9.12×10^{-3}	1.00×10^{-1}	5.25×10^{-2}
4	295,132	6.66×10^{-4}	2.47×10^{-4}	5.65×10^{0}	1.06×10^{2}	4.74×10^{2}
5	506,908	1.10×10^{-3}	4.28×10^{-4}	3.87×10^{2}	3.91×10^{2}	4.68×10^{3}

Table 4.14: LHWRBI operation speed - 30 bit hash (CPU seconds)

Cube	N	put	get	slice	dice	roll-up
1	16,758	1.70×10^{-3}	1.54×10^{-3}	—	1.71×10^{1}	—
2	761	1.80×10^{-5}	1.65×10^{-5}	8.65×10^{-3}	4.76×10^{-2}	4.13×10^{-2}
3	951	7.12×10^{-5}	6.80×10^{-5}	9.46×10^{-3}	9.91×10^{-2}	5.30×10^{-2}
4	295,132	6.40×10^{-4}	2.44×10^{-4}	5.52×10^{0}	1.08×10^{2}	4.76×10^{2}
5	506,908	1.09×10^{-3}	4.28×10^{-4}	3.86×10^{2}	3.94×10^{2}	4.68×10^{3}

Table 4.15: LHWRBI operation speed - 30 bit hash (wallclock seconds)

Cube	N	put	get	slice	dice	roll-up
1	16,758	1.42×10^{-4}	2.39×10^{-5}	—	1.05×10^{0}	—
2	761	1.59×10^{-5}	1.18×10^{-5}	6.35×10^{-3}	3.57×10^{-2}	3.30×10^{-2}
3	951	4.89×10^{-5}	3.69×10^{-5}	7.11×10^{-3}	5.85×10^{-2}	4.58×10^{-2}
4	295,132	1.68×10^{-4}	4.64×10^{-5}	2.11×10^{-1}	3.90×10^{1}	4.33×10^{1}
5	506,908	1.86×10^{-4}	5.39×10^{-5}	1.46×10^{0}	1.11×10^{2}	6.81×10^{1}

Table 4.16: LHWRBI operation speed - 15 bit hash (CPU seconds)

Cube	N	put	get	slice	dice	roll-up
1	16,758	1.45×10^{-4}	2.58×10^{-5}	—	1.06×10^{0}	—
2	761	1.60×10^{-5}	1.17×10^{-5}	6.68×10^{-3}	3.57×10^{-2}	3.48×10^{-2}
3	951	4.88×10^{-5}	3.68×10^{-5}	7.33×10^{-3}	5.72×10^{-2}	4.87×10^{-2}
4	295,132	1.67×10^{-4}	4.67×10^{-5}	2.21×10^{-1}	4.13×10^{1}	4.43×10^{1}
5	506,908	1.85×10^{-4}	5.36×10^{-5}	1.51×10^{0}	1.13×10^{2}	6.83×10^{1}

Table 4.17: LHWRBI operation speed - 15 bit hash (wallclock seconds)

The `slice(), dice()`, and `rollup()` operations are also considerably improved when considering fewer bits. This improvement is due to a combination of two main factors. First, `slice(), dice()`, and `rollup()` require numerous `put()` operations. Since `put()` is more efficiently executed, these operations are also improved. Furthermore, `slice()` and `dice()` may also be benefitting from a more useful iterator. The iterator, when used to select a subset of the entire range of data points, first determines if any buckets are guaranteed to be outside of the range and can thus be excluded from scanning. Since the points are more evenly distributed over the buckets, it is more likely when buckets are excluded from the scan that they actually contain data points, and are not simply empty buckets.

All of the LHWRBI results are primarily CPU intensive. This fact is encouraging as processing power increases at a much faster rate than secondary storage access speeds.

Each dimension is approximately equal in terms of slicing efficiency (see Tables 4.18 and 4.19). The differences in times can be attributed to the number of data points found in the scope of the `slice()`, thus bounding the efficiency of the `slice()` operation to that of the `put()` operation. This explains why slicing the third dimension is slower than slicing the first two. The fact that slicing on different dimensions appears not to affect the efficiency of the operation eliminates the benefit of storing the dimensions in a different order, as was done in the Rotated B+ Tree.

Cube	0	1	2
2	4.00×10^{-3}	4.00×10^{-3}	1.60×10^{-2}
3	4.00×10^{-3}	5.00×10^{-3}	1.80×10^{-2}
4	3.30×10^{-2}	4.10×10^{-2}	7.41×10^{0}
5	5.02×10^{-1}	4.40×10^{-1}	1.41×10^{1}

Table 4.18: LHWRBI slicing times on different dimensions (CPU seconds)

Cube	0	1	2
2	4.64×10^{-3}	3.94×10^{-3}	1.63×10^{-2}
3	4.40×10^{-3}	4.91×10^{-3}	1.82×10^{-2}
4	3.25×10^{-2}	4.11×10^{-2}	7.74×10^{0}
5	5.11×10^{-1}	4.42×10^{-1}	1.55×10^{1}

Table 4.19: LHWRBI slicing times on different dimensions (wallclock seconds)

Varying the bucket size appears to have a large impact on the speed of most operations, without negatively impacting the size of the structure on disk. A smaller bucket size yields the most efficient results. When the bucket size becomes too small, however, the operation times increase.

The `put()` and `get()` operations (see Figure 4.5) are substantially faster with smaller bucket sizes. The improvement of the `get()` operation is expected as hashing to a specific bucket can be done in constant time, while the smaller buckets are faster to scan for the specific key. The `put()` operation encounters many more splitting operations when the bucket size is small, which would be expected to decrease the efficiency of the `put()` operation. The number of splits that occur is offset however, by the fact that each bucket created is smaller in size, requiring less disk accessing. Additional splits when the bucket size is small also has the advantage of forcing the hashing function to the less significant bits of the key, resulting in fewer overflow buckets as the cube is populated.

The `put()` operation is optimized at a bucket size of 100. At buckets sizes smaller than 100, the number of splits encountered greatly reduces the efficiency of the `put()` operation.

The `slice(), dice()`, and `rollup()` operations all appear to have proportional decreases in efficiency as the bucket size increases (see Figure 4.6). The initial difference in timings for each of the bucket sizes can be closely related to the size of the resultant cube. The size of the resultant cube is smallest for `slice()`, larger for `dice()`, and largest when performing a `rollup()`. These results are mostly explained by the efficiency of the `put()` operation. At a bucket size of 100, where `put()` achieved its best results, the `slice()`, and `dice()` operations also achieved their best results. The `rollup()` operation's best results were recorded at a bucket size of 75 because of the efficiency of the `get()` operation. Due to aggregation, many of the `put()` operations performed by the `rollup` would over-write previously inserted values in the resultant cube. This reduces the number of split operations. Since the `put()` operation's efficiency decreases faster than that of

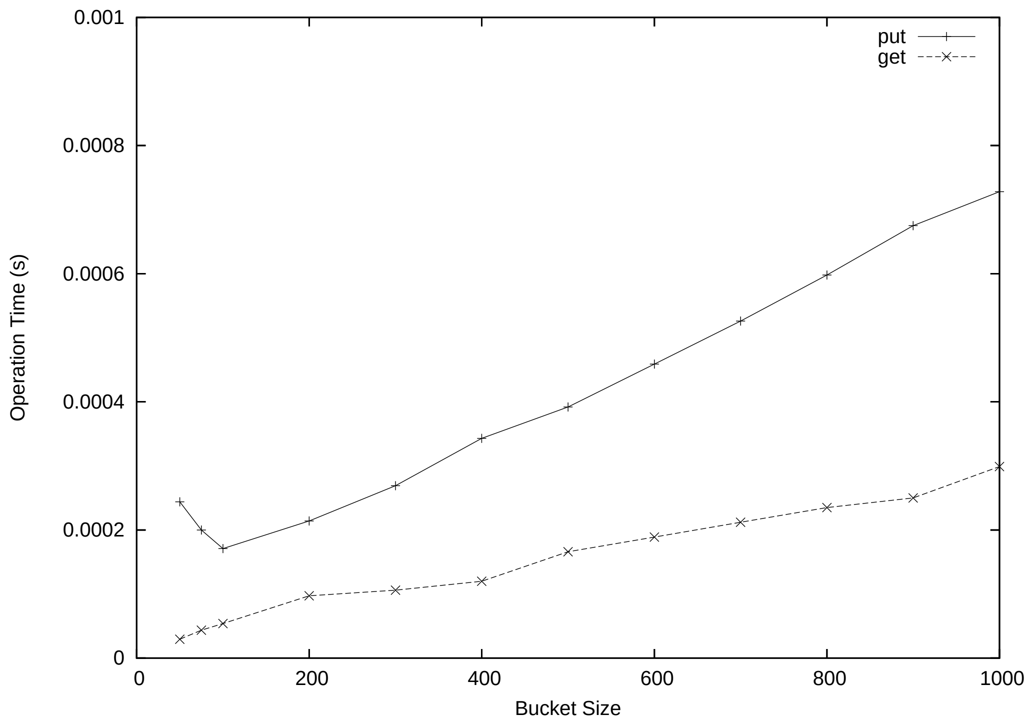


Figure 4.5: LHWRBI - Cube 5 - `put()` and `get()` operation speed with varying bucket sizes

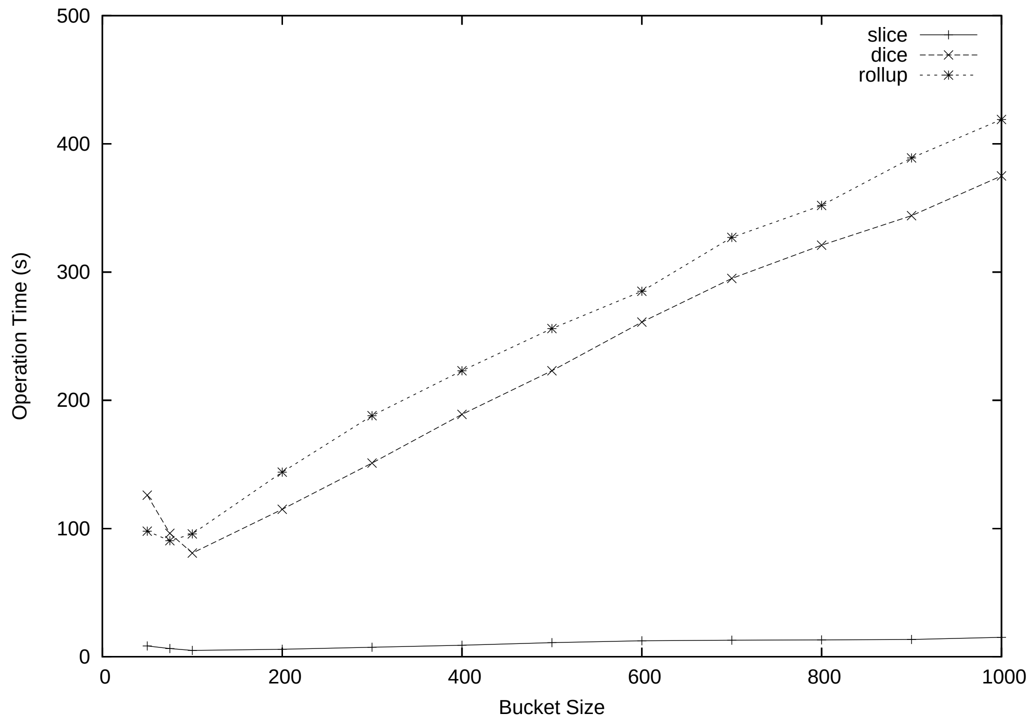


Figure 4.6: LHWRBI - Cube 5 - `slice(), dice()`, and `rollup()` operation speed with varying bucket sizes

the `get()` operation, the `rollup()` operation begins to be more expensive at bucket sizes lower than 75.

The sizes of the structure remain approximately constant (see Figure 4.7). This is expected, since each bucket contains on average $\tau \times \text{size}(B)$ data points. Since the proportion of data points to empty storage locations in the primary buckets has little variability, the total size of the structure is approximately constant. The size of the structure is not entirely constant, however, since the number of buckets influences how many bits are used in hashing, and thus the distribution of the data points among the buckets. There are additional overflow buckets in some cases if the hashing does not evenly distribute the data points. When all points had been inserted into the cube, the final bucket usage varied depending on the bucket size. This resulted in the slight variation of disk usage described in Figure 4.7.

Disk usage was actually much higher when small bucket sizes were used as the size listed was the sum of all file sizes given by the operating system in bytes which does not account for unused space at the end of disk blocks. Ennui had a block size of 4096 bytes, therefore, when the bucket size was 50, only 804 bytes were actually used in the block.

4.4.5 Database Management System

The DBMS results (see Tables 4.20 and 4.21) are slower than some of the other data structures, but most operations appear highly scalable in that the timings increase at a lower rate than the other storage structures with respect to N. The tablewriter method of populating the cubes appears highly efficient and comparable to the other methods. Performing `put()` operations in bulk is more efficient as fewer SQL queries must be issued. A single `put()` operation without using the tablewriter would require approximately the same amount of time as a `get()` operation.

The scalability of the `put()` operation speed with respect to N is promising. The B+ Tree and LHWRBI storage structures' `put()` operation time increased as the total number of data points in the cube increased. Even in

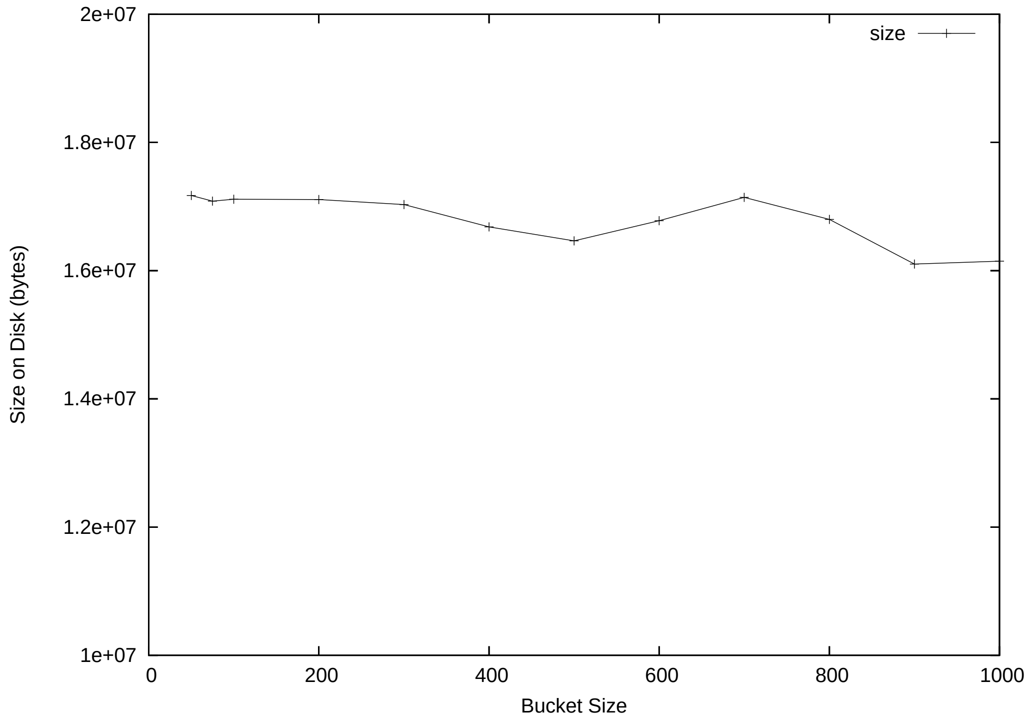


Figure 4.7: LHWRBI - Cube 5 - size on disk with varying bucket sizes

Cube	N	put	get	slice	dice	roll-up
1	$16,758$	5.97×10^{-5}	5.32×10^{-4}	—	1.45×10^{0}	—
2	761	9.23×10^{-5}	4.74×10^{-4}	1.52×10^{-3}	3.40×10^{-3}	2.10×10^{-3}
3	951	9.24×10^{-5}	4.43×10^{-4}	2.54×10^{-3}	3.21×10^{-3}	2.45×10^{-3}
4	$295,132$	9.44×10^{-5}	5.05×10^{-4}	2.22×10^{-3}	1.25×10^{0}	2.76×10^{-3}
5	$506,908$	9.35×10^{-5}	5.40×10^{-4}	2.00×10^{-3}	4.33×10^{-2}	3.47×10^{-3}

Table 4.20: DBMS operation speed (CPU seconds)

Cube	N	put	get	slice	dice	roll-up
1	$16,758$	1.45×10^{-4}	4.07×10^{-2}	—	5.62×10^{2}	—
2	761	1.85×10^{-4}	1.26×10^{-2}	1.33×10^{0}	1.19×10^{0}	1.11×10^{0}
3	951	1.83×10^{-4}	1.30×10^{-2}	1.33×10^{0}	1.27×10^{0}	1.13×10^{0}
4	$295,132$	2.07×10^{-4}	4.98×10^{-1}	2.51×10^{0}	3.07×10^{3}	9.59×10^{0}
5	$506,908$	1.98×10^{-4}	8.41×10^{-1}	3.51×10^{0}	2.24×10^{2}	1.37×10^{1}

Table 4.21: DBMS operation speed (wallclock seconds)

the LHWRBI, which has constant insertion time complexity, the disk locality resulted in longer `put()` operations when the cube became larger. Despite the scalability of the `put()` operation, it remains to be determined if it would ever be faster than the LHWRBI or B+Tree `put()` operation times.

The `get()` operation is considerably slower than the `put()` operation. Most data structures that were considered had comparable `put()` and `get()` operation times. Since `get()` operations must be a single, self contained SQL queries, the expense of connecting to the DBMS and issuing the query cannot be amortized as is done when inserting a large number of data points at once.

The `slice()` and `rollup()` operations have the advantage of being single SQL queries, while the implementation of `dice()` requires multiple SQL queries. The current definition of `dice()` allows for multiple values in any single dimension to be included. Each value in that dimension is then mapped to a new value, to minimize the maximum index in each dimension. An SQL query must be issued for each mapping that occurs and is, therefore, highly inefficient. For example, if the aggregation map contains $\{2 \rightarrow 0, 9 \rightarrow 1\}$ it would be impossible to select index 2 and 9 and have them inserted at index 0 and 1 respectively. The `rollup()` and `slice()` operations are inefficient for small cubes as there is considerable overhead in performing even a single query. On those queries where the result is easily obtainable, the overhead involved in issuing an SQL query to the DBMS results in a minor portion of the total expense required. The additional overhead is involved in creating the table used to store the result of the operation as well as the function and trigger that are associated with every table created by this storage structure.

The DBMS operation cost is primarily incurred in accesses to the DBMS itself. This is evident by the difference in CPU and wallclock timings. Since we are issuing calls to the PostgreSQL DBMS, the amount of CPU time required by PostgreSQL is not included in the CPU seconds measurement of Table 4.22. The difference in results of Tables 4.22 and 4.23 can therefore be declared as the time spent waiting for DBMS transactions which includes time PostgreSQL spent waiting for I/O.

Slicing results indicate that it is preferable to perform a `slice()` on the first dimension. The differences in `slice()` operation times on different dimensions increases as the number of data points in the cube increases.

Since `slice()` is achieved by issuing a single SQL call, all of the CPU timings are approximately equal. The wallclock timings indicate the amount of time required by the PostgreSQL DBMS to process and complete the slicing query.

Cube	N	0	1	2
2	1.76×10^{-3}	2.45×10^{-3}	2.24×10^{-3}	
3	1.95×10^{-3}	1.76×10^{-3}	2.06×10^{-3}	
4	1.79×10^{-3}	1.89×10^{-3}	2.71×10^{-3}	
5	2.07×10^{-3}	2.25×10^{-3}	2.39×10^{-3}	

Table 4.22: DBMS slicing times on different dimensions (CPU seconds)

Cube	0	1	2
2	1.39×10^{0}	1.42×10^{0}	1.46×10^{0}
3	1.44×10^{0}	1.48×10^{0}	1.51×10^{0}
4	2.00×10^{0}	2.22×10^{0}	3.86×10^{0}
5	2.44×10^{0}	3.05×10^{0}	5.60×10^{0}

Table 4.23: DBMS slicing times on different dimensions (wallclock seconds)

4.4.6 Operation Speed Comparison

To evaluate the relative operational efficiency of the five operations being evaluated for each storage method, two comparisons are performed. The first comparison includes plotting the operation speeds for each storage method against the various sizes of the test cubes. From these plots a visual comparison of the operation speeds is easily obtained. Finally, the operation speeds will be normalized so that the timings received from the various test cubes can be combined. This allows each of the storage method's operations to be compared. Only the wallclock timings are considered in these comparisons as they take into consideration both the CPU time and the time spent waiting for I/O operations to occur.

The efficiency of each storage method's `put()` operation is presented in Figure 4.8. The FBF storage method is faster than any of the other storage methods for the smaller cubes it is capable of storing. The LHWRBI and DBMS storage methods appear to converge to similar `put()` operation times for larger cubes. While B+ Trees appear to have the fastest insertion time for smaller cubes, the rate at which the B+ Tree's insertion time is increasing with the cube size suggests that eventually the `put()` into LHWRBI and DBMS will be faster.

The efficiency of each storage method's `get()` operation is presented in Figure 4.9. The FBF storage method is again much faster than any of the other storage methods for the cubes that are small enough for it to store. The DBMS storage method is especially weak when performing a `get()` since it requires a single SQL call to be issued to the database which carries a notable amount of overhead. The B+ Tree and Rotated B+ Tree storage methods have almost identical `get()` operation times since they perform almost identical operations as was previously discussed in Section 4.4.3. The LHWRBI storage method performs `get()` operations slower than the B+ Tree, however, it appears as though the B+ Tree's `get()` operation time is increasing faster as N increases than that of the LHWRBI storage method. It is possible that the LHWBRI storage method will be faster for larger N than was considered in this exploration.

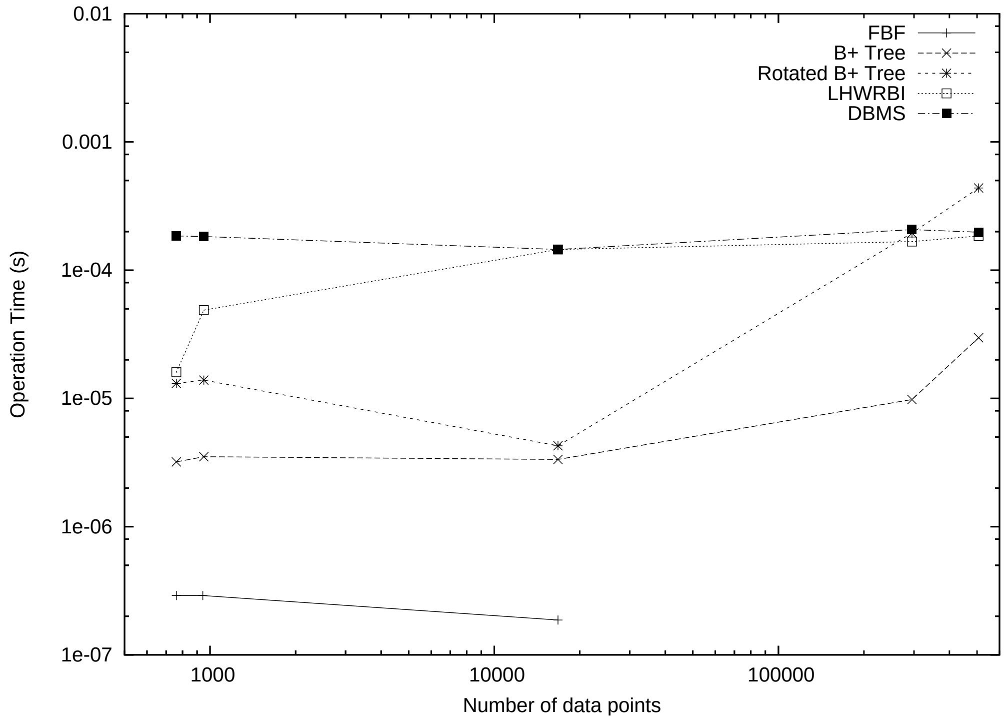


Figure 4.8: Speed of the `put()` operation

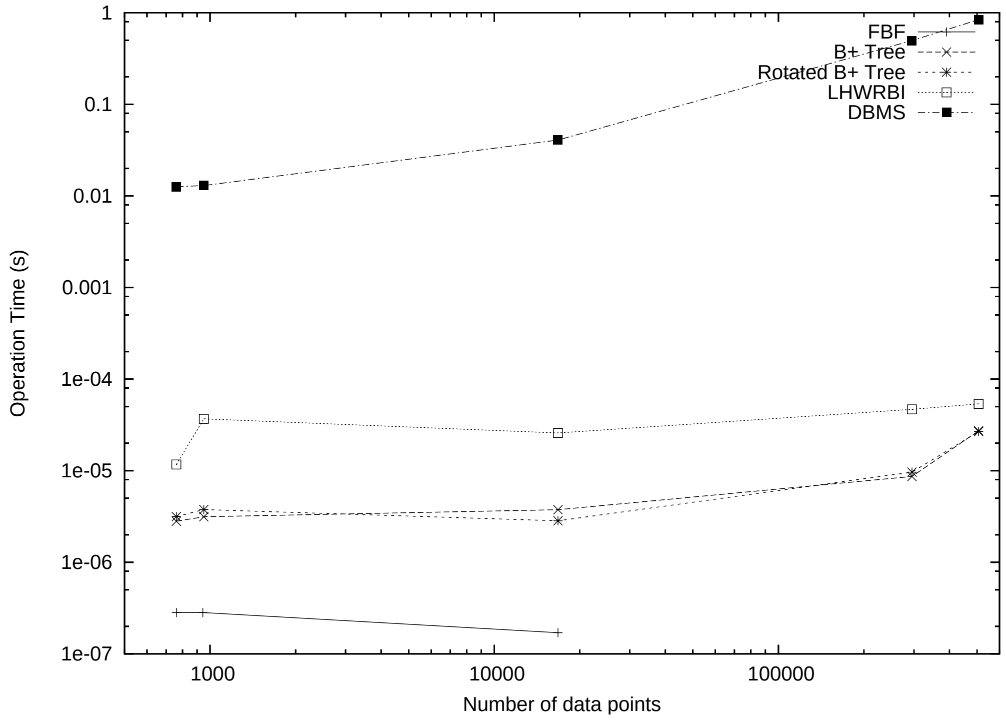


Figure 4.9: Speed of the `get()` operation

The efficiency of each storage method's `slice()` operation is presented in Figure 4.10. The FBF storage method is able to `slice()` faster than the other storage methods, in large part due to that method's fast `put()` and `get()` operation speeds. The LHWRBI storage method appears more efficient at slicing smaller cubes than remaining storage methods. As this storage structure has higher `put()` and `get()` operation times than the B+ Tree storage method, the extra efficiency must be attributed to the initial creation of the storage structure in memory or a more efficient iterator. The Rotated B+ Tree is most efficient for large cubes as it is always possible to slice on the first dimension of one of the structure's B+ Trees, greatly improving the performance of the iterator. The DBMS `slice()` operation speed shows the least amount of increase as N becomes large.

One puzzling result was the `slice()` operation performed on the final test cube using the LHWRBI storage method. This storage method had the largest increase in operation time from the fourth test cube to the fifth. The reason for this result is due to the distribution of data within the final test cube. When the result of the `slice()` is stored using the LHWRBI storage method, the data points are not evenly distributed throughout the buckets. In fact, during construction up to nine overflow buckets were created in a single chain to accommodate the clustered points. When points are inserted into a bucket having an overflow bucket an additional disk access is required since each bucket is contained in its own file. Therefore, when a bucket has a chain of nine overflow buckets, a `put()` operation is ten times as expensive

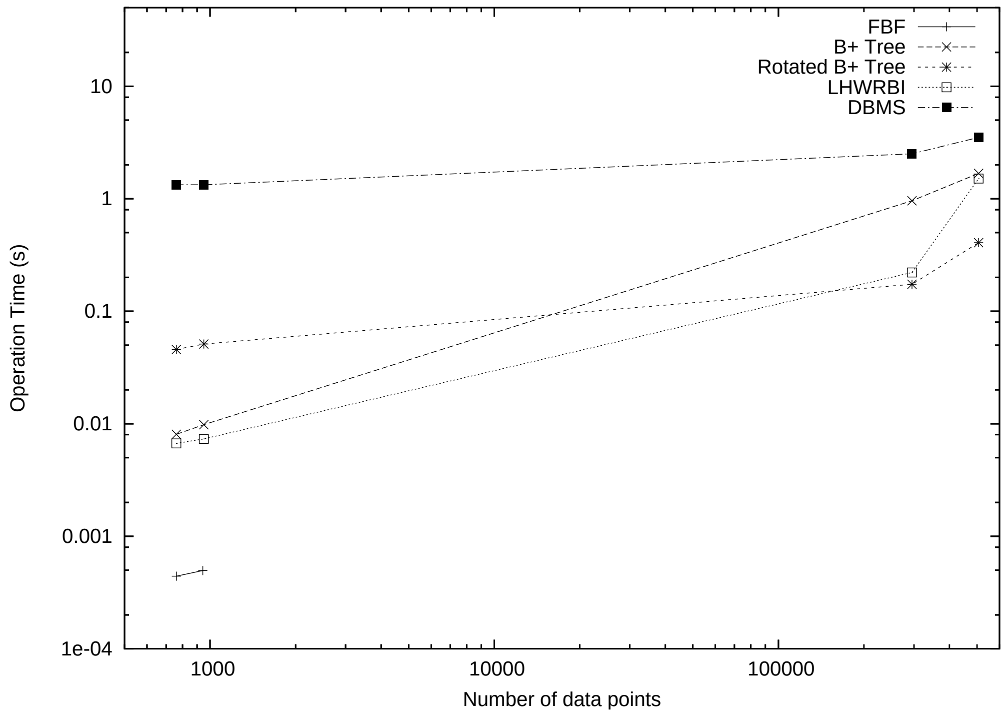


Figure 4.10: Speed of the `slice()` operation

as a `put()` into a bucket having space for the data point.

The efficiency of each storage method's `dice()` operation is presented in Figure 4.11. It appears as though the cost of the `dice()` operation increases approximately linearly with respect to N for all storage methods with the exception of the DBMS storage method. The dicing results for the largest test case using the DBMS storage structure are surprisingly efficient compared to the fourth test cube dicing results. This surprising result is currently unexplained, however, it is likely due to the fact that the fifth test cube was much denser (having relatively fewer keys pointing to undefined values) than

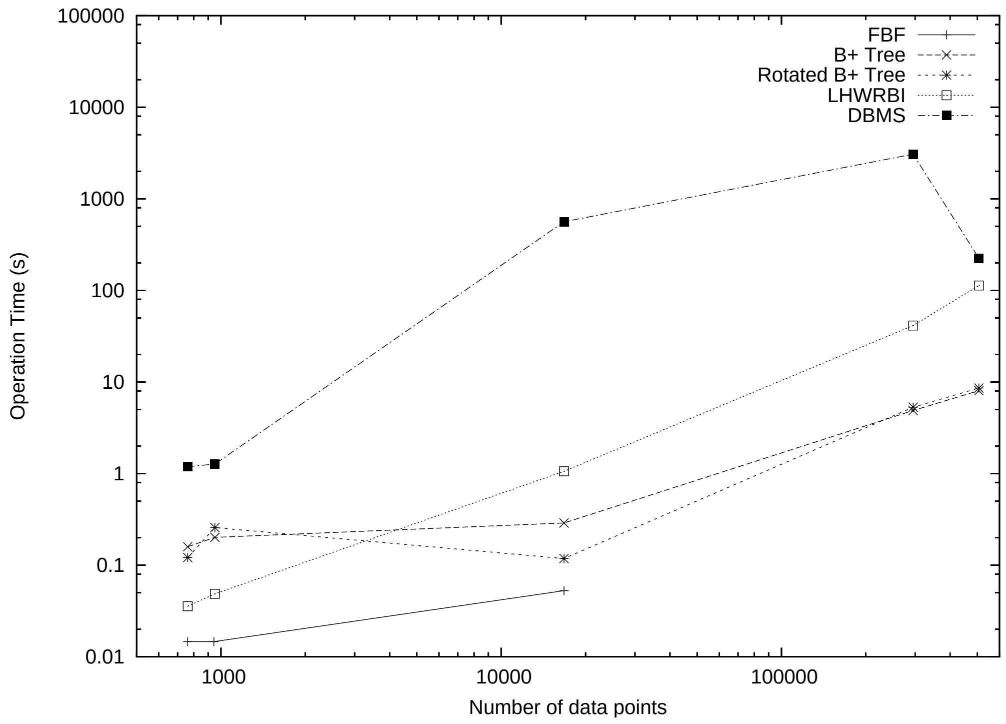


Figure 4.11: Speed of the `dice()` operation

the fourth test cube. The FBF storage structure is again the most efficient for the cubes it is capable of storing while the B+ Tree and Rotated B+ Tree are most efficient for larger cubes.

The efficiency of each storage method's `rollup()` operation is presented in Figure 4.12. This operation is an expensive operation for all but the DBMS storage methods because of the number of `put()` and `get()` operations that are required. The performance of each method reflects that method's `put()` and `get()` operation speeds. The DBMS storage method, however, requires only a single SQL call to perform a `rollup()` operation. Furthermore, the

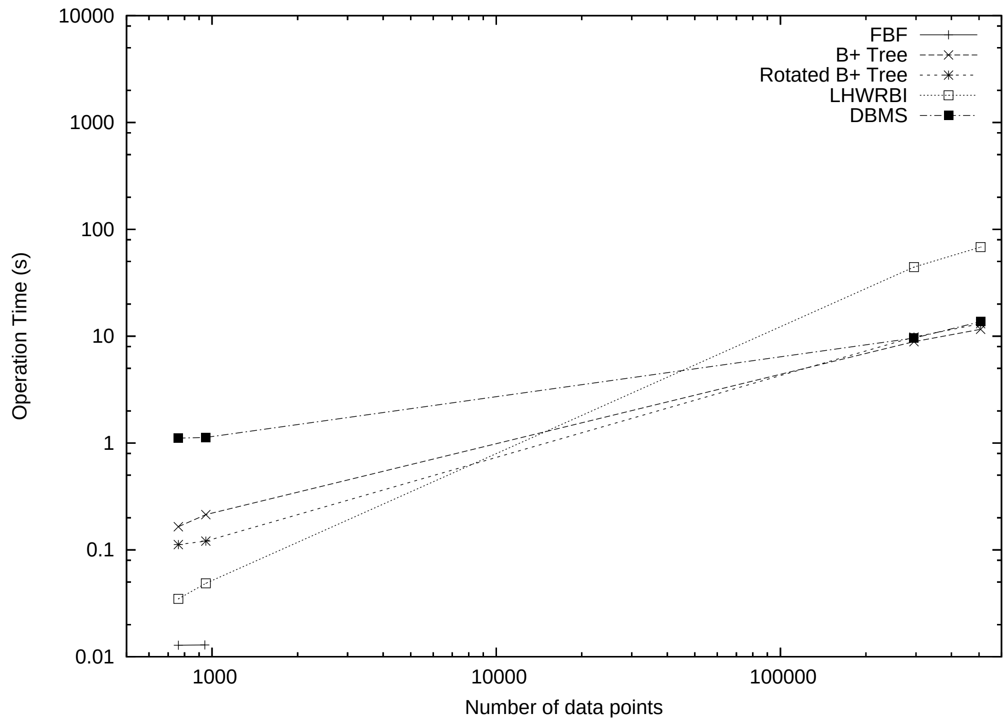


Figure 4.12: Speed of the `rollup()` operation

`GROUP BY` SQL statement is regularly used and is likely highly optimized within the PostgreSQL DBMS.

Tables 4.24 and 4.25 present the relative operation speed efficiencies for each of the storage methods. Each value contained within the table represents the *average inverse of the normalized operation speed* over all of the test cubes. For each storage method, the operation speed is normalized with respect to all of the other storage methods using Equation 4.1.

$$\begin{gathered}\text{Assume } t_i \text{ is the operation speed using storage method i.} \\ \text{norm}(t_i) = \frac{t_i}{\min(t)}\end{gathered} \tag{4.1}$$

When the results have been normalized, the values are inverted so that the most efficient method is given the greatest value. Finally, the inverse of the normalized operation speeds are averaged over the test cubes. The small and large test cubes are averaged separately for two reasons. First the FBF storage structure cannot store the larger cubes. Second, some of the storage methods are efficient for small cubes and inefficient for larger cubes when performing specific operations.

Method	put	get	slice	dice	roll-up
FBF	**100.00**	**100.00**	**100.00**	**100.00**	**100.00**
B+ Tree	7.64	7.87	5.27	11.55	6.89
Rotated B+ Tree	2.89	7.50	0.97	20.77	11.04
LHWRBI	0.85	1.28	6.68	25.28	31.64
DBMS	0.15	0.00	0.04	0.80	1.15

Table 4.24: Summary of relative operation speed efficiency for small cubes. Each value represents the average inverse of the normalized operation speed over all small test cubes.

Table 4.24 presents the operation speed efficiency for each storage method when the operations are performed on small cubes. The FBF storage method is able to perform all of the operations faster than any of the other methods. The DBMS is at least six times slower at performing operations than all other storage methods. Though the LHWRBI storage method has slower operation speeds for the `put()` and `get()` operations, it is faster than the B+ Tree and Rotated B+ Tree methods when performing the three OLAP operations.

Table 4.25 presents the operation speed efficiency for each storage method when the operations are performed on large cubes. The B+ Tree is most efficient for the large test cubes used in these experiments with the exception of the `slice()` operation. The Rotated B+ Tree is most efficient at slicing since it is always capable of slicing on the first dimension of a B+ Tree. The average `put()` operation time of the Rotated B+ Tree is much less than a third of the B+ Tree insertion time since the Rotated B+ Tree structure is multiple times larger on disk resulting in additional I/O seek time. The LHWRBI storage method is able to perform a `slice()` faster than the B+ Tree, however, it is especially slow when performing `rollup()` operations. The DBMS is most efficient at slicing and rolling up, since these two operations can be issued as single SQL calls.

Method	put	get	slice	dice	roll-up
B+ Tree	**100.00**	**99.08**	21.18	**100.00**	**100.00**
Rotated B+ Tree	5.90	94.87	**100.00**	92.75	89.59
LHWRBI	10.96	34.27	52.84	9.47	18.54
DBMS	9.86	0.00	9.26	1.88	88.74

Table 4.25: Summary of relative operation speed efficiency for large cubes. Each value represents the average inverse of the normalized operation speed over all large test cubes.

4.4.7 Size Considerations

The actual disk utilization of each storage method is considered (see Table 4.26) since efficient disk usage is an important aspect of a data warehouse. Measurements of disk usage were calculated by using the UNIX function call `stat`, which, among other things, returns the number of bytes used in storing the file on disk. Storage structures that use less disk space leave additional space for intermediate and summary results to remain on disk. These intermediate and summary results can then be used in the future when similar or identical queries are executed, removing the need to recalculate those results. This greatly improves the query evaluation time of complex queries.

Cube	N	FBF	LHWRBI	B+ Tree	Rotated B+ Tree	DBMS
1	16,758	80,708	290,268	131,821	131,823	679,936
2	761	7,776	27,292	131,821	395,469	40,960
3	951	6,936	35,312	131,821	395,469	49,152
4	295,132	—	9,561,468	8,263,978	27,507,326	14,229,504
5	506,908	—	17,113,100	16,494,753	52,378,645	24,428,544

Table 4.26: Storage method sizes (bytes)

The FBF disk usage is dependant on the shape of the cube and not the number of inserted values. For this reason the FBF storage structure is ideal for dense cubes, as disk space is not used for storing many empty cells of the cube. As noted in Section 3.2, test cubes 4 and 5 exceeded the maximum file size set by the operating system and were therefore not included in the results.

The LHWRBI storage structure has a size that increases linearly with respect to the number of data points inserted into the cube. A previous exploration of the LHWRBI storage structure concluded that a value of $\tau = 0.7$ (where τ is the splitting threshold) would provide the most efficient use of disk without negatively affecting the performance of operations [22]. Therefore, each bucket within the LHWRBI will be, on average, 70% full of data points, leaving an average of 30% of the total structure size empty. In the worst case, all data points could hash to the same bucket, thus leaving $\frac{N}{\text{size}(B)\times\tau} - 1$ empty buckets and a single bucket with $\frac{N}{\text{size}(B)} - 1$ overflow buckets.

B+ Tree storage sizes increase as the leaves of the B+ Tree become saturated with points. When a leaf is split into multiple leaves, additional space is allocated for the additional leaf and possibly an additional inner node used for indexing, causing the size of the B+ Tree to increase by large increments. The first 3 test cubes are all contained within a structure of the same size, despite containing a very different number of values. For large cubes it appears as though the B+ Tree provides the minimum storage requirements.

The Rotated B+ Tree size is a multiple of the B+ Tree size with an additional small amount of storage required to manage the various B+ Trees that have been created in the object. In Table 4.26, all of the different dimensions are rotated to the first dimension, providing fast slicing and dicing support on all dimensions. The first test cube is approximately the same size as the B+ Tree, since there is only a single dimension in the cube. All other cubes are approximately three times larger than their corresponding B+ Tree since they have 3 dimensions. The Rotated B+ Tree uses more disk space than any of the other storage methods.

Determining the amount of space used by PostgreSQL to store a table is achieved by querying the `pg_class` table which stores, among other things, the number of 8,192 byte pages that are being used to store each table. By multiplying the number of files by 8,192, we are able to determine the disk usage of storing a cube in the DBMS.

The DBMS storage structure has expensive storage requirements in comparison to all but the Rotated B+ Tree. Normally one would expect that a larger storage structure would allow for faster query results, however, the overhead of the DBMS appears to contradict that expectation.

As previously mentioned, PosgreSQL divides the data stored in tables into specially formatted files. PostgreSQL divides a table into separate files on disk referred to as *pages*. Each page is 8 kibibytes ($8,192$ bytes) in size and is divided into five parts; header, pointers, free space, items, and special space. The header is a constant 20 bytes in size. There is one pointer that is 4 bytes in size for every row in the table. The free space between the pointers and the items decreases as more rows are stored in the file. The items are the rows themselves, varying in size depending on number and size of each column. Each row requires its own additional 27 byte header in addition to the space required to store the actual data. Finally, the special space is reserved for additional indexing information, and varies in size.

As an example, the size of the fourth test cube, C_4, can be broken down into its parts. The cube C_4 contained $295,132$ data points, each represented in the database as a 4 column row in a table. Each column was of type Integer which is 4 bytes in size, thus, each row required 16 bytes of actual disk space. This results in $16 \times 295,132 = 4,722,112$ bytes required to store the actual data points. Thus, only 33% of total disk usage reported by C_4 was used in storing the data. Each row has a 4 byte pointer and a 27 byte header, accounting for an additional $31 \times 295,132 = 9,149,092$ bytes of disk space. Therefore, a total of $13,871,204$ bytes are accounted for. If each page was completely full and had no special space, $1,698$ pages would be required to store the table. If the absolute minimum number of pages were used in storing the table and absolutely no special space was required, a minimum of $13,910,016$ bytes were required to store C_4.

Since there is a constant overhead of 31 bytes per data point inserted into the DBMS storage structure, it is possible that the DBSM may more efficiently store cubes with many dimensions. However, further experimentation would be required to verify this claim.

4.5 Conclusions

The following conclusions have been drawn from the observations obtained through the previously described experiments. These conclusions summarize the strengths and weaknesses of each storage method by outlining the circumstances in which one storage method would prove efficient compared with the other storage methods discussed in this thesis.

The FBF storage structure is ideal for any small cube in that it efficient in terms of both operation speed and disk usage. Since the FBF maps a key directly to an address on disk, the keys do not require storage within the structure saving considerable disk space. When a cube is sliced, diced, or rolled up, the resultant cube is only a fraction of the size. One disadvantage of the FBF is that it requires defining the maximum values in each dimension prior to the creation of the cube. Therefore it would be difficult to use this structure without prior knowledge of the data distribution.

In the case that disk usage and creation time are not considered important, the Rotated B+ Tree is an ideal storage method as it is capable of performing all three OLAP operations relatively fast (see Table 4.25. The Rotated B+ Tree is at least twice as large as any of the other storage methods, thus it would not be ideal in any situation where disk usage was an important factor.

The LHWRBI storage method works very well with evenly distributed data points. It is also capable of slicing cubes nearly as fast as the Rotated B+ Tree without storing the data points numerous times. The scalability of the `put()` and `get()` operations, when N increases, is promising. This indicates that the LHWRBI storage method will likely store very large cubes faster than any of the other methods. The `dice()` and `rollup` operations are not very efficiently when this structure is used. Therefore, the LHWRBI storage method is ideal for storing large cubes that will be sliced frequently as long as the data points are reasonably distributed.

The B+ Tree storage method is highly versatile, using the least amount disk space when storing large cubes as well as performing fast operations. The B+ Tree is less affected by clustered data than the LHWRBI storage method, however the B+ Tree is unable to slice all dimensions with the same efficiency. Slicing on the first dimension is preferable since the B+ Tree's iterator is able to take advantage of the lexicographical ordering of the data points. The B+ Tree storage method is also able to `dice()` and `rollup()` cubes faster than any of the other methods on large cubes. Therefore, the B+ Tree method of storing a cube is ideal when none of the other specialized storage methods meet the cube's storage requirements in terms of disk usage or operation speed.

The DBMS storage method did prove to be scalable, however, it used more space and performed operations slower than most of the other storage methods. The evidence indicates that customized storage methods outperform the DBMS, and that the additional overhead required by this method makes it an inefficient method of storing cubes in a literary data warehouse.

Chapter 5

Conclusions

The following chapter summarizes the major points of the thesis and gives a brief description of some areas that have yet to be explored relating to this research.

5.1 Summary

The focus of this thesis has been to evaluate the performance of various storage methods in an effort to determine an efficient means of storing cubes in a literary data warehouse. The various storage methods included in this investigation were selected to investigate a wide scope of previously established external data structures.

The first method, referred to as the Flat Binary File storage method, was a memory mapped file in which the key was used to calculate an unique offset address in the file. This method proved efficient at storing and executing operations on small, dense cubes.

The B+ Tree and Rotated B+ Tree storage methods were selected to evaluate the ability of tree structures to efficiently store and execute operations on cubes. The B+ Tree proved to be a versatile storage method that occupied little space on disk yet was still able to quickly perform operations on the cube. One disadvantage of the B+ Tree storage structure was its inability to efficiently slice more than the first dimension of a cube. This disadvantage lead to the creation of the Rotated B+ Tree.

The Rotated B+ Tree is storage method that stores redundant copies of a B+ Tree in which the keys have been rotated allowing each dimension the ability to be the first dimension in one of the B+ Trees. Though dramatically increasing the disk usage, this structure proved the most efficient at slicing all dimensions of a cube. Since multiple B+ Trees are created in this structure, its construction time is a major disadvantage.

Linear Hashing with Reversed Bit Interleaving proved an interesting multidimensional, order preserving hashing method with linear growth on disk. This method has various parameters that could be altered to influence its performance. The LHWRBI storage method was able to slice cubes on all dimensions with approximately equal performance without any additional storage requirements. The main disadvantage of this storage method was its reliance on well distributed data points. Clustered data points tended to yield poor results when stored using this method.

These storage methods were compared against storing the cube using the relational database management system, PostgreSQL. Storing the data points in relational tables proved to be inefficient, mostly due to the overhead incurred using the DBMS. Though separate operations were developed to exploit the capabilities of the DBMS, this storage method proved inefficient with respect to both operation speed and disk usage.

Finally, the conditions under which each storage method is likely to prove efficient were outlined.

5.2 Future Work

The results of this thesis have raised numerous questions which could be explored in the future.

First, the manner in which the `dice()` operation is evaluated by the DBMS is inefficient due to the number of SQL calls that are required. This number can be reduced if we store the aggregation maps in SQL tables. Populating the aggregation map tables should not be overly expensive since the `put()` operation has proved scalable. With the aggregation maps stored in tables, the `dice()` operation could be performed by joining the aggregation maps with the cube's dimensions. By selecting the mapped values, and not the original dimension values, from the join, the result of the `dice()` can be evaluated in a single SQL call.

Next, the efficiency of each storage structure should be determined for larger cubes. The results obtained in this thesis suggest that the LHWRBI storage method may be capable of performing numerous operations faster than the B+ Tree storage method if the cube's size, N, is adequately large. Also the DBMS proved to be a highly scalable method, showing little increase in the `put(), slice()`, and `rollup()` operation times as N increased. This storage method may prove more competitive for very large sizes of N.

Finally, a method of predicting properties of the resultant cubes of the `slice(), dice()`, and `rollup()` operations on a cube in the data warehouse would be beneficial. If the properties of the resultant cube are known before it is created, the most efficient available storage method could be selected. For example if the result of a `rollup()` is expected to be small, the resultant cube should be stored in a FBF storage structure.

Bibliography

[1] ATLAS.ti Scientific Software Development, GmbH, *ATLAS.ti*, http://www.atlasti.de/, last accessed June 22, 2005.

[2] H. Baayen, H. van Halteren, A. Neijt, and F. Tweedi, *An experiment in authorship attribution*, Journees Internationales d'Analyse Statistique des Donnees Textuelles (2002).

[3] Jon Louis Bentley and Jerome H. Friedman, *Data structures for range searching*, ACM Comput. Surv. **11** (1979), no. 4, 397–409.

[4] Grady Booch, James Rumbaugh, and Ivar Jacobson, *The unified modeling language user guide*, Addison Wesley, October 1999.

[5] W. A. Burkhard, *Interpolation-based index maintenance*, BIT **23** (1983), no. 3, 274–294.

[6] F. Can and J. Patton, *Change of writing style with time*, Computers and the Humanities **38** (2004), no. 1, 61–82.

[7] E.F. Codd, *Providing OLAP (on-line analytical processing) to user-analysis: an IT mandate*, Tech. report, E.F. Codd and Associates, 1993.

[8] Cognos, *Business intellegence and performance management software solutions from Cognos*, http://www.cognos.com, last accessed May 29, 2006.

[9] J. Diederich, J. Kindermann, E. Leopold, and Gerhard Paass, *Authorship attribution with support vector machines*, Applied Intelligence **19** (2003), 109–123.

[10] Alan Dix, *online! Moore's law*, 2004, http://www.hcibook.com/e3/online/moores-law/ last accessed June 21, 2005.

[11] D. Foster, *Elegy by W.S.*, Associated University Presses, Mississauga, Ontario, Canada, 1989.

[12] ______, *Author unknown: on the trail of anonymous*, Henry Holt and Company, LLC, New York, New York, USA, 2000.

[13] Hector Garcia-Molina, Jeffrey Ullman, and Jennifer Widom, *Database systems: The complete book*, ch. 1, p. 14, Prentice Hall, 2002.

[14] GBorg, *The libpqxx Project – libpqxx The official C++ client API for PostgreSQL*, http://gborg.postgresql.org/project/libpqxx/projdisplay.php, last accessed Feb 27, 2006.

[15] Sanjay Goil, *High performance on-line analytical processing and data mining on parallel computers*, Ph.D. thesis, Dept. ECE, Northwestern University, 1999.

[16] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh, *Data cube: A relational aggregation operator generalizing group-by, cross-tab, and subtotal*, ICDE '96, 1996, pp. 152–159.

[17] Jim Gray and Alex Szalay, *The world-wide telescope*, Commun. ACM **45** (2002), no. 11, 50–55.

[18] Radhakrishna Hiremane, *From Moore's law to Intel innovation–prediction to reality*, Tech. report, Intel Corporation, April 2005.

[19] Hyperion, *Hyperion - the business performance management software leader*, http://www.hyperion.com, last accessed May 29, 2006.

[20] IBM, *IBM software - DB2 cube views - features and benefits*, http://www-306.ibm.com/software/data/db2/db2md/features.html, last accessed April 18, 2006.

[21] InTEXT SYSTEMS, *Intelligent Internet Tools*, http://www.intext.com/, last accessed June 22, 2005.

[22] Steven Keith, *Linear hashing with reversed bit interleaving*, http://v5o5jotqkgfu3btr91t7w5fhzedjaoaz8igl.unbf.ca/∼n7sh4/Report.pdf, December 2004.

[23] Steven Keith, Owen Kaser, and Daniel Lemire, *Analyzing large collections of electronic text using OLAP*, Tech. Report TR-05-001, UNBSJ CSAS, June 2005.

[24] ———, *Analyzing large collections of electronic text using OLAP*, APICS 2005, October 2005.

[25] D. E. Knuth, *The art of computer programming. vol. 3: sorting and searching*, Addison-Wesley, Reading, Mass, 1973.

[26] Yannis Kotidis, *Aggregate view management in data warehouses*, Handbook of Massive Data Sets (J. Abello et al., ed.), Kluwer, 2002, pp. 711–741.

[27] Cognitive Science Laboratory, *WordNet - Princeton University Cognitive Science Laboratory*, http://wordnet.princeton.edu/, last accessed April 18, 2006.

[28] Leeds Electronic Text Centre, *The Signature textual analysis system*, http://www.etext.leeds.ac.uk/signature/, last accessed June 21, 2005.

[29] Daniel Lemire, Owen Kaser, and Steven Keith, *Lemur OLAP library*, http://savannah.nongnu.org/projects/lemur, last accessed April 10, 2006.

[30] Jiexun Li, Rong Zheng, and Hsinchun Chen, *From fingerprint to writeprint*, Commun. ACM **49** (2006), no. 4, 76–82.

[31] W. Litwin, *Linear hashing: A new tool for file and table addressing*, Proceedings of the 6th International Conference on Very Large Databases (VLDB) (F. H. Lochovsky and R. W. Taylor, eds.), 1980, pp. 212–223.

[32] M. C. McCabe, J. Lee, A. Chowdhury, D. Grossman, and O. Frieder, *On the design and evaluation of a multi-dimensional approach to information retrieval*, SIGIR '00 (New York, NY, USA), ACM Press, 2000, pp. 363–365.

[33] T. Mendenhall, *The characteristic curves of composition*, Science **IX** (1887), 237–249.

[34] Microsoft, *Microsoft SQL server: Business intelligence solutions*, http://www.microsoft.com/sql/solutions/bi/default.mspx, last accessed April 18, 2006.

[35] ———, *Welcome to the MSDN Library*, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsql90/html/sql2k5xml.asp, last accessed Mar 27, 2006.

[36] Mikio Hirabayashi, *QDBM: Quick Database Manager*, http://qdbm.sourceforge.net/, last accessed Feb 21, 2006.

[37] Gorden E. Moore, *Cramming more components onto integrated circuits*, Electronics **38** (1965), no. 8.

[38] MySQL, *MySQL AB :: GIS and Spatial Extensions with MySQL*, http://dev.mysql.com/tech-resources/articles/4.1/gis-with-mysql.html, last accessed Mar 27, 2006.

[39] J. A. Orenstein, *A dynamic hash file for random and sequential accessing*, Proceedings of the 9th International Conference on Very Large Databases (VLDB) (M. Schkolnick and eds. C. Thanos, eds.), 1983, pp. 132–141.

[40] J. A. Orenstein and T. H. Merrett, *A class of data structures for associative searching*, Proceedings of the 3rd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), 1984, pp. 181–190.

[41] M. Ouksel and P. Scheuermann, *Storage mappings for multidimensional linear dynamic hashing*, Proceedings of the 2nd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), 1983, pp. 90–105.

[42] PostgreSQL, *PostgreSQL: Documentation: Manuals: PostgreSQL 8.1: Extensibility*, http://www.postgresql.org/docs/current/static/gist-extensibility.html, last accessed Mar 27, 2006.

[43] ______, *PostgreSQL: The world's most advanced open source database*, http://www.postgresql.org/, last accessed Mar 1, 2006.

[44] E. Pourabbas and M. Rafanelli, *Hierarchies and relative operators in the OLAP environment*, ACM Sigmod Record **29** (2000), no. 1, 32–37.

[45] Project Gutenberg Literary Archive Foundation, *Project Gutenberg*, http://www.gutenberg.org/, 2005.

[46] Raghu Ramakrishnan and Johannes Gehrke, *Database management systems*, 3 ed., pp. 344–364, McGraw-Hill, 2003.

[47] Sunita Sarawagi and Michael Stonebraker, *Efficient organization of large multidimensional arrays*, ICDE: 10th International Conference on Data Engineering, IEEE Computer Society Technical Committee on Data Engineering, 1994.

[48] Michael Scott, *Oxford wordsmith tools, version 4*, Oxford University Press, see also http://www.lexically.net/wordsmith/, last accessed June 22, 2005.

[49] Seagate Technology, Inc., *ST336706LC Configuration and Specifications*, http://www.seagate.com/support/disc/specs/scsi/st336706lc.html, last accessed Feb 21, 2006.

[50] E. Stamatatos, N. Fakotakis, and G. Kokkinakis, *Computer-based authorship attribution without lexical measures*, Computers and the Humanities **35** (2001), 193–214.

[51] D. Sullivan, *Document warehousing and text mining: Techniques for improving business operations, marketing, and sales*, John Wiley & Sons, 2001.

[52] D. Theodoratos, *Exploiting hierarchical clustering in evaluating multidimensional aggregation queries*, DOLAP (2003), 63–70.

[53] P.D. Turney and M.L. Littman, *Learning analogies and semantic relations*, Tech. report, National Research Council Canada, Institute for Information Technology, 2003.

[54] F.J. Tweedie and R.H. Baayen, *How variable may a constant be? measures of lexical richness in perspective*, Computers and the Humanities **32** (1998), 323–352.

[55] Stephen Tweedie, *Ext3, journaling filesystem*, http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html, July 2000.

[56] J. Yang, K. Karlapalem, and Q. Li, *Algorithms for materialized view design in data warehousing environment*, VLDB (1997), 136–145.

Vita

Author: Steven Keith

University attended:

University of New Brunswick, Saint John (2000-2004)

BScCS

Publications:

Steven Keith, Owen Kaser, and Daniel Lemire, *Analyzing large collections of electronic text using OLAP*, APICS 2005, October 2005.