Fast Access to File-based Data Warehouses

A Project Report

Submitted in partial fulfilment of the

requirements for the Degree of

Master of Engineering

in

Faculty of Engineering

by

Amol V. Wanjari

Department of Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012

JULY 2004

To my beloved Mom, Dad and my cute little sisters Bhavana and Rupali for always being there.

Acknowledgments

Giving thanks is a tricky business: I want to favor some who went out of their way to help me with a special mention, but I risk offending others by omission.

If you feel that I have not properly recognized your contribution to my work in this acknowledgement, please accept my apology, know that I meant no disrespect, and forgive my imperfections!

No amount of words in acknowledgment can record my appreciation for my project advisor, Prof Jayant R. Haritsa. His passion for his work and friendly approach to students, his worldly-wise approach to problems are some of many appealing qualities which made my association with him an extremely valuable experience. Truly saying, he is not only my project advisor but a source of inspiration for me. I believe that the freedom and moral support which I got from him during my project is one of the prime factors in accelerating the work.

I'm thankful to the staff of Yahoo Software Development Centre, Bangalore for helping me during the tenure of the project.

I would specially thank my colleage Shipra for her useful suggestions in the project work. I'm also thankful to fellow *DSLites* Abhijit, Anoop, Kumaran, Maya, Parag, Srikanta and Suresha who made DSL a homely place to work. I am grateful to all my friends at IISc whose constant support and inspiration has been a driving factor to me.

With a deep sense of belonging, I wish to express my sincere gratitude to all the staff members of the CSA Department who have helped me directly or indirectly during the period of my association with the department.

Contents

Abstract **vi**

1 Introduction **1**

2 Schema Design **4**

2.1 Partitioning 5

2.2 Definitions and notations 9

2.3 Fundamental concepts 10

2.4 The algorithm 12

3 Compression **14**

3.1 Compression: Pros and Cons 15

3.2 Compression Techniques 16

3.2.1 Huffman Coding 17

3.2.2 LZW Coding 18

3.2.3 gzip 18

3.2.4 Run length Encoding 19

3.3 Compression Granularity 20

3.4 Overheads 21

3.5 Summary 22

4 Indexes **23**

4.1 Indexing mechanism 23

4.2 Design decisions about indexing 24

4.3 Handling Updates 25

5 Case Study: System-Y **26**

6 Performance Evaluation **29**

6.1 Schema 29

6.2 Optimal chunk size 30

6.3 Cost model 31

6.4 Query execution algorithms 31

6.4.1 Query type Q_1, Q_2, Q_3 32

6.4.2 Query type Q_4 35

6.5 Compression Performance 35
6.6 Execution time 36
6.6.1 Without Index 37
6.6.2 With Index 37

7 Related Work 39

8 Yippee: Software prototype 41
8.1 Features 41
8.2 System Architecture 45

9 Conclusions and Future Work 47

Bibliography 47

List of Figures

2.1 Storage Architecture 6
2.2 Cycle and extension 10
2.3 Partition 12

6.1 Partitions obtained after applying graph algorithm 30
6.2 Determination of optimal chunk size 31
6.3 Total Space Requirement 36
6.4 Execution Time without index 37
6.5 Execution Time with index 38

8.1 Yippee: Generate Partition module 42
8.2 Yippee: Create Partition module 43
8.3 Yippee: Query Execution module 44
8.4 System Architecture of Yippee 45

Abstract

Efficient query processing is critical in a data warehouse environment because the warehouse is very large, queries are often ad hoc and complex, and decision support applications typically require interactive response times. Our work focuses on the improvement of response time for a given query workload. We have studied the effect of schema design, compression and indexing on the performance of data warehouses. We propose a new storage layout that combines the power of schema design and compression and also adapted the existing index structures to match this storage layout. The data warehouse implemented is based on file system approach instead of traditional relational engine which are incapable of handling such large data which is in orders of tera-byte. To evaluate the efficacy of the proposed techniques we conducted a case study on real datasets obtained from a very popular web-portal and evaluated our system against their existing system - *System-Y*. Our experiments show four orders of performance improvement for range queries using indexes and 40% boost without using indexes over existing setup used by *System-Y*. Moreover there is almost no space overheads due to indexes which is achieved due to compression on partitioned data. We have designed and implemented *Yippee*, a software prototype which integrates the concepts put forward in this report. This tool has an automated schema designer and a query execution engine.

Chapter 1

Introduction

Data warehouses are large, special purpose databases that contain historical data integrated from a number of independent sources, supporting users who wish to analyze the data for trends and anomalies. The process of analysis is usually done by queries that aggregate, select and group the data in a number of ways. Efficient query processing is critical because the data warehouse is very large, queries are often complex, and decision support applications typically require interactive response times.

Our motivation is to study the effect of schema design, compression and indexing on the query performance. According to case study conducted on *System-Y*, the size of typical data warehouses are very large in the order of few terabytes. Since traditional relational database systems fail to handle such large data we resort to file system to handle storage. Current RDBMS have limits on various parameters - maximum number of columns in a relation is 1000, maximum number of indexes on a relation is 32, maximum LOB size is 4GB. Apart from these restrictions, populating the relations with rows and building indexes on such a large data is time consuming. One may also go for simple system based on files to avoid the overhead added due to extra functionality provided by general purpose RDBMS.

The performance of query processing depends on the physical schema design. A highly normalized schema offers superior performance and efficient storage where only a few attributes are accessed by a particular query. The star schema [18] provides similar benefits for data warehouses where most queries aggregate a large amount of data. A star schema consists of a large central fact table which has predominantly numeric data and several smaller dimension tables which are all related to the fact table by a foreign key relationship. The dimension tables have more descriptive attributes. Such a schema is an intuitive way to represent the multi-dimensional data so typical of business, in a relational system. The queries usually filter rows based on dimensional attributes and then group by some dimensional attributes and aggregate the attributes of the fact table . We achieve normalization via vertical partitioning which is essentially column-wise split of the original relation. Storing the data column-wise leads to better compression as data in a column is semantically associated. Moreover if most of the queries access few columns then we save unnecessary disk access leading to better performance. In this regard we exploit the workload information to come up with partitioning scheme for given relation.

Given the large size of data warehouses, storage costs are very high and so is the cost of storage due to index structures. Compression has several benefits. Apart from reducing storage costs, it also could reduce query processing time by reducing the number of disk accesses [25]. Although decompression adds to query processing cost, [25] showed that in databases the reduced number of disk accesses more than compensates for the increased processing time. We believe that it should in fact be even more efficient to compress a data warehouse as most queries access a large amount of data and most of the decompressed data will actually be required to evaluate the query. Hence the number of disk accesses will be much less leading to performance boost.

Data warehouses are typically updated only periodically, hence they are mostly read-only in nature. Hence the problems of maintaining indexes in the presence of concurrent updates is no longer an issue, it is possible to use sophisticated indexes to speed up query evaluation. We have adapted traditional indexing structures like B-Tree index, Hash index to match the proposed storage format.

The framework for compressed data warehouse presented in this paper is generalized but to evaluate on a real system we have undergone a case study of a leading web portal - say *System-Y*. The problem is thus to design a data warehouse, where the data in in compressed form and moreover to come-up with indexing mechanisms that are applicable to compressed data.

The remainder of this paper is organized as follows. Chapter 2 explains the architecture of storage layer for data warehouse which employs compression and partitioning to achieve performance boost. The graphical vertical partitioning algorithm that we employed for schema design is also discussed in Chapter 2. In Chapter 3 we review various compression techniques that we evaluated and Chapter 4 talks about our modifications to existing index structures. In Chapter 5 we conduct a case study of System-Y data ware house engine and compare it against our system. Chapter 6 presents the performance figures, Chapter 8 discusses the software prototype of a tool named *Yippee* developed as an aid to database designers and Chapter 9 concludes the paper presenting future avenues to explore.

Chapter 2

Schema Design

In this chapter we discuss how the physical schema design affects the performance of queries and how we employed vertical partitioning as a means to achieve that. Physical design is the process to come up with proper structuring of data in storage so that good performance is guaranteed. A highly normalized schema offers superior performance and efficient storage where only a few records are accessed by a particular transaction. It is not possible to come up with better physical schema unless prior knowledge of workload is known. The information required should consist of the nature of queries and their expected frequencies. For each query we should specify the following:

- The files that will be accessed by the query.
- The attributes on which any selection predicates are specified.
- The attributes whose values will be retrieved by the query.

Apart from the workload characteristics we must also take into account their expected frequency of invocation. This frequency information along with the attribute information of each query can be used to compute a cumulative statistics of expected access frequency for all the queries. This is expressed as the expected frequency of accessing each attribute in each file in a selection predicate over all the queries.

2.1 Partitioning

The partitioning of a global schema into fragments can be done in two ways viz., *vertical partitioning* and *horizontal partitioning*. Since a normalized schema may lead to joins when accessing columns in different normalized relations we simply do vertical partitioning. Moreover vertical partitioning has its own advantages as discussed later in this section.

n	1	2	3	4	5	6	7	8	9	10	11	12	13
B_n	1	2	5	15	52	203	877	4140	21147	115975	678570	4213597	27644437

Table 2.1: Bell numbers

Vertical partitioning is the process that divides a *global object* which may be a single relation or more like a universal relation into groups of their attributes, called *vertical fragments* [23, 22, 8]. Formally speaking, if C is attribute set of a relation then *partition set* P is set of subsets of attribute set C. $P = \{P_1, P_2, \ldots, P_k\}$ such that

$$\forall i, j : i \neq j : P_i \subseteq C, P_j \subseteq C, P_i \bigcap P_j = \phi, \bigcup_{i=1}^{k} P_i = C$$

For a set of size n the number of unique partition sets is given by Bell number B_n [5].

$$B_n = \sum_{i=0}^{n} S(n, k)$$

where $S(n, k)$ is second order Stirling number given by following recursive function

$$S(n, k) = k \cdot S(n-1, k) + S(n-1, k-1)$$

$$S(n, n) = S(n, 1) = 1$$

The first few Bell numbers are given in table 2.1.

Vertical partitioning is used during design of a database to enhance the performance of query execution[23]. In order to obtain improved performance, fragments must closely match the requirements of the query workload.

Vertical partitioning involves splitting the relations along the columns into partitions all having equal number of rows. Each partition now act as a separate relation but we preserve the row ordering in all the partitions as it was in the original relation. It should be noted that each partition may contain more than one column. However when columns in multiple partitions are accessed instead of join we just need to do pasting of columns.

The advantages of vertical partitioning are as follows: If query involves only few columns then we avoid unnecessary fetching of other columns. This saves the I/O bandwidth and avoids unnecessary processing. Moreover data in a column belongs to the same domain e.g., values in salary column will be numeric within some range. This similarity in data can be well exploited by compression algorithms and better compression ratios can be achieved.

The data under consideration is of the order of terabytes. Since the data is large enough to be handled by traditional relational database system we resort to file system for storage. Each partition as described above is stored as a separate file. Since the row ordering is maintained we do not store the row identifiers in the file. The meta data stores information regarding the partitions of a given relation such as number of partitions, columns in each partition etc.

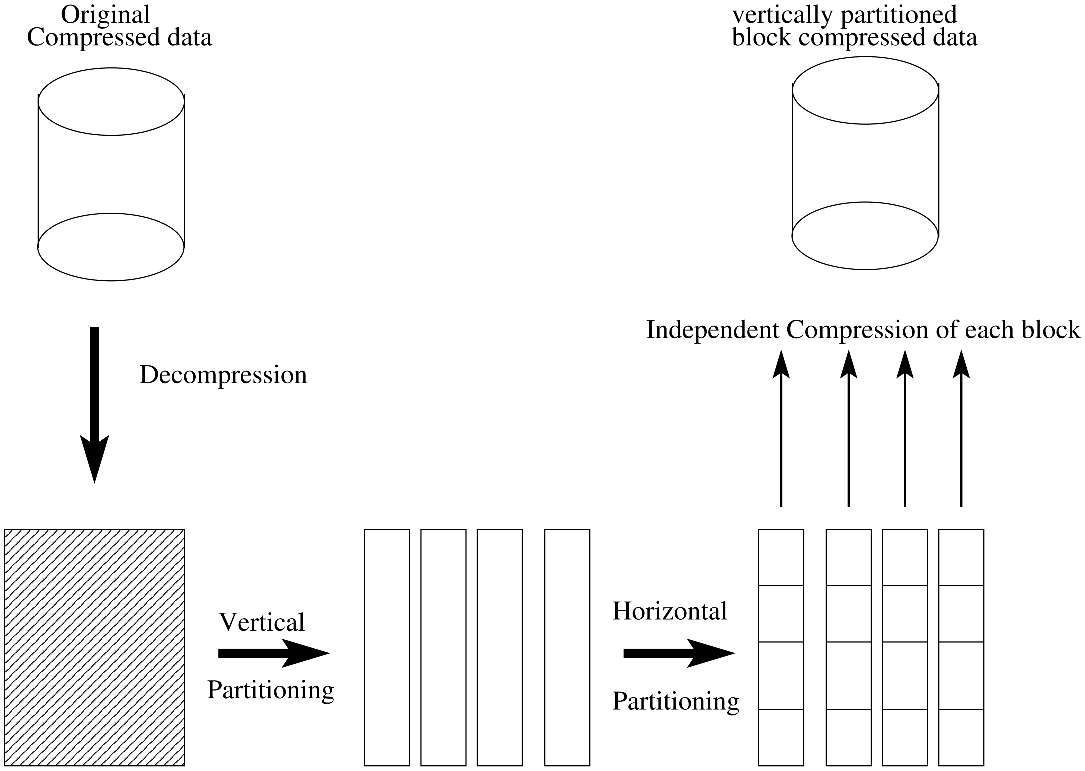


Figure 2.1: Storage Architecture

After the first step of vertically partitioning the data into partitions we horizontally divide each partition into chunks. The process is depicted in Figure 2.1. Each chunk in turn is then compressed individually. Each chunk contains equal number of row, this is done to simplify indexing as will be discussed in Chapter 4. We preserve row boundary across chunks so that in order to access a row one and only one chunk is decompressed. The reason behind this compression granularity as opposed to attribute-level compression is better compression factor without incurring high decompression overhead. Another important aspect is the optimal chunk size at which compression ratio and decompression overhead are balanced. At higher levels of compression granularity, the compression ratio is better but query processing requires decompressing a large portion of data, whereas at lower compression granularities the compression ratio is poor but query processing requires a small part of data to be decompressed. The tradeoffs between compression granularity and compression ratio will be discussed in Chapter3.

In fact the problem of finding optimal partitions given a cost function reduces to classical Set Partition Problem which is NP-Hard. But is this really a hard problem for smaller input sizes (like 13 in case of our test data), as can be seen in Table 2.1 the Bell number increases rapidly with the number of attributes,(even for 13 columns number of ways of partitioning is close to 28 million) hence the search space for optimal partition set is quite huge to be explored by brute force methods. Hence we resort to heuristic based approaches, proposed in literature for vertical partitioning [8, 14, 23, 22, 21, 24]. The discussion on vertical partitioning algorithm in this section is taken from [24]. We have chosen [24] because of the following reasons:

(a) There is no need for iterative binary partitioning as discussed in [23]. The major weakness of iterative binary partitioning is that at each step two new problems are generated increasing the complexity; furthermore, termination of the algorithm is dependent on the discriminating power of the objective function.

(b) The method requires no objective function. The objective function controls the process of partitioning. The empirical objective functions in [23] were selected after some trial and error experimentation to see that they possess a good discriminating power. Although reasonable, they constitute an arbitrary choice. This arbitrariness has been eliminated in this methodology. The algorithm starts from the *attribute affinity (AA) matrix*, which is generated from the attribute

	Attribute usage matrix													Frequency
Attributes→	0	1	2	3	4	5	6	7	8	9	10	11	12	
Queries ↓														
T1	1	0	0	1	0	0	1	0	0	0	0	0	0	Acc 1 = 1
T2	0	1	1	0	0	0	0	0	1	0	0	0	0	Acc 2 = 1
T3	0	0	1	1	0	0	0	1	0	0	0	0	0	Acc 3 = 1
T4	0	0	1	0	0	1	0	1	0	0	0	0	0	Acc 4 = 1
T5	0	1	1	0	0	0	0	0	0	0	0	0	0	Acc 5 = 1
T6	1	0	1	0	0	0	1	0	0	0	0	0	0	Acc 6 = 1
T7	1	0	1	1	0	0	1	0	0	0	0	0	0	Acc 7 = 1
T8	0	0	0	0	0	0	0	0	0	0	1	0	0	Acc 8 = 1
T9	0	0	0	0	0	0	0	0	0	1	0	0	0	Acc 9 = 1
T10	0	0	0	0	0	0	0	0	1	0	0	0	0	Acc 10 = 1

Table 2.2: Attribute usage matrix for experimental workload

usage matrix using the method of [23]. The attribute usage matrix represents the use of attributes in important queries. Each row refers to one query; the 1 entry in a column indicates that the query accesses the corresponding attributes. The attribute usage matrix for 13 attributes and 10 queries is shown in Table 2.2. This workload corresponds to the test workload that we used for experimental purpose.

Attribute affinity is defined as

$$aff_{ij} = \sum_{k \in \tau} acc_{kij}$$

where acc_{kij} is the number of accesses of query referencing both attributes i and j. The summation occurs over all queries that belong to the set of important queries τ. This definition of attribute affinity measures the strength of an imaginary bond between the two attributes, predicated on the fact that attributes are used together by queries. Based on this definition of attribute affinity, the attribute affinity matrix is defined as follows: It is n×n matrix for the n-attribute problem whose (i, j) element equals aff_{ij}. Table 2.3 shows the attribute affinity matrix which was formed from Table 2.2. A diagonal element $AA(i, i)$ equals the sum of the elements in the attribute usage matrix for the column which represents a_i. This is reasonable since it shows the “strength” of that attribute in terms of its use by all queries.

In previous approaches, they apply a clustering algorithm to the AA matrix. In [24], however, they consider the AA matrix as a complete graph called the *affinity graph* in which an edge value represents the affinity between the two attributes. Then, forming a linearly connected spanning tree, the algorithm generates all meaningful fragments in one iteration by considering a cycle as a fragment. A “linear connected” tree has only two ends. Note that the AA matrix serves as a data structure for the affinity graph.

Attributes	0	1	2	3	4	5	6	7	8	9	10	11	12
0	6	0	3	2	0	1	3	0	0	1	1	0	0
1	0	2	2	0	0	0	0	0	1	0	0	0	0
2	3	2	6	2	0	1	2	1	1	0	0	0	0
3	2	0	2	3	0	0	2	1	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0
5	1	0	1	0	0	1	0	0	0	0	0	0	0
6	3	0	2	2	0	0	3	0	0	0	0	0	0
7	0	0	1	1	0	0	0	1	0	0	0	0	0
8	0	1	1	0	0	0	0	0	2	0	0	0	0
9	1	0	0	0	0	0	0	0	0	1	0	0	0
10	1	0	0	0	0	0	0	0	0	0	1	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 2.3: Attribute Affinity matrix for experimental workload

2.2 Definitions and notations

The following notation and terminology is used in the description of the algorithm.

- $A, B, C, ...$ denotes nodes.
- $a, b, c, ...$ denotes edges.
- $p(e)$ denotes the affinity value of an edge e.

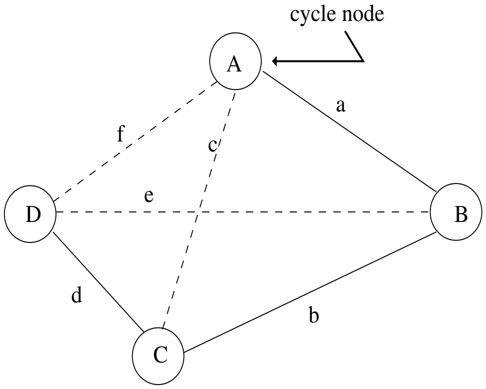


Figure 2.2: Cycle and extension

- *primitive cycle* denotes any cycle in the affinity graph.
- *affinity cycle* denotes a primitive cycle that contains a cycle node.
- *cycle completing edge* denotes a “to be selected edge” that would complete a cycle.
- *cycle node* is that node of the cycle completing edge, which was selected earlier.
- *former edge* denotes an edge that was selected between the last cut and the cycle node.
- *cycle edge* is any of the edges forming a cycle.
- *extension of a cycle* refers to a cycle being extended by pivoting at the cycle node.

The above definitions are used in the proposed algorithm to process the affinity graph and to generate possible cycles from the graph. They will become clearer when they will be explained further in this section. Each cycle gives rise to a vertical fragment.

2.3 Fundamental concepts

Based on the above definitions the mechanism of forming cycles will be explained. For example, in Figure 2.2, suppose edges a and b were selected already and c was selected next. At this time, since c forms a primitive cycle, we have to check if it is an affinity cycle. This can be done by checking the possibility of a cycle. *Possibility of a cycle* results from the condition that *no former edge exists*, or $p(former\ edge) \leq p(all\ the\ cycle\ edges)$. The primitive cycle a, b, c is an affinity cycle because it has no former edge and satisfies the possibility of a cycle. Therefore the primitive cycle a, b, c is marked as a candidate partition and node A becomes a cycle node.

The explanation of how the extension of a cycle is performed will be discussed next. In Figure 2.2, after the cycle node is determined, suppose edge d was selected. At this time, d is checked as a potential edge for extension. It can be done by checking the possibility of extension of the cycle by d. $Possibility\ of\ extension$ results from the condition of $p(edge\ being\ considered\ or\ cycle\ completing\ edge) \geq p(any\ one\ of\ the\ cycle\ edges)$. Thus the old cycle a, b, c is extended to the new cycle a, b, d, f if the edge d under consideration, or the cycle completing edge f, satisfies the possibility of extension which is: $p(d)\ or\ p(f) \geq minimum\ of\ (p(a), p(b), p(c))$. Now the process is continued: suppose e was selected as the next edge. But we know from the definition of the extension of a cycle that e cannot be considered as a potential extension because the primitive cycle d, b, e does not include the cycle node A. Hence it is discarded and the process is continued.

The next concept that is explained corresponds to the relationship between a cycle and a partition. There are two cases in partitioning.

1. Creating a partition with a new edge.

In the event that the edge selected next for inclusion (e.g. d in Figure 2.2) was not considered before, lets call it a $new\ edge$. If a new edge by itself does not satisfy the possibility of extension, then continue to check for an additional new edge called cycle completing edge (e.g. f in Figure 2.2) for the possibility of extension. In Figure 2.2, new edges d and f would potentially provide such a possibility of extension of the earlier cycle formed by edges a, b, c. If d, f meet the condition for possibility of extension stated above (namely $p(d)\ or\ p(f) \geq minimum\ of\ (p(a), p(b), p(c))$), then the extended new cycle would contain edges a, b, d, f. If the condition were not met, then produce a cut on edge d (called the $cut\ edge$) isolating the cycle a, b, c. This cycle can now be considered a $partition$.

2. Creating a partition with a former edge.

After cutting in (1), if there is a former edge, then change the previous cycle node to that node

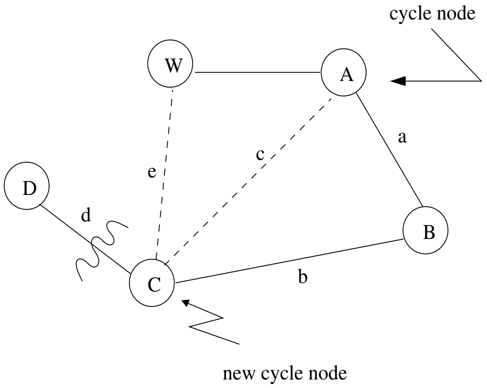


Figure 2.3: Partition

where the cut edge was incident, and check for the possibility of extension of the cycle by the former edge. For example, in Figure 2.3, suppose that a, b, and c form a cycle with A as the cycle node, and that there is a cut on d, and that the former edge w exists. Then the cycle node A is changed to C because the cut edge d originates in C. We are now evaluating the possibility of extending the cycle a, b, c into one that would contain the former edge w. Hence we consider the possibility of the cycle a, b, e, w. Assume that w or e does not satisfy the possibility of extension, i.e., if $p(w)\ or\ p(e) \geq minimum\ of\ (p(a), p(b), p(c))$ is not true. Then the result is the following: (i) w will be declared as a cut edge, (ii) C remains as the cycle node, and (iii) a, b, c becomes a partition. Alternately, if the possibility of extension is satisfied, the result is: (i) cycle a, b, c is extended to cycle w, a, b, e, (ii) C remains as the cycle node, and (iii) no partition can yet be formed.

Intuitively, the algorithm presented below achieves the decision of partitioning in the following manner. Keeping the pivot on a present cycle node, extension of the cycle is attempted by considering either new edges or former edges which would expand the area under the cycle.

2.4 The algorithm

An algorithm for generating the vertical fragments by the affinity graph is described below. Each partition of the graph generates a vertical fragment.

The brief description of the algorithm is given as follows.

Step 1: Construct the affinity graph of the attributes of the object being considered. Note that the AA matrix is itself an adequate data structure to represent this graph. No additional physical storage of data would be necessary.

Step 2: Start from any node.

Step 3: Select an edge which satisfies the following conditions:

- It should be linearly connected to the tree already constructed.
- It should have the largest value among the possible choices of edges at each end of the tree.

This iteration will end when all nodes are used for tree construction.

Step 4: When the next selected edge forms a primitive cycle:

- If a cycle node does not exist, check for the possibility of a cycle and if the possibility exists, mark the cycle as an affinity cycle. Consider this cycle as a candidate partition. Go to step 3.
- If a cycle node exists already, discard this edge and go to step 3.

Step 5: When the next selected edge does not form a cycle and a candidate partition exists:

- If no former edge exists, check for the possibility of extension of the cycle by this new edge. If there is no possibility, cut this edge and consider the cycle as a partition. Go to step 3.
- If a former edge exists, change the cycle node and check for the possibility of extension of the cycle by the former edge. If there is no possibility, cut the former edge and consider the cycle as a partition. Go to step 3.

The algorithm takes a time $O(n^2)$, which is less than that of [23], namely, $O(n^2 logn)$.

Chapter 3

Compression

The discussion in this section is based on [25]. There are couple of issues related to compression (i) compression algorithm (ii)compression granularity (iii) meta-data compression

Compression algorithm: The compression algorithm determines the compression ratio and compression/decompression speed. There are several well-known compression algorithms viz. Huffman coding [15], Arithmetic coding [30], LZW algorithm [29], LZ77 algorithm [32], run-length encoding [10] etc. Some of these algorithms come in adaptive and non-adaptive flavors.

We used `zlib` library that uses LZ77 algorithm combined with Huffman coding. The choice was dictated by decompression speed and compression ratio as will be shown in Section 6. We have conducted study of various compression techniques viz. Huffman coding, Run-length encoding, LZW algorithm and gzip.

Compression granularity: Compression granularity denotes the amount of data that is compressed independently. For instance, the entire relation stored in a file can be compressed together, this is file-level compression. Similarly, we can have page-level, record-level, attribute-level compression. At each level of compression there is trade-off between query processing time and compression ratio. As we go from file-level to record-level the compression ratio may fall and the query processing time may increase. This is because at higher level the amount of data seen by the compression algorithm is more and hence the probability of finding recurring pattern is more which leads to higher compression ratios. But this is has inverse effect on query processing since now to retrieve a record, the amount of data that needs to decompressed increases, e.g. in file-level compression to extract even a record the entire file is decompressed before any processing is done which adds to extra disk accesses.

We chose chunk-level compression preserving row boundary since it ensures that a single chunk needs to be decompressed to extract a row while preserving respectable compression ratio. Since only a portion of data needs to be read into memory we save disk accesses improving execution time. One more aspect is the optimal size of chunk where the compression factor balances the decompression overhead. We conducted experiments to find the optimal chunk size empirically.

Meta-data compression: Another issue is whether meta-data should be compressed. Since meta-data in form of catalogs or indexes are small in comparison to data files, it appears that the small storage savings that are achieved by compression are more than offset by the increased processing complexity. A related question is whether the indexes should be built using the original keys or their compressed equivalents. Here again it appears that retaining the original keys is far more attractive since the lexical ordering provided by index leaf level is retained.

We did not go for meta-data compression to avoid processing complexity.

3.1 Compression: Pros and Cons

We describe the benefits and drawbacks typically associated with compression. For the drawbacks, we discuss how these may be addressed in modern database systems.

Apart from the query processing improvement, compression results in several other benefits also [12, 3, 16, 11] by virtue of storing data in a reduced space: Firstly, disk seek times are reduced since the compressed data fits into a smaller physical disk area. Secondly, related objects can be clustered closer together. Thirdly, data compression increases disk bandwidth by increasing the information density of the transferred data. Finally, network communication costs are reduced in distributed databases and client-server applications due to reduced data transfer.

From a transaction processing perspective, there are two further benefits [11]: First, the buffer hit rate increases since a larger fraction of the database now fits into the buffer pool. Second, the disk I/O to log devices is decreased since the log records are shorter.

The drawbacks typically associated with compression [3, 11, 16] are: Firstly, data compression and decompression may result in considerable overhead at the CPU. Secondly, compression results in data records becoming variable-sized. However, many modern database systems are already equipped to handle variable-sized records in order to efficiently support schema transformation [16] or variable length attributes (as required in SQL92), so this is not a real problem. Finally, compression, by virtue of reducing redundancy in data, reduces the ability to recover from errors. For example, a single bit error in the output may result in the decoder misinterpreting all subsequent bits. Problems of this nature, however, are taken care of by current communication protocols and disk controllers.

3.2 Compression Techniques

Most data compression techniques are based on one of two models: statistical or dictionary. In statistical modeling, each distinct character of the input data is encoded, with the code assignment being based on the probability of the character's appearance in the data. In contrast, dictionary-based compression schemes maintain a dictionary which contains a list of commonly occurring character strings in the data and their corresponding codes. While encoding, these schemes search for the longest string of input characters that also appear in their dictionary. Once this string match is identified, the code of the matched string is used in place of the entire character string.

Yet another dimension of lossless compression algorithms is that they may be adaptive or non-adaptive. In adaptive schemes no prior knowledge about the input data is assumed and statistics are dynamically gathered and updated during the encoding phase itself. On the other hand, non-adaptive schemes are essentially ”two-pass” over the input data: during the first pass, statistics are gathered, and in the second pass, these values are used for encoding.

In this study, we have considered the popular compression techniques: Huffman, LZW and gzip (which uses LZ77 combined with Huffman encoding). The Huffman coding technique implement the statistical model, while the LZW scheme is dictionary-based. In addition to these techniques, we have also looked at the simple Run length encoding (RLE)[10] scheme which is supported in many current database systems (e.g. IMS [16]). The RLE scheme does not use either the statistical or the dictionary model - it simply recognizes successive repetitions of characters.

In the remainder of this section, we describe the salient features of the above-mentioned compression algorithms.

3.2.1 Huffman Coding

In Huffman coding [15], a tree is constructed with the characters of the input alphabet forming the leaves of the tree. The links in the tree are labeled with either 0 or 1 and the code for a character is the label sequence that is obtained by traversing the path from the root to the leaf node corresponding to that character in the Huffman tree. The tree is built such that the most frequent characters in the input data are assigned shorter codes and the less frequent characters are assigned longer codes.

As mentioned earlier, both adaptive and non-adaptive versions of Huffman coding exist. In non-adaptive Huffman coding, the Huffman tree is completely built before encoding starts, using the known frequency distribution of the characters in the data to be compressed. The tree remains unchanged for the entire duration of the encoding process. The decoder builds the same tree using the same frequency distribution before decoding the compressed data. On the other hand, adaptive Huffman coding starts off with a Huffman tree that is built using an assumed frequency distribution of the characters in the input data. A common practice is to assume that all characters are equally likely to occur. As the encoding process proceeds and more data is scanned, the Huffman tree is modified based on the data seen up to that point. Therefore, the Huffman tree changes dynamically during the encoding phase and the same character can have different codes depending on its position in the data being compressed (unlike non-adaptive Huffman).

3.2.2 LZW Coding

The LZW (Lempel-Ziv-Welch) algorithm [29] is a popular compression technique, used in both Unix compress and DOS pkzip. The scheme is organized around a string translation table. This table contains a set of character strings and their corresponding code values. The string table has a prefix property that for every string in the table its prefix is also in the table. That is, if string ωK, composed of some string ω and some single character K, is in the table, then ω is also in the table.

In LZW, the input data is scanned sequentially and the longest recognized input string (that is, a string which already exists in the string table) is parsed off each time. The recognized string is then replaced by its associated code. Each parsed input string, when extended by its next input character, gives a string that is not yet present in the string table. This new string is added to the string table and is assigned a unique code value. In this manner, the string table is built incrementally during the compression process.

3.2.3 gzip

The compression algorithm used by gzip [1] is a variation of LZ77(Lempel-Ziv 1977) algorithm [32]. It finds duplicated strings in the input data. The second occurrence of a string is replaced by a pointer to its previous occurrence, in the form of a pair (distance, length), where length is that of the string and distance is the number of symbols between the string and its last occurrence. Distances are limited to 32 KBytes, and lengths are limited to 258 bytes. When a string does not occur anywhere in the previous 32 KBytes, it is emitted as a sequence of literal bytes. Literals or match lengths are compressed with one Huffman tree, and match distances are compressed with another tree. The trees are stored in a compact form using Huffman coding at the start of compressed data. It efficiently decides on the fly whether to apply static Huffman coding or dynamic Huffman coding. Duplicated strings are found using a hash table. All input strings of length 3 are inserted in the hash table. A hash index is computed for the next 3 bytes. If the hash chain for this index is not empty, all strings in the chain are compared with the current input string, and the longest match is selected. The hash chains are searched starting with the most recent strings, to favor small distances and thus take advantage of the Huffman encoding. There are no deletions from the hash chains, the algorithm simply discards matches that are too old.

To avoid a worst-case situation, very long hash chains are arbitrarily truncated at a certain length, determined by a compression level (-1 to -9). So gzip does not always find the longest possible match but generally finds a match which is long enough. gzip also defers the selection of matches with a lazy evaluation mechanism. After a match of length N has been found, gzip searches for a longer match at the next input byte. If a longer match is found, the previous match is truncated to a length of one (thus producing a single literal byte) and the longer match is emitted afterwards. Otherwise, the original match is kept, and the next match search is attempted only N steps later. The lazy match evaluation is also subject to a compression level. If the current match is long enough, gzip reduces the search for a longer match, thus speeding up the whole process. If compression ratio is more important than speed, gzip attempts a complete second search even if the first match is already long enough. The lazy match evaluation is not performed for the fastest compression levels (speed options -1 to -3). For these fast modes, new strings are inserted in the hash table only when no match was found, or when the match is not too long. This degrades the compression ratio but saves time since there are both fewer insertions and fewer searches.

3.2.4 Run length Encoding

Run length encoding (RLE) [10] is an extremely simple and old compression technique. It takes advantage of consecutive repetitions (or runs) of the same character. For example, consider the string ”cccccaabbbb”. In normal 8-bit ASCII representation, the string would require 11 bytes (since the string has 11 characters). RLE, however, can encode the string in 8 bytes, as ”OEc5aaOEb4”. In this coding scheme, OE is a special character (usually a non-printable character such as ASCII 255), which denotes the beginning of a run. It is followed by the repeated character and the length of its run.

3.3 Compression Granularity

In a typical relational database system, compression can be conceptually applied at four different levels, namely the file level, the page level, the record level and the attribute level. At the file level, each relation in the database is compressed as a whole. Since compression techniques generally work better with larger data sets, we may expect that the best compression ratios would be realized at this level. In particular, adaptive models may find the large size favorable (since they have to dynamically gather statistics). With respect to query processing, however, file level compression requires the entire relation to be decompressed each time data in the relation is accessed. Further, if the relation is modified (insert, delete or update), the whole relation has to be re-compressed. This compression/decompression overhead may result in extremely slow query processing times.

An alternative scheme is for relations to be compressed at the page level. Since the page size in most commercial databases is around 4K, this gives scope for deriving good compression ratios. However, page compression has a problem similar to that described above for file compression: The whole page has to be decompressed and possibly re-compressed when data within the page is referenced or modified. This may again result in poor query processing times. Moreover, since the compressed pages are of variable size, additional complications arise: Firstly, a compressed page will occupy only a fraction of a disk block (assuming that the uncompressed pages are of disk block size, as is usually the case). Since disk transfer is usually in units of a disk block, fetching the compressed page will also bring in unnecessary data corresponding to other pages. Secondly, compressed pages may cross disk block boundaries. So, two disk block accesses have to be made to fetch the single compressed page to memory. Thirdly, when data in a page is updated, the size of the compressed page may change. In that case, the page has to be relocated to some other block, creating a hole in its previous position.

The next level of compression is the record level. Since, on average, record sizes vary between 40 to 120 bytes [16], the limited data size may cause a significant fall in the compression ratio, especially for adaptive algorithms. However, from the query processing viewpoint, this option is more attractive since only the records that potentially contribute to the result need to be decompressed and compressed. The decompression overhead is also limited due to the small size of a record. Since record level compression deals with fixed size pages, we can use disk-block sized pages to facilitate efficient disk I/O. Moreover, with fixed sized pages, the buffer manager of the DBMS does not have to be modified to account for compression. The only additional memory requirement is a record sized buffer which is used to hold uncompressed records when needed.

The lowest possible level at which compression can be done is at the attribute level. The common data types of attributes are integers, floating point numbers and character strings. Generally, the size of integers varies from 1 to 4 bytes, that of floating point numbers from 4 to 8 bytes and that of characters strings from 10 to 32 bytes [16]. This means that the data size is really limited and may result in poor compression ratios, especially for adaptive techniques. From the query processing viewpoint, however, the attribute level appears to be the most attractive because it permits precise queries, that is, where decompression is necessary only for the result tuples. This ability to execute a query entirely in the compressed domain appears extremely desirable from a performance viewpoint. Further, even if the need arises to decompress/compress an attribute, only that attribute and not the entire tuple needs to be decompressed/compressed. This is a distinct advantage over record level compression where entire tuples need to be decompressed/compressed. In addition, attribute level compression can be implemented with fixed sized pages, as in the case of record level compression.

Apart from having different compression ratios on the input data, each of the above granularities has different amounts of space overhead involved in their implementations. These overheads reduce the effective compression ratio. In certain situations, as explained in Section 3.4, the overheads may even exceed the compression ratio, leading to an expansion of the input file. Therefore, it is critical to include overhead effects in evaluating compression schemes, and this aspect is analyzed below.

3.4 Overheads

In file level compression, the file is compressed as a whole and no overhead is involved. In page level compression, pointers are stored for random access to the variable sized pages. The number of such pointers is equal to the number of pages in the relation, and therefore the space overhead as a fraction of relation size is insignificant. In record level compression, due to the variable-sized records, a pointer to the beginning of each record has to be maintained. So the percentage overhead is directly proportional to the compression ratio. At the attribute level, a pointer is stored for each attribute of each record. This overhead can, therefore, be significant, especially if the number of attributes in the relation is large. We will hereafter use the term pointer overhead to refer to the storage overhead arising out of pointers. An important point to note here is that the problem of tracking variable sized attributes is inherent to databases that support variable sized attributes (which many modern databases do). So, the pointer overhead that arises out of compression cannot really be considered as an additional overhead in these systems since the same overheads will also be present in the uncompressed case. However, in order to be conservative in our estimates of performance improvement due to compression, we assume that pointer overhead is present only for the compressed files.

3.5 Summary

From the above discussions, we observe that there are inherent difficulties in simultaneously achieving the desired goals of efficient query processing and good compression ratio.

Chapter 4

Indexes

Indexes are primarily used to speed up the retrieval of records in response to certain search conditions. They provide secondary access paths, which provide alternative ways of accessing the records without affecting the physical placement of the records on disk. They enable efficient access to records based on the indexing fields that are used to construct the index. The most prevalent types of indexes are based on ordered files and tree data structures like B$^+$ trees. Indexing can also be constructed based on hashing or other search data structures.

4.1 Indexing mechanism

After applying partitioning and compression as discussed in Sections 2 and 3 respectively, we have to study the effect of indexing. Since existing index structures are built on data that is not compressed, we need to adapt the indexing mechanism. In this regard we propose two indexing structures viz. *indexes* and *maps*.

Indexes: The indexes used here are traditional index structures like B$^+$-tree and Hash indexes. Instead of storing the file offsets for a given key we store the starting file offset of the compressed chunk containing that key. When we index using a given key value, we get the start file offset of compressed chunk from the index entry for that key. We then fetch the compressed chunk size S which is stored at the starting file offset of the chunk. We read the compressed chunk which is stored in the next S bytes. We decompress the chunk in memory using appropriate decompression algorithm, then do linear search to get the key. Since a chunk may contain multiple occurrences of the same key, we scan the entire decompressed chunk. Note that only the compressed chunk is transferred from the disk to memory, which saves I/O bandwidth. If there are duplicate keys stored but occurring in different chunks then we store both of them in index. However if a key occurs multiple times in chunk then only one entry is created in index. Thus the index has unique (key value,chunk offset) pair.

Maps : Map acts as translation table from row identifiers to file offsets of compressed chunks, i.e. given a row identifier they provide the starting offset of the compressed chunk containing that row in the file corresponding to given partition. Maps come handy when values in columns stored in different partitions of the same row needs to be combined. In such case we need to retrieve data corresponding to a given $rowid$ and hence need a map which provides the translation. By storing equal number of rows in each chunk, we do not need to store the range of row identifiers for a given chunk in the maps. Thus a map simply contains starting file offsets of compressed chunks in the file. The size of map is considerably smaller than actual indexes. Note that since we are dealing with file sizes greater than 2GB (which use 32 bit file offsets) we use 64 bit file offsets. If a partition has M rows and each compressed chunk has k rows then size of map will be $8M/k$ bytes, which is typically of order of hundreds of kilobytes for our datasets. This map can be stored in memory which speeds up access further. Moreover we need only one map per vertical partition.

Each partition essentially has a map whereas a given column may or may not have an index. The map is built for a partition whereas an index is built for a column. Note that we are not compressing the indexing structures to avoid processing complexity.

4.2 Design decisions about indexing

The attributes whose values are required in equality or range predicates and those that are keys or that participate in join predicates require fast access paths. The performance of queries largely depends upon what indexes or hashing schemes are exploited for the processing of selection and joins. Insertions, deletions or updates adds to the overhead of index maintenance. Since we are concerned with data warehouses which do not undergo frequent updates and are almost static over large window of time, we seldom encounter these overheads.

The physical design decisions for indexing are as follows:

1. Whether to index an attribute
2. What attribute or attributes to index on
3. Whether to set up a clustered index
4. Whether to use a hash index over a tree index

4.3 Handling Updates

Though data warehouses are not subjected to frequent updates, but still we look at this aspect of handling unexpected update which may occur due to incorrect computation while insertion which needs to be rectified. The index structure we proposed can handle updates. Moreover as we are using file offsets instead of disk addresses the data and the indexes can be shipped to other machine without any modification.

If a new row is added to data then, we simply need to do insertion of the key value into index. If addition of row into data creates a new chunk then an extra entry is appended to the map otherwise the map remains unchanged but the last chunk is decompressed, new row is added , the chunk is compressed and appended to existing partition.

When a value in a row needs to be modified to reflect corrections in the data, then the chunk size in which that row appears may change. If the chunk size becomes smaller than previous size then a hole will be created in the partition and no changes are made to the map. But if the chunk becomes larger then we can use standard database technique of creating a overflow chunk for this chunk at the end of the partition file. Hence an extra chunk will be required to read while processing this chunk. Under such situations the key needs to be deleted and the corrected key is inserted.

Chapter 5

Case Study: System-Y

The techniques presented in this paper are intended to be applicable to general data warehouse system. To evaluate our system on real data we conducted a case study on System-Y dataset obtained from a popular web-portal. The dataset is generated from web logs collected from various servers followed by some processing over it. The data is stored in plain text format and is compressed using gzip [1] algorithm. Their data warehouse system say *System-Y*, provides SQL-like interface to applications running on top of it which are primarily data mining tasks. The size of the gzip compressed file is around 3GB and the size of the original uncompressed data is around 28GB. The data had 13 columns having nine alphanumeric and four numeric fields. The columns along with their example values are explained in Table 5.1. The values A and N in the $Type$ column refer to alpha-numeric and numeric respectively. The total number of rows were close to 140 million. The queries that run on System-Y are primarily C++ programs that decompresses the whole gzipped file before processing it. The data is not partitioned vertically, hence even for a query involving only few columns all the columns will have to retrieved. The query execution consists of linear scan of the decompressed file or external sorting in case of *group-by* type queries. However the data is sorted on column BC so in order to avoid sorting for group-by queries, the system prevents users from asking a group-by query on other columns. System-Y does not use any indexes.

Broadly speaking the workload on System-Y can be classified into four types of queries given as follows.

Column name	Type	Example
RT	A	p/c
SI	N	15050006
BC	A	bj000q8tp5tt2&b=2
LC	A	jesusmiguel_chuy/y
RW	A	7q
IPA	A	18d8040f
AI	A	624029B̂3135262B̂150620
PC	A	m2bvvca60100g
TS	N	1000482594
OSE	N	1,2
BRE	N	1,2
MC	A	q1=PGPgABIAYYAAUA–;q2=PGMiqQ–
E	A	QUFBQOFDQUR

Table 5.1: Dataset Attributes

Q_1: `select A...B from T`

`where C relop` k

Q_2: `select A...B from T`

`where C relop` k_1 `logop D relop` k_2

Q_3: `select A...B from T`

`where C in list; list` is in memory list

Q_4: `select A...B from T group by C;`

where `A,B,C,D` are columns, `T` is relation k_1, k_2, k are constants, *logop* is logical operator (and,*or*) and *relop* is either of (≤,≥,=,<,>)

The aim was to improve the query processing for this setup. We modified the system so as to incorporate compression in such a way that indexes can be built directly on the compressed data. Our system does not need the entire file to be decompressed in order to process a part of the data. This is because we compress chunks of a vertical partition obtained after horizontal partitioning. The indexes store the file offsets of these compressed chunks, so in order to retrieve a row you just need to decompress the data in the chunk. Whereas in their system we would have to decompress the entire file and then scan each row to reach up to a particular row. Thus our system achieves better performance at the expense of extra indexing structures. However as will be shown in Section 6 there is minimal space overhead.

Chapter 6

Performance Evaluation

We conducted our experiments on Pentium IV 2.4GHz 512GB RAM, FreeBSD machine. Database routines and indexes were implemented using C API provided by Berkeley DB library 4.2.52. For gzip, Huffman and run-length encoding compression primitives we used *zlib* version 1.2.1 library. System-Y also uses zlib library for compression hence in order to do fair comparison we have also chosen it. We implemented the LZW algorithm in C.

6.1 Schema

The test workload consisted of queries of type $Q_1, \ldots Q_4$ as mentioned in Section 5. For the experimental workload the attribute affinity matrix is given in Table 2.3 and the attribute usage matrix is given in Table 2.2. After applying the graphical vertical partitioning algorithm to the affinity graph of the given test workload we obtain the partitioning as shown in Figure 6.1, which also depicts the steps of the partitioning algorithm. Our schema design consisted of 10 vertical partitions as shown in Figure 6.1. The node numbers in Figure 6.1 corresponds to column number.

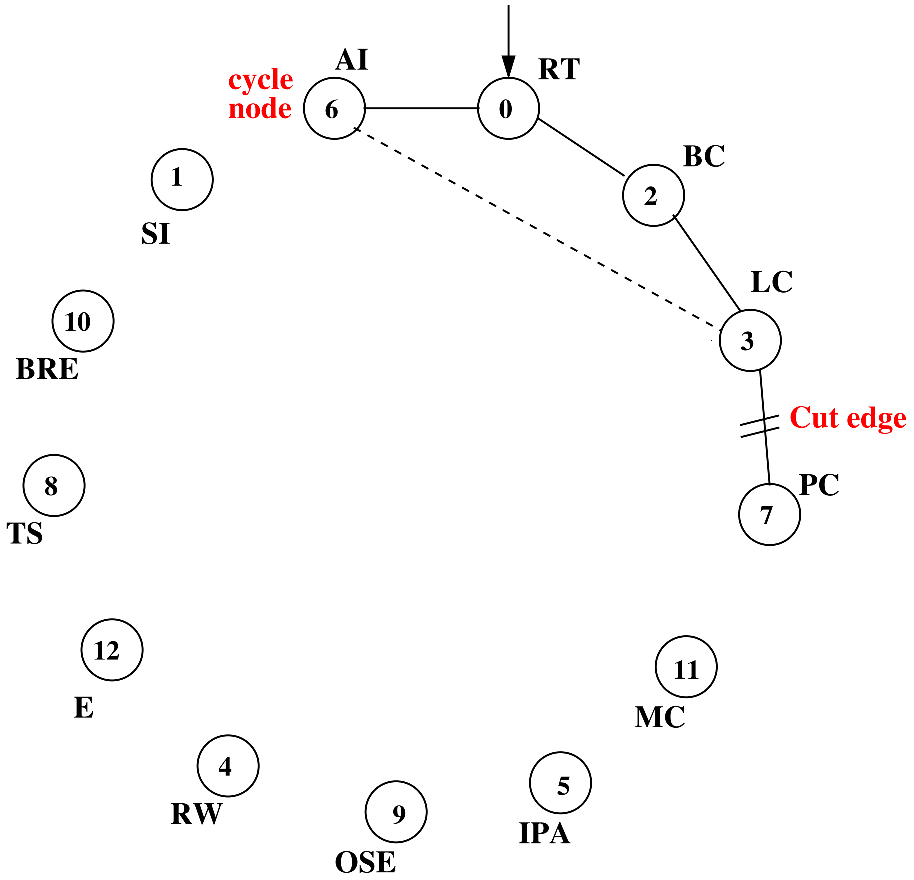


Figure 6.1: Partitions obtained after applying graph algorithm

6.2 Optimal chunk size

After finding the partitioning scheme using the workload the next step is creating the partitions. This involves determining the chunk size and compression algorithm. In order to determine the optimal chunk size so that decompression overhead is balanced by compression factor, we varied the chunk sizes from ten rows to the entire file size and computed the total space taken by all the partitions. We used gzip algorithm for this purpose. Alternative way is to compute optimal chunk size for each partition. so that we use different chunk sizes for different partitions. This will require storing the chunk sizes in the map itself.

From Figure 6.2 we find that the knee of the curve appears at 1000 rows hence the optimal chunk size is 1000 rows. For further experimentation we kept chunk size equal to 1000 rows.

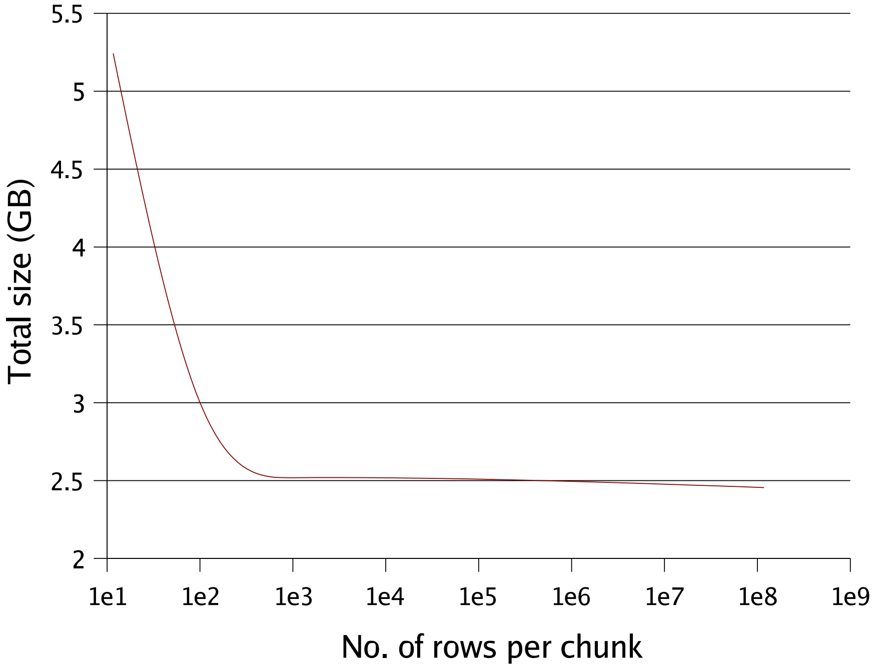


Figure 6.2: Determination of optimal chunk size

6.3 Cost model

To evaluate a candidate solution {partition set, indexed columns, type of each index, compression technique} from the search space , we need a cost model which gives the cost due to query execution time and storage space required. Our proposed cost model is as described in Tables 6.2 and 6.3. Note that the numbers expressed in the tables refer to algorithms that were used for the query execution. e.g., 1.4 refers to algorithm number 1.4 in Section 6.4.

6.4 Query execution algorithms

The algorithms that were used for executing the queries that belong to query types Q_1 to Q_4 are explained in this section. The cost of each of this algorithm is mentioned in the Tables 6.2 and 6.3.

Description	Notation
k^{th} column	C_k
No. of rows in original data	N
No. of columns in original data	M
No. of disk blocks in original data	B
No. of disk blocks in compressed file	B_C
No. of disk blocks occupied by column k	B_k
No. of disk blocks occupied by compressed column k	B_C^k
Ratio of range of values of a column and no. of blocks the values appear in	$spread$
Result cardinality(rows)	N_r
Original file size in Bytes	S
Cardinality of list for query type Q_4	$\|\text{list}\|$
Number of duplicate block addresses obtained from index	ndb
No. of separately compressed gzip chunks ($Z = 1$ if whole file is compressed together)	Z

Table 6.1: Notations for cost model

6.4.1 Query type Q_1, Q_2, Q_3

Horizontal

1.1 Read each block sequentially, decompress and evaluate the condition on C_k for each row. Output rows which satisfy the condition.

1.2 Binary search is performed on column C_k since it is already sorted.

1.3 Access the index on C_k and get the desired block offsets, sort them to remove duplicate block addresses and fetch the corresponding blocks. Decompress the block and check condition on C_k for each row, output the rows which satisfies the condition.

1.4 For Q_1, Q_2 (Same as 1.3). For Q_3 do sort-merge, i.e. sort list in memory and merge with C_k. This requires single scan.

	NO INDEX		INDEX ON C_k	
	Dataset not sorted on C_k	Dataset sorted on C_k	Data not sorted on C_k	Data sorted on C_k
Horizontal				
Storage	B_C	B_C	B_C + Index on C_k	B_C + Index on C_k
Disk Access	**1.1** B_C	**1.2** $log(B_C)$	**1.3** $(N_r - DB)\lceil(B_C/Z)\rceil$	**1.4** Disk Access for Q_1, Q_2 $Nr(Z/N)\lceil(BC/Z)\rceil$ Disk Access for Q_3 B_C sort-merge
Vertical				
Storage	B_C	B_C	B_C + Index on C_k	B_C + Index on C_k
Disk Access	**2.1** B_C^k $+$ $(N_r - ndb)\lceil B_C^i/Z_i\rceil m$	**2.2** $log(B_C^k)$ $+$ $N_r(Z/N)\lceil B_C^i/Z_i\rceil m$	**2.3** $(N_r - ndb)\lceil B_C^i/Z_i\rceil m$ $+$ Index Access	**2.4** Disk Access for Q_1, Q_2 $N_r(Z_i/N)\lceil(B_C^i/Z_i)\rceil m$ $+$ Index Access Disk Access for Q_3 $N_r(Z_i/N)\lceil(B_C^i/Z_i)\rceil m+$ B_C^k sort-merge

Table 6.2: Cost for query types Q_1, Q_2 and Q_3

Vertical

2.1 Read each compressed block of column C_k, decompress it and apply condition on each row. Get the row ids which satisfy the condition, get the corresponding rows of columns $C_i \ldots C_j$ using their maps.

2.2 Binary search on column C_k, followed by extracting $C_i \ldots C_j$ as in 2.1. For Q_3 do sort-merge as in 1.4.

2.3 Access the index on C_k to get the desired block offsets, sort them and remove duplicates. Get row ids in C_k that satisfy condition and get corresponding rows using maps of $C_i \ldots C_j$.

2.4 Same as 2.3 for Q_1, Q_2. For Q_3 do sort-merge.

	NO INDEX		INDEX ON C_k	
	Data not sorted on C_k	Data sorted on C_k	Data not sorted on C_k	Data sorted on C_k
Horizontal				
Storage	$B_C + (B_C^i + \ldots + B_C^j) + B_C^k$	B_C	B_C + Index on C_k	B_C + Index on C_k
Disk Access	**1.1** B_C + $m(B_C/M)log(mB_C/M)$	**1.2** B_C	**1.3** Index block access + $(N_r - ndb)\lceil BC/Z \rceil$	**1.4** B_C
Vertical				
Storage	$B_C + B_C^k$	B_C	B_C + Index on C_k	B_C + Index on C_k
Disk Access	**2.1** $(B_C^k + B_k log(B_k) + B_C^i + \ldots + B_C^j$	**2.2** $B_C^k + B_C^i + \ldots + B_C^j$	**2.3** Index access + B_C^k + $(N_r - ndb)\lceil B_C^i/Z_i \rceil m$	**2.4** $B_C^k + B_C^i + \ldots B_C^j$

Table 6.3: Cost for query type Q_4

6.4.2 Query type Q_4

Horizontal

1.1 Extract column $C_i \dots C_j, C_k$ and Sort on C_k. Find groups in one scan of this sorted data.

1.2 Find groups in one scan of this sorted data.

1.3 For each entry in index of C_k get the set of block offsets, these corresponds to a group. Access $C_i \dots C_j$ corresponding to the rows in a group.

1.4 Same as 1.2

Vertical

2.1 Sort the column C_k, group, get values in desired order from $C_i \dots C_j$ using map

2.2 Group on C_k in single scan, get values in desired order from $C_i \dots C_j$ using map

2.3 Using index on C_k, make groups, get column $C_i \dots C_j$ values in desired order using map

2.4 Same as 2.2

6.5 Compression Performance

The characteristics of data set are as follows. The data had around 139 million rows and the total size was 27 GB uncompressed and 2.5 GB gzip compressed. We evaluated the compression algorithms discussed in Section 3 on the given dataset for compression ratios. This evaluation is depicted in Figure 6.3 which shows the total space consumption after applying compression on vertical partitioned chunk-compressed data. The total space consumption as shown in Figure 6.3 consisted of partitions, indexes and maps. In Figure 6.3 *Gzip-chunk*, *RLE*, *Huf* and *LZW*corresponds to system which uses chunk-level compression on vertically partitioned

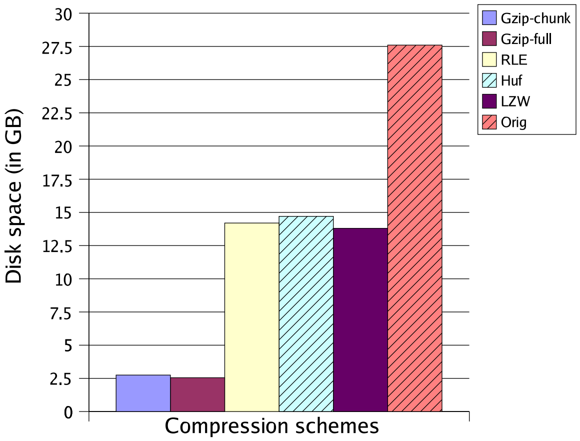


Figure 6.3: Total Space Requirement

data using gzip, run-length encoding, Huffman encoding and LZW compression algorithms respectively. *Gzip-full* corresponds to System-Y which uses file-level compression using gzip algorithm. *Orig* corresponds to original uncompressed file size. The map size was same for each partition i.e. 1.1 MB.

It is quite clear from Figure 6.3 that Gzip-chunk and Gzip-full achieve almost same compression ratio. The reason for this behavior is that after we increase the window for finding recurring patterns beyond few kilobytes the compression algorithm does not improve on compression ratio. LZW, RLE, Huf behave similarly achieving poor compression as compared to gzip algorithm. Thus we conclude that to achieve better compression ratio gzip algorithm should be used which achieves a compression factor of about 10.

6.6 Execution time

For the given setup we compared the execution times of each query type on our system against System-Y as shown in Figures 6.4 and 6.5. The comparison was done on systems with index and without index varying the compression algorithms. Note that that the data configurations are the same for these experiments.

6.6.1 Without Index

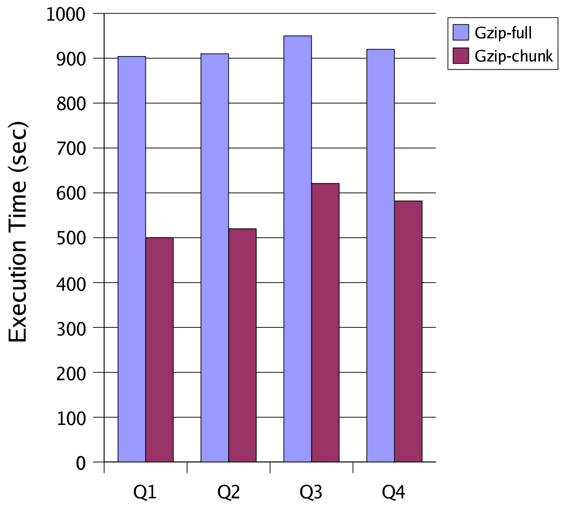


Figure 6.4: Execution Time without index

The values on X-axis are the query types Q_1 to Q_4. From Figure 6.4 it is evident that for all the query types partitioned schema with compression provides better performance. This is because only the partition containing the columns in the query are accessed there by saving extra disk accesses and processing of undesired columns. For query types Q_1 and Q_2, gzip-chunk achieves around 45% improvement and for Q_3 and Q_4, it achieves 37% improvement. Thus gzip-chunk is a desirable technique for compressing partitioned data.

6.6.2 With Index

The indexes were built on column 1 (SI) and column 2 (BC) and were B$^+$-tree indexes. The index construction time is not taken into account as it is amortized over executions of multiple queries. Note that *Gzip-full* scheme corresponds to System-Y. The experiments were done only for query types Q_1 and Q_2 because only these queries required index for their execution. From Figure 6.5 it is evident that for query types Q_1 and Q_2 (which took just below a second to execute) we achieve four orders of magnitude improvement due to use of indexes which System-Y lacks. The total size of the indexes was around 300 MB which is 10% of compressed

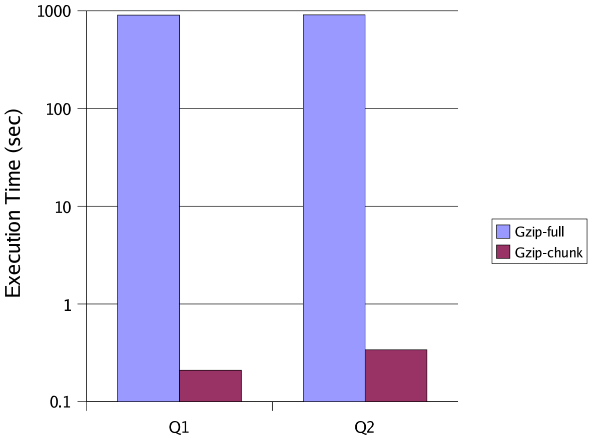


Figure 6.5: Execution Time with index

data size and 1% of uncompressed data size. For these queries System-Y will have to do single scan of entire data, whereas we need to read the appropriate chunk containing that key value. It is not surprising that the our technique with index is more efficient than the one without it.

Chapter 7

Related Work

Most data ware house implementation are centered around relational database. We have considered data warehouse whose size cannot be handled by traditional RDBMS. The general ideas of database compression have been discussed in [27, 28, 7, 3, 26]. These studies evaluate the various types of compression techniques from a storage perspective but do not consider their suitability for database query processing. Considerable work has been done in the field of compression of scientific and statistical databases (e.g. [4, 9, 21, 2, 19]). However, the nature of data and the types of operations in these databases differ significantly from those in commercial databases, which is our target domain. While [11] brings out the potential of compression with respect to query processing, [16] is an excellent investigation of the implementation practicality of these ideas.

Several vertical partitioning algorithms have been proposed in the literature. Hoffer and Severance [14] measure the affinity between pairs of attributes and try to cluster attributes according to their pairwise affinity by using the bond energy algorithm (BEA) [21]. Hammer and Niamir [13] use a file design cost estimator and a heuristic to arrive at a ”bottom up” partitioning scheme. Navathe, et al [23] extend the BEA approach and propose a two phase approach for vertical partitioning. Cornell and Yu [8] apply the work of Navathe [23] to physical design of relational databases. Ceri, Pernici and Wiederhold [6] extend the work of Navathe [23] by considering it as a ‘divide’ tool and by adding a ‘conquer’ tool. In addition to these vertical partitioning algorithms, there are many data clustering techniques [17], traditionally used in pattern recognition and statistics, some of which can be adapted to partitioning of a database. These data clustering algorithms include Square-error clustering [17], Zahn's clustering [31], Nearest-neighbor clustering [20] and Fuzzy clustering [17].

Chapter 8

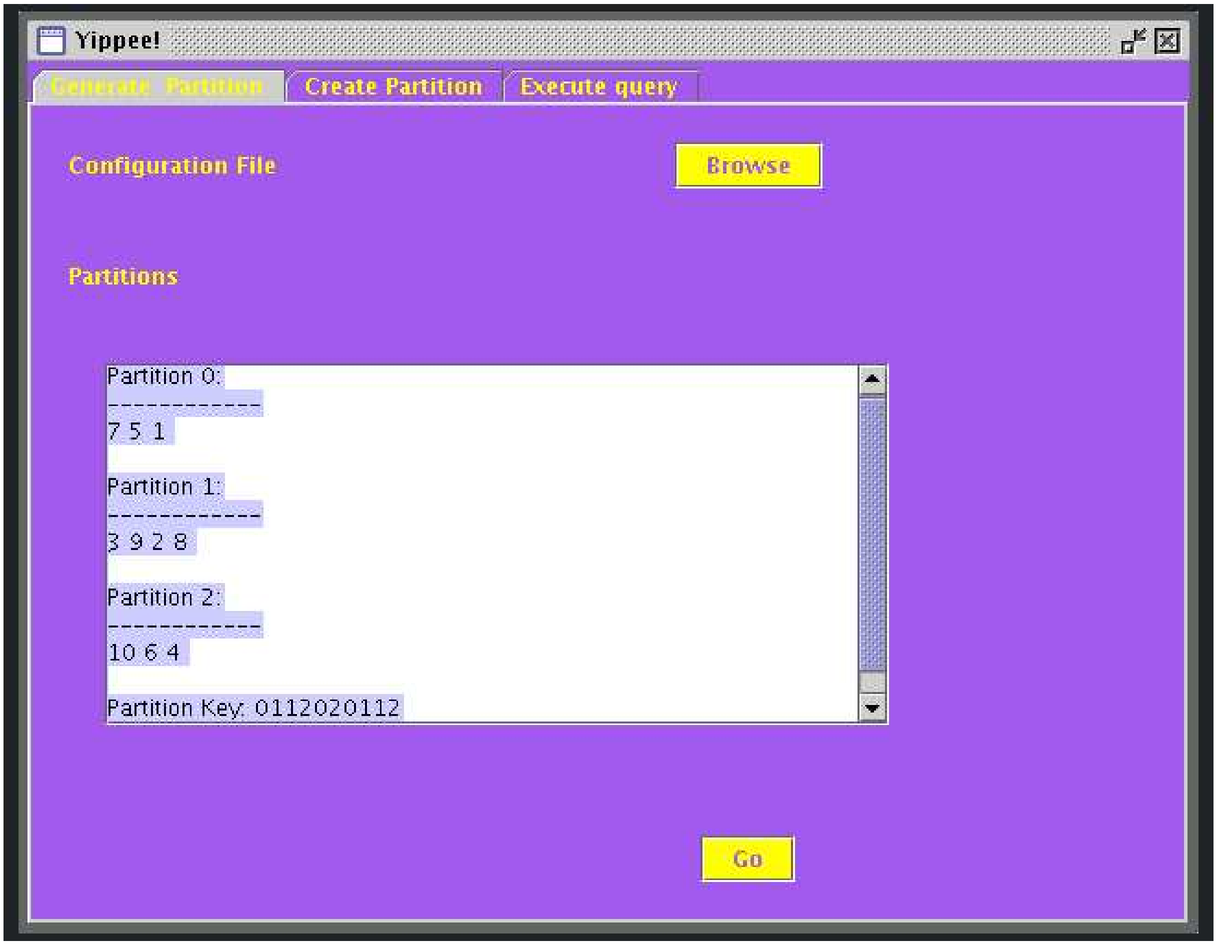
Yippee: Software prototype

In this chapter we present a software prototype of the system we developed which integrates the concepts of partitioning, compression and indexes.

8.1 Features

The software prototype developed implements the techniques we have proposed in this report. The system is called Yippee and it serves as a tool to data warehouse designers. The backend is implemented in C and Perl which is around 7000 LOC whereas the GUI is implemented using Java Swing (1000 LOC).

The snapshots of Yippee are shown in Figures 8.1, 8.2, 8.3. Yippee has three modules. First one takes the query workload as input and comes up with partitioning scheme based on graph-based algorithm as discussed in Section 2. The second module takes the partitioning scheme as input. It has various configurable parameters like number of rows per chunk and compression algorithm. Currently, Yippee has support for compression algorithms like gzip, Huffman, RLE and LZW. The third module is the query execution engine, which given a user query executes it with the specified configuration. It gives the query output together with time statistics.



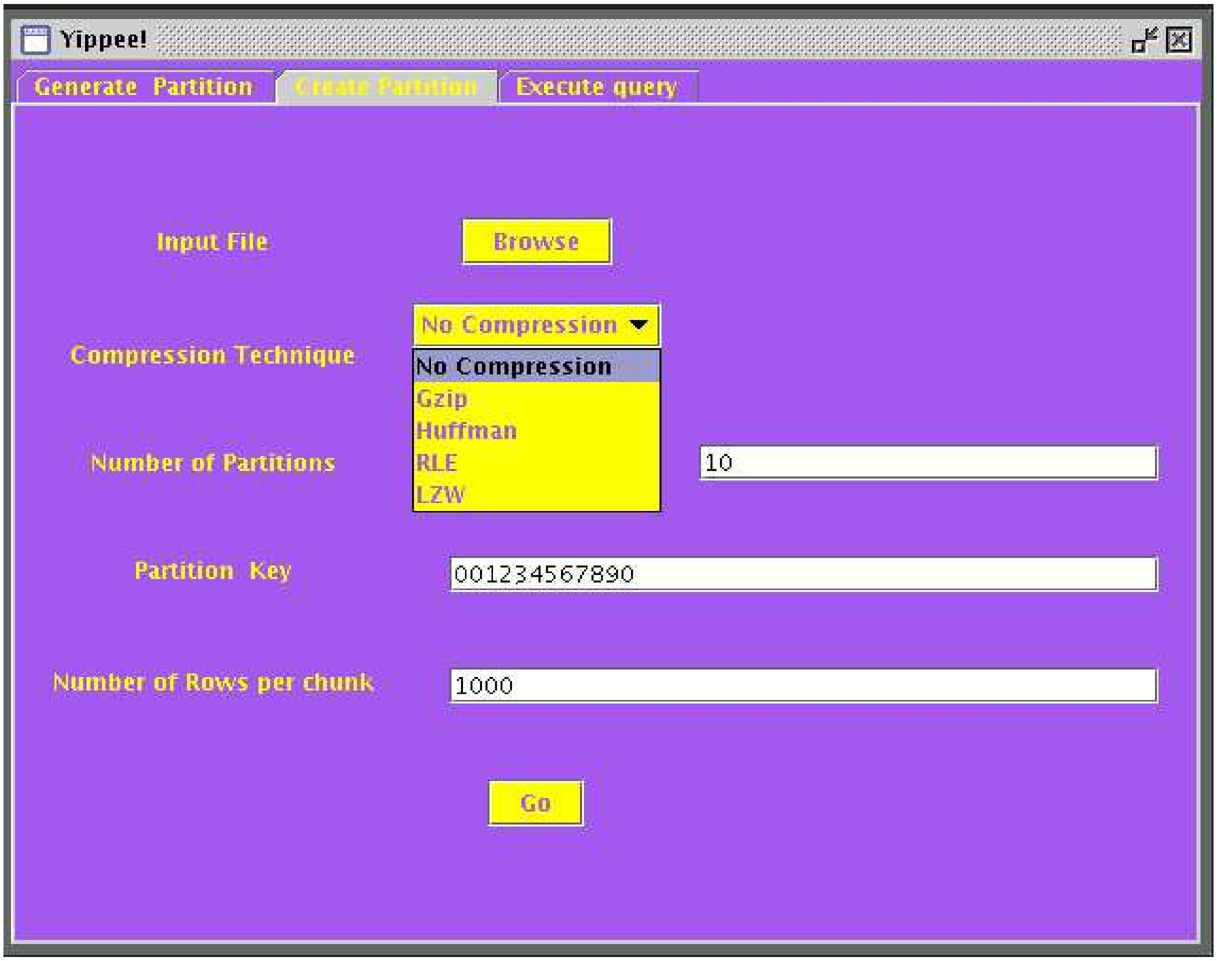
```
Partition 0:
------------
7 5 1

Partition 1:
------------
3 9 2 8

Partition 2:
------------
10 6 4

Partition Key: 0112020112
```

Figure 8.1: Yippee: Generate Partition module



Input File

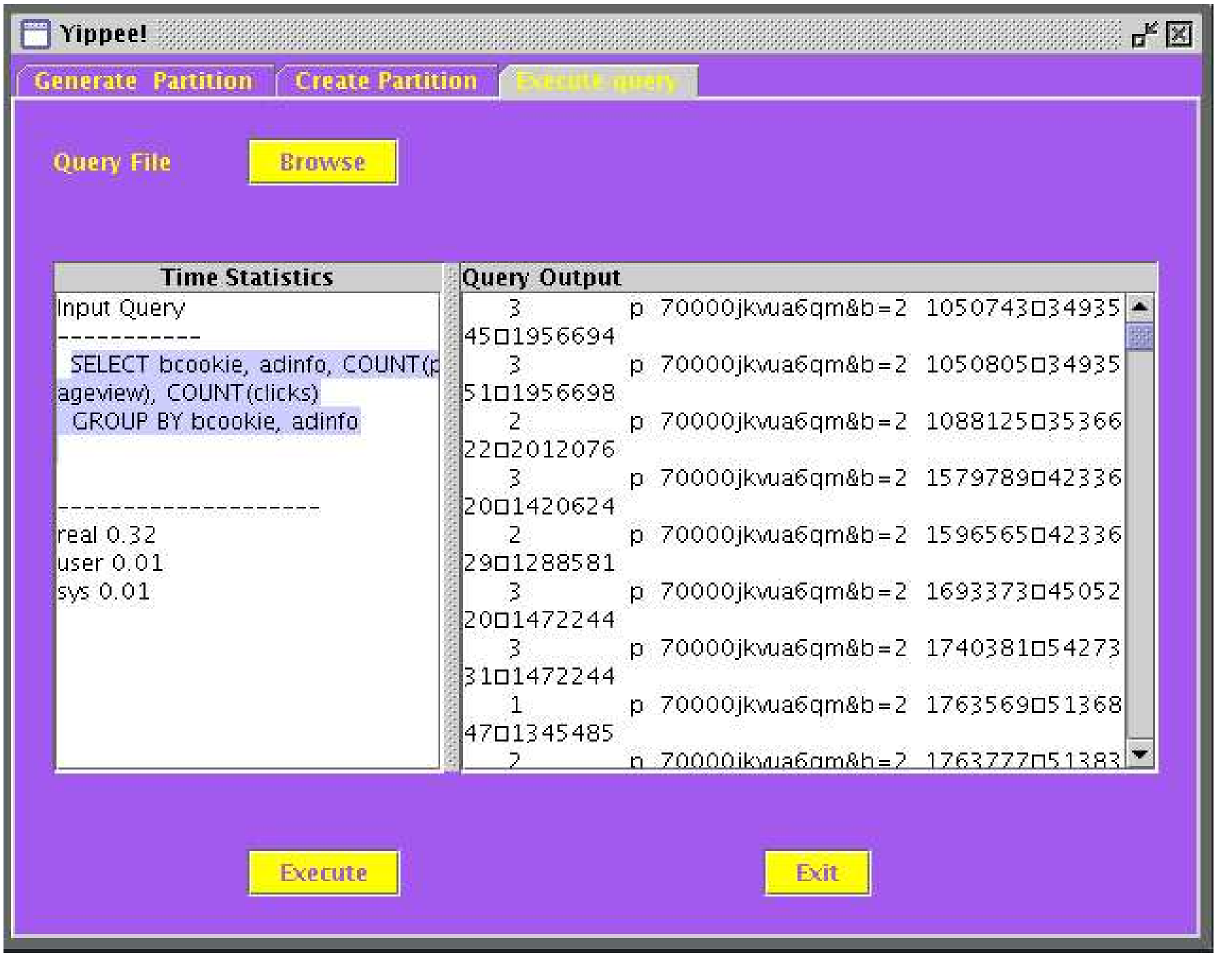
Compression Technique: No Compression

Number of Partitions: 10

Partition Key: 001234567890

Number of Rows per chunk: 1000

Figure 8.2: Yippee: Create Partition module



```
Input Query
-----------
 SELECT bcookie, adinfo, COUNT(p
ageview), COUNT(clicks)
 GROUP BY bcookie, adinfo



--------------------
real 0.32
user 0.01
sys 0.01
```

```
     3          p  70000jkvua6qm&b=2  1050743□34935
45□1956694
     3          p  70000jkvua6qm&b=2  1050805□34935
51□1956698
     2          p  70000jkvua6qm&b=2  1088125□35366
22□2012076
     3          p  70000jkvua6qm&b=2  1579789□42336
20□1420624
     2          p  70000jkvua6qm&b=2  1596565□42336
29□1288581
     3          p  70000jkvua6qm&b=2  1693373□45052
20□1472244
     3          p  70000jkvua6qm&b=2  1740381□54273
31□1472244
     1          p  70000jkvua6qm&b=2  1763569□51368
47□1345485
     2          p  70000jkvua6qm&b=2  1763777□51383
```

Figure 8.3: Yippee: Query Execution module

8.2 System Architecture

The system architecture of Yippee is shown in Figure fig:yippee. It automatically generates vertical partitions based on workload information, compression algorithm and chunk size. It also allows the user to specify SQL queries which run on the vertically partitioned data.

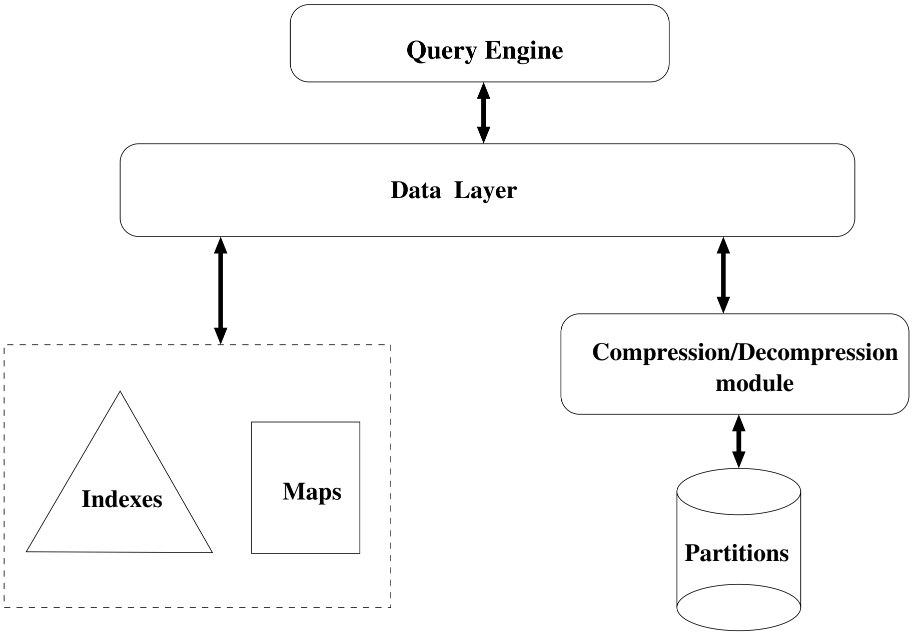


Figure 8.4: System Architecture of Yippee

The architecture consists of the following components:

- **Query Engine**

  It provides support for executing SQL queries. The current version of Yippee has limited support for SQL. It only supports equality, range, group-by and order-by queries.

- **Data Layer**

  This layer provides the required data to the SQL operators from the storage. It gets the requests from query engine and it then uses the indexes and maps and fetches the required data from the partitions stored on the disk. The data layer maintains sufficient information about the indexes and maps and the mappings from logical table names to files and column names to partitions.

- **Compression/Decompression module**

  This module receives requests from data layer in form of (partition name, chunk offset), this layer than fetches the chunk from the given file storing that partition, decompresses it using appropriate compression algorithm and returns the decompressed chunk to data layer.

Chapter 9

Conclusions and Future Work

We have analyzed the effect of schema design, compression and indexing on the performance of file-based tera-byte size data warehouse system. We have tried graphical vertical partitioning algorithm for schema design. We have also studied the effect of various compression techniques on space requirement and execution time of query. We modified existing index structure to couple with vertically partitioned chunk compressed data format. We have shown that the techniques suggested in this report lead to four orders of performance benefit for selection queries and 45% improvement in execution time for group by queries, with a small space overhead of 10% due to index structures. We have developed a software prototype - Yippee which integrates the concepts proposed in this paper. Yippee serves as a tool to database schema designer.

We look forward to automate some parts of Yippee. We plan to add automated index selector to Yippee. We also want to modify our system to scale to distributed environment.

Bibliography

[1] The zlib library. http://www.gzip.org/zlib.

[2] M. Bassiouni and K. Hazboun. Utilization of Character Reference Locality for Efficient Storage of Databases. In *Proceedings of International Workshop on Statistical Database Management*, 1983.

[3] M. A. Bassiouni. Data Compression in Scientific and Statistical Databases. In *IEEE Transaction on Software Engineering*, October 1985.

[4] D. Batory. Index Coding: A Compression Technique for Large Statistical Databases. In *Proceedings of International Workshop on Statistical Database Management*, 1983.

[5] E. T. Bell. Exponential numbers. In *American Math. Monthly 41, 411-419*, 1934.

[6] S. Ceri, S. Pernici, and G. Weiderhold. Optimization Problems and Solution Methods in the Design of Data distribution. In *Information Sciences Vol 14, No. 3, p 261-272*, 1989.

[7] G. V. Cormack. Data Compression in Database Systems. In *Communication of ACM*, 1985.

[8] D. Cornell and P.S. Yu. A Vertical Partitioning Algorithm for Relational Databases. In *Proceedings of ICDE*, 1987.

[9] S. Eggers, F. Olken, and A. Shoshani. A Compression Technique for Large Statistical databases. In *Proceedings of VLDB*, 1981.

[10] S.W. Golomb. Run-length encoding. In *IEEE Transactions on Information Theory*, 1966.

[11] G. Graefe. Options in Physical Database. In *SIGMOD Record*, September 1993.

[12] G. Graefe and L. Shapiro. Data Compression and Database Performance. In *Proceedings of ACM/IEEE Computer Science Symposium on Applied Computing*, April 1991.

[13] M. Hammer and B. Niamir. A heuristic approach to attribute partitioning. In *Proceedings ACM SIGMOD Int. Conf. on Management of Data*, 1979.

[14] J.A. Hoffer and D.G. Severance. The Use of Cluster Analysis in Physical Database Design. In *Proceedings of VLDB*, 1975.

[15] D.A. Huffman. A Method for Construction of Minimum- Redundancy Codes. In *Proceedings of the IRE*, 1952.

[16] B. R. Iyer and D. Wilhite. Data Compression Support in Databases. In *Proceedings of VLDB*, September 1994.

[17] A. Jain and R. Dubes. *Algorithms for clustering Data*. Prentice Hall Advanced Reference Series, Englewood Cliffs, NJ, 1988.

[18] Ralph Kimball. *The Data Warehousing Toolkit*. John Wiley and Sons, 1996.

[19] E. Lefons, A. Silvestri, and F. Tangorra. An Analytic Approach to Statistical Databases. In *Proceedings of VLDB*, 1983.

[20] S. Lu and K. Fu. A sentence-to-sentence clustering procedure for pattern analysis. In *IEEE Transactions on Systems, Man and Cybernetics SMC 8, 381-389*, 1978.

[21] W.T. McCormick, P. Schweitzer, and T.W White. Problem Decomposition and Data Reorganization by a Clustering Technique. In *Operations Research*, 1972.

[22] S.B. Navathe and S. Ceri. A Comprehensive Approach to Fragmentation and Allocation of Data in Distributed Databases. In *IEEE Tutorial on Distributed Database Management*, 1985.

[23] S.B. Navathe, S. Ceri, G. Wiederhold, and J. Don. Vertical Partitioning Algorithms for Database Design. In *ACM Transactions on Database Systems*, 1984.

[24] S.B. Navathe and Minyoung Ra. Vertical Partitioning for Database Design : A Graphical Algorithm. In *Proceedings of SIGMOD*, 1989.

[25] G. Ray, J. R. Haritsa, and S. Seshadri. Database Compression: A Performance Enhancement Tool. In *International Conference on Management of Data*, 1995.

[26] H. K. Reghbati. An Overview of Compression Techniques. In *IEEE Computer*, 1981.

[27] M. A. Roth and S. J. Van Horn. Database compression. In *SIGMOD Record*, 1993.

[28] G. D. Severance. A Practitioner's Guide to Database Compression - A Tutorial. In *Information Systems*, 1983.

[29] T.A. Welch. A Technique for High Performance Data Compression. In *IEEE Computer*, 1984.

[30] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic Coding for Data Compression. In *Communications of ACM*, 1987.

[31] C. Zahn. Graph-theoretical methods for detecting and describing Gestalt Clusters. In *IEEE Transactions on Computers*, 1971.

[32] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. In *IEEE Transactions on Information Theory*, 1977.