An Algebraic Compression Framework for Query Results[1]

Zhiyuan Chen and Praveen Seshadri
Cornell University
zhychen, praveen@cs.cornell.edu, contact: (607)255-9124, fax:(607)255-4428

Abstract

Decision-support applications in emerging environments require that SQL query results or intermediate results be shipped to clients for further analysis and presentation. These clients may use low bandwidth connections or have severe storage restrictions. Consequently, there is a need to compress the results of a query for efficient transfer and client-side access. This paper explores a variety of techniques that address this issue. Instead of using a fixed method, we choose a combination of compression methods that use statistical and semantic information of the query results to enhance the effect of compression. To represent such a combination, we present a framework of "compression plans" formed by composing primitive compression operators. We also present optimization algorithms that enumerate valid compression plans and choose an optimal plan. Our experiments show that our techniques achieve significant performance improvement over standard compression tools like WinZip.

1. Introduction

Most database applications have multi-tier client-server architectures. The database back-end server is used to run queries over the stored data. The query results are shipped across a hierarchy of clients, with possible transformations along the way. Increasingly, these "clients" reside on a desktop machine, a laptop, or a palmtop computer. In this paper, we apply compression to the results of SQL queries.

1. In OLAP applications, the query results need to be moved to the clients for ready visualization. The clients may download the query results across an expensive and/or a slow connection such as a wireless network.

2. In mobile computing applications, the clients are usually disconnected, and consequently need to download large query results for offline use. For the emerging class of palmtop computers, storage is severely constrained. For instance, the only storage for a PalmPilot III is 2MB of RAM (there is no hard disk). Usually users only need to access a small portion of the results. For instance, a user may browse the results and only access those tuples that can be shown on the screen. Compression is a very effective approach under such situations because the clients can keep the results compressed and only decompress the portion being accessed. Therefore, for efficient transfer and client-side storage conservation, the query results need to be compressed.

3. In heterogeneous database systems, a query is also often divided into sub-queries executed on several sites separately and each site will transfer intermediate results across an often slow and unreliable network. Recent research [MP99] has demonstrated the use of client-side functions (UDFs) within SQL queries. This requires that partial results be shipped from the server to the client. A slightly different motivation underlies the research on semantic caching [DFJ96] that also expects that the results of queries are cached at clients.

This work is motivated by all these environments, where the ability to compress query results enhances the usability, functionality and/or performance of the application.

1.1. Summary of contributions

Data compression has traditionally been applied to database indexing structures [WAG73, COM79, GRS98]. While there has been some work on compression techniques for query evaluation [RH93, RHS95, GRS98, GS91], this activity is typically restricted to the internals of decision-support systems. There is also much existing research in the data compression community, mostly focused on compression algorithms for specific data types (like text and multimedia). Three issues make it non-trivial to apply compression in database systems:

1. Compression and decompression are CPU intensive operations. The cost is especially important at low-end clients.

2. Database applications often need random access to data in small pieces (tuples, attributes, etc). This makes some popular compression methods such as LZ77 (used in WinZip, PKZIP, and Gzip) not appropriate for low-end clients. The reason is such methods only work well for large

[1] This work on the Cornell Jaguar project was funded in part through an IBM Faculty Development award and a Microsoft research grant to Praveen Seshadri, through a contract with Rome Air Force Labs (F30602-98-C-0266) and through a grant from the National Science Foundation (IIS-9812020).

chunks of results, and a large chunk of results must be decompressed even if only a small piece is needed. Therefore, these methods have prohibitive decompression costs for low-end clients because both the processors of these clients are slow and the power supplies are limited (these clients usually use batteries).

3. A DBMS holds domain specific knowledge of the query results, which can be used to enhance the effect of compression. For instance, values of the same attribute have many similarities while values of different attributes usually have very different characteristics. This implies that a single method may not be appropriate for all attributes. Instead, a combination of compression methods that compress each attribute separately based on the domain knowledge usually can achieve better effect [BBC 99]. The choice of an appropriate combination is not obvious.

Our research makes the following contributions:

1. We analyze SQL queries and demonstrate how the semantics of the queries, the statistics computed in query processing or query optimization, and the underlying tables can identify compression opportunities.

2. We present an algebraic framework to represent the compression of an entire query result using the combination of many methods. We define primitive *compression operators*, and a *compression plan* as a sequence of compression operators.

3. We demonstrate through implementation that well-chosen compression plans can result in significantly better (89% on average in our experiments) compression ratios (the size of uncompressed results divided by the size of compressed results) than standard compression methods like LZ77. We apply compression plans to modified versions of all the queries in the TPC-D benchmark and report on the resulting compression ratios. We also show the superiority of well-chosen compression plans in an experiment on handheld devices.

4. The choice of an appropriate compression plan requires "compression optimization". The performance of a combination of compression methods depends on various factors such as the concerns of the client (whether the storage is limited or the processor is slow, or both), the characteristics of the query results, the network transmission rate, etc. Further, compression decisions may have to be made before the query is executed and before the query result is materialized. We present a cost-based exhaustive algorithm to enumerate valid compression plans and choose the best plan, as well as a faster heuristic algorithm to choose a reasonably good plan.

We briefly summarize related work in the next section. Section 2 shows how the use of domain knowledge can enhance the effect of compression. A general framework for the combination of compression methods is presented in section 3. Section 4 demonstrates the effect of compression plans. Section 5 discusses compression optimization issues. Section 6 presents the conclusion and the future work.

1.2. Brief summary of related work

A number of researchers have considered compression methods on text and multimedia data. The methods they consider can be roughly divided into two categories, statistical and dictionary methods. Statistical methods include Huffman encoding [Huf52], arithmetic encoding [WNC87], etc. Dictionary methods include LZ78 [WEL84], LZ77 (LZ77 is used in popular compression tools like WinZip and Gzip) [LZ77, LZ76], etc. A comprehensive survey of compression methods can be found in [SAL98].

Most related work on database compression has focused on the development of new algorithms and the application of existing techniques within the storage layer [ALM96, COR85, GOY83, GRA93, LB81, SEV83]. [GRS98] discusses page level offset encoding on indexes and numerical data. The tuple differential algorithm is presented in [NR95]. COLA (Column-Based Attribute Level Non-Adaptive Arithmetic Coding) is presented in [RHS95]. Considerable research has dealt with compression of scientific and statistical databases [BAS85, EOS81]. In commercial database products, SYBASE IQ [SYB98] uses a compression technique similar to LZ77. DB2 [IW94] also uses Ziv-Lempel compression method in the storage layer.

Other researchers have investigated the effect of compression on the performance of database systems [RH93, RHS95, GRS98]. [GS91] discusses the benefits of keeping data compressed as long as possible in query processing. Our research is complementary to the above work because we focus on the compression of query results leaving the DBMS. To the best of our knowledge, there is no research that directly addresses this topic.

Another related research area is mobile databases. Much work has been done on caching [CSL98], data replication and data management [HSW94], and transaction processing [DL98]. However, our focus is on the compression of query results shipped to handheld devices.

2. Compression methods

2.1. Data compression methods

2.1.1. Standard lossless compression methods. Table 2.1 presents a list of the standard lossless compression methods that are used in this paper. Each method is described briefly, and detailed algorithmic explanations can be found in [RH93]. While this list is not exhaustive, any specific database system will implement a finite number of compression algorithms, and our work assumes this finite list.

While we intend to leverage well-known data compression algorithms, we also present two database-specific compression methods - normalization and grouping - in the next two subsections.

Name	Description
Differential encoding	Compute the difference of adjacent values (same column or same tuple).
Offset encoding	Encode the offset to a base value.
Null suppression	Omit leading zero bytes for numerical values and leading or ending spaces for strings.
Non adaptive dictionary encoding	Use a fixed dictionary to encode data.
LZ77, LZ78	Adaptive dictionary encoding.
Huffman encoding	Assign fewer bits to represent more frequent characters.
Arithmetic encoding	Represent a string by interval according to probabilities of each character.

Table 2.1 standard compression methods.

2.1.2. Normalization as a compression method. Stored tables are usually normalized to eliminate data redundancy. However, when queries involve foreign key joins, the results become unnormalized. For instance, assume there are two tables *R1 (A,B)* and *R2 (A,C)*. *A* is a primary key in *R1* and a foreign key in *R2*. Therefore, there exists a functional dependency *A->B*. Assume the query is:

*select * from R1, R2 where R1.A = R2.A and R1.A < 1000 order by R1.A;*

Suppose the result contains 1000 distinct *A* values and 100,000 tuples. *A* and *B* values are stored 100,000 times instead of 1000 times (they only have 1000 distinct values)! Such redundancy can be reduced by normalizing the result into two relations, *AB* and *AC* such that *B* values are only stored 1000 times.

The simple normalization technique has two problems (a) values of attribute *A* still have redundancy, (*A* values are stored 101,000 times) (b) a join is needed to regenerate the result. However, when the results are already sorted on *A* (the LHS of the functional dependency), there is a more efficient normalization algorithm that partitions the table into equality partitions on *A* (and thereby on *B* as well). The first tuple in each partition stores all attribute values while the other tuples in this partition only store *C* values. There is also a bitmap indicating the first tuple in each partition such that the table can be decompressed at the attribute-level. This physical representation of the normalized tables has little redundancy (and hence occupies fewer bytes). We call such a method *sorted-normalization*. However, the application of this method is more limited than that of the naïve normalization because the former requires certain order of results.

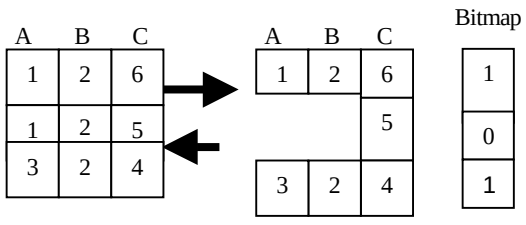


Figure 2.1 Application of sorted-normalization method on attributes A and B.

Figure 2.1 shows the procedure of compression and decompression using sorted normalization.

2.1.3. Grouping as a compression method. Even if there is no functional dependency, a similar method is to group together duplicate attribute values and merely record each distinct value once along with the matching values of other attributes. If the query results are sorted, the grouping method can be applied on the sorted attributes without significant additional cost.

2.2. Using compression methods

Clearly, it is possible to apply a general-purpose compression technique like LZ77 compression to the result of any query. However, this approach has following drawbacks:

(a) A large chunk of result tuples needs to be decompressed in order to access individual tuples or attributes. The decompression overhead may defeat the purpose of compression in some cases.

(b) The LZ77 method uses no available domain knowledge about the query result.

(c) Since each table consists of different types of data in each column, there is no reason to believe that a single compression method is ideal for the whole table.

Consequently, we are faced with the non-trivial issue of composing multiple compression methods in a consistent and efficient way. We discuss the consistency issue in Section 3 (what is a valid compression plan?) and the efficiency issue in Section 5 (how is a good compression plan chosen?). In the rest of this section, we further motivate our work by demonstrating that domain knowledge can be effectively used to accomplish better compression ratios on query results.

2.3. Sources of domain knowledge for compression

A database system can acquire the domain knowledge about the result of a query from:

- The semantics of the query.
- The statistics computed during query optimization. Many techniques [JKM98, HS92, PIHS96] such as histograms and sampling have been proposed to maintain such information[2].

[2] If we assume that the query result is materialized, then it is possible to obtain much of the necessary statistical information during or after query processing. However, we do not make this simplified assumption, and instead use estimates from query optimization so that we can decide the compression strategy before query processing. Either approach is valid.

- The statistical and semantic information kept in the catalog, which serves as the basis for statistics derived during query optimization.

Example 2.1. We use an example to illustrate the use of such semantic and statistical information. Example 2.1 selects for each supplier, his nation, phone number, and revenue on every possible order date and ship date. The underlying database is a 100 MB scaled database from the TPC-D benchmark [TPC95].

select *S_SUPPKEY, N_NAME, S_PHONE, O_ORDERDATE, L_SHIPDATE, SUM (L_EXTENDEDPRICE*(1- L_DISCOUNT)) AS REVENUE*
from *LINEITEM, SUPPLIER, NATION, ORDER*
where *L_SHIPDATE < O_ORDERDATE + 3 months*
AND S_SUPPKEY = L_SUPPKEY
AND S_NATIONKEY = N_NATIONKEY
AND L_ORDERKEY = O_ORDERKEY
group by *S_SUPPKEY, N_NAME, S_PHONE, O_ORDERDATE, L_SHIPDATE*
order by *S_SUPPKEY, O_ORDERDATE*
having *REVENUE between 10,000 AND 100,000*

Useful information is available at three granularities: (a) about individual attributes, (b) about each tuple, (c) about the entire relation.

For each individual attribute, it is useful to know:

- *Range of values*: If the range of the values of an attribute is small, each value of this attribute may be encoded by the offset from the lower bound of the range. For example, in Example 2.1, from the *having* clause, *REVENUE* is larger than 10,000 and smaller than 100,000. Therefore, the ASCII form of *REVENUE* requires no more than eight characters. Therefore, four bits can be used for each numeric character in the ASCII format of *REVENUE*.
- *Number of distinct values:* When an attribute (such as *N_NAME*) only has a small number of distinct values, non-adaptive dictionary compression (refer to table 2.1) may be applied.
- *Character distribution for strings:* Some string encoding methods (e.g. Huffman) can be applied if the character distribution of strings is available. For instance, in example 2.1, *S_PHONE* represents the phone number and a phone number can only hold characters corresponding to digits and '-'. Therefore, each character of *S_PHONE* can be encoded with four bits.

For each tuple, it is useful to know the following "cross-attribute" information:

- *Value Constraints:* In example 2.1, the *where* clause constrains *L_SHIPDATE <= O_ORDERDATE + 3* months; so *L_SHIPDATE – O_ORDERDATE* can only differ by less than 3 months. Therefore, *L_SHIPDATE* can be encoded with 2 bits for the month difference and 5 bits for the day.
- *Functional dependencies:* Functional dependencies between attributes indicate opportunities for normalization. The FD information is available in catalogs where *"the key of the table -> other attributes in this table"* is a trivial FD. In example 2.1, FD "*S_SUPPKEY -> N_NAME, S_PHONE"* indicates the opportunity for normalization compression method.

At the level of the entire relation, the order of the results suggests grouping or sorted-normalization compression methods. In example 2.1, the *order by* clause indicates that the results are sorted on *S_SUPPKEY*, which suggests the sorted-normalization method.

2.4. Example of performance improvement

To motivate the rest of the paper, we present a quantitative example of the effects of query result compression. Consider the compression of the result of the SQL query (approx. 10 MB) in example 2.1. Plan *A* (refer to appendix for details) is the combination of various methods discussed above to form a suitable "compression plan". It allows individual tuples and attribute values to be decompressed. Plan *B* extends Plan *A* by applying LZ77 to each column of the relation. While this gains additional compression, the entire relation needs to be decompressed to access any tuple or attribute value. We compare these plans with WinZip (using file level LZ77) which of course requires that the entire relation be decompressed. We present the compression ratio, compression and decompression throughput (uncompressed result size divided by compression or decompression time). The experiment is run on a machine with a Pentium-II 450 MHz processor and 256 MB RAM.

Compression Plans	WinZip	Plan *A*	Plan *B*
Compression ratio	4.25	5.57	10.68
Compression Throughput	1.25 MB/s	1.76 MB/s	1.39 MB/s
Decompression Throughput	4.60 MB/s	3.46 MB/s	2.93 MB/s

Table 2.2 Comparison on compression ratio, compression and decompression throughput.

Both plan *A* and *B* achieve more compression and higher compression throughput than WinZip because of the use of domain knowledge. Moreover, plan *A* allows the client to decompress tuples and attributes individually. However, the decompression throughput of plan *A* and *B* is lower than that of WinZip due to the overhead of combining multiple decompression methods.

We should view query result compression in the context of an end-to-end system. Assume that a PC client communicates with the database server over a 56.6 kbs modem. The "cost" of accessing the query result is some combination of the compression, decompression and transmission time. Consider that we simply add these times to compute the cost. Using no compression requires 1,408 seconds; using WinZip requires 341 seconds; using plan *A* requires 262 seconds; using Plan *B* requires 143 seconds, which is 42% of the end-to-end time of WinZip, 55% of that of plan *A*, and 10% of that of no compression.

3. A framework for query result compression

Section 2 shows that the combination of different compression methods applied on different columns of the query results has better performance than the naïve WinZip compression method. We now present a framework that captures the combination of individual compression methods as a composite compression strategy. Therefore, the framework we present contains three components:

(a) Compressed table (Section 3.1): This is the basic abstraction representing the relational data in compressed form. The uncompressed query result is also a special-case instance of a compressed table.

(b) *Compression operator (Section 3.2):* This is the basic abstraction representing a compression method. It operates on a (input) compressed table and defines a new (output) compressed table.

(c) *Compression plan (Section 3.3):* This is the abstraction representing the entire combination of compression methods.

3.1. Compressed tables

We first present some intuition and then give the formal definition of a compressed table.

Attributes and tuples may become inaccesable in a compressed table because a compression method may compress several attributes from several tuples to a single unit. For instance, figure 3.1 shows an uncompressed table on the left-hand side. The uncompressed table contains two attributes – "*Customer ID*" and "*Order Number*". Four tuples are in the table, where the first two tuples have the same "*Customer ID*" and so do the next two tuples. The grouping compression method is applied on *"Customer ID"* and the offset encoding is applied on *"Order Number"* with the offset from 1,000,000. The figure shows that the grouping method compresses the *"Customer ID"*s from the first two tuples in the uncompressed table to a single unit (block 1).

Customer	*Order*
C1	1000000
C1	1000090
C2	1000999
C2	1000102

Grouping	*Offset-encoding*
C1 (*Block 1*)	0 (Block 3)
	90(Block4)
C2 (Block 2)	999(Block5)
	102(Block6)

Figure 3.1 A table compressed by grouping and offset-encoding methods.

Therefore, we use the notion *compressed data blocks* (*blocks* in abbreviation) to represent the minimal accessable units in a compressed table. In a compressed table, a block contains a given value, which is compressed from a set of attributes in one or several tuples. For instance, block 1 in the compressed table shown in figure 3.1 contains a string *"C1"*, which is compressed from the "*Customer IDs*" of the first two tuples.

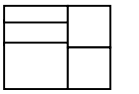
Now we define the compressed table based on the intuition.

Definition 1. A ***compressed table*** is a collection of compressed data blocks and a ***compression schema*** that represents meta information.

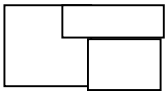
- Each compressed data block is a value with a certain data type that is compressed from a set of attributes in one or several tuples. Each attribute from a tuple must be compressed to one and only one compressed data block.
- The compression schema consists of the following meta information that enables the decompression:
  - For each compressed data block, what set of attributes from what tuples are compressed to that block.
  - For each compressed data block, what compression method is used to generate that block.
  - The original relational schema of the uncompressed table that the compressed table is compressed from.

An uncompressed table can be seen as a special compressed table where each block just contains one attribute in one tuple.

Although our definition of compressed table allows a compressed data block includes arbitrary set of attributes from arbitrary set of tuples, we only consider more regular data blocks in this paper. Here we assume that the blocks in the compressed table will have no indents down the column (as shown in figure 3.2). Further, we define a block *belongs* to a set of columns if the value contained in the block is compressed from attribute values from and only from those columns. We also assume that blocks belong to the same set of columns will be compressed by the same compression method because usually there are more redundancies down the column than across the column [RHS95].



A regular compressed table.



An irregular compressed table.

Figure 3.2

3.2. Compression operators

A compression operator specifies how a compression method is applied on one or several columns of an input compressed table to generate an output compressed table.

We define a *compression operator* as a mapping from a set of blocks in the input table and belonging to columns that the operator will compress (input blocks in abbreviation) to a single block in the output table (an output block in

abbreviation). For instance, a per-page LZ77 operator maps each page of blocks in the input table to a single block in the output table. Blocks in the input table belonging to columns not compressed by the operator are not changed by the application of the compression operator.

A compression operator consists of three components to specify the mapping:

a) ***The compressed columns.*** The columns the compression method is applied on.

b) ***Compression granularity*** defines what set of input blocks will be compressed to a single output block.

The observation is that different compression methods should be applied on different granularities to achieve the best performance. For instance, LZ77 should be applied on a large granularity (i.e. a page) to be effective.

To guarantee that each input block will be compressed to one and only one output block, we define the compression granularity of a compression operator as an ***equivalence relation*** on input blocks belonging to the compressed columns. Each equivalence class of input blocks will be mapped to a single output block. For instance, the compression operator with the grouping method in Figure 3.1 maps *"Customer IDs"* with the same value to a single output block. Hence, the granularity of the compression operator is an equivalence relation on *"Customer IDs"* such that two *"Customer IDs*" are equivalent if and only if they have the same value. We represent the equivalence relation as a first-order logic Boolean expressions *E(x, y)* where *x* and *y* are input blocks belonging to compressed columns and *E* may include physical or logical properties of *x* and *y* as predicates. For instance, the equivalence relation on *"Customer Ids"* can be represented as:

E(x, y) ≡ "Column_Name (x) = Column_Name(y) = {'Customer ID}' and Value(x) = Value(y)".

(Column_Name(x) returns the set of columns block x belongs to and Value(x) returns the value of x).

According to this compression granularity, the *"Customer ID"s* of the first two tuples in the uncompressed table in figure 3.1 will be mapped to a single block (block 1) in the output table.

c) ***The compression method*** and ***meta information*** used in compression.

The compression method has a compression routine, a decompression routine and the desired data types of input blocks and output blocks. The meta information is the information used in compression, such as the range of values of an attribute.

Therefore, given an input table, we can type-check whether a compression operator is applicable on it as follows:

1) The data type of input blocks should agree with the input data type of the compression method.

2) The compression granularity must view each input block as a single unit because otherwise some input blocks need to be decompressed before compression. For instance, in figure 3.1, after a per page LZ77 compression method is applied on the column of *"Customer ID"*, any compression method compressing an individual *"Customer ID"* can not be applied because individual *"Customer ID"s* become not visible in the input table.

If a compression operator type-checks with an input table, we call the operator a *valid* operator for the table.

Therefore, given an input compressed table and a valid compression operator, the application of the compression operator will generate the output compressed table as follows:

(a) First, blocks in the output table are generated as follows:

- Input blocks not belonging to the compressed columns remain unchanged.
- Input blocks belonging to the compressed columns are divided into equivalence classes by the equivalence relation (compression granularity) and each such equivalence class of blocks is mapped to one output block by the application of the compression routine.

(b) The compression schema of the output compressed table is deduced from the compression schema of the input table and the operator as follows:

- The relational schema of the output table is the same as that of the input table.
- The meta information about output blocks not belonging to the compressed columns remain unchanged.
- The value of each newly generated output block is compressed from all individual attributes compressed in the equivalent class of input blocks mapped to that output block.
- The compression method applied on the new output block is just the compression method of the compression operator.

3.3. Compression plans

Definition 2: A ***compression plan*** is defined as:

- A compressed table, or
- A valid compression operator applied to a compressed table (which may itself be the result of an operator).

We represent the compression plan containing *N* compression operators (*Operator 1* to *Operator N*) in a nested form as

"Operator N (Operator N-1 (… (Operator 1 (Input Table))….)"

For instance, the compressed table in figure 3.1 can be expressed as the output of the following compression plan:

Offset-encoding operator on attribute "Order Number" (Grouping operator on attribute "Customer ID" (The uncompressed table in figure 3.1)).

3.4. Plan execution

We now provide an operational semantics for the execution of a compression plan. Due to the space constraints, we focus on the compression phase of a compression plan. The decompression phase is the reverse of the compression phase. We first present the *serial execution* semantics, then present the equivalence of two executions of a compression plan, and finally discuss pipelining in executing a compression plan.

A serial execution executes operators one-by-one. Each operator execution applies the operator to a compressed input table and generates a compressed output table, which acts as the input for the next operator.

Any plan execution that produces the same result as a serial execution is considered equivalent and therefore correct. It is likely that several equivalent execution strategies may exist. Observe that the orders between compression operators in a plan may be exchangeable. For instance, in Figure 3.1, the order of grouping and offset-encoding operator can be exchanged and we still get the same result. To specify what operators can not be exchanged, we define a partial order between two compression operators if they compress the same column.

To reduce the storage of each intermediate table, the execution of many operators in the plan can be pipelined. Each pipelined operator processes a chunk of input blocks at a time. This chunk is called the *pipelining unit*. This is only one of several forms of parallelism during plan execution (we have not yet investigated other "bushy" parallel execution strategies).

4. Experimental results

This section demonstrates that the combination of compression methods using domain knowledge has superior performance than the naïve WinZip compression method. We first compare plans optimized for network transmission (where minimizing the data size is critical). We also run decompression code on a handheld device to explore realistic decompression costs. All the compression plans are manually constructed. We derive intuition from these plans to guide our later choice of a heuristic optimization algorithm in Section 5. The compression algorithms are implemented on top of the Cornell Predator ORDBMS. The server runs on a Pentium II 450 MHz processor with 256 MB RAM.

4.1. Compression plans optimized for network transmission vs. WinZip

We need to choose a set of queries that are realistic and have large result size (needing compression). We accomplish this by adapting queries from the TPC-D benchmark [TPC95]. We justify this choice because:

- Queries in the TPC-D benchmark are characteristic of a wide range of decision support applications.
- The data in the TPC-D benchmark contains data with various data types and various redundancies.

Most queries include grouping and aggregation, making their result sizes small. As we mention in the introduction, in the environment we consider, aggregations and analysis are performed locally at the clients rather than within the database server. Therefore, we adapt the queries by removing the grouping and aggregation.

Our goal is to minimize the compressed size of the query result. We compare the performance of four categories of plans.

1) WinZip (*W* in abbreviation) applied on the results.
2) WinZip applied per column (*PW*).
3) Compression plans that allow attribute level decompression (each attribute value can be decompressed individually) and use the domain knowledge of the query results. We call these plans *small-granularity* plans (*S*). They have similar forms:
   - First, they use the sorted-normalization or grouping method if such methods can be applied.
   - Then, appropriate per column compression methods with small granularity are applied on each column. More than one such operator may be applied on the same column.
4) The *small-granularity* compression plans plus per-column WinZip (*SPW*).

The database size is 100 MB. The average result size is 2.5 MB. Each tuple has 52 bytes and 5 columns on average.

4.1.1. Compression ratio. Figure 4.1 shows the compression ratios for the 17 adapted queries. Since the compression ratios for query 15 and 16 are much higher than other queries, they are shown separately. Table 4.1 shows the average compression ratios.

W	*PW*	*S*	*SPW*
3.43	4.41	3.10	6.51

Table 4.1 Average compression ratio of the four categories of compression plans

Figure 4.1 shows that the *SPW* compression plans always have the best compression ratio, on average 89% better than that of *W* and 110% better than that of the *S* plans. The reason is that *S* plans only exploit redundancies indicated by the domain knowledge, which is in general about the whole attribute values. However, WinZip can exploit redundancies between bytes of attribute values. Therefore, the combination of WinZip and small-granularity plans exploits both types of redundancies and achieves better compression.

PW is the second best and it achieves an average compression ratio 29% better than WinZip because of the use of the domain knowledge that there are more similarities down the column than across the column.

The *S* plans have a comparable compression ratio with *W* plans (14% worse on average).

To evaluate the grouping and sorted normalization methods, we delete grouping and sorted normalization

operators from the *S* and *SPW* plans (called *S'* and *SPW'*). We compare their compression ratios with the compression ratios of those plans using grouping and sorted-normalization methods.

Figure 4.2 shows the results. Sorted normalization is applicable in query 3, 10, and 15. Grouping is applicable in other queries. Table 4.2 shows the average compression ratios of these four categories of plans.

S	*S'*	*SPW*	*SPW'*
3.10	1.69	6.51	6.33

Table 4.2

Table 4.2 shows that the grouping and sorted-normalization methods are important for *S* plans because other compression methods with small compression granularities do not exploit the repetitions of values in consecutive tuples. However, the effect of the sorted-normalization and grouping methods becomes insignificant when per-column WinZip is applied because per-column WinZip also exploits the repetitions of values in consecutive tuples.

4.1.2. Overall performance. Table 4. 3 shows the average compression (decompression) throughput of the four categories of compression plans listed in the beginning of section 4.1.1. The *S* and *SPW* plans have lower throughput.

	Avg. Compression Throughput	Avg. Decompression Throughput
W and *PW*	1.25 MB/s	4.60 MB/s
S	1.18 MB/s	3.39 MB/s
SPW	0.87 MB/s	2.37 MB/s

Table 4.3

We measure the overall performance by computing the end-to-end time as the sum of the compression time, network transmission time, and decompression time. We compare *W*, *PW*, and *SPW* plans. *S* plans are not shown because they have lower compression ratios and higher compression/decompression cost than *W* plans. Figure 4.3 shows the average end-to-end time for these three sets of plans normalized to the end-to-end time of no compression. The x-axis is the network transfer rate, ranging from 20 KB/s to 100 KB/s in figure 4.3 (a), and from 100 KB/s to 0.9 MB/s in figure (b). The results when the network bandwidth is over 0.9 MB/s are not shown because in that case the compression cost and decompression cost become dominant and compression has no benefits. Table 4.4 shows the average normalized end-to-end time of these plans when network speed is 56.6 kb/s.

W	*PW*	*SPW*
29.8%	23.3%	16.2%

Table 4.4. Average normalized end-to-end time of the three sets of compression plans.

Figure 4.3 shows that compression does reduce the end-to-end time for a slow network. Moreover, per-column WinZip is always better than WinZip because they have the same compression and decompression costs while the former has better compression ratios. When the network speed is slow, *SPW* plans are the best. Due to the higher compression and decompression costs, the improvement on end-to-end time is less than the improvement on the compression ratio.

As the network transfer rate increases, the network transfer time becomes less important and the compression and decompression time becomes more critical. Per-column

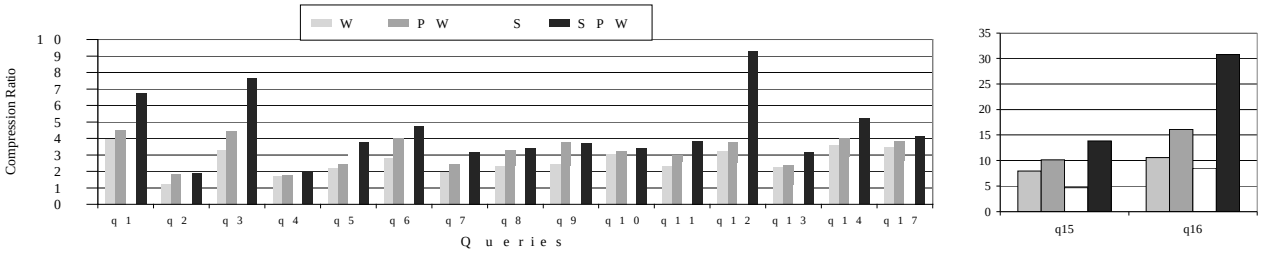


Figure 4.1 Comparison of compression ratios

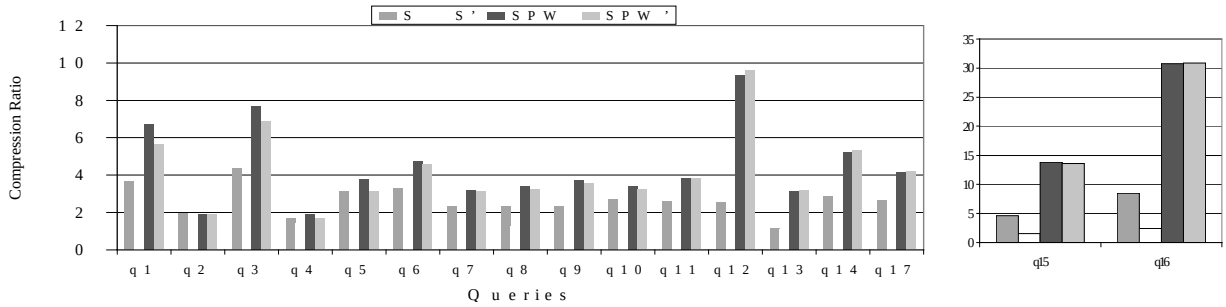


Figure 4.2 Effect of using grouping/sorted normalization methods

WinZip becomes more competitive when the network transfer rate exceeds 200 KB/s because it has a lower compression and decompression costs than *SPW* plans.
Per-column WinZip method has two extra advantages:
1) It is very simple and does not require collecting domain knowledge and choosing appropriate compression methods.
2) DBA may use per attribute compression methods such as offset-encoding to compress the data tables. Therefore, the compression ratio on query results will be lower. However, as table 4.1 shows, per-column WinZip can be applied to improve the compression ratio.

4.2. Decompression on handheld devices

We now consider handheld client devices. We run an experiment on a palm-size CASSIOPEIA E-10 device, running Windows CE on a MIPS 4000 processor with 4 MB RAM (2MB of persistent data storage and 2 MB of program storage). The query is the same as example 2.1 and the result size is 3.72 MB, making it too large to store without compression. We also assume that the client needs random access to tuples, and clearly it is not feasible to decompress the entire results in an initial step. Instead, we store the results in compressed form and only decompress the results being accessed (decompression *on demand*).

Windows CE uses a default page-level compression for all persistent storage. Therefore, the compression plans should meet two (possibly conflicting) requirements: (a) they should exploit redundancies that Windows CE's compression method does not exploit (b) they should allow tuple-level access to reduce the decompression cost of random access. Specifically, we compare the behavior of:

- The small-granularity compression plan *A*.
- Compression plan *C* is the same as plan A except that plan *C* does not include the sorted-normalization operator. Both plan *A* and *C* allow attribute-level decompression.
- Plan *D*: A plan using the default Windows CE compression.
- Plan *E*: LZ77 applied on each page of tuples.

Table 4.5 shows the storage usage (including both the persistent and program memory), and the time to randomly access 1000 tuples of the result. The average tuple size is 52 bytes.

	A	*C*	*D*	*E*
Storage	0.75 MB	0.95 MB	1.42 MB	1.43 MB
Accessing Time	2.85 second	2.77 second	2.48 second	23.1 second

Table 4.5 Performance on handheld devices.

Plan *E* is not competitive because it has no improvement in storage usage because the Windows CE's method may be the same as LZ77 and applying the same compression method twice can not achieve additional compression. The decompression time of plan *E* is also prohibitive because whenever a tuple is accessed, a whole page has to be decompressed.

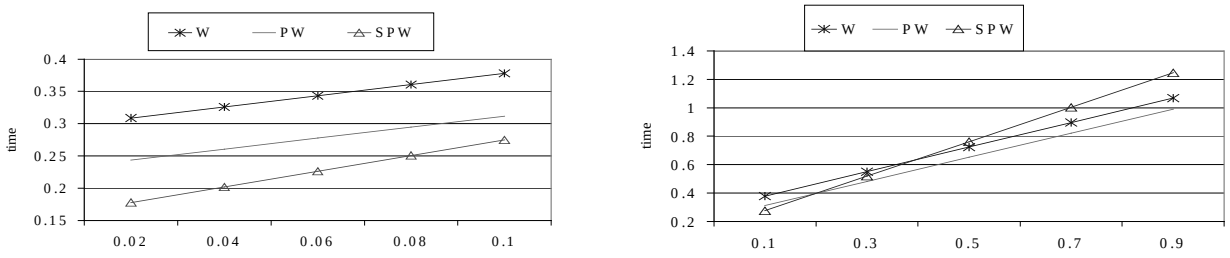
Plan *A* uses 53% the space of Windows CE's compression method and Plan *C* uses about 67%. This shows that the use of domain knowledge does improve the compression ratio. Further, the accessing time of plan *A* plus Windows CE's method is just 15% more than that of Windows CE's method because plan *A* allows tuple-level decompression. Plan *A* is also better than plan *C* because the former only uses 79% the space of the latter and has just 3% more accessing time.

This result shows that using plans exploiting knowledge on query results and allowing decompression in small granularities can improve the performance on handheld devices.

The size of the program using plan *A* is slightly bigger than the size of program using Windows CE's default method (less than one KB increase using Visual C++ 5.0 compiler) due to the implementation of multiple decompression methods. This is insignificant compared with the extra compression plan *A* achieves.

5. Optimization issues

Compression plan optimization is necessary because there are many valid combinations of compression methods and these combinations have different performance according to the specific characteristics of the query results, the network transfer rate, and the resource constraints of the clients. After query optimization, a cost-based compression plan optimization phase automatically generates a minimal



(a) For bandwidth <= 0.1 MB/s **(b) For bandwidth >= 0.1 MB/s**

Figure 4.3 End-to-end time of using compression.

cost or low cost compression plan based on the query plan, the domain knowledge of the query results, the network transfer rate, the client resource constraints, and the available compression methods. We first discuss cost estimation. Then we present an exhaustive but inefficient naïve algorithm. Finally, we present a polynomial heuristic algorithm that is based on the intuition derived from our experimental results.

5.1. Cost estimation

The cost of a plan equals the compression overhead minus the compression benefit. The compression overhead is:

*w1 * compression cost + w2 * decompression dost.*

The compression benefit is defined as:

*w3 * saving on network transfer + w4 * saving on client side storage.*

This formula can fit various goals of optimization such as minimizing the network transition cost (set *w1, w2, and w4 = 0*) and minimizing client side cost (set *w1* and *w3* = 0).

5.1.1. Compression and decompression cost. The compression cost of each operator includes the CPU cost and the I/O cost. The CPU cost depends on the CPU speed, the compression method, and the input result size. The I/O cost includes the cost for reading the input and writing the output, which can be saved if the plan is pipelined in a unit that can be held in memory. For a serially executed plan, the compression cost equals the sum of the cost for each operator.

The decompression cost depends on the compression plan and the client's access pattern of the result. If the client decompresses the query result immediately, as in section 4.1, the decompression cost is just the cost to decompress the whole result. It can be computed by adding up the cost for individual operators. If the client stores the result in compressed form and needs *N* random accesses to certain units (tuples, attributes, etc.), the decompression cost equals the cost to decompress blocks containing each access unit times *N*. If the client needs *N* sequential accesses instead, we assume that the client caches the decompressed units and the cost is reduced to the cost to decompress all blocks containing the attributes values being accessed.

5.1.2. Compression benefit. The compression benefit includes the saving on network transmission and client-side storage.

The saving on transmission = Transmission cost of the uncompressed result – that of the compressed result.

The transmission cost equals a function of the result size, which also depends on network properties. The saving on client-side storage depends on whether on-demand decompression is required. If not, the saving is zero. Otherwise, the saving is:

Uncompressed result size – compressed result size – uncompressed size of blocks accessed each time – extra program memory used for decompression.

The compressed result size is computed based on the compression methods, the uncompressed result size, and other information such as the average attribute size and the number of attributes. The details are specific to each method and are omitted for reasons of space.

5.2. A naïve algorithm

The naïve algorithm starts with an empty plan (the plan has no compression operator). The algorithm repeatedly enumerates new plans by adding new and valid operators (operators that do not appear in the existing plan and also type-check with the output of the existing plan) to existing plans, and stops when no new plans are generated. The algorithm evaluates the cost of each plan (as described in section 5.1) and returns the minimal cost plan. Clearly, this algorithm is exhaustive because it has considered all possible combinations of compression operators. This algorithm also halts if the number of possible methods, columns, and compression granularities is limited for two reasons. First, the total number of possible plans is finite because there are a finite number of valid operators (defined by combinations of possible methods, columns applied on and compression granularities). Second, the same operator is never included twice in any plan.

However, the naïve algorithm is not efficient because the algorithm needs to check all sequences of compression operators, which requires execution time exponential to the number of columns in query results.

5.3. A heuristic algorithm

In order to reduce the search space, we derive the following heuristics from the experiments in section 4:

1. First, apply small-granularity compression methods using the domain knowledge because both the domain knowledge is only valid for the uncompressed results and compression methods with small granularities can not be applied after compression methods with large granularities (refer to the type-checking in section 3.2). After such "semantic" compression, methods with large granularity and not using domain knowledge (such as LZ77) may further enhance the compression.
2. Use local per-column optimization because the compression overhead or the benefit of the whole plan can be seen as the sum of the overhead or benefit on individual columns.
3. For compression methods not using domain knowledge, per-column operators are preferred because usually there are more similarities down the column than across columns [RHS95].

4. If the client applies on-demand decompression and needs random access in a certain unit, the compression plan should allow decompression in that access unit to minimize the decompression cost.

5. For methods not using the domain knowledge, larger compression granularities are preferred because such methods work better for large chunks of data. The compression granularity should be also no larger than the pipelining unit to ensure that input blocks can be accessed within the pipelining unit.

The heuristic algorithm is as follows:

1. First consider methods with small granularity and using domain knowledge as follows:

(a) For each column, apply the naïve algorithm to enumerate all valid combinations of operators compressing this column using the domain knowledge.

(b) Choose the minimal cost plan for each column and combine these plans to a global plan, which contains all operators in individual plans. The partial orders of the combined plan include the union of partial orders of each individual plan and the partial orders between operators in different plans. This plan should also type check.

2. Consider other methods for each column as follows:

(a) Choose an appropriate method based on available information. For instance, if a column is not compressed and we have the character distribution table, arithmetic or Huffman encoding is the choice. Otherwise, methods like LZ77 will be favored.

(b) Choose an appropriate compression granularity by heuristics 4 and 5.

(c) Form a new plan by adding the new operator to the global plan generated by step 1. Accept the operator only if it can reduce the cost. If per-column LZ77 is applied, grouping/normalization operators will be deleted. The resulted plan is the final plan.

For instance, suppose we want to compress the table in figure 3.1 for two clients: client 1 is a handheld device that decompresses result on demand and needs random access to tuples; client 2 downloads the result through a slow network and only cares about the result size. The available compression methods are offset encoding, sorted-normalization, and LZ77. The plan is also required to be pipelined per page.

Assume after step 1 a), grouping is the minimal plan for "*Customer ID*" and Offset encoding is the minimal plan for "*Order Number*". Step 1 b) generates a global plan as:

Plan 1: offset encoding on "Order Number"
(Grouping on "Customer ID" (The uncompressed table))

Then step 2 a) chooses LZ77 for each column. For client 1, the random access unit is per tuple, so LZ77 should have small granularity. For client 2, the compression granularity should be per page by heuristics 5.

For client 1, LZ77 operators are rejected because as shown in [RHS95], the granularity is so small that LZ77 does not work well. Therefore, the final plan is *plan 1*. For client 2, per-column LZ77 operators are accepted since per page LZ77 can improve the compression for a large chunk of data. The grouping operator is also deleted. The final plan is:

Per column LZ77 with per page compression granularity
(Offset encoding on "Order Number"(
The uncompressed table))

Assume the number of columns is *M*, and for each column, there are *X* valid combinations of operators using information on query results, the cost for step 1 is O(*M*X*). The cost for step 2 is O (*M*). Therefore, this algorithm has the cost O (*M*X*).

6. Conclusion and future work

This paper addresses the issue of compressing query results and makes the following contributions:

- We present a framework to model the combination of compression methods. The framework models a compression plan as a sequence of compression operators.
- We describe the use of domain knowledge about the query to improve the effect of compression on query results.
- We present experiments on queries adapted from the TPC-D benchmark. The experiments show that on average, our handcrafted compression plans achieve an 89% improvement in compression ratio over a standard compression algorithm (WinZip).
- We also present an experiment with a handheld device acting as the client. This demonstrates the importance of using domain knowledge and allowing decompression in small granularities in such resource-constrained environments.
- We present a naïve and a heuristic optimization algorithm that choose low-cost compression plans for a variety of different requirements.

We are currently exploring the joint optimization problem of query plans and compression plans. Currently, the compression optimization is based on the query plan returned by the query optimization. However, the overall cost of a combination of a query plan and a compression plan is different from the cost of the query plan. For instance, a more expensive query plan may sort the result in an order so that the sorted-normalization method can be applied , which may lead to a plan with lower overall cost.

Another interesting topic for future work is to apply the overall methodology in this paper to compress databases as they are stored. The algebraic framework of result compression and the idea of "algebraic optimization" are also applicable.

Acknowledgements

We thank Philippe Bonnet and Flip Korn for their detailed feedback on this paper and Ryan Patrick Kennedy for his implementation of a GUI for the application running on Windows CE devices.

References

[ALM96] G. Antoshenkovc, D. Lomet, and J. Murray. Order preserving compression. In Proc. IEEE Conf. on Data Engineering, pages 655-663, New Orleans, LA, USA, 1996.

[BAS85] M. A. Bassiouni. Data Compression in Scientific and Statistical Databases. IEEE Trans. on Software Eng.: 1047-1058 October 1985.

[BBC99] Philippe Bonnet, Kyle Buza, Zhiyuan Chen, etc. The Cornell Jaguar System: Adding Mobility to PREDATOR. In demo sessions of the Proceedings of ACM SIGMOD International Conference on Management of Data, page 580-581, Philadelphia, June, 1999.

[COM79] D. Comer. The ubiquitous B-tree. ACM Computing Surveys, 11(2):121-137, 1979.

[COR85] G. Cormack. Data Compression in a database system. Commnications of the ACM: 1336-1342, Dec. 1985.

[CSL98] Boris Y. L. Chan, Antonio Si, Hong Va Leong: Cache Management for Mobile Databases: Design and Evaluation. ICDE 1998: 54-63.

[DFJ96] Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, Divesh Srivastava, Michael Tan: Semantic Data Caching and Replacement. VLDB 1996: 330-341.

[DK98] Margaret H. Dunham, Vijay Kumar: Location Dependent Data and its Management in Mobile Databases. Ninth International Workshop on Database and Expert Systems Applications: 414-419, 1998.

[DL98] Ravi A. Dirckze, Le Gruenwald: A Toggle Transaction Management Technique for Mobile Multidatabases. Proceedings of the 1998 ACM CIKM International Conference on Information and Knowledge Management, 371-377, Bethesda, Maryland, USA, November 3-7, 1998.

[EOS81] S. J. Eggers, F. Olken and A. Shoshani. A Comporession Technique for Large Statistical Data Bases. In VLDB, pages 424-434, 1981.

[GOY83] P. Goyal, Coding methods for text string search on compressed databases, Information System, 8,3: 231-233, (1983).

[GRA93] G. Graefe. Options in Physical Databases. SIGMOD Record 22(3): 76-83, September 1993.

[GRS98] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compression relations and indexes. In Proc. IEEE Conf. on Data Engineering: 370-379, Orlando, FL, USA, 1998.

[GS91] G. Graefe and L. Shapiro. Data compression and database performance. In Proc. ACM/IEEE-CS Symp. On Applied Computing: 22-27, Kansas City, MO, April 1991.

[Huf52] D. Huffman. A method for the construction of minimum-redundanc codes. Proc. IRE, 40(9): 1098-1101, Septementer 1952.

[HS92] Peter J. Haas, Arun N. Swami: Sequential Sampling Procedures for Query Size Estimation. SIGMOD Conference 1992: 341-350

[HSW94] Yixiu Huang, A. Prasad Sistla, Ouri Wolfson: Data Replication for Mobile Computers. SIGMOD Conference 1994: 13-24.

[IW94] B. Iyer and D. Wilhite. Data compression support in databases. In Proc. of the Conf. on Very Large Databases, pages 695-704, Santiago, Chile, Sept. 1994.

[JKM98] H. V. Jagadish, Nick Koudas, S. Muthukrishnan, Viswanath Poosala, Kenneth C. Sevcik, Torsten Suel: Optimal Histograms with Quality Guarantees. VLDB 1998: 275-286.

[LB81] L. Lynch and E. Borwinrigg. Application of Data Compression to a Large Bibliography Data Base. VLDB 1981, 435.

[LZ76] A . Lempel and J. Ziv. On the complexity of finite sequences. IEEE Transactions on Information Theory, 22(1):75-81, 1976.

[LZ77] A . Lempel and J. Ziv. A universal algorithm for sequential data compression. IEEE Transactions on Information Theory, 31(3):337-343, 1977.

[MP99] Tobias Mayr and Praveen Seshadri. Client-site Query Extensions. In the Proceedings of ACM SIMGOD International Conference on Management of Data: 347-358, 1999.

[NR95] W. K. Ng, C.V. Ravishankar. Relational Database Compression Using Augmented Vector Quantization. ICDE 1995: 540-549.

[PIHS96] Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas, Eugene J. Shekita: Improved Histograms for Selectivity Estimation of Range Predicates. SIGMOD Conf. 1996: 294-305

[RH93] M. Roth and S. Van Horn. Database compression. ACM SIGMOD Record, 22(3): 31-39, Sept., 1993.

[RHS95] Gautam Ray, Jayant R. Harista, S, Seshadri. Database Compression: A Performance Enhancement Tool. Advances in Data Management '95, Proceedings of the 7th International Conference on Management of Data (COMAD), 1995, Pune, India.

[SAL98] David Salomon. Data compression, the complete reference. Springer-Verlag New York, Inc, 1998.

[SEV83] D. Severance. A practitioner's guide to database compression. Information Systems 8(1): 51-62, 1983.

[SYB98] Sybase IQ white Paper. http://www.sybase.com/products/dataware/iqwpaper.html.

[TPC95] Transaction Processing Performance Council TPC. TPC benchmark D (decision support). Standard Specification 1.0, Transaction Processing Performance Council (TPC), May 1995. http://www.tpc.org.

[WAG73] R. Wagner. Indexing designing considerations. IBM Systems Journal, 12(4):351-367, 1973.

[WEL84] T.A. Welch. A Techinique for High Performance Data Compression. IEEE Computer 17(6): 8-19, 1984.

[WNC87] I. Witten, R. Neal and J. Cleary. Arithmetic coding for data compression. Communications of the ACM, 30(6):520-540, June 1987.

Appendix

Compression plan A in example 2.1.

Method	Columns
Sorted-normalization	*S_SUPPKEY, N_NAME, S_PHONE*
Offset encoding	*S_SUPPKEY*
Non adaptive dictionary	*N_NAME*
Encoding each digit with four bits.	*S_PHONE*
Using one byte to encode difference with *O_SHIPDATE*	*L_SHIPDATE*
Encode float numbers with ASCII format.	*REVENUE*