A Compression Framework for Query Results

Zhiyuan Chen and Praveen Seshadri

Cornell University

zhychen, praveen@cs.cornell.edu, contact: (607)255-1045, fax:(607)255-4428

Abstract

Decision-support applications in emerging environments require that entire SQL query results be shipped to clients for further analysis and presentation. These clients may use low bandwidth connections (like modems) or have severe memory restrictions (like palmtops). Consequently, there is a need to compress the results of a query for efficient transfer and client-side storage. This paper explores a variety of techniques that address this issue. We model the problem as the choice of an appropriate compression plan and present a framework to model acceptable compression plans. The factors that influence this choice include schema information and statistics on stored tables. Importantly, we demonstrate that the query itself and its evaluation plan can provide semantic information that can be used to compress the result. We demonstrate that these techniques can result in 75% greater compression than standard compression tools like WinZip on queries adapted from the TPC-D benchmark. We identify two topics for future research: the choice of an optimal compression plan, and the integration of query result compression into the regular query evaluation plan.

1 Introduction

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, SNG93, GRS98, GS91], this activity is typically restricted to the internals of decision-support systems. In this paper, we apply compression to the results of SQL queries, in order to derive efficiencies when these results are delivered to clients. Since query results are highly structured and contain different data types, an efficient compression strategy typically involves a combination of multiple primitive compression techniques. We develop a framework to express the concept of a *compression plan,* which is a sequence of primitive *compression operators*.

The compression plan uses information derived from an analysis of the query and the particular query plan used to evaluate it. It also uses schema information as well as statistical information on stored tables. This semantic information enables much higher compression ratios than are achieved using traditional compression algorithms (e.g. using WINZIP). We implement appropriate compression plans for queries adapted from the TPC-D benchmark and demonstrate that the compression ratios achieved are 75% greater than with standard compression algorithms. We also identify two topics for future research that build upon the basic framework presented in this paper. One topic deals with the generation of efficient compression plans (analogous to query optimization). While we do not present a complete algorithm, we present guidelines that could be used to address this issue. The other topic deals with the integration of the compression plan with the query evaluation plan, for reasons of increased efficiency.

1.1 Motivation

While most database systems research examines the internals of database query processing, this paper is motivated by the use of queries within applications. 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 one class of applications, the query results need to be moved to the clients for ready visualization (e.g. OLAP). The clients may need to download the query results across an expensive and/or a slow connection. In another category of applications, the results need to be moved to clients that normally operate in disconnected mode. The best examples of such mobile computing are palmtop computers that have severe memory constraints (e.g. the PalmPilot III has 2MB of RAM for applications and their data). For reasons of efficient transfer and client-side memory conservation, the query results need to be compressed. In this paper, we examine techniques to accomplish query result compression.

There is a tremendous volume of existing research in the data compression community. Most of this work has focused on compression algorithms for specific data types (like text and multimedia types). Three issues make it non-trivial to apply compression in database systems:

1. The data is well structured and we usually have some a-priori knowledge about the data values. For instance, for a schema (*Name, Age, Sex*), we may know that the *Name* is solely composed of letters from the alphabet, *Age* should be from 0 to 125, and *Sex* can only be male or female.

2. 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. For instance, in the above example, the ideal case may use Huffman encoding [Huf52] or LZW[LZ77] to compress "*Name*", and use a single byte to encode Age and Sex (seven bits for Age, one bit for Sex).

3. Not only can multiple compression methods be applied to the data; they may be applied at different granularities. For example, [GRS98] partitions the data into pages, and the data on each page are compressed individually. Further, it is possible to apply different compression techniques one after the other to the same data partitions to achieve better compression ratio. For instance, if the "*name*" field in the above example has a lot of leading spaces, we may apply null suppression before LZW. This strategy usually achieves a higher compression ratio than using LZW alone because we explore the redundancy of leading spaces more efficiently by simply deleting them.

A compression method is effective when it finds redundancies in data. The more semantic information a compression method uses, more likely it is to find redundancies that other operators ignore. For example, an attribute known to be numeric need not be treated merely as a string of bytes. Compression methods specialized for numerical values may recognize some redundancies string oriented operators will ignore. For instance, suppose attribute A has continuous integer value from 1 to 10000. Differential compression can discover that the difference between adjacent values is 1 for all tuples. LZW can not find this redundancy because it does not know attribute A contains a sequence of values that have small difference. Our paper identifies and is guided by the following principle: **"more semantic information leads to more compression**".

1.2 Summary of Contributions

The field of data compression is well studied; consequently, we utilize well-known techniques rather than invent new compression algorithms. Our research makes the following contributions:

1. We analyze SQL queries and demonstrate how the semantics of the queries and the underlying tables can be used to identify compression opportunities.

2. Query results are typically unnormalized, since the queries often involve foreign-key joins. We present normalization as a compression technique for query results. We use the physical properties of query plan (primarily the sort ordering) to apply efficient normalization algorithms. This

is a novel use of normalization, which has typically been studied in the past in the context of eliminating update anomalies and minimizing disk storage.

3. We present a framework to understand the choice of compression methods. We define primitive compression operators, and a composite compression plan as a sequence of compression operators.

4. We demonstrate through implementation that well-chosen combinations of compression methods can result in significantly greater (around 75%) compression ratios than standard compression methods as LZW (used in gzip, WinZip). We apply compression plans to modified versions of all the queries in the TPC-D benchmark and report on the resulting compression ratios.

5. The choice of an appropriate compression plan requires "compression optimization", analogous to query optimization. We present some initial ideas towards compression optimization, and motivate it as a topic for future research.

6. Since compression plans are applied to query results, which themselves are generated by query plans, a natural idea is to merge the two to achieve greater efficiency. We present initial ideas in this direction, and motivate it as a topic for future research.

The combination of these contributions provides a significant initial step towards using data compression for query results. Figure 1.1 below shows various stages at which data compression plays a role within a DBMS. Our work is in the "compression to client" module. Note that our work is complementary to the body of existing work on data compression for stored data managed by the database system; that focus is on data being brought into the database system.

.

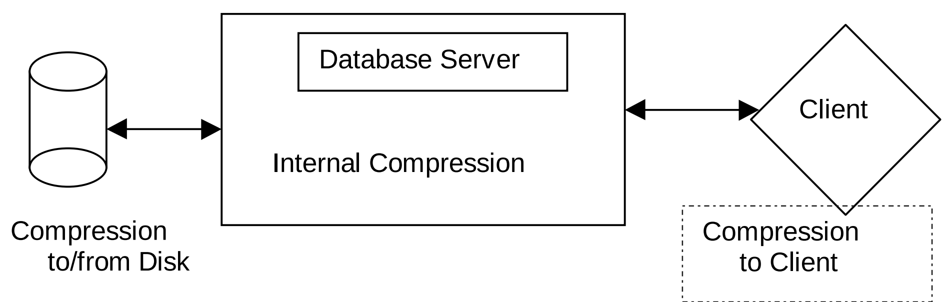


Figure 1.1

1.3 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 LZW [WEL84], LZ77 [LZ77, LZ76], etc. A comprehensive survey of compression methods is given 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 [GHS95]. Considerable research has dealt with compression of scientific and statistical databases [BAS85, EOS81]. In commercial database products, SYBASE IQ [SYB98] uses compression technique similar to gzip. 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, SNG93, GRS98]. [GS91] discusses the benefits of keeping data compressed as long as possible in query processing. The performance of TPC-D benchmark on compressed database is presented in [WKHM98]. Our research is complementary to the above work because we focus on compression of query results leaving the DBMS. To the best of our knowledge, there is no research that directly addresses this topic.

2 Compression Methods

2.1 Data compression methods

We intend to leverage well-known data compression algorithms. Here we present a non-exhaustive list of the standard methods that are considered in this paper. Compression methods can be divided into two categories, adaptive methods and non-adaptive methods. Adaptive methods assume no prior knowledge of the data and collect information from the context. Non-adaptive methods do not collect information on the fly, instead, they assume the information is collected in advance. Traditional methods such as LZW, Huffman encoding and arithmetic encoding all have adaptive and non-adaptive variants. For each method, we provide a brief description, the conditions under which it is likely to be effective and the kind of data on which it is best applied.

Name	Description	Condition	Data type
Differential encoding	Compute the difference of adjacent values (same column or same tuple)	The difference between adjacent values << actual values.	Numerical
Offset encoding	Encoding the offset to a base value	The offset << actual value.	Numerical
Null suppression	Omit leading zero bytes for numerical values and leading or ending spaces for strings.	Data contains a lot of null bytes (zero byte for numerical value, leading space for string)	Numerical, string
Non adaptive dictionary encoding.	Use a fixed dictionary to encode data.	#of distinct values / total occurrence << 1 and dictionary entry size << value size	String, numerical
LZW	Adaptive dictionary encoding	Data contains a lot of repetitions of patterns.	String, numerical
Huffman	Assign fewer bits to represent more frequent characters	Character distribution is skewed.	String, numerical
Arithmetic	Represent a string by interval according to probabilities of each character.	Character distribution is skewed.	String, numerical

We now present two database-specific compression methods -- normalization and grouping.

2.2 Normalization as a compression method

In database design, we normalize the tables to eliminate redundancy and anomalies. However, when queries involve foreign key joins, the result becomes unnormalized. For instance, assume we have tables R1 (A,B) and R2 (A,C). A is a primary key in R1 and a foreign key in R2. Therefore, we have a functional dependency A->B. Assume we run the query

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

Then we can normalize the result to remove redundancies. Suppose there are 1000 distinct A values and 100,000 tuples in result. Instead of storing A and B values 1000 times (because actually they only have 1000 distinct values), they are stored 100,000 times! We can reduce such redundancy by normalizing the result into two relations, AB and AC such that B values are only stored 1000 times.

However, the simple normalization algorithm has two problems (a) values of attribute A still have redundancy, (A values are stored 101,000 times) (b) we need a join to regenerate the result. However, when the data is already sorted on A (the left-hand side 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). There are 1000 such partitions. A and B are stored just once for each partition, along with each of the matching C values. This physical representation of the normalized tables has little redundancy (and hence occupies fewer bytes).

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

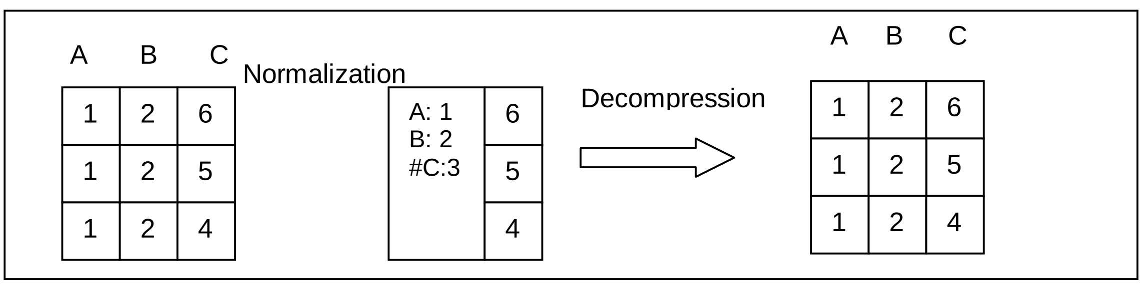


Figure 2.1

Suppose the size of attributes appearing in FD (functional dependency) is s and the total size of each tuple is t. The number of distinct values of attributes on the left hand side of the FD is m and the table cardinality is n. The compression ratio of normalization is:

$$(t * n)/(s * m + (t-s)*n).$$

Normalization uses the semantic information from the query (functional dependency) to find the duplicates.

2.3 Grouping as a compression method

Even if there is no functional dependency, we may utilize a similar compression technique when the data is sorted. We can group together duplicate attribute values and merely record the value once along with the number of occurrences. This is essentially run-length-encoding compression at the attribute level.

Since query results are often sorted, the grouping method can be applied on the sorted attributes without significant cost. Like normalization, grouping is more efficient than traditional compression methods in removing duplicate attribute values, since the sort order provides semantic information.

3 Opportunities for semantic compression

Clearly, it is possible to apply a general-purpose compression technique like LZW compression (instantiated in WINZIP, gzip, etc.) to the result of any query. Such an approach uses no knowledge of the semantics of the query or of the underlying database. Further, it does not utilize statistical information available from the database catalogs or information from the query plan describing the physical properties of the query result. In this section, we will look at the possibilities that exist if this information is used. We first look at the information that would be useful to have, and then at how that information might be obtained. The sources of information are the query itself, the underlying relations, and the query evaluation plan.

3.1 Useful information for compression

Our goal is to compress the relation produced as the result of a query. We can utilize semantic information at three granularities: (a) about individual attributes, (b) about each tuple, (c) about the entire relation. For each individual attribute, it would be useful to know the following:

- *Range of attribute values*: If the range is small, we may encode each value by its offset from the lower bound of value range. We may further use null suppression to encode the offset.
- *Number of distinct values of an attribute:* If this number is small, we can use non-adaptive dictionary compression.
- *Difference between consecutive values:* If this difference is much smaller than the actual values, we can represent each value by its difference from the preceding value.
- *Character distribution for strings:* This information is useful for string encoding methods (e.g. Huffman). For instance, if there are only 26 characters in English name, we need no more than 5 bits to represent each character (as against the 8 bits typically used).

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

- *Value Constraints:* If there is a constraint of the form $B - 5 < A < B + 5$ on attributes A and B. It might be desirable to represent A by the value A-B+5 (since this value always lies between 1 and 9).
- *Functional dependencies:* Functional dependencies between attributes serve to indicate opportunities for normalization (which are also opportunities for compression). Moreover, since the values of attributes on left hand side of FD decide the values of attributes on the right, the number of distinct values of all attributes in the FD equals the number of distinct values of left hand side attributes. Thus we can use dictionary compression on all the attributes in FD rather than compress each of them separately.

At the level of the entire relation, it is useful to know the order in which tuple are produced. Depending on the order, certain grouping compressions are naturally suggested. This is "physical" information, rather than logical information about the relation.

3.2 Information from the database catalogs

The database catalogs maintain information about each attribute, including domain and key constraints. Domain constraints define the basic type domain of an attribute value. Key constraints provide the following information.

- The key has the same number of distinct values as the whole table.
- There exists a trivial functional dependency of the form key -> other attributes in this table.

Other constraints, especially the domain constraints, give us information on the possible values of an attribute. From this information, we can sometimes infer the number of distinct values. For instance, we may know that *L_DISCOUNT* is constrained to lie within the range of 0 - .99. For strings, domain constraints may provide information of the character distribution.

Because query optimization needs to estimate result size, statistical information is often kept in the catalogs, which can also be useful for compression. For instance, the number of distinct values and range are often stored as statistic information. Further, we may require that the DBMS collect extra information such as character distribution for compression purpose. However, this information should be used cautiously by the compression algorithms, since it is not guaranteed to be accurate. On the other hand, constraints are guaranteed to be accurate, since they are based on semantic information about the data.

3.3 Information from SQL query

An SQL query has the following general form:

```
SELECT [DISTINCT] target-list
FROM relation-list
[WHERE predicate]
[GROUP BY subset-of-target-list
 [HAVING predicate]]
[ORDER BY subset-of-target-list]
```

We adapt query 3 and query 15 from the TPC-D benchmark[TPC95] to demonstrate the possible use of semantic information for compression. Example 3.1 selects all orders with a certain SHIPPRIORITY and whose ORDERDATE is before some time and the SHIPDATE is no later than 3 month after the order date. Example 3.2 selects the daily revenue (L_EXTENDEDPRICE*(1- L_DISCOUNT)) and supplier information for each supplier.

Example 3.1

> ***select*** *L_ORDERKEY, O_CUSTKEY, O_ORDERDATE, O_SHIPPRIORITY, L_LINENUMBER, L_EXTENDEDPRICE, L_DISCOUNT, L_SHIPDATE*
>
> ***from*** *ORDER, LINEITEM*
> ***where***
> *C_CUSTKEY = O_CUSTKEY AND L_ORDERKEY = O_ORDERKEY*
> *AND O_ORDERDATE < '1995-07-01' AND L_SHIPDATE > O_ORDERDATE*
> *AND L_SHIPDATE <= O_ORDERDATE + 3 month AND O_SHIPPRIORITY = 0*
> ***order by*** *L_ORDERKEY, O_ORDERDATE*

Example 3.2

> ***select*** *S_SUPPKEY, S_NAME, N_NAME, R_NAME, S_ADDRESS, S_PHONE, L_SHIPDATE, SUM (L_EXTENDEDPRICE*(1- L_DISCOUNT)) AS REVENUE*

***from** LINEITEM, SUPPLIER, NATION, REGION*
where

L_SHIPDATE >= '1998-1-01' AND L_SHIPDATE < '1998-7-01'
AND S_SUPPKEY = L_SUPPKEY AND S_NATIONKEY = N_NATIONKEY
AND N_REGIONKEY = R_REGIONKEY

***group by** S_SUPPKEY, L_SHIPDATE*
***order by** S_SUPPKEY, L_SHIPDATE*
***having** REVENUE > 100,000*

- *SELECT clause:* This specifies the attributes in the result. Since the attributes are derived from attributes of stored relations, some of their properties may be inferred from the corresponding attribute properties in the catalogs. Indeed, a standard query optimizer would maintain these properties for the result of any query.

In this example, the following information can determined:
1) The data types of the different attributes are determined. In Example 3.1, all attributes are of numerical type.
2) Range information: We know from statistics in the catalog that *L_LINENUMBER* only has the range from 0 to 20. *O_CUSTKEY* has range from 0 to 6000. *O_ORDERDATE* is later than 1/1/1995. *L_DISCOUNT* only has 2 digits of precision and all values fall in range from 0 to .99. In Example 3.2, there are many string attributes, (*S_NAME, N_NAME, R_NAME, S_ADRESS, S_PHONE*. Note that an attribute like *S_PHONE* can only hold characters corresponding to digits and '-'. Further, S_*NAME* only has characters corresponding to letters and lower and upper case are not distinguished.
3) The number of distinct values: *N_NAME* has only about 100 distinct values and *R_NAME* has about 5 values. The other attributes have many distinct values.

- *FROM clause:* This specifies the tables accessed. The catalog entries for these tables specify the functional dependencies defined. In this case, we have the following FD derived from the primary keys:

  - *L_ORDERKEY -> O_CUSTKEY, O_ORDERDATE, O_SHIPPRIORITY*
  - *S_SUPPKEY -> S_NAME, N_NAME, R_NAME,S_ADDRESS, S_PHONE*
  - *N_NAME -> R_NAME*

- *WHERE clause:* The WHERE clause constrains the values of the output tuple. We consider the following categories of WHERE clause predicates:
  - *O_SHIPPRIORITY = 0*: This is an important special-case where we know that the range of attribute values is of size 1. The constant value need only be stored once.
  - *O_ORDERDATE < '1995-07-01'*: Based on the schema constraints, we know that *O_ORDERDATE* is from *1995-01-01* to *1995-07-01*. Therefore, we only need 1 byte to encode the 6 possible month values and 31 possible day values.
  - *L_ORDERKEY = O_ORDERKEY*: Every such equi-join predicate is an extreme form of a value constraint (here, the two attributes are constrained to have exactly the same values).
  - *L_SHIPDATE > O_ORDERDATE* and *L_SHIPDATE <= O_ORDERDATE + 3 month*: The presence of non-equality join predicates can be used to impose cross-attribute range constraints. In this case, we know *L_SHIPDATE – O_ORDERDATE* can only differ by less than 3 month. Therefore, we can use 2 bits for the month difference and 5 bits for the day difference. In general, if we know A > B and we know A's range is [1,100] and B's range is [1,1000], we can infer B's range is [1,100].

- *GROUP BY clause:* GROUPBY tells us the number of distinct values of non-grouping attributes. All non-grouping attributes listed in select clause and the last attribute in the GROUPBY clause have exactly one value per group. Therefore, in Example 3.2, we do not need to consider any method that looks for duplicate values of *L_SHIPDATE*.

- *HAVING clause:* The HAVING clause defines the condition that aggregation attributes in each group must satisfy. Therefore, we can get similar information as from the WHERE clause. For example, in Example 3.2, REVENUE is larger than 10,000.

- *ORDER BY clause:* The ORDER BY clause defines the physical order of result, thereby providing information for the normalization and grouping compression methods. Further, we may use differential or offset encoding on the sorted attributes, since we expect the difference between adjacent tuples to be small.

3.4 Information from query plan

The query plan also gives us physical information about the final result so that we may reduce the cost of compression. For instance, if relation R1 joins with R2 on attribute A using a sort-merge join, we know the result is sorted on A.. In the case of a tuple-at-a-time indexed nested loop join, we expect that result tuples with the same values of the join attribute of the outer relation are grouped together.

3.5 Demonstration of semantic compression

We now "demonstrate" how this wide variety of semantic information may be used to compress the two example queries. More detailed experiments are presented later. In example 3.1, one possible sequence of compression steps is as follows:

- The normalization method is considered first. Based on the FD "*L_ORDERKEY -> O_CUSTKEY, O_ORDERDATE, O_SHIPPRIORITY"* and the information that the result is sorted on *L_ORDERKEY*, we use normalization on these attributes.
- We then choose an appropriate compression method for each attribute.
    - Since we know *L_ORDERKEY* is sorted, we expect the difference between adjacent values in the result to be small. Therefore, we choose the differential compression method first. Since we are not sure how much the difference actually may be, we use null suppression next to compress zero bytes
    - For *O_ORDERDATE,* we know it has the same year value, 6 possible month values and 31 possible day values. Thus we use 2 bytes to store the default year once. Then for each *O_ORDERDATE* value, only one byte is used (3 bits is used for the month and 5 bits for the day).
    - For C_CUSTKEY, we know the range is 0 to 6000. We choose null suppression on it so that at most two bytes are needed.
    - *O_SHIPPRIORITY* has constant value and we only need to store it once.
    - Since *L_LINENUMBER* has range 0 to 20*,* null suppression is used because only one byte is needed.
    - We only need to encode two digits for *L_DISCOUNT* because it falls in the range of 0 to .99. Further, each digit falls in range from 0 to 9, so only 4 bits are necessary per digit. Therefore we can encode the *L_DISCOUNT* with one byte.
    - Similarly, *L_EXPENDEDPRICE* can be encoded using 4 bits for each character in the ASCII format (without leading zeros).
    - *L_SHIPDATE* satisfies the constraint that the gap between *L_SHIPDATE* and *O_ORDERDATE* is less than three months. Therefore, only one byte is needed (2 bits for difference of month and 5 bits for the day value of *L_SHIPDATE*).
- Finally, we can compress each column in a page separately using standard LZW method.

For example 3.2: one possible sequence of compression steps is as follows:

- Normalization using FD "*S_SUPPKEY -> S_NAME, N_NAME, R_NAME,S_ADDRESS, S_PHONE*" is applied first.
- For each attribute:
    - Since all the string fields contain some leading and ending spaces, null suppression is used first for each.
    - *S_SUPPKEY* is sorted and has the range of 1 to 400, so differential encoding is appropriate because in most cases only one byte is needed.
    - Based on that characters in *S_NAME* can only be letters, a simple dictionary encoding that encodes each character using 5 bits is applied next.
    - Only four bits are used for each character in *S_PHONE* since it must be digit or '-'.
    - S_ADDRESS only contains letters, digits, some symbols like '-', '#', '.', ','. Therefore, we can use a simple dictionary compression that compresses each character with six bits.
    - Since *SHIPDATE* always has the same year value, we can encode it with only month and day value that cost us two bytes each.
    - We also expect the ASCII form of *REVENUE* to have length no more than 16. Then we may use four bits for each character in ASCII format.
    - For *N_NAME* and *R_NAME*, we can infer the number of distinct values is around 100 based on the information that *N_NAME* has 100 distinct values and there is a FD *N_NAME -> R_NAME*. Thus, dictionary compression on these two attributes is applied.
- As with Example 3.1, we can further compress each field using LZW to remove other redundancy.

For each example query, we present the compression ratio (original table size divided by compressed table size) of fivecompression strategies:

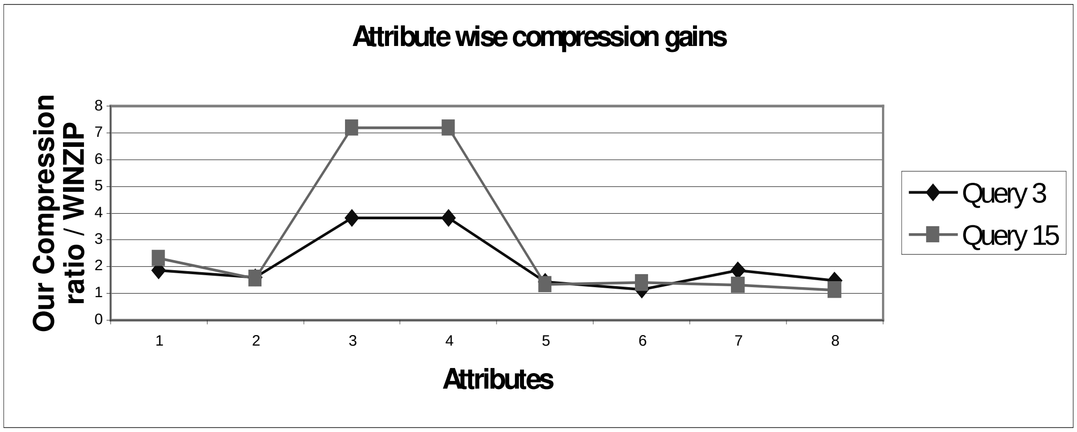
- Applying WINZIP on the entire result.
- Applying WINZIP on each column separately, this can be modeled as WINZIP using schema information of attributes.
- Our proposed strategy, without LZW applied to the result.
- Our proposed strategy, with file level LZW applied to the result.
- Our proposed strategy, with column-level LZW applied to the result.

The table below shows that WINZIP applied to the whole result file (what WINZIP would do) has the worst compression ratio. WINZIP at the column-level is better, which implies that the schema information helps the compression. It is interesting to note that our strategy performs comparably with WINZIP, even though it does not use the LZW compression algorithm. Our strategy with LZW applied to its result is significantly better than WINZIP on column level. With our strategy and LZW on each individual column, we perform almost 75% better than naïve file-level LZW.

Method	WINZIP on whole file	WINZIP on column	Our Strategy without LZW	Our strategy with file level LZW	Our strategy with LZW on column
Compression ratio for example 3.1	2.86	3.42	3.25	4.21	4.98
Compression ratio for example 3.2	12.5	18.1	16.1	20.9	24.6

Figure 3.1 improves our understanding of the source of the improved compression ratios. It shows the relative benefits of our strategy (column 5 of the table) to LZW (column 2 of the table) on a *per-attribute* basis. The gains of using semantic information are obvious. For instance, for the *O_ORDERDATE*

attribute in example 3.1 (query 3), the cross-attribute value constraint improves the compression ratio by a factor of more than 7.



Attributes	1	2	3	4	5	6	7	8
Query 3	*L_ORDERKEY*	*O_CUSTKEY*	*O_ORDERDATE*	*O_SHIPPRIORITY*	*L_LINENUMBER*	*L_EXTENDEDPRICE*	*L_DISCOUNT*	*L_SHIPDATE*
Query 15	*S_SUPPKEY*	*S_NAME*	*N_NAME*	*R_NAME*	*S_ADDRESS*	*S_PHONE*	*L_SHIPDATE*	*REVENUE*

Figure 3.1

4 A Framework for Query Result Compression

Compression can be seen as a mapping from the original data to compressed data. In text or image compression, the compression method itself is enough to express the mapping. However, in database systems, we may need to use different compression methods on different parts of the data, and the same data may be compressed multiple times. We saw two examples of possible compression strategies in the previous section. Here we present a general framework to model such a strategy as a *compression plan,* which is a sequence of *compression operators*, each of which accepts an input relation and produces an output relation. This framework is a central contribution of the paper.

4.1 Relations

The input and output of each compression operator is a relation. The result relation of the query acts as the input to the first compression operator. Every relation has the usual logical structure: it is a set of tuples, each of which has a fixed number of typed attribute values. For the sake of this discussion, we will assume that every tuple has a unique identifier, and that the tuple identifiers range from *zero* through *N-1*, each with *M* attribute values. Consequently, *A[I,j]* uniquely identifies the *j*'th attribute in tuple *I.*

Compression is a physical operation that does not affect the logical properties of the relation. Therefore, we need to represent the physical properties in an appropriate manner. We represent the physical compression properties of every relation by a set of *compression granules*, each of which is the minimal physical access unit within the relation. A compression granule contains:

- A possible composite data type representing the kind of data in the granule.

- A set of attributes *A[I,j]* that are contained in the granule. This is called the *attribute-granule mapping*. Any attribute is contained in exactly one granule (i.e., the mapping is N-to-1).
- A decompression method that can be applied to the granule to generate the individual attributes.
- Any additional information needed for the decompression method.

At a physical level, a relation is modeled as a set of granules, each of which defines the various components described. In a relation that has not been compressed, there is one trivial granule per attribute value. The effect of any compression operator is either to compress individual attribute values or to compress groups of attribute values together.

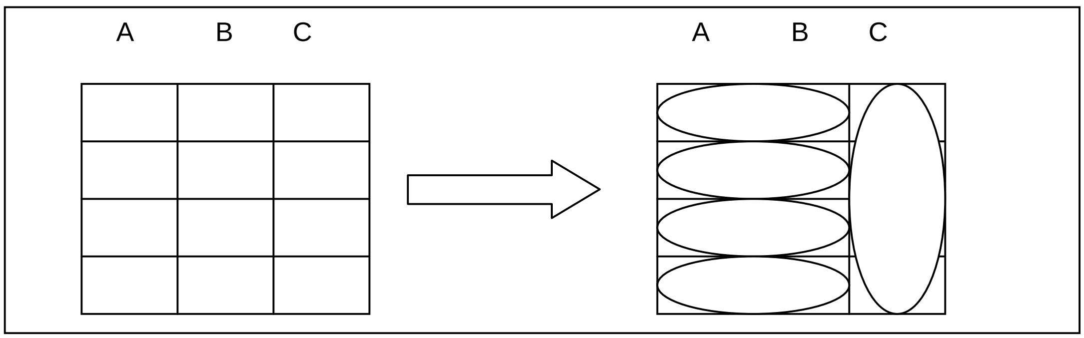


Figure 4.1

Figure 4.1 illustrates the physical view of a relation wit attributes A, B and C, after some compression operators have been applied. The figure shows that we compress A and B values for each tuple to a single granule and the C values of all tuples to a single granule.

4.2 Compression Operators

A compression operator takes an input relation, applies a compression algorithm to it, and produces an output relation. Each compression operator has certain properties of its own which help describe the properties of the output relation:

- *Output Granules:* The operator defines the set of output granules, and the properties of each granule are determined by the following operator properties.
- *Granule Scope:* This specifies the set of input granules that map to each output granule. Every granule in the input relation must map to exactly one output granule; there is a N-1 mapping from input granules to output granules (stated informally, a compression operator can only increase the compression granularity, but cannot shrink it). Consequently, the granule mapping of the operator indirectly defines the attribute-granule mapping of the output relation.
- *Compression Mask:* We emphasize that it is unlikely that a single compression operator compresses all the granules of a relation. It is more likely to compress the granules corresponding to a specific attribute, or group of attributes. The compression mask specifies the input granules that will be compressed. Any input granule that is eliminated by the compression mask is passed unchanged to the output relation.
- *Compression Scope:* This defines for each output granule, the set of input granules whose values contribute to the compressed granule value (this is not the same as the input granules actually represented by the compressed value). The compression scope for each granule is a superset of the granule scope. Further, each input granule can be part of the compression scope of multiple output granules. For instance, for differential encoding operator, the compression scope includes both the attribute value of the tuple that the operator works on and the attribute value of the preceding tuple.

- *Compression Method:* This is the method to apply to the compression scope, and it might generate some meta-information when compressing each granule. The compression method expects a certain data type for the input granules, and generates a compressed granule of a specific output data type.
- *Decompression Method:* This is the method that can be applied to each output granule along with its meta-information to decompress it.

For example, let us examine compression operators of two well-known compression methods.

- The differential method operates on each value of attribute A.
    - The output granules are still the individual A values, each in its granule.
    - The granule scope is a 1-1 mapping from input attributes to output attributes.
    - The compression mask includes all A values in input relation.
    - The compression scope includes A values of the current tuple and the preceding tuple because the compressed value depends on the difference between A values of current and preceding tuple.
    - The compression method is differential encoding and the decompression method is differential decoding.
- For the grouping method on attribute A,
    - Each output granule includes a distinct A value and the number of its occurrence.
    - The granule scope includes all occurrences of this A value.
    - The compression mask is still all A values.
    - The compression scope is all values of *A,* because we have to look at all A values to find all occurrences of a specific A value.
    - The compression method is grouping and the decompression method is its converse operation.

Having defined this notion of compression operators, we can now classify well-known compression strategies in these terms. The strategies in WINZIP [WEL84], SYBASE IQ [SYB98], DB2 [IW94] and tuple differential method [NR95] use exactly one compression operator. The page level offset encoding strategy [GRS98] and COLA [GHS95] contain a sequence of operators with similar properties. Table 4.1 summarizes the properties of the operators in these strategies.

Strategy	Compression Methods	Granule Scope	Compress Granule	Compression Mask
WINZIP, GZIP	LZW	The whole relation	The whole relation	The whole relation
SYBASE IQ	LZW	Tuple within a page	Tuple within a page	The whole relation
DB2	Non adaptive Ziv-Lempel	Each tuple	The whole relation	The whole relation
Page level offset encoding.	Offset encoding	Individual attribute for each tuple	Values of an attribute within a page	Values of an attribute in the whole relation
Block-Oriented-tuple differential	Offset encoding	All attributes of a tuple	Tuple within a page	The whole relation
COLA	Non adaptive arithmetic encoding	Individual attribute for each tuple	Values of an attribute in the relation	Values of an attribute in the relation

Table 4.1

4.3 Compression Plans

A compression plan is a valid sequence of compression operators. The output of each operator acts as the input of the next operator in the sequence. The validity of the sequence is determined by checking that each operator is compatible with its input relation (this corresponds to “type-checking” the plan). An operator is compatible with its input relation if the following properties hold:

- The data type of the input granules included by the compression mask is the same as the input data type for the compression method.

- The input granules used to specify the granule scope and compression scope of the operator correspond to the granules of the input relation (in other words, the expected input granularity matches the actual input granularity). Clearly, if we have compressed several input granules into the output granule, we can no longer access the individual input granule without decompression. Formally, for each granule in input relation, its attribute granule mapping must either disjoint with the granule scope of current operator or be a subset of some input granule in granule scope, i.e. compression granules can only grow.

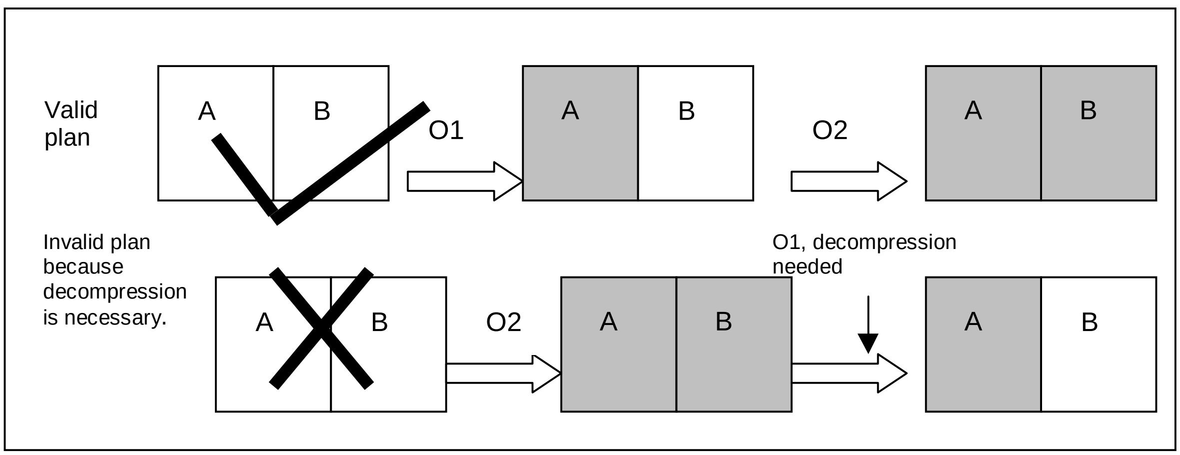


Figure 4.2

For instance, figure 4.2 shows a plan consists of two operators O1 and O2. O1 has granule scope (page, attribute A) and O2 has granule scope (page, attribute A and B). {O1, O2} is a valid plan while {O2, O1} is not. The dark area shows the granule scope.

4.4 Examples of Compression Plans

In the previous section, we presented a sample compression strategy for Example 3.2. We now reexamine that strategy, presenting it as a compression plan. It consists of the following sequence of compression operators: (the compression masks are always the whole columns of attributes appearing in granule scope)

Method	Granule Scope	Compress Scope
Normalization	*Group of S_SUPPKEY, S_NAME, N_NAME, R_NAME, S_ADDRESS,*	These attributes in the relation.

	S_PHONE values	
Differential Encoding	Each distinct *S_SUPPKEY* value	This attribute in the whole relation
NULL Suppression for each string attribute	Each string attribute	This attribute in the whole relation
Dictionary	*N_NAME, R_NAME* in each tuple	These attributes in the relation
Six bits encoding for characters including letters, '.', ',', '#', etc.	*S_ADDRESS* in each tuple	*S_ADDRESS* in current tuple.
Four bits encoding for characters including digits, '-'	*S_PHONE* in each tuple	*S_PHONE* in current tuple.
Use one byte to encode month and day.	*L_SHIPDATE* in each tuple	*L_SHIPDATE* in each tuple
Encode ASCII form of float value.	*REVENUE* in each tuple	*REVENUE* in each tuple
LZW encoding for each column (attribute *N_NAME* and *R_NAME* are treated as one column)	Each attribute in the whole relation	Each attribute in the relation

Table 4.2

5. Experimental Results

This section demonstrates the compression ratios that can be achieved by semantic compression plans. We needed to choose a set of queries that are both realistic and have large result size so that result compression is meaningful. We accomplished this by adapting the queries in the TPC-D benchmark [TPC95]. We consider this a representative choice because:

- Queries in the TPC-D benchmark are considered characteristic of a wide range of decision support systems. However most of queries include group by and aggregation, making their result sizes small. In the environment that 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 GROUP BY clause and aggregation predicates in SELECT clause.

- The data in the TPC-D benchmark contains various data types such as string, integer, decimal and character. The character distribution of string fields also varies and some of the fields are easy to compress while others are not.

Our primary metric of interest is the compression ratio (size of original query result divided by size of compressed query result). We expect our experiments to demonstrate the following:

- Semantic compression leads to significantly higher compression ratios than naïve compression techniques.

- The compression ratio varies from query to query, depending on the nature of the query.

In short, we expect the results in this section to justify our interest in semantic compression of query results.

5.1 Improvement on Compression Ratio

We first test whether semantic information can improve the compression ratio of query results. We run a handcrafted compression plan for each adapted query and compare the compression ratio with that using WINZIP on the whole file. Figure 5.1 shows the results for 17 adapted queries. Except for query 17, using semantic information improves the compression ratio.

Table 5.1 shows the average compression ratio for WINZIP and our plans:

	WINZIP	Our Plans
Average Compression Ratio	5.16	9.18

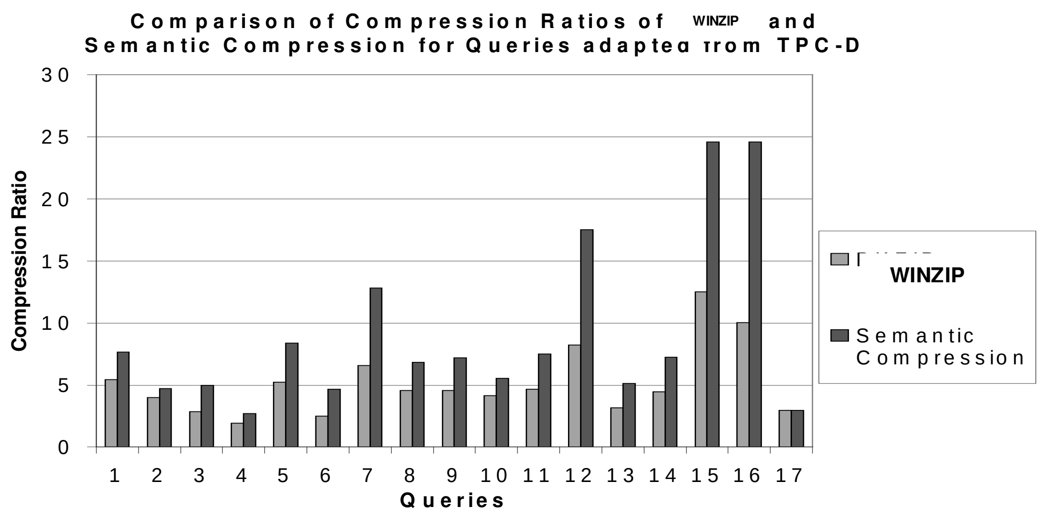


Figure 5.1 Compression Ratios

Figure 5.1 shows that using WINZIP can achieve an average compression ratio of 5.16, which is higher than the general compression ratio for database tables. The reason may be there are more redundancies in the query result than original data. Our plans achieve an average compression ratio of 9.18, which is 77% higher than that of WINZIP. While space does not allow us to present each of our compression plans, we include the plan for Query 16 in the appendix (since this has a high compression ratio).

We define the ***gain*** for each query as follows:

Gain = (Compression Ratio of our plan) / (Compression ratio of WINZIP) - 1.

The gain is a measure of the improvement due to the use of semantic information. Figure 5.2 shows the compression ratios for eight out of the seventeen queries are improved by 40 to 60 per cent. For other six queries, the improvement is over 60%. Only three queries (query 2, 10 and 17) have lower than 40% improvement in compression ratios.

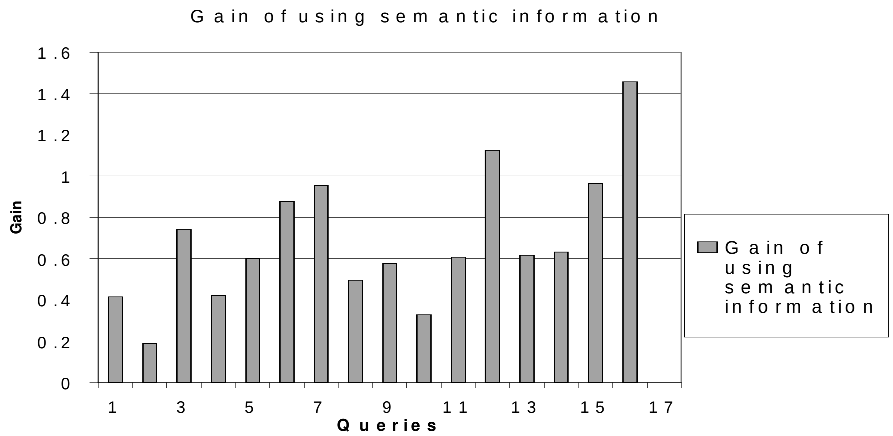


Figure 5.2 Gain of our plans over WINZIP (in percentage)

Query 17 has no improvement at all. The reason is that the result of query 17 only contains one attribute with the floating-point data type. Since there is no additional semantic information, the compression plan using semantic information is not expected to outperform WINZIP. For query 2 and query 10, there are two large string fields, *C_ADDRESS* and *C_COMMENT,* whose values are generated based on uniform distribution. Therefore, there is no useful semantic information for these two fields and we can not achieve better compression on these two fields. Since these two fields form a substantial proportion of the whole data (about 50%), the gain is not that significant.

Therefore, the result agrees with the "more semantic information, more compression" principle. Using semantic information generally can improve the compression ratio significantly (75% on average).

5.2 Comparison of Column-based Compression Plans

We compare our plans with **column-based** compression plans that use WinZIP on each attribute. Column-based plans only use the information that values of the same attribute usually have more similarities than those values from different attributes. Therefore, this experiment should indicate whether this level of semantic information is sufficient, or whether the fancier information used by our plans are really necessary. In Figure 5.3, we show the total gain of our compression plans (this is the same as Figure 5.2) and extra gain over column-based plans using WinZIP. The extra gain is computed as the compression ratio of our plans minus that of column-based plans. We also plot the ratios between the two in Figure 5.4.

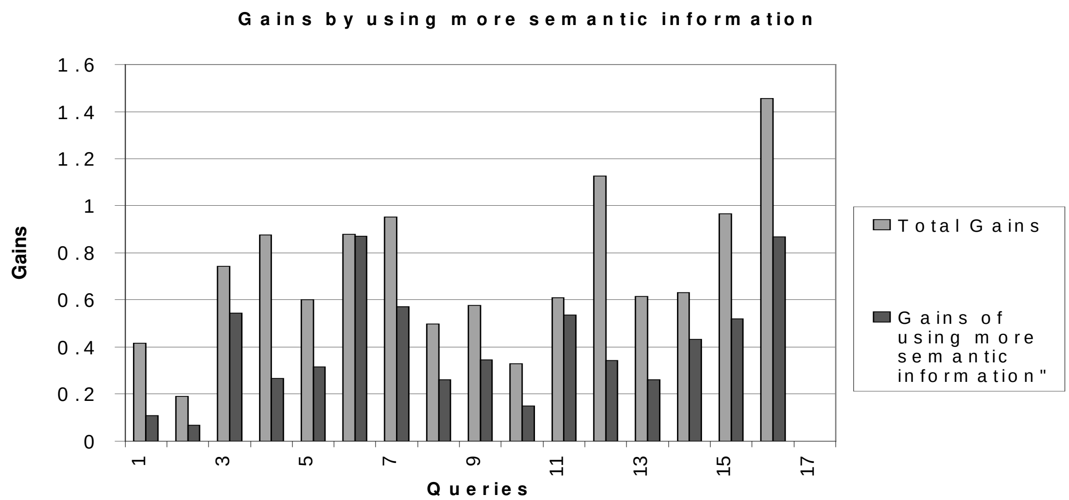


Figure 5.3 Comparison of gain of our plans and extra gain over column-based plans.

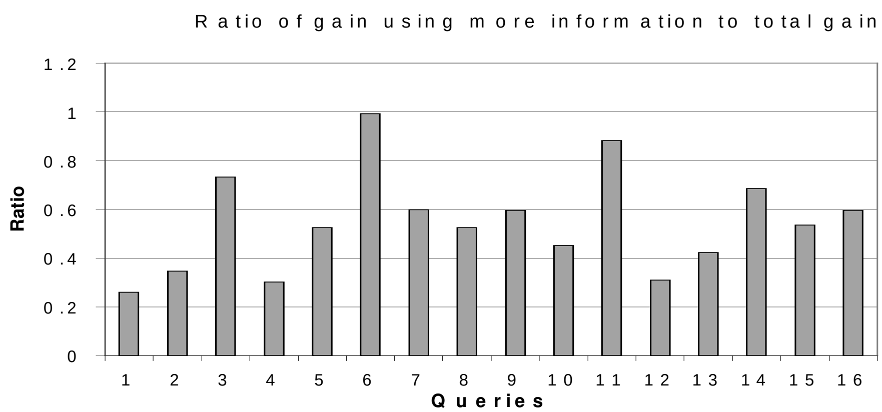


Figure 5.4 Ratio of extra gain to total gain.

Figure 5.4 shows that for most queries, using extra semantic information contributes more than 50% of the total gain. For five queries (query 4, 6, 11, 12, 16), extra semantic information accounts for more than 80% of the total gain. Only for three queries (query 1, 2 and 10) does the use of extra information contribute less than 50% of the compression gain. The reason is that there is not much useful extra information in these three queries. For instance, all fields in query 1 are floating-point numbers and not much information is available. Query 2 has a functional dependency as:

S_SUPPKEY -> S_NAME, S_ACCTBAL, N_NAME, N_ADDRESS, S_PHONE, S_COMMENT.

However, the data is not ordered on S_SUPPKEY, so this information can not be used.

5.3 Comparison of Gains Between Different Categories of Query Result

We divide the query results into three categories:

- Mostly numerical data. Such results have more than 60% numerical data.
- Mostly string data. Such results have less then 20% numerical data.
- Those in between.

Figure 5.5 shows the gains for queries in each category.

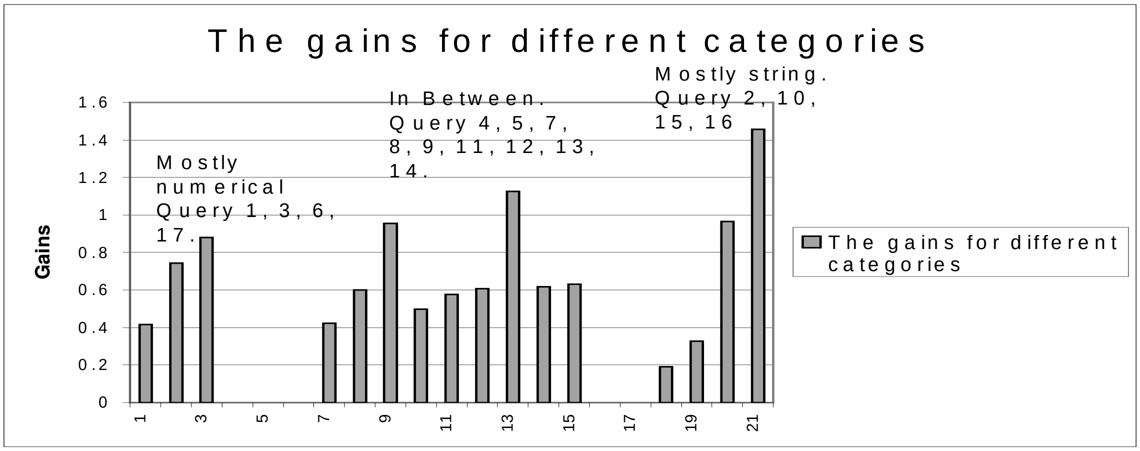


Figure 5.5 Gains for different categories of query results

The figure does not show any clear difference of gains between different categories. This suggests that the achieved compression ratio depends more on the available semantic information rather than on the data types of the attributes.

6 Optimization Issues

We have presented the compression plan framework and demonstrated that compression plans conforming to this framework can create highly compressed query results. However, these plans were hand-crafted rather than automatically produced. Clearly, there is a need for a "compression plan optimizer" that acts like a relational query optimizer to find the "best" compression plan. We do not propose a concrete optimization algorithm; it is a topic of ongoing research. Instead, we present our initial thoughts on the issues and suggest directions that appear promising.

6.1 Goal of Optimization

The goal of compression plan optimization may be to:

(a) maximize the compression ratio, which measures the network and client memory savings,

(b) minimize the compression CPU cost, which measures the effort that the server needs to process the data,

(c) minimize the decompression cost, which measures the effort that the client needs to use to access the data.

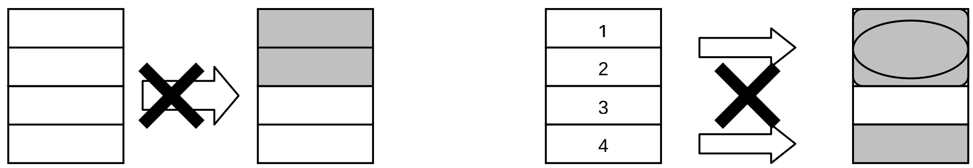
Clearly, the optimization goal in practice is to accomplish some weighted combination of these conflicting individual goals. For now, we will assume that maximizing compression ratio is the primary goal.

6.2 Limiting the Search Space

An exhaustive optimization algorithm would enumerate all valid sequences of compression operators and choose that with the highest compression ratio. This is clearly an unreasonable approach. We have identified some practical constraints that might limit the search space.

* ***Constraint 1***: The compression mask must include one or several attributes of all tuple. We do not consider such operators that only compress some subset of the tuples.

* ***Constraint 2***: The granule scopes and compression scopes are homogeneous across all tuple.



Violation of constraint 1, dark area is compression mask.

Violation of constraint 2, Tuple 1, 2 are mapped to one output granule and tuple 4 is mapped to one output granule.

Figure 6.1

Therefore, we can express the granule scope in a general form as a pair (X, Y), where X defines the horizontal level of granule scope and Y defines the vertical level. X can be tuple, group of tuple, page or file. Y is a set of one or more attributes.

* ***Constraint 3***: We never decompress a previous result in order to use a new compression operator. Then the granule scopes of operators in a valid compression plan can only grow.

* ***Constraint 4***: Each compression operator has a strict memory constraint and disk I/O is not permitted. This means:

    (a) The granule scope can not be larger than the memory constraint.

    (b) Compression plans must be able to be pipelined so that no intermediate results need to be saved.

    (c) No operators can require reordering of the entire input relation.

    (d) No operators can use an extra scan of the result to collect semantic and statistical information.

We may also use certain heuristics to further limit the space of options and guide the search through that space.

* ***Heuristic 1***: Operators with fewer constraints on subsequent operators should be considered early. This allows us to have less strict validity constraints in choosing subsequent operators, thus increases the chance that we find a better plan. Which operators have looser constraints on subsequent operators? As we discussed earlier, the granule scope of operators can only grow in a valid compression plan. Thus, operators with smaller granule scopes have fewer constraints than those with larger granule scopes (which are more likely to invalidate any subsequent operators with smaller granule scopes).

* ***Heuristic 2***: Apply operators that use semantic-based methods earlier rather than later. This agrees with the *"more semantic information, more compression"* principle. The semantic information is usually available on the uncompressed version of the data. Consequently, the introduction of other compression operators earlier makes it more likely that the semantic information will become useless.

* ***Heuristic 3***: Choose operator having higher compression ratios earlier.

* ***Heuristic 4***: The order between operators with disjoint compression masks are usually irrelevant. For instance, figure 6.1 (in section 5.1) shows two operators compressing attribute A and B separately.

The compression masks of these two operators do not intersect. Thus, there is no difference between compressing A then compressing B and compressing B then compressing A. In fact, the two compression algorithms will probably be combined into one compression operator.

6.3 Costs and Statistics

The compression ratio of each operator can be estimated based on the statistical and semantic information. For instance, for a differential operator compressing each attribute value separately, the compression ratio is:

$$\frac{\text{The size of average difference between adjacent attribute values}}{\text{the size of actual attribute value}}$$

At times, it is difficult to estimate the compression ratio accurately, and we will require estimation capabilities to predict compression ratios. After we estimate the compression ratio of an operator, we can estimate the size of the output relation as following:

$$\text{size of data not in compression mask} + \frac{\text{size of data under compression mask}}{\text{compression ratio of the operator}}$$

We can compute the size of output relation for each operator in turn and then the compression ratio for the whole plan is:

$$\frac{\text{Size of output relation}}{\text{Size of the input relation}}$$

For example, figure 6.2 shows size of the output relation for two operators O1 and O2, where the dark area is the compression mask. The size of the initial input relation is 10M and the size of the final output relation is 1M. Thus, the overall compression ratio is 10.

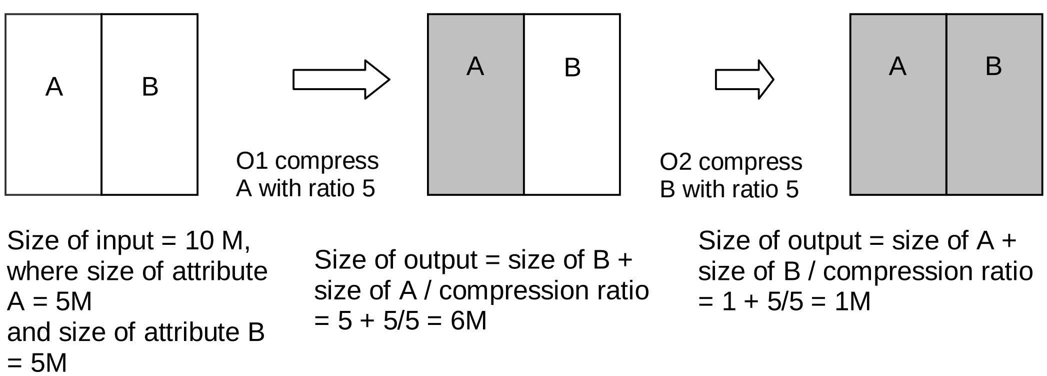


Figure 6.2

We must also propagate semantic information through the compression plan. Every operator might modify the semantic information that is valid of its result. In general, we hope to infer new semantic information from the original information on the input table and properties of the operator.

We are currently in the process of designing and building an appropriate compression optimizer based on some these observations. Compression optimization appears to be an area with significant potential for future research.

6.4 Compression Plans and Query Plans

Another promising area of future work is the interaction between compression plans and query plans. This paper has assumed that the results of queries are compressed, but the query evaluation itself is unaware of this use of its results. However, if we treat the result compression as part of the regular query execution, many interesting issues arise. Portions of the compression plan may be pushable into the query evaluation plan, without compromising the correctness of the answer. For example, a individual attribute compression methods may be applied as early as possible to limit the size of an intermediate query result. Another intriguing example is to move a normalization operator closer to the join that actually created the unnormalized data. This kind of optimization provides new challenges to the relational query optimizer, as well as to the compression optimizer.

7 Conclusion and Future Work

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

- We present a framework to model compression strategies. The framework models a compression plan as a sequence of compression operators. This is similar in spirit to a relational query plan.

- We describe the use of semantic information from the database catalogs and the queriy to compress query results.

- We present experiments on queries adapted from the TPC-D benchmark to show that the use of semantic information generally improves compression ratio by a factor of 75% over WinZip.

- We identify open research problems in the areas of optimization of compression plans and the interaction of compression and regular query processing.

Acknowledgement

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).

References

[ALM96] G. Antoshenkov, 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.. October 1985.

[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, Dec. 1985.

[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 (1983).

[GRA93] G. Graefe. Options in Physical Databases. SIGMOD Record, September 1993.

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

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

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

[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.

[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.

[NR95] W. K. Ng, C.V. Ravishankar. Block-Oriented Compression Techniques for Large Statistical Databases. IEEE Trans. for Knowledge and Data Engineering, Taipeh, Taiwan, 1995.

[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.

[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, 1983.

[SNG93] Leonard Shapiro, Shengsong Ni, Goetz Graefe. Full-Time Data Compression: An ADT for Database Performance. Portland State Univ., OR, USA, 1993.

[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. Comm. Of ACM. June 1984.

[WKHM98] Till Westmann, Donald Kossmann, Sven Helmer, Guido Moerkotte. The implementation of performance of compressed databases.

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

Appendix

<u>Compression Plan for Adapted Query 16</u>

The adapted query 16 is as following:

```
select P_BRAND, P_TYPE, P_SIZE, PS_SUPPKEY
from PARTSUPP, PART
WHERE
        P_PARTKEY = PS_PARTKEY AND
        P_BRAND <> 'Brand#45' AND
        P_TYPE NOT LIKE 'SMALL ANODIZED%' AND
        P_SIZE IN (27, 1, 19, 17, 32, 12, 11, 33) AND
        PS_SUPPKEY NOT IN
          (SELECT S_SUPPKEY FROM SUPPLIER WHERE S_COMMENT LIKE
          '%Better Business Bureau%Complaints%')
order BY P_BRAND, P_TYPE, P_SIZE, P_SUPPKEY
```

Following table shows the compression plan:

Method	Granule Scope	Compress Scope
Grouping	*Group of P_BRAND*	This attribute in the relation.
Grouping	Group of *P_TYPE*	This attribute in the whole relation
NULL Suppression for each string attribute	Each string attribute	This attribute in the whole relation
Dictionary	*Each P_TYPE value*	This attributes in the relation
NULL Suppression	*P_SIZE, P_SUPPKEY*	This attributes in the relation
LZW encoding for each column	Each attribute in the whole relation	Each attribute in the relation

Following table shows the compression ratio of our plan divided by the compression ratio of column-wise LZW for each attribute.

	P_BRAND	*P_TYPE*	*P_SIZE*	*PS_SUPPKEY*
Our ratio/ LZW	3	1.8	1.45	1.5

Note that we have order information about *P_BRAND* and *P_TYPE*, so that we can use two grouping operators. This is the main reason we achieve much higher compression ratio than WINZIP.