# Query Optimization In Compressed Database Systems

Zhiyuan Chen*
Cornell University
zhychen@cs.cornell.edu

Johannes Gehrke*
Cornell University
johannes@cs.cornell.edu

Flip Korn
AT&T Labs–Research
flip@research.att.com

## ABSTRACT

Over the last decades, improvements in CPU speed have outpaced improvements in main memory and disk access rates by orders of magnitude, enabling the use of data compression techniques to improve the performance of database systems. Previous work describes the benefits of compression for numerical attributes, where data is stored in compressed format on disk. Despite the abundance of string-valued attributes in relational schemas there is little work on compression for string attributes in a database context. Moreover, none of the previous work suitably addresses the role of the query optimizer: During query execution, data is either *eagerly* decompressed when it is read into main memory, or data *lazily* stays compressed in main memory and is decompressed on demand only.

In this paper, we present an effective approach for database compression based on lightweight, attribute-level compression techniques. We propose a Hierarchical Dictionary Encoding strategy that intelligently selects the most effective compression method for string-valued attributes. We show that eager and lazy decompression strategies produce suboptimal plans for queries involving compressed string attributes. We then formalize the problem of compression-aware query optimization and propose one provably optimal and two fast heuristic algorithms for selecting a query plan for relational schemas with compressed attributes; our algorithms can easily be integrated into existing cost-based query optimizers. Experiments using TPC-H data demonstrate the impact of our string compression methods and show the importance of compression-aware query optimization. Our approach results in up to an order speed up over existing approaches.

## 1. INTRODUCTION

Over the last decades, improvements in CPU speed have outpaced improvements in main memory and disk access speeds by orders of magnitude [6]. This technology trend has enabled the use of data compression techniques to improve performance by trading reduced storage space and I/O against additional CPU overhead for compression and decompression of data. Compression has been utilized in a wide range of applications from file storage to video processing; the development of new compression methods is an active area of research.

In a *compressed database system*, data is stored in compressed format on disk and is either decompressed immediately when read from disk or during query processing. Compression has traditionally not been used in commercial database systems because many compression methods are effective only on large chunks of data and are thus incompatible with random accesses to small parts of the data. In addition, compression puts extra burden on the CPU, the bottleneck resource for many relational queries such as joins [29]. Nonetheless, recent work on attribute-level compression methods has shown that compression can improve the performance of database systems in read-intensive environments such as data warehouses [13, 29].

The main emphasis of previous work has been on the compression of numerical attributes, where coding techniques have been employed to reduce the length of integers, floating point numbers, and dates [13, 25]. However, string attributes (*i.e.*, attributes declared in SQL of type `CHAR(n)` or `VARCHAR(n)`) often comprise a large portion of the length of a record and thus have significant impact on query performance. For example, the TPC-H benchmark schema contains 61 attributes, out of which 26 are string-valued, constituting 60% of the total database size. Surprisingly, there has not been much work in the database literature on compressing string attributes. Classic compression methods such as Huffman coding [18], arithmetic coding [31], Lempel-Ziv [32, 33] (the basis for `gzip`), and order-preserving methods [4] all have considerable CPU overhead that offsets the performance gains of reduced I/O, making their use in databases infeasible [12]. Hence, existing work in the database literature employs simple, lightweight techniques such as NULL suppression and dictionary encoding [6, 29]. This paper contributes such an effective and practical database compression method for string-valued attributes. Our method achieves achieves better compression ratios than existing methods while avoiding high CPU costs during decompression.

An important issue in compressed database systems is when to decompress the data during query execution. Traditional solutions to this problem consisted of simple strate-

*Supported in part by NSF Grant IIS-9812020, NSF Grant EIA-9703470, and a gift from AT&T Corporation.


*ACM SIGMOD 2001* May 21-24, Santa Barbara, California, USA

```
Select  S_NAME, S_COMMENT, L_SHIPINSTRUCT, L_COMMENT
From    Supplier, Lineitem
Where   S_SUPPKEY = L_SUPPKEY and
        S_COMMENT <> L_SHIPINSTRUCT
Orderby S_NAME, L_COMMENT
```

**Figure 1: Example Query**

gies, while we view this problem in the larger framework of compression-aware query optimization. In the following we survey well-known and new strategies for decompression in query plans. Then we show for an example query how efficient query plans can only be generated by fully integrating query optimization with the decision of when and how to decompress.

Early work in the database literature proposed *eager decompression* whereby data is decompressed when it is brought into main memory [19]. Eager decompression has the advantage of limiting the code changes caused by compression to the storage manager. However, as Graefe et al. point out, eager decompression generates suboptimal plans because it does not take advantage of the fact that many operations such as projection and equi-joins can be executed directly on compressed data [15].

Another strategy is *lazy decompression*, whereby data stays compressed during query execution as long as possible and is (explicitly) decompressed when necessary [12, 29]. However, this decompression can increase the size of intermediate results, and thus increase the I/O of later operations in the query plan such as sort and hashing. Westmann et al. suggest explicitly compressing intermediate results [29], but as pointed out by Witten et al. [30], compression is usually quite expensive and can wipe out achievable benefits. We assume in the remainder that compression never occurs during query execution and that an attribute once uncompressed will stay uncompressed in the remainder of a query.

We contribute a new decompression strategy that we call *transient decompression*. In transient decompression, we modify standard relational operators to temporarily decompress an attribute $x$, but keep $x$ in compressed representation in the output of the operator. We refer to such modified operators that input and output compressed data as *transient operators*.

Note that since numerical attributes are cheap to decompress, transient decompression usually outperforms lazy and eager decompression for numerical attributes. Unfortunately, for string attributes the choice between the three decompression strategies is not so easy. And query plans involving compressed string attributes must be chosen judiciously. On the one hand, transient decompression on string-valued attributes can result in very significant I/O savings because (1) string attributes are typically much longer than numerical values, and (2) string attributes are often easy to compress (*e.g.*, string attributes with small domains can be compressed to one or two bytes). On the other hand, decompressing string attributes is much more expensive than decompressing numerical attributes, and transient operators may need to decompress the same string value many times (*e.g.*, consider a nested loops join where the join attribute is a string). The following example illustrates that choosing the right query plan is an important decision.

Figure 1 shows an example query from the TPC-H benchmark [1]. The query joins the `Supplier` and `Lineitem` relations on a foreign key; the query includes an additional selection condition involving two string attributes. The string attributes `S_COMMENT`, `L_SHIPINSTRUCT`, `L_COMMENT`, and `S_NAME` are compressed using attribute-level dictionary compression with different dictionaries; none of the compression methods is order-preserving except the method used for `S_NAME` (Section 2 describes our compression methods in detail). Thus, during the execution of the example query, the attributes `S_COMMENT` and `L_SHIPINSTRUCT` need to be decompressed for computing the join and `L_COMMENT` needs to be decompressed for the sort.

We ran this query on a modified version of the Predator Database Management System [2] where we implemented the eager and lazy strategies, as well as variants of the transient decompression strategies (A detailed description of our experimental setup can be found in Section 4).

Table 1 reports the execution time of different query execution plans. Plans 1 to 4 use a block-nested-loops join followed by a sort.[1] Plans 1 uses eager decompression and Plan 2 uses lazy decompression; the running times were 1515 and 1397 seconds, respectively. Plan 3 explicitly decompresses attributes for the join operator and uses transient decompression for the sort operator, which improves the running time by about a factor of two to 712 seconds. The reason for this improvement is that the size of the intermediate results (the input to the sort operator) in Plan 3 is significantly smaller than the intermediate results in Plan 1 and 2. In Plan 3, the long string attribute `L_COMMENT` stayed compressed, whereas the attribute is already decompressed in Plans 1 and 2. Since the performance of the sort operator is very sensitive to the size of the input relation, keeping `L_COMMENT` compressed leads to better overall performance (the sort took 612 seconds in Plan 3 versus 1307 seconds in Plan 2).

If we choose transient decompression for both the join and the sort operators, the execution time jumps to 3195 seconds, as shown in Plan 4 in Table 1. Plan 4 keeps the join attributes `S_COMMENT` and `L_SHIPINSTRUCT` compressed in the intermediate results, leading to better performance for the sort operator (the sort time drops from 612 seconds to 102 seconds). However, the nested-loops join needs to test the join condition for pairs of (`S_COMMENT`,`L_SHIPINSTRUCT`) values. Thus, transient decompression is invoked a quadratic number of times in the sizes of the input relations, leading to a prohibitive CPU overhead (the join time increases from 90 seconds to 3093 seconds). This is an extreme case of the classic "CPU versus I/O" trade-off, demonstrating that transient operators should not be deployed arbitrarily.

Whereas Plans 1-4 use block-nested-loops join, Plan 5 uses a sort-merge join, which is less efficient in the view of a traditional optimizer, but Plan 5 also uses transient decompression for both the join and the sort operator. Surprisingly, its execution time drops to 302 seconds, more than a factor of two improvement over the previously best plan (Plan 4). Although Plan 5 takes more time for processing the join

[1]Predator contains block-nested-loops and sort-merge joins. A traditional optimizer enhanced with a cost model takes into account both I/O benefits of compression and CPU overhead of decompression chose a block-nested-loops join because the `Supplier` relation fits into the buffer pool, but the `Lineitem` table does not, thus block-nested-loops join has lower overall cost.

Table 1: Execution times (in seconds) for the query in Figure 1 (different decompression strategies)

| *Plans* | *Strategy* | *Total Time* | *Join Time* | *Sort Time* |
|---|---|---|---|---|
| Plan 1 | Eager with BNL-join | 1515 | 96 | 1419 |
| Plan 2 | Lazy with BNL-join | 1397 | 90 | 1307 |
| Plan 3 | BNL-join: explicit; sort: transient | 712 | 90 | 612 |
| Plan 4 | BNL-join: transient; sort: transient | 3195 | 3093 | 102 |
| Plan 5 | SM-join: transient; sort: transient | 302 | 182 | 102 |

(182 seconds versus 90 seconds for Plan 3), it keeps the intermediate results compressed. This lowers the cost for the sort operator (102 seconds versus 612 seconds for Plan 3), since the intermediate results of Plan 5 fit into the buffer pool. Plan 5 illustrates a central point: Query optimization in compressed database systems needs to *combine* the search for optimal plans with the decision of how and when to decompress.

In this paper, we study the problem of compression-aware query optimization in a compressed database system. We make the following contributions:

- We propose a Hierarchical Dictionary Encoding strategy that intelligently selects the most effective compression methods for string-valued attributes. (Section 2)
- We formalize the problem of compression-aware query optimization, and propose three query optimization algorithms: a provably optimal and two fast heuristic algorithms. Our algorithms can easily be integrated into existing cost-based optimizers. (Section 3)
- We present an extensive experimental evaluation using a real database system on TPC-H data to show the importance of compression-aware query optimization. The presence of string attributes makes query processing particularly sensitive to the choice of plan. Our methods result in up to an order of magnitude speedup over existing strategies. (Section 4)

We discuss related work in Section 5 and conclude in Section 6.

## 2. DATABASE COMPRESSION

The nature of query processing in databases imposes several constraints in choosing a suitable database compression method. First, the decompression speed must be extremely fast. Since we intend to apply transient operators, the query processor may need to decompress the data many times during query execution. Second, only fine-grained decompression units (*e.g.*, at the level of a tuple or a attribute) are permissible in order to allow random access to small parts of the data without incurring the unnecessary overhead of decompressing a large chunk of data.

Common compression methods include Lempel-Ziv [32, 33], Huffman coding [18], arithmetic encoding [31], and predicative coding [9]. Unfortunately, the decompression speeds of these methods are not fast enough; for example, the performance difference between LZ and simple methods, like offset encoding (encoding a numerical value as the offset from a base value) is an order of magnitude [12]. Hence, we limit our consideration to lightweight methods. Dictionary-based encoding involves replacing each string $s$ by a key value $k$ that uniquely identifies $s$; a separate table called the *dictionary* stores the necessary $\langle s, k\rangle$ associations. *Adaptive* compression methods (such as LZ) build the dictionary on-the-fly during compression. However, as pointed out by Chen et al. [8] and Goldstein et al. [12], adaptive compression methods require large chunks of data as inputs to achieve good compression ratios. Even when adaptive methods are applied on a page-level they are insufficient for our needs because access to a single tuple requires decompressing the entire page. To allow fine-grained decompression, only methods that are *static* (*i.e.*, the dictionary is fixed in advance) or *semi-static* (*i.e.*, the dictionary is built during preprocessing and fixed thereafter) are acceptable.

Since the attributes of a relational database typically consist of heterogeneous attribute types, the most suitable compression method for each attribute may be different, and should be chosen separately. Following Kossman et al. [29], we compress numerical attributes by applying offset encoding to integers and by converting 8-byte double-precision floats to 4-byte single-precision floats if there is no loss in precision. For string-valued attributes, we propose a simple hierarchical semi-static dictionary-based encoding, which we describe next.

Existing work has applied simple dictionary-based encoding to the set of strings in an attribute (*i.e.*, one dictionary entry for each distinct string). But repetition in string attributes often exists at different levels of granularity, and applying dictionary encoding at the appropriate level can greatly improve the compression ratio. We thus consider dictionary encoding at the whole-string level, the "word" level (*e.g.*, English text), the prefix/suffix level (*e.g.*, URLs and e-mail addresses), and adjacent-character-pair level (*e.g.*, phone numbers).

Given this hierarchy of dictionary encodings, we can determine the level most suitable for each attribute separately as follows. Each level of granularity $\ell$ has an associated substring unit $u_\ell$ (*e.g.*, whole-string, word, *etc.*). Let $W^\ell = \{w_i^{(\ell)}\}$ be the set of distinct unit $u_\ell$ substrings of a given attribute (*e.g.*, the set of words) and let $n_\ell$ be the cardinality of $W^\ell$ (*e.g.*, the number of distinct words). Let $N_\ell$ be the total number of (non-distinct) substrings (*e.g.*, the number of word occurrences including duplicates). We choose the level $\ell$ that minimizes $b * N_\ell + \left(\sum_i^{n_\ell} |w_i^{(\ell)}| + b * n_\ell\right)$ where $|w_i^{(\ell)}|$ denotes the length of the (unit $u_\ell$) substring $w_i^{(\ell)}$, and $b$ is the length of the key value (in bytes). This is the size of the encoded attribute plus the size of the dictionary. As we demonstrate in Section 4.2, HDE is very effective for string-valued data, achieving higher compression than existing compression methods.

# 3. COMPRESSION-AWARE QUERY OPTIMIZATION

Section 1 illustrated that the difference between the use of simple heuristics for choosing query plans suing decompression and the use of compression-aware optimization is significant. This section starts with a formal introduction of the query optimization problem for compressed databases (Section 3.1). We then briefly discuss the relationship of compression-aware query optimization with the problem of query optimization with expensive predicates (Section 3.2). We then propose new query optimization algorithms in Sections 3.3 and 3.4.

## 3.1 Problem Definition

We adopt the notion of properties, also called tags, to describe which attributes are compressed in intermediate results of a plan. The property concept extends the idea of an interesting order from Selinger et al. [26]. We associate with each relation $r$ a so-called *tag*, denoted $tag(r)$, which contains the set of attributes in $r$ that are compressed. The *tag* of a plan $p$, $tag(p)$ is the tag of the output relation of $p$.

Let us extend the physical algebra with a *decompression operator* $D_X$ that decompresses a set of attributes $X$. The decompression operator takes as input a relation $r$ whose tag is a set of attributes $\mathcal{X}$ that is a superset of $X$: $X \subseteq \mathcal{X}$. Its output is the same relation but with a tag that is reduced by the decompressed attributes: $\mathcal{X} \setminus X$.

We also extend the physical algebra with transient versions of the traditional operators. A traditional physical algebra operator takes as input relations $r_1, \ldots, r_k$, each with an empty tag, and produces an output relation $r$, also with an empty tag. The *transient* version $o^T$ of operator $o$ takes as input relations $r_1, \ldots, r_k$ with possibly non-empty tags and produces an output relation $r$ with a possibly non-empty tag $X$, which is the union of the tags of $r_1, \ldots, r_k$ minus the set of attributes that were dropped by $o$. If an attribute $x$ appears in the output relation $r$ of operator $o^T$ and $x$ was compressed in the input, then $x$ is also compressed in $r$ and thus $x \in tag(r)$. Vice versa, if $x$ was not compressed in the input it will not be compressed in the output. Thus operator $o^T$ decompresses attributes only transiently as necessary, while attributes compressed in the input remain compressed in the output.

We can now characterize eager and lazy decompression. In eager decompression, every query plan contains decompression operators directly after each base relation scan. Thus all the tags of the intermediate relations are empty because we decompress all attributes directly when the base relations are read into memory. In lazy decompression, we insert a decompression operator $D_X$ directly before a physical algebra operator $o$ if $o$ requires access to the attributes in $X$, which have not been decompressed yet. Thus we delay the decompression of each attribute as much as possible.

In order to define the search space of compression-aware query plans, let us first introduce the notion of a query plan. A query plan $q$ has two components: (1) a *query plan structure* $(V, E)$, consisting of nodes $V$ and edges $E \subset V \times V$, and (2) a *query plan tagging*, which is a function $tag$ that maps each node $v \in V$ onto the set of attributes $tag(v)$ that are still compressed in $v$'s result. The internal nodes of the tree are instances of operators in the physical algebra of the database system (including decompression operators and transient operators), and the leaf nodes are base relation scans. Edges in $E$ lead from children nodes $v_1, \ldots, v_k$ to parent node $v$, indicating that the output of operators $v_1, \ldots, v_k$ is the input of operator $v$. The tag of each node $v \in V$, $tag(v)$, represents the set of compressed attributes in the output relation of the query plan fragment rooted at node $v$.

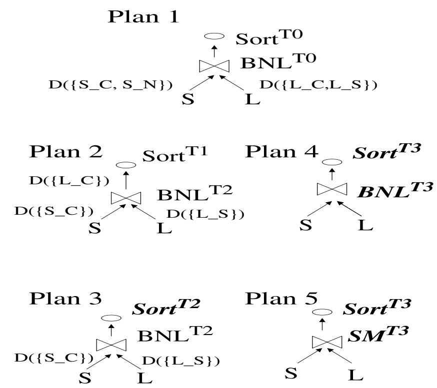



**Figure 2: The compression-aware query plans listed in Table 1. $S$ represents `Supplier` table and $L$ represents `Lineitem`. Tags $T_0 = \emptyset$, $T_1 = \{$`S_NAME`$\}$, $T_2 = \{$`S_NAME`, `L_COMMENT`$\}$, $T_3 = \{$`S_NAME`, `L_COMMENT`, `L_SHIPINSTRUCT`, `S_COMMENT`$\}$.**

We say that a query plan is *consistent* if the tagging of its nodes matches the actual changes of tags imposed by the operators in the tree. Let $v$ be a node in the tree with associated output relation $r$. Then the tags of $v$ satisfy the following properties: (1) $tag(v)$ is a subset of the attributes in $r$. (2) If $v$ is a leaf node (a base relation scan), then $tag(v)$ equals the set of attributes that is compressed in the associated base relation. (3) $tag(v) = \emptyset$ if $v$ is the root of the tree (the output of the query is decompressed). (4) If attribute $x$ is an attribute in $r$, but $x \notin tag(v)$, then $x \notin tag(v')$ for all ancestors $v'$ of $v$ in the query tree (Once an attribute is decompressed, it stays decompressed in the remainder of the query plan). (5) If attribute $x$ is in the output of node $v$ and $x$ is in one of the tags of $v$'s child nodes, then $x$ is in the tag of $v$ unless $v$ is a decompression operator $v = D_X$ that decompresses $x$ ($x \in X$).

Thus we can (informally) define the problem of compression-aware query optimization as the search for the least-cost consistent query plan. Note that the space of traditional query plans (with only empty tags) partitions the search space of compression-aware query plans into equivalence classes. We can map each compression-aware query plan to a traditional query plan by deleting all tags and all explicit decompression operators.

As an example, consider the query plans from Table 1, which were discussed for the query in Example 1 from Section 1. Figure 2 shows these query plans. The tags of each relational operator are specified as superscripts (the tags of the root nodes of each query plan are omitted because they must be the empty set). Transient operators are displayed in italic. Decompression operators are placed between relational operators.

Suppose there are $m$ attributes in the base tables and that we consider the space of consistent query plans with $n$

internal nodes. Any compression-aware plan $q$ is fully specified by the placement of decompression operators because the tagging of transient operators is determined by the tagging of its children. For each decompression operator, there are at most $n$ possible placements in the query plan; thus given a search space of size $s$ for a traditional optimizer, the size of the search space of the compression-aware query optimization problem is $O(s \cdot n^m)$. For a System R-style optimizer that searches only left-deep plans, the search space size of the compression-aware query optimization problem is thus $O(n \cdot 2^{n-1} \cdot n^m)$. In the remainder of this paper, we investigate how to make optimizers based on dynamic programming compression-aware.

## 3.2 Compression and Expensive Predicates

Our optimization problem bears some analogy to the work on optimizing queries with expensive predicates, such as user-defined procedures [7, 17]. The analogy is that a decompression operator can be thought of as an expensive predicate with 100% selectivity and a resulting increase in the tuple length. The traditional heuristic of pushing predicates down towards the leaves of the query plan does not apply when a predicate incurs a significant cost during query processing, since there is a tradeoff between the I/O savings by pushing down a predicate and the extra CPU processing of doing so. Similarly, the pulling up (delaying) of a decompression operator must weigh the I/O savings of keeping data compressed against the CPU overhead of transient decompression. Chaudhuri et al. propose a polynomial-time algorithm for placing expensive predicates in a query plan assuming that the cost formulas for relational operators are regular [7].

For example, suppose $r_1$ and $r_2$ are two input relations to a block-nested-loops join. Suppose $[r_1]$ and $[r_2]$ are the number of pages of $r_1$ and $r_2$, and B is the number of pages in the buffer pool. Then the cost of the join equals:

$$[r_1] \cdot [r_2]/B + [r_1] = [r_1] \cdot ([r_2]/B + 1) + 0$$

That is, the cost can be expressed in the form of $[r_1] \cdot a + b$, where a and b are constants irrelevant to the placement of predicates on $r_1$ (the placement of predicates will only change the input size $[r_1]$). As an expensive predicate $\sigma$ is applied on input $r_1$, the size of input becomes $[r_1']$ and the cost becomes

$$[r_1'] \cdot [r_2]/B + [r_1'] = [r_1'] \cdot ([r_2]/B + 1) + 0,$$

thus both factors $a$ and $b$ remain constant.

Now consider the cost of a block-nested-loops join operator in our problem of placing decompression operators, and assume that the join needs to decompress attribute $x$ in relation $r_1$. Let us consider the two cases of (1) explicitly decompressing the attribute $x$ before the join, and (2) executing the operator as an transient operator. In case (1), the input size of $r_1$ has increased to $[r_1^d]$ due to the decompression. Hence, the cost of the join is as follows:

$$[r_1^d] \cdot [r_2]/B + [r_1^d] = [r_1^d] \cdot ([r_2]/B + 1) + 0.$$

Assuming that our cost formulas would be regular, we can calculate the factors $a = ([r_2]/B + 1)$ and $b = 0$, both independent of $r_1$. Now assume that we "pull" the decompression operator on $A$ over the join. Join attribute $A$ needs to be decompressed transiently $n_1 \cdot n_2$ times, if there are $n_1$ tuples in $r_1$ and $n_2$ tuples in $r_2$. Assuming that the unit cost of decompression is a (usually small) constant $d$, the cost of the join becomes:

$$[r_1^c] \cdot [r_2]/B + [r_1^c] + n_1 \cdot n_2 \cdot d = \\ [r_1^c] \cdot ([r_2]/B + 1) + n_1 \cdot n_2 \cdot d.$$

Note that the size of $r_1$ has decreased to $[r_1^c]$. Comparing with the previous cost formula, we observe that the factor $a$ stayed constant, but $b$ changed from 0 to $n_1 \cdot n_2 \cdot d$. Thus the cost formulas for transient operators are no longer regular, and the polynomial algorithm proposed by Chaudhuri et al. cannot be not applied.

Note that if we exclude transient decompression, we can reduce our problem of placing decompression operators to the problem of expensive predicate placement. A full elaboration of this reduction is beyond the scope of this paper; in addition we showed in Section 1 that transient decompression results in query plans with very attractive costs in many cases. Thus we concentrate in the remainder of this section on the case where transient decompression is included.

## 3.3 Finding the Optimal Plan

In this section, we describe a query optimization algorithm based on dynamic programming which always finds the optimal plan within the space of all left-deep query plans. The following two observations serve as the basis of our dynamic programming algorithm:

- Critical attributes. We only need to decompress two types of attributes: (1) Attributes that are involved in operations that cannot process compressed data directly, and (2) Attributes that are required in the output of the query. We call such attributes *critical attributes*l they are the attributes we need to consider during query optimization.

- Pruning of suboptimal plans. Assume we are given two query plans $p$ and $q$ that represent the same select-project-join subexpressions of a given query and assume that $p$ and $q$ have the same physical properties (such as sort orders of the output relation). It is easy to see that if $tag(p) = tag(q)$ and $cost(p) < cost(q)$, then we can prune plan $q$ and all its extensions from the search space without compromising optimality.

Figure 3 shows the OPT Algorithm, our dynamic programming algorithm for finding the optimal plan. To simplify the presentation, we only consider joins; OPT can be easily extended to plans including other operators (e.g., a sort operator is just a degenerated case of join).[2]

The algorithm selects the join order, access methods, and join methods exactly in the same way as the system R optimizer. The main difference is that an optimal plan needs to be stored *for each distinct tag $t'$ of each intermediate join result $s$*. Note that the tagging of a plan determines the placement of decompression operators, and whether the operators in the query plan are transient operators or work on attributes that are already uncompressed. The algorithm enumerates bottom-up each possible join combination (lines 03-05), but at the same time also enumerates every possible tag that a query plan fragment can be labeled with (lines 06-10); the set of tagged plans is stored in `optPlan`.

[2] OPT is based on the optimal algorithm for placing expensive predicates by Chaudhuri and Shim [7].

**OPT Algorithm**
**Input:** A set of relations $r_1, \ldots, r_n$ to be joined
**Output:** The plan with the minimum cost

(01) Initialize each $r_1, ...r_n$'s tag with the subset of critical attributes compressed in $r_i, 1 \leq i \leq n$.
(02) **for** i := 2 to n **do**
(03)     **for** all $s \subseteq \{r_1, ...r_n\}$ s.t. $||s|| = i$ **do**
(04)         initialize array `bestPlan` to a dummy plan with infinite cost
(05)         **for** all $r_j$, $s_j$ s.t. $s = \{r_j\} \cup s_j$ and $\{r_j\} \cap s_j = \emptyset$ **do**
(06)             **for** all plans $p$ stored in `optPlan`$[s_j]$ **do**
(07)                 $t = tag(p) \cup tag(r_j)$
(08)                 **for** all $t' \subseteq t$ **do**
(09)                     $q := GenJoinPlan(p, r_j, t')$
(10)                     **if** $(cost(q) < cost(bestPlan[t']))$ $bestPlan[t'] := q$ **fi**
(11)         **endfor endfor endfor**
(12)         copy plans in bestPlan to $optPlan(s)$
(13) **endfor endfor**
(14) finalPlan := a dummy plan with infinite cost.
(15) **for** all plans $p \in$ `optPlan`$(\{r_1, ..., r_n\})$ **do**
(16)     **if** $(complete\_cost(p) < cost(finalPlan))$ finalPlan := completed plan of $p$ **fi endfor**
(17) return (finalPlan)

**Figure 3: OPT Algorithm for finding the optimal plan.**

In Line 06 we loop over the set of existing query plans for the join of $i - 1$ relations (called $s_j$) with different tags, creating the largest possible tag for the resulting relation in Line 07. Lines $08 - 10$ explore all possible ways to insert decompression operators before the join while maintaining the best possible plan for each possible output tag by calling subroutine `GenJoinPlan()` shown in Figure 4. Line 12 stores the currently best plans for each possible tag in `optPlan`, our "memory" for the dynamic programming. Lines $14 - 17$ select the final plan with overall lowest cost. Note that the tag of the final result of the query has to be the empty set, thus function `complete_cost()` potentially introduces decompression operators at the end of query plans whose tag is not the empty set.

In Example 1, there are four critical attributes: `L_COMMENT`, `L_SHIPINSTRUCT`, `S_NAME`, and `S_COMMENT`. Thus $2^4 = 16$ different output tags will be generated for the join node, and 16 differently tagged plans will be stored as inputs to the sort operator. Algorithm OPT returns Plan 5 as the optimal plan. Plan 4 will be pruned as we enumerate plans for the join node because the join fragment of the plan using transient decompression has the same tag ($T_3$) as the join fragment of Plan 5, but with higher cost (see Table 1).

Showing that Algorithm OPT finds the plan with the overall least cost within the space of left-deep plans is straightforward. We omit the details here due to space constraints. To study the complexity of OPT, let $m$ be the number of critical attributes and $n$ be the number of relations in the query. The System R optimizer has space complexity $O(2^n)$ and time complexity $O(n \cdot 2^{n-1})$. In the worst-case, over $m$ critical attributes, we may have to store as many as $2^m$ possible tags, each with an optimal subplan. Thus, the space complexity of the OPT algorithm is $O(2^n \cdot 2^m)$. At each step when we enumerate plans joining an existing plan $p$ with a relation $r_j$, if there are $\ell$ attributes compressed in $p$ and $r_j$, then there can be at most $2^\ell$ different output tags. In the worst case, the input has $m$ critical attributes in the input, thus there are at most $\binom{m}{\ell}$ cases when $\ell$ attributes are compressed in the input. Therefore, the total number $num$ of plans enumerated for each relational operator is:

$$num = \sum_{\ell=0}^{m} \binom{m}{\ell} 2^\ell = 3^m,$$

and the total time complexity of OPT is $O(n \cdot 2^{n-1} \cdot 3^m)$.

## 3.4 Heuristic Algorithms

The OPT algorithm can be easily integrated into an existing System R-style optimizer. However, the time and space complexity of the OPT algorithm increases by a factor that is exponential in $m$, the number of critical attributes. In this section, we propose two heuristic algorithms with sharply reduced space and time complexity.

### *3.4.1 Two-Step*

Our first algorithm allows for an easy integration with existing System R-style dynamic programming query optimizers. If we assume that the plan $p$ returned by a traditional query optimizer is structurally close to the optimal plan, then we can use $p$ to bootstrap a subsequent placement of decompression operators, thus transforming the traditional plan with empty tags into a fully tagged plan.

The Two-Step Algorithm first generates a traditional query plan $p$ with empty tags, and then in a second step executes a degenerated version of Algorithm OPT, which enumerates all possible taggings of $p$ while maintaining join order, join methods, and access methods from the first step. Due to the orthogonality of the two steps, the space complexity of Two-Step is $O(2^n + 2^m)$ and the time complexity of Two-Step is $O(n \cdot 2^{n-1} + n \cdot 3^m)$ In Example 1, a traditional optimizer returns a plan with a block-nested-loops join with `Supplier` as the outer table. Two-Step will then find the optimal decompression strategy for that plan, resulting in Plan 3 (see Figure 2).

Procedure GenJoinPlan($p$, $r_j$, $t'$)
**Input:** Query fragment $p$, base relation $r_j$, tag $t'$
**Output:** Physical algebra join plan $q$ with output tag $t'$ and suitable decompression operators
$q =$ a dummy plan with maximal cost.
**for** each possible join method $J$ **do**
    generate a plan $q'$ that joins $p$ and $r_j$ with $J$ as the join method
    add a decompression operator $d_1(tag(p) - t')$ to $q'$ between $p$ and the join node.
    add a decompression operator $d_2(tag(r_j) - t')$ to $q'$ between $r_j$ and the join node.
    $tag(q') = t'$
    **if** $cost(q') < cost(q)$ $q := q'$ **fi endfor**
return $q$

**Figure 4: Algorithm GenJoinPlan: Searches the space of physical join methods for a given plan.**

### 3.4.2 *Min-K*

The Min-K Algorithm is based on the OPT Algorithm from Section 3.3. It uses the following two heuristics to reduce the search space:

**Heuristic 1:** For an intermediate query plan fragment, instead of storing plans for each possible tagging of the output relation, we only store the $K$ plans with the least cost (change line 12 in Figure 3).

**Heuristic 2:** Instead of considering every possible tagging of the output relation of an operator (see loop lines 08–12 in Figure 3), we only consider the following two taggings $t_1$ and $t_2$: $t_1 = tag(p) \cup tag(r_j)$, and $t_2 = (tag(p) \cup tag(r_j)) \setminus X$, where $X$ is the minimal set of attributes that needs to be decompressed for the join method to perform the join on uncompressed attributes. Tagging $t_1$ makes the join operator a transient operator without inserting any decompression operators, whereas $t_2$ inserts decompression operators for attributes $x \in X$ that are required in the join. The intuition for this heuristic is that transient decompression usually helps for I/O intensive join operators, whereas for CPU intensive join operators, explicitly decompressing all join attributes can avoid prohibitive decompression overhead during join processing.

In the Min-K Algorithm, we store at most $K$ possible tags (and thus different plans) for each query plan operator. Thus, the space complexity of Min-K is $O(2^n \cdot K)$. At each step we enumerate plans joining an existing plan $p$ and a relation $r_j$. Since $r_j$ is a base table with one unique tag (the set of attributes compressed in $r_j$), there are at most $K$ possible tags in the inputs. Hence, there are at most $2K$ output tags and corresponding extended plan fragments enumerated. Therefore, the total time complexity is $O(n \cdot 2^{n-1} \cdot 2K)$.

As an example, assume that $K = 2$ in Example 1. For the join node, Min-K will enumerate two tags ($T_2$ and $T_3$ in Figure 2). Thus, two query plan fragments will be stored, one as the join fragment of Plan 5 with tag $T_3$, the other as the join fragment of Plan 3 with tag $T_2$. For the sort node, Min-K enumerates four possible tags: $T_2$, $T_1$ as $T_2 \setminus \{\texttt{L\_COMMENT}\}$, $T_3$, and $T_3 \setminus \{\texttt{L\_COMMENT}\}$. Min-K returns Plan 5 as the plan with least cost.

# 4. EXPERIMENTS

This section presents an experimental evaluation of (1) the new HDE compression strategy for string attributes and (2) our new query optimization algorithms. We start with a description of our experimental setup in Section 4.1. Section 4.2 presents a short evaluation of HDE, and Section 4.3 describes and evaluation of our algorithms for compression-aware query optimization.

## 4.1 Experimental Setup

We implemented the Hierarchical Dictionary Encoding (HDE) compression strategy proposed in Section 2 in the Predator database system [2]. We modified the query execution engine to run queries on compressed data. We made the following two changes to our cost model. First, we take the effect of compression on the length of records into account by estimating the tuple size of intermediate results based on the tags associated with each operator. Second, we added decompression time to the optimizer cost formulas. We experimentally tested our revised cost model and the results show that our cost model correctly preserves the relative order between different query plans as imposed by actual query execution times. As comparison to our proposed algorithms, we also implemented strategies for eager, lazy, and transient-only decompression. We refer to these three strategies as *baseline strategies*.

**Data:** We used TPC-H data scaled to 100MB both for our experiments on compression and on the optimization strategies. Clustered indices were built on primary keys and unclustered indices were built on foreign key attributes. Indices are not compressed. TPC-H data contains 8 tables and 61 attributes, 23 of which are string-valued. The string attributes account for about 60% of the total database size. We also used a 4MB of dataset with US census data, the adult data set [5] for experiments on compression strategies. The adult dataset contains a single table with 14 attributes, 8 of them string-valued, accounting for about 80% of the total database size.

**Execution Environment:** All experiments were run on an Intel Pentium III 550 Mhz PC with 512 MB RAM running Microsoft Windows 2000. The database was stored on a 17GM SCSI disk. The query execution time reported is the average of three executions. Predator implements index-nested-loops, block-nested-loops, and sort-merge join. We plan to implement hash join in the future and study its impact on our techniques.

**Queries:** We are interested in queries (particularly joins) involving compressed string-valued attributes. Although queries involving strings are quite common in practice (*e.g.*, "find people with the same last name" or "find papers written by the same author"), TPC-H queries only have foreign key joins on numerical attributes. Hence, we modified the

TPC-H queries by randomly adding secondary join conditions on string attributes as follows: First, we randomly pick two joinable relations that appear in a TPC-H query, then we randomly pick a string attribute from each relation, and finally, we choose a join condition from equality/prefix/suffix/substring matching of the two chosen attributes with equal probability. We also add a negation for the matching condition with 50% possibility and add the chosen string attributes to the final output with 50% probability. Unlike numerical attributes, string attributes usually have different domains and decompression will be necessary for evaluating the added join conditions on those two string attributes. We formed four query workloads, based on the number of join conditions we added. Workload W0, W1, W2, and W3 contain zero, one, two, and three join conditions on string attributes for each TPC-H query, respectively. Since TPC-H queries each contain a different number of join *tables* (as opposed to join conditions), we further divide each workload into three groups, containing 1-2 join tables, 3-4 join tables, and 5 or more join tables, respectively.

**Metrics:** Following Chaudhuri and Shim [7], we use the following two metrics to evaluate our query optimization strategies.

1. *Relative cost*, which measures the quality of plans. The relative cost equals the execution time of the plan returned by the optimization algorithm divided by the execution time of the optimal plan. A relative cost of 1 means the plan is optimal; the higher the relative cost, the worse the quality of the plan. We used OPT to determine the optimal plan (see Section 3).

2. *Multiplicative factor*, which measures the time complexity of optimization strategies. The multiplicative factor equals the number of plans enumerated by the algorithm being studied divided by the number of plans enumerated by a standard optimizer. A small factor implies a fast algorithm.

## 4.2 Effectiveness of HDE

To isolate the effect of compressing string attributes, we compress all numerical attributes in the data sets using techniques proposed by Westmann et al. [29], but vary the compression methods applied to string attributes. We compared the effectiveness of HDE with the following attribute-level compression strategies on string attributes:

1. Numerical-Only: We only compress numerical attributes.

2. Attribute-Dic: Dictionary compression on the whole attribute for string attributes with low cardinality. This is the strategy used by Westmann et al. [29].[3]

3. S-LZW: Semi-static LZW [28] on every string attribute. A tuple-level version of this method was employed by Iyer and Wilhite [19].

4. Word-Dic: Word-level dictionary compression on each string attribute. This is the technique used for information retrieval queries by Witten et al. [30].

[3] Westmann et al. also used NULL suppression (deleting ending blanks) on other string attributes, but Predator automatically stores long fixed length string attributes (`char(n)`) as variable length strings (`varchar(n)`) such that blanks are automatically deleted.

**Table 2: Comparison of different compression strategies on TPC-H data.**

| *Strategy* | *Data Size* | *Scan-ND* | *Scan-D* |
|---|---|---|---|
| Uncompressed | 100% | 100% | 100% |
| Numerical-Only | 91% | 92% | 94% |
| Attribute-Dic | 70% | 71% | 77% |
| S-LZW | 61% | 62% | 97% |
| Word-Dic | 56% | 58% | 84% |
| HDE | 50% | 51% | 77% |

Table 2 reports the results of applying various compression methods to the TPC-H database, normalized by the size of the uncompressed data. We also measured the time to scan all tables in the database without decompressing (referred to as *Scan-ND*), and the time to scan all tables with decompressing (referred to as *Scan-D*). We can make the following observations:

1. HDE achieves the best space savings. HDE beats Attribute-Dic and Word-Dic because it intelligently chooses the most appropriate level of dictionary compression rather than using a fixed level. Numerical-Only does not save much space because the majority of the attributes are strings that are not compressed in Numerical-Only. S-LZW uses more space than HDE because there are many short fixed lengthed string attributes with very low cardinality, which can be compressed to one or two byte fixed length integers by dictionary compression on the whole attribute level (the method selected by HDE). In contrast, S-LZW generates variable-length codes and needs to use extra bytes to store the length.

2. The I/O benefits are proportional to the space savings (although slightly lower); HDE achieves the best performance.

3. HDE also achieves the best balance between I/O savings and decompression overhead because it has the shortest time for scan with decompression. Numerical-Only has worse performance because the I/O savings are insignificant although decompression is very fast. S-LZW and Word-Dic have good I/O savings but the decompression overhead is too high. Only Attribute-Dic has similar performance to HDE.

Table 3 reports the results for the Adult data set; the observations are similar except that Attribute-Dic works equally well as HDE because all string attributes in the Adult data set have low cardinality. All compression strategies also added about 1 sec/MB compression overhead when the database was loaded, mainly due to the preprocessing pass over the data to build the dictionary. However, in a read-intensive environment, this penalty is offset by the improvement in query performance.

**Table 3: Comparison of different compression strategies on Adult data.**

| *Strategy* | *Data Size* | *Scan-ND* | *Scan-D* |
|---|---|---|---|
| Uncompressed | 100% | 100% | 100% |
| Numerical-Only | 88% | 90% | 93% |
| Attribute-Dic | 24% | 25% | 32% |
| S-LZW | 77% | 79% | 110% |
| Word-Dic | 55% | 57% | 94% |
| HDE | 24% | 25% | 32% |

## 4.3 Compression-Aware Optimization

This section evaluates the various optimization strategies by the quality of returned plans and the time complexity for optimization strategies. Our main findings are as follows:

1. The Min-K strategy with $K = 2$ is optimal for all the queries we tried at various buffer pool sizes, and the optimization cost is very low.

2. The Two-Step strategy sometimes finds near-optimal plans, especially when the buffer size is large.

3. The Transient-Only strategy is optimal only when queries do not contain join conditions on strings or the buffer pool size is large. Otherwise, it often produces inefficient plans.

**Average Quality of Plans:** We first fix the buffer pool size at 5 MB, but vary the query workload. Figures 5 (a), (b), and (c) report the average relative cost of different groups of queries as we vary the number of join conditions on string attributes. The x-axis shows the number of join conditions we added (each number corresponds to one of the four workloads). For the Min-K algorithms, we show two cases: $K = 1$ and $K = 2$. The relative costs for running the queries on uncompressed data and data where only the numerical attributes are compressed are also displayed. These costs are 2-10 times that of optimal plans over compressed string attributes, confirming the performance benefits from string attribute compression.

The Transient-Only strategy is best only when there are no join conditions on string attributes; numerical attributes are inexpensive to decompress, and it is usually better to decompress them transiently so that intermediate results remain compressed. Otherwise, the average relative cost of plans returned by the Transient-Only strategy is up to an order worse than OPT, demonstrating that transient operators must be applied selectively to be effective.

As the number of join conditions on string attributes increases, the performance of plans returned by all strategies except Min-2 and OPT deteriorates. The reason is that string attributes are expensive to decompress and the right decision of whether to keep string attributes compressed to save I/O makes more and more of a difference. The ranking of relative costs for plans returned by the different optimization strategies is as follows:

$$\text{OPT} \sim \text{Min-2} < \text{Min-1} \sim \text{Two-Step} < \text{Baseline strategies}$$

Among the baseline strategies, Lazy is significantly better than Eager (up to a factor of 3) because, using Lazy strategy, decompression does not occur until necessary and the intermediate results are smaller. The Transient-Only strategy achieves more I/O savings than Lazy by always keeping string attributes compressed. However, arbitrary use of transient operators may lead to prohibitive decompression overhead when relational operators require repeated access to compressed string attributes (*e.g.*, in a block-nested-loops join).

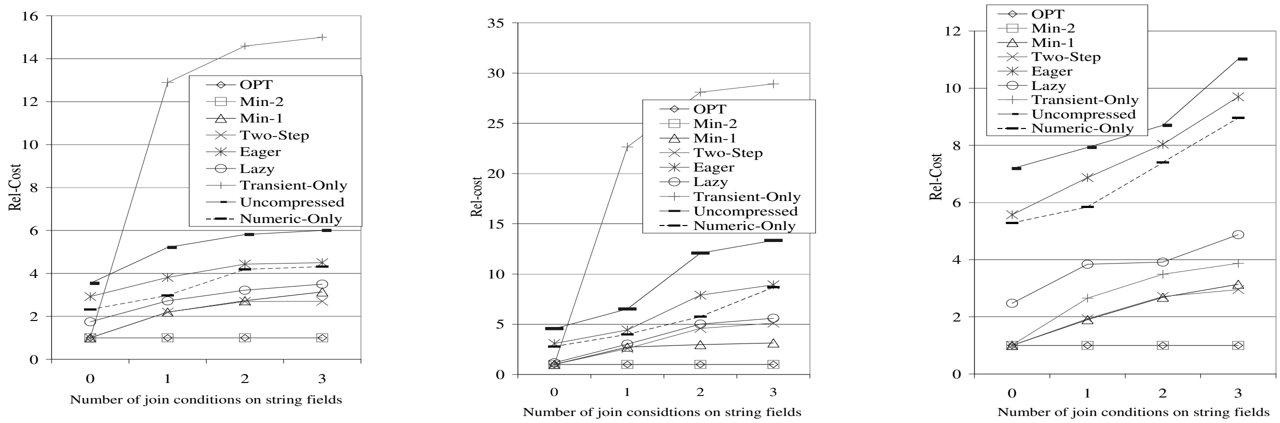
Min-2 always gave optimal plans in our experiments, whereas both Min-1 and Two-Step often gave suboptimal plans. Min-K considers up to two cases for each join operator: One is to transiently decompress all string attributes during the join and leave them compressed afterwards, such that relational operators later in the plan will get extra I/O benefits. The other is to decompress all those string attributes before the join to avoid the prohibitive overhead of repeated decompressions during the join. However, since the I/O savings for future relational operations cannot be decided locally, a local decision can be suboptimal. Hence, the choice of $K$ is crucial: If $K = 2$, optimal plans for both cases are kept, while if $K = 1$, only the local minimum is kept, allowing globally suboptimal results. Similarly, the Two-Step heuristic is less effective than Min-2 because the decision of join order and join methods is made independently of the decision on the decompression strategy.

In summary, the optimizer has to combine the search for optimal plans with the decision of how and when to decompress (as in OPT and Min-2). Using straightforward, simple heuristics such as Eager, Lazy, and Transient-Only, or making the optimization too local such as in Min-1 can lead to significantly worse performance.

**Distribution of Query Performance:** We examined the performance distribution of individual queries in the workload W2 using a 5 MB buffer pool size (the results for other workloads are similar). We found that Min-2 gave an optimal plan for all 22 queries; Two-Step gave an optimal plan for 11 queries, but had 7 queries with relative cost greater than 5; Min-1 had 5 queries with an optimal plan but 13 queries with relative cost greater than 5; Eager was the only strategy that never found the optimal plan; Transient-Only was very unstable, with optimal plans for 7 queries but relative cost over 20 for 6 queries.

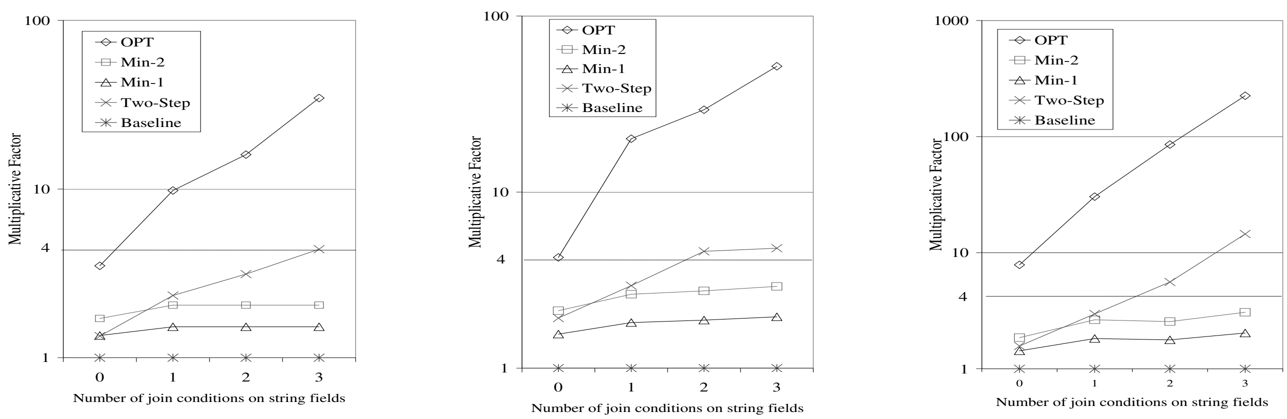
In practice, it is usually suffices to return a "good" plan instead of the optimal plan. We plot the number of queries with relative cost lower than 2 (*i.e.*, with cost within twice of the optimal cost) for various strategies in Figure 7 (a). Again, OPT and Min-K always return good plans. The number of good plans returned by other strategies decreases as the number of join conditions on string attributes increases. The total number of good plans over all workloads is reported in Figure 7 (b).

**Number of Plans Enumerated:** Figures 6 (a), (b), and (c) report the average multiplicative factor of different groups of queries as we vary the number of join conditions on string attributes. The three baseline strategies have a multiplicative factor of 1 because no additional plan is enumerated after standard optimization. The multiplicative factor of OPT increases rapidly as the number of join conditions on string attributes and the number of join tables increases, and soon leads to prohibitive optimization overhead. Our proposed heuristic algorithms reduce the search space greatly. Note that Min-2 never enumerates more than four times as many plans as the standard optimizer, regardless of the number of join conditions added or of the number




(a) Queries with 1-2 join tables (b) Queries with 3-4 join tables (c) Queries with $\geq 5$ join tables

**Figure 5: Relative cost of various strategies vs. number of join conditions on strings.**




(a) Queries with 1-2 join tables (b) Queries with 3-4 join tables (c) Queries with $\geq 5$ join tables

**Figure 6: Multiplicative factor of various strategies vs. number of join conditions on strings.**

of join tables. Given that the plans returned by Min-2 are close to optimal, Min-2 appears to be the most attractive strategy.

**Effect of Buffer Pool Size:** The size of the buffer pool is an important determinant of query performance. We ran the experiments with buffer pool sizes 5, 20, and 100 MB. Since the trends we observed for the different workloads were similar, we report the results from workload W2 only. Figures 8 (a) and (b) show the average relative costs of different query groups using different strategies against varying buffer pool size (the results for the query group with 3-4 join tables were similar to that with 1-2 join tables and are omitted).

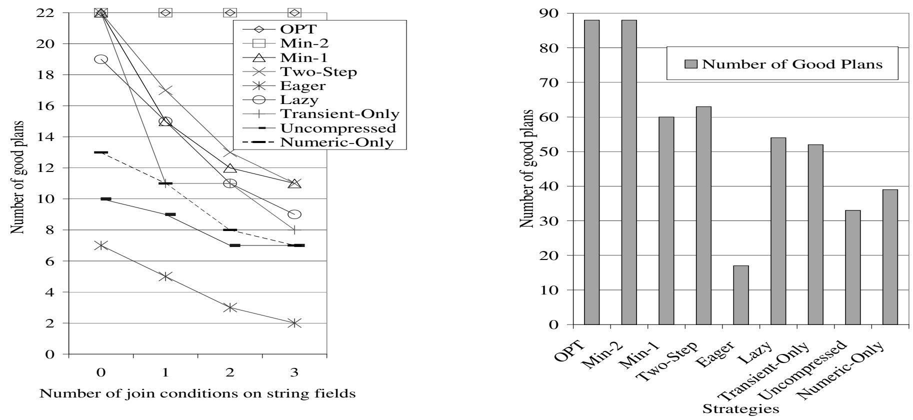
Not surprisingly, the performance benefits resulting from string compression decrease as the buffer pool becomes larger, since compression has less impact on the amount of data that can be brought into the buffer. Nonetheless, the savings from compression are still substantial (ranging from 70-300%) even when the whole database fits into the buffer pool (100 MB), due to the CPU savings of transient decompression (*e.g.*, fewer memory copies). Also, as reported by Lehman et al. [20], when the whole database fits into the buffer pool, the choice of join methods becomes simpler because CPU cost becomes the only dominant factor. Hence, the plan returned by a traditional optimizer becomes good enough, thus Two-Step often finds optimal plans. Moreover, for this buffer size transient-decompression seems to be a good choice for most queries, and thus Transient-Only strategy is close to optimal as well.

# 5. RELATED WORK

Data compression has been a very popular topic in the research literature, and there is a copious amount of work on this subject. Well known methods include Huffman coding [18], arithmetic coding [31], and Lempel-Ziv [32, 33].

Most existing work on database compression focused on designing new compression methods [4, 8, 10, 11, 12, 14, 19, 22, 23, 24, 27]. However, despite the abundance of string-valued attributes in databases, most existing work has focused on compressing numerical attributes.

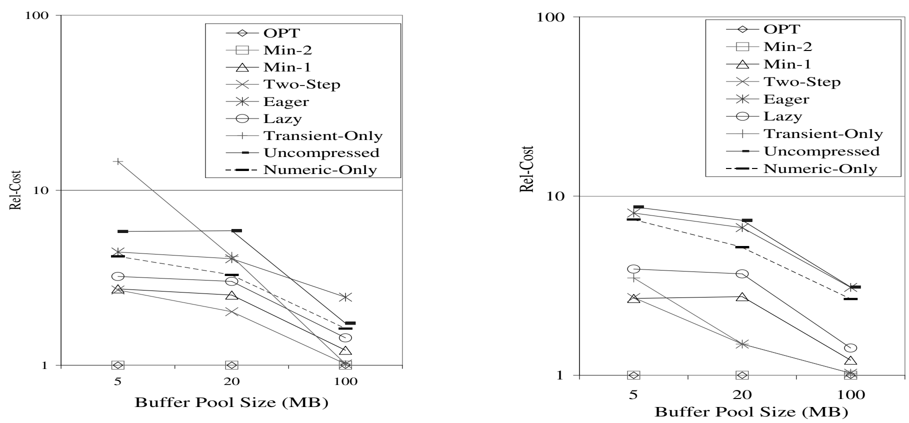
Recently, there has been a resurgence of interest on employing compression techniques to improve performance in a database. Greer uses simple compression techniques in the Daytona database system, but does not consider how




(a) Number of good plans vs. number of join conditions

(b) Total number of good plans

**Figure 7: Distribution of query performance.**




(a) Queries with 1-2 join tables

(b) Queries with $\geq 5$ join tables

**Figure 8: Rel-Cost of various optimization strategies varying buffer pool size.**

to exploit this in the query optimizer [16]. Goldstein et al. propose attribute-level offset encoding where the data is only decompressed lazily as needed [12, 13]; little consideration is paid to query optimization other than a modified cost model. Westmann et al. propose a collection of lightweight, attribute-level compression methods and shows how to modify the query execution engine [29]; the authors briefly mention that the cost model should be modified and the issue of whether to compress intermediate results, but no query optimization algorithm was proposed. Boncz et al. [6] consider some attribute-level compression techniques (dictionary-based encoding) for improving join performance in a main-memory database; the focus of this work is on the design of new join algorithms, but without any consideration of query optimization. Li et al. [21] consider aggregation algorithms in a compressed Multi-dimensional OLAP databases; however, they have not addressed querying more general compressed relational databases. The only work we are aware of which considers query optimization over compressed data is by Amer-Yahia and Johnson [3], but they focus on bitmaps. Finally, as discussed in Section 3, there is some similarity between our work and that of Chaudhuri and Shim [7] and Hellerstein et al. [17], which considers the optimization of queries over expensive predicates. However, for the reasons put forth in Section 3, their algorithms do not apply in our case.

# 6. CONCLUSIONS

In this paper, we studied the use of compression to improve database performance. We observed that compressing string attributes is important for query performance. Due to the heterogeneous nature of string attributes, a single compression method is inferior to our Hierarchical Dictionary Encoding, a comprehensive strategy that chooses the most effective encoding level for each string attribute.

In addition, we observed that the placement of string decompression in a query plan is crucial for query performance. A traditional optimizer enhanced with a cost model that takes both I/O benefits of compression and the CPU overhead of decompression into account, does not necessarily achieve good plans. (The Two-Step algorithm is an instantiation of this approach.) We proposed two new query optimization algorithm, OPT and Min-K, that combine the search for optimal plans with the decision of when and where to decompress. Our experiments show that the combination of effective compression methods and compression-aware query optimization is crucial for query performance – usage of our compression methods and optimization algorithms achieves up to an order improvement in query performance over existing techniques. The significant gains in performance suggests that a compressed database system should have the query optimizer modified for better performance.

There are several interesting future research directions. First, it would be interesting to study how caching of intermediate (decompressed) results can reduce the overhead of transient decompression. Second, we plan to study how our compression techniques can handle updates.

**Acknowledgments.** We thank Praveen Seshadri, Philippe Bonnet, Divesh Srivastava, and Tobias Mayr for useful discussions.

## 7. REFERENCES


[1] Transaction processing performance council. *TPC-H benchmark, http://www.tpc.org*, 1999.

[2] Predator DMBS. *http://www.cs.cornell.edu/database/predator, Cornell Univ., Computer Science Dept.*, 2000.

[3] S. Amer-Yahia and T. Johnson. Optimizing queries on compressed bitmaps. In *Proc. of VLDB*, pages 329–338, 2000.

[4] G. Antoshenkov, D. B. Lomet, and J. Murray. Order preserving compression. In *Proc. of ICDE*, pages 655–663, 1996.

[5] C. Blake and C. Merz. UCI repository of machine learning databases. *http://www.ics.uci.edu/∼mlearn/MLRepository.html*, 1998.

[6] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proc. of VLDB*, pages 54–65, 1999.

[7] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *TODS*, 24(2):177–228, 1999.

[8] Z. Chen and P. Seshadri. An algebraic compression framework for query results. In *Proc. of ICDE*, pages 177 – 188, 2000.

[9] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Trans. on Communications COM-32(4)*, pages 396–402, April 1984.

[10] G. Cormack. Data compression in a database system. *Commnications of the ACM*, pages 1336–1342, Dec. 1985.

[11] S. J. Eggers, F. Olken, and A. Shoshani. A compression technique for large statistical data-bases. In *Proc. of VLDB*, pages 424–434, 1981.

[12] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In *Proc. of ICDE*, pages 370–379, 1998.

[13] J. Goldstein, R. Ramakrishnan, and U. Shaft. Squeezing the most out of relational database systems. In *Proc. of ICDE*, page 81, 2000.

[14] G. Graefe. Options in physical databases. *SIGMOD Record 22(3)*, pages 76–83, Sept. 1993.

[15] G. Graefe and L. Shapiro. Data compression and database performance. In *ACM/IEEE-CS Symp. On Applied Computing*, pages 22–27, April 1991.

[16] R. Greer. Daytona and the fourth-generation language cymbal. In *Proc. of SIGMOD*, pages 525–526, 1999.

[17] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proc. of SIGMOD*, pages 267–276, 1993.

[18] D. Huffman. A method for the construction of minimum-redundanc codes. In *Proc. IRE, 40(9)*, pages 1098–1101, Sept. 1952.

[19] B. R. Iyer and D. Wilhite. Data compression support in databases. In *Proc. of VLDB*, pages 695–704, 1994.

[20] T. J. Lehman and M. J. Carey. Query processing in main memory database management systems. In *Proc. of SIGMOD,*, pages 239–250, 1986.

[21] J. Li, D. Rotem, and J. Srivastava. Aggregation algorithms for very large compressed data warehouses. In *Proc. of VLDB*, pages 651–662, 1999.

[22] H. Liefke and D. Suciu. Xmill: An efficient compressor for XML data. In *Proc. of SIGMOD*, pages 153–164, 2000.

[23] W. K. Ng and C. V. Ravishankar. Relational database compression using augmented vector quantization. In *Proc. of ICDE*, pages 540–549, 1995.

[24] G. Ray, J. R. Harista, and S. Seshadri. Database compression: A performance enhancement tool. In *the 7th Int'l Conf. on Management of Data (COMAD)*, Pune, India, 1995.

[25] M. A. Roth and S. J. V. Horn. Database compression. *SIGMOD Record*, 22(3):31–39, 1993.

[26] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. of SIGMOD*, pages 23–34, 1979.

[27] D. Severance. A practitioner's guide to database compression. *Information Systems 8(1)*, pages 51–62, 1983.

[28] T. Welch. A technique for high-performance data compression. *IEEE Computer 17(6)*, pages 8–19, June 1984.

[29] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record 29(3)*, Sept. 2000.

[30] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Giga Bytes - Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, Inc, 1999.

[31] I. H. Witten, R. Neal, and J. Cleary. Arithmetic coding for data compression. *Communications of the ACM, 30(6)*, pages 520–540, June 1987.

[32] J. Ziv and A. Lempel. On the complexity of finite sequences. *IEEE Trans. on Information Theory, 22(1)*, pages 75–81, 1976.

[33] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. on Information Theory, 22(1)*, pages 337–343, 1977.