Query-Aware Compression of Join Results

Christopher M. Mullins
University of Hawai‘i at Mānoa
Honolulu, HI, USA
cmmullin@hawaii.edu

Lipyeow Lim
University of Hawai‘i at Mānoa
Honolulu, HI, USA
lipyeow@hawaii.edu

Christian A. Lang
Acelot Inc.
Santa Barbara, CA, USA
christian.a.lang@gmail.com

ABSTRACT

Client-server database query processing has become an important paradigm in many data processing applications today. In cloud-based data services, for example, queries over structured data are sent to cloud-based servers for processing and the results relayed back to the client devices. Network bandwidth between client devices and cloud-based servers is often a limited resource and the use of data compression to reduce the amount of query result data transmitted would not only conserve bandwidth but also help with battery lifetime in the case of mobile client devices. For query result compression, current data compression methods do not exploit redundancy information that can be inferred from the query structure itself for greater compression. In this paper we propose a novel query-aware compression method for compressing query results sent from database servers to client applications. Our method is based on two key ideas. We exploit redundancy information obtained from the query plan and possibly from the database schema to achieve better compression than standard non-query aware compressors. We use a collection of memory-limited dictionaries to encode attribute values in a lightweight and efficient manner. Each dictionary in the collection of dictionaries are also dynamically resized to adapt to changing temporal access characteristics. We evaluated our method empirically using the TPC-H benchmark show that this technique is effective especially when used in conjunction with standard compressors. Our results show that compression ratios of up to twice that of gzip are possible.

Categories and Subject Descriptors

E.4 [**Coding and information theory**]: Data compaction and compression; H.2.4 [**Database Management**]: Systems—*Query Processing*

General Terms

Algorithms

1. INTRODUCTION

Client-server database queries form the backbone of many data-intensive applications ranging from cloud-based applications and services to parallel/distributed data warehousing systems. In cloud-based services, queries may be sent to cloud-based database servers from mobile client devices and the results relayed back to those devices. Such mobile client devices are often limited by network bandwidth and battery life. Any effort to reduce the amount of data transmitted across the network would not only conserve bandwidth but also help with the battery life, because network communications typically consume significant amounts of energy. In parallel data warehousing systems, user queries are partitioned into sub-queries to be executed on worker nodes. Results from worker nodes are either transmitted to other worker nodes for further processing or consolidated to a coordinator to be returned to the user. A reduction in the amount of data transmitted between worker nodes, say, via data compression, would have a significant impact on query latency. Data compression methods and their applications to data communications and database processing is not new. A key insight of this paper is that in the case of client-server style database query processing, the queries themselves offer clues to the redundancy structure of the query result set that can be exploited for greater compression of the query results. That said, not all query results will exhibit the kind of redundancy that can be exploited for compression. For such query results, standard data compression methods suffices, but for query results that do exhibit sufficient redundancy structure, greater compression can be achieved.

In this paper, we propose a novel query-aware (QA) compression method for compressing join query results sent from database servers to client applications. Our method is based on two key ideas. First, we exploit redundancy information obtained from the query plan and possibly from the database schema as well. Second, we use a collection of nested memory-limited dictionaries to encode attribute values efficiently.

Consider the result set of the SQL query illustrated in Fig. 1(e) and Fig. 1(d) respectively. The tuples in the result set contain much redundancy that would be amenable to compression: in the columns `(A,B)`, the values `(a1,b1)` and `(a2,b1)` are repeated twice; in the columns `(B,D)`, the value `(b1,d1)` is repeated four times; in the columns `(B,C)`, the value `(b1,c1)` is repeated twice. Standard column-wise or row-wise dictionary compression techniques would not be able to capture this type of redundancy, because they would have no knowledge that the result set tuples are gen-

A	B
a1	b1
a2	b1
a1	b2
a2	b3

(a) R(A,B)

B	C
b1	c1
b1	c2
b2	c3
b4	c3

(b) S(B,C)

B	D
b1	d1
b2	d2
b4	d3
b5	d3

(c) Q(B,D)

```
SELECT   R.A, R.B, S.C, Q.D
FROM     R, S, Q
WHERE    R.B=S.B
         AND S.B=Q.B
         AND Q.B < b5
```

(d) Query

A	B	C	D
a1	b1	c1	d1
a1	b1	c2	d1
a2	b1	c1	d1
a2	b1	c2	d1
a1	b2	c3	d2

(e) Query Result

Figure 1: The base relations, query and result set used in the running example.

erated from a join of the three relations `R(A,B)`, `S(B,C)`, and `Q(B,D)`.

To infer this type of redundancy from the result set tuples themselves is a combinatorial problem and clearly infeasible in practice. Instead, our proposed compression scheme obtains the knowledge of this redundancy from the query itself in the form of a join tree and uses the join tree to compress the result set using a hierarchy of dictionaries. In the worst case, the size of the hierarchy of dictionaries may be in the order of the size of the base relations used in the query. Since the decompressor at the client needs to materialize the hierarchy of dictionaries for decompression, the memory requirements of the decompressor may be impractical for lightweight clients. To address issue, we use a memory-limited dictionary data structure for our compression and decompression algorithms.

Note that the proposed compression method is not equivalent to sending the base relations of the query (after any selection and projection operations) to the client and performing the joins and any additional projections at the client. Such an approach would require the client to materialize all the base relations (albeit after any selections and projections) and actually evaluate the join conditions. The proposed compression method does not evaluate the join conditions at the client : the query is fully evaluated by the database engine and the result set is compressed by the proposed method using a hierarchy of dictionaries. Moreover, through the use of our memory-limited dictionary data structure, the proposed method does not need to materialize the base relations in their entirety at both the compressor and the decompressor.

Our QA compression method is applicable in any applications where database servers need to transmit query result sets over some network to client applications. The compressed result sets would require less bandwidth for transmission and hence improve query latency at the client as well. Hence, we envision that the method can be incorporated into the ODBC/JDBC layers and the distributed relational database architecture (DRDA) protocol as well. In the context of parallel and distributed databases, our method can be used to compress results of sub-queries that are executed at (remote) database nodes. Our method is sufficiently general and can be applied on an arbitrary relation if a lossless-join decomposition of the relation is available.

Contributions.

- We designed a novel query-aware (QA) compression method for query result set by exploiting redundancies obtained from the query. To the best of our knowledge, the concept of query-aware compression is new.
- We designed a memory-limited dictionary data structure that bounds the memory requirement upfront thereby enabling a more space-efficient (de-)compression algorithms.
- We evaluated our compression method experimentally and demonstrated its effectiveness and efficiency.

The rest of the paper is organized as follows. Related Work is discussed in Section 2. We describe our algorithm in detail in Section 3. Experiments testing the efficacy of QA compression are given in Section 4. Lastly, conclusions are given in Section 5.

2. RELATED WORK

Several compression schemes have been devised in the past. Some are generic, some are tailored to specific data structures, and some are optimized for streaming data. We will discuss examples for each category in the following.

The most widely used compression schemes are based on Huffman codes [15], arithmetic coding [25], and dictionary encoding such as the algorithms devised by Lempel and Ziv [26, 27]. They use a dictionary to encode arbitrarily long fragments of data from the input stream. More frequent fragments are typically encoded with fewer bytes via Huffman encoding for example. Since join results exhibit complex nested repeating patterns, the dictionary may become very large (and thus decompression costly) or only the base tuples are encoded (thus losing compression potential).

More recently, grammar-based compression has received some attention. The input data is replaced by a small context-free grammar (CFG) that describes the generation of the input. Since finding the smallest CFG is an NP-hard problem, various heuristics have been proposed. Sakamoto et al. [22] propose a linear-time algorithm that outperforms dictionary based algorithms for highly repetitive inputs.

Applying data compression techniques to compress relational tables and indexes in a relational database system is a well studied problem. Cormack [7] proposes an algorithm based on Huffman codes designed for databases. Roth and Van Horn [21] propose a number of ways to apply compression algorithms in a relational database context. Ng and Ravishankar [18] provide an algorithm designed to work well with local decompression, allowing for common database operations to be performed on compressed data. Ray et. al [20] argue that improvements in query processing performance due to the use of compressing databases make it attractive even when storage capacity is not a concern, and propose an attribute-level compression algorithm. Shapiro and Graefe [12] discuss performance improvements gained by leaving databases compressed during query processing.

Most of the previous work apply compression to either pages or tuples to reduce the storage requirement of relational data and the associated indexes. Recent database compression research [1, 14, 19] apply compression to database tables with the objective of improving query processing. Database query processing is often IO-bound, so any technique that reduces the size of data that needs to be read or written to disk potentially improves performance. Much attention has been given to compression inside the storage layer of database systems. For example, Antoshenkov et al. [3] propose compression for string data within tables for faster query processing. Goldstein et al. [11] and Bhattacharjee et al. [5] propose compression schemes for low cardinality fields and database indexes. Most commercial database products provide storage layer compression. However, compression for query result transmission is mostly handled outside the database system.

All compression schemes described so far focus mainly on non-streaming data or offline compression. Compression for streaming data has traditionally focused on audio and video data. A prominent example is the MPEG-1 Layer 3 [16] for audio streaming. Since compression for these applications is typically lossy, these algorithms cannot be applied to our use case. One exception is the work by Maruyama et al. [17] that improve upon Sakamoto's grammar-based compression scheme by transforming the algorithm into a true online algorithm that does not require knowledge of the entire data stream in advance. It may be possible to describe join results by a CFG and thereby employ grammar-based compression schemes. However, due to their heuristic nature, no strong time and space usage guarantees can be given.

ODBC [24] and JDBC [13] provide limited query result compression capabilities. They typically allow the user to specify one of the offline compression schemes (such as GZIP [9]) which is then applied to each record or a set of records before transmission. Chen and Seshadri [6] propose an algebraic compression framework for query results in the context of low-bandwith and low-memory query clients. They analyze the data distribution of query results to detect patterns and functional dependencies that lend themselves to specific compression techniques. They then derive a "compression plan" that may consist of the application of various compression algorithms to different parts of the query result. The important difference to our proposed scheme is the fact that we exploit the knowledge of the query plan that led to the result set.

A somewhat related aspect of SQL result compression is the order in which fields of database records are stored. Traditionally, database systems employ a row-oriented storage scheme in which fields of one row are followed by fields of the next row. Recently, column-oriented storage and query processing has been proposed [2] in which all values of a single column are stored, followed by all values of the next column. Besides accelerating certain query types (e.g., decision support queries), this way of storing records can also help compression since values of the same column are of the same type and are more likely to exhibit repetitions [1]. Our proposed join result compression scheme can be applied to column-oriented databases equally well.

As opposed to compressing data for the purpose of reducing storage requirements or for improving query processing performance, the compression techniques proposed in this paper specifically address the problem of transmission of join query result sets. Hence the proposed techniques exploit redundancy information from the join query plan. To the best of our knowledge, no prior work has studied this problem.

The most closely related work is Goh et al. [10] where association rule mining algorithms are used to obtain association rules from the database tables that are then used to compress the data. Association rules are quite different from functional dependencies and the join query tree that are exploited in our compression method.

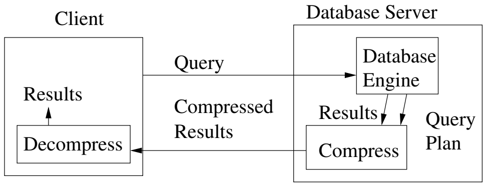


Figure 2: QA compression in a client-server framework.

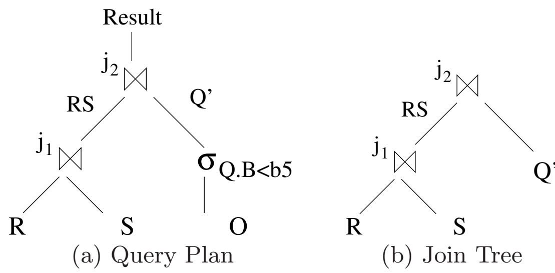


Figure 3: Query execution plan for the SQL query in Fig. 1(d) and the corresponding join tree.

3. QUERY-AWARE (QA) COMPRESSION

3.1 Overview

A schematic diagram of the compression and decompression process is outlined in Fig. 2. The client sends a query to the database server. The database engine processes the query and sends both the result set and the query execution plan (defined in the next paragraph) to the proposed compressor. The compressor compresses the result set using the redundancy information obtained from the query execution plan and transmits the compressed bit stream over the network to the client. The client receives compressed bit stream and forwards it to the decompressor for decompression. The decompressed result set is then made available to the client applications. The proposed compression algorithm is symmetric; hence the decompression algorithm mimics the state of the compression algorithm to decode the compressed bit stream. We defined several terms next before giving an overview of the proposed method using our running example.

A *query execution plan* (or simply query plan) is a tree of relational algebra (RA) operators. The root node denote the result set of the query, the leaf nodes represents the relations that the query operates on. Internal nodes represent standard RA operators ($\sigma, \pi, \bowtie$). Edges represent tuple flows (from bottom up) between nodes. The proposed method can also apply to extended RA operators such as projections that allow expressions and the group-by operator as long as these operators can be pushed down so as to minimize the number of join operators beneath those operators. Without loss of generality query plans are assumed to have selection, projection and extended RA operators pushed down as close to the base relations as possible[1].

[1]A standard operator push-down rewriting algorithm can always be applied to ensure that condition.

A *join tree* is a binary tree where the internal nodes denote join operators and the leaf nodes represent relations that could be actual relations or views in the database or temporary results of some relational operator. A join tree for the result set of a query is typically obtained from the query plan of the query by merging each subtree of the query plan that do not contain the join operator into a (temporary) relation. For query plans that contain extended RA operators, join operators that are beneath the extended RA operators in the query plan may not participate in the join tree. Fig. 3 shows a query plan and the corresponding join tree. For the purpose of this paper, there is no loss of generality if we consider query plans where selection and projection operators are pushed down close to the leaf nodes.

Our proposed compression method will encode each row of the query result set using a set of nested dictionaries – one dictionary for each non-root node of the join tree. In addition each column of the result set will be encoded using a column-based dictionary. Since the decompressor is symmetric, the join tree and the schema of the leaf relations are first serialized and sent to the decompressor before sending any encoding of the result set using the nested dictionaries. The dictionaries at the decompressor are then kept in sync with those at the compressor as entries are received and inserted into the dictionaries in exactly the same way that the compressor is maintaining its dictionaries. For the join tree in Fig. 3(b), we will initialize four empty dictionaries $\{D(R), D(S), D(Q'), D(j_1)\}$ for each non-root node, and four empty column dictionaries $\{D(A), D(B), D(C), D(D)\}$ for each column in the result set. As new entries are added to a dictionary, these entries and the dictionary identifier are also sent to the decompressor, so that the dictionaries remain in sync at the decompressor. A dictionary entry flag is sent to denote that the value sent is a dictionary entry and not an encoded tuple fragment. Our compression algorithm loops through each row of the result set and encodes the row using the join tree in a bottom up (depth first search order) sequence. Each field of a row is encoded using the column dictionaries and the fragment to be encoded by each join tree node is constructed either from the encoded columns (for the leaf nodes) or recursively from the encoded fragments of the child nodes. For non-root nodes, the constructed fragment is further encoded using the dictionary associated with that node. For the root node, the constructed fragment is transmitted without further encoding. For the join tree in Fig. 3(b), we will encode each row in the result set using the following sequence of dictionaries $\langle D(A), D(B), D(R), D(C), D(S), D(j_1), D(D), D(Q')\rangle$.

3.2 Compression

The proposed QA compression algorithm is outlined in Algorithm 1. The algorithm takes as input a join tree T and the result set W produced by the database query execution engine. The result set W is a set of rows whose schema conforms to the SELECT clause of the SQL query. The compressor first serializes the join tree and sends it to the decompressor. The schemas of the relations associated with the leaf nodes of the join tree are also transmitted to the decompressor. We assume that the schemas are logically part of the join tree that was supplied as input to the algorithm. The join tree and schemas are then used to initialize the set of dictionaries – one for each non-root node of the join tree. We denote the dictionary associated with a non-root node v in the join tree as $D(v)$ and the set of dictionaries for the entire join tree as $\mathcal{D}$. Each of these dictionaries will be used to encode the relational tuples associated with each non-root node of the join tree. For leaf nodes of the join tree, additional dictionaries are initialized for encoding each column of the relation associated with the leaf node. We use $D(c)$ to denote the (column) dictionary for a column c (assuming that column identifiers are unique across all the tables in the query). We will often refer to the relation associated with a node in the join tree using just the node identifier. Once the dictionaries are initialized, the compressor loops through each tuple in the result set and compresses each tuple using the join tree and the associated dictionaries.

Algorithm 1 COMPRESS(T, W)

Input: Join tree T, Result set W
Output: Sends compressed result set to decompressor

```
1: send join tree $T$ and schema of leaf nodes
2: $\mathcal{D} \leftarrow \emptyset$
3: for all non-root node $v \in T$ do
4:    initialize dictionary $D(v)$
5:    $\mathcal{D} \leftarrow \mathcal{D} \cup D(v)$
6:    if $v$ is a leaf node then
7:       for all columns $c$ of relation $v$ do
8:          initialize dictionary $D(c)$
9:          $\mathcal{D} \leftarrow \mathcal{D} \cup D(c)$
10: for all row $r \in W$ do
11:    send COMPRESSROW($root(T), \mathcal{D}, r$)
```

Algorithm 2 COMPRESSROW($t, \mathcal{D}, r$)

Input: Join tree node t, collection of dictionaries $\mathcal{D}$, a row r to be compressed
Output: Sends dictionary entries to decompressor and returns compressed row

```
1: $rowcode \leftarrow \epsilon$
2: if $t$ is a leaf then
3:    for all columns $c$ of relation $t$ do
4:       $code \leftarrow$ DICTLOOKUP( $r[c], D(c)$)
5:       if $code = \epsilon$ then
6:          $code \leftarrow$ DICTADD($D(c), r[c]$)
7:          send $DENTRY, c, r[c]$,
8:       $rowcode \leftarrow rowcode \cdot code$
9: else
10:    $leftcode \leftarrow$ COMPRESSROW($t.left, \mathcal{D}, r$)
11:    $rightcode \leftarrow$ COMPRESSROW($t.right, \mathcal{D}, r$)
12:    $rowcode \leftarrow leftcode \cdot rightcode$
13: if $t$ is non-root then
14:    $code \leftarrow$ DICTLOOKUP( $rowcode, D(t)$)
15:    if $code = \epsilon$ then
16:       $code \leftarrow$ DICTADD($D(t), rowcode$)
17:       send $DENTRY, t, rowcode$,
18:    $rowcode \leftarrow code$
19: return $rowcode$
```

Note that the dictionaries at the decompressor are maintained by receiving entries interlaced with encoded tuple data sent by the compressor. This allows the QA compression and decompression algorithm to operate in a streaming fashion.

Compressing One Row. The steps to compress one row of

DE, D(A), a1,	DE, D(C), c2,
DE, D(B), b1,	DE, D(S), 1,
DE, D(R), 0, 0,	DE, D(j_1), 0, 1,
DE, D(C), c1,	TF, 1, 0
DE, D(S), 0,	
DE, D(j_1), 0, 0,	
DE, D(D), d1,	
DE, D(Q′), 0	
TF, 0, 0	

Figure 4: Logical encoding of the first two rows of the result set in Fig. 1(e) using the join tree in Fig. 3(b). The symbol 'DE' denotes the dictionary entry flag, 'TF' the tuple fragment flag, and '$D(i)$' the dictionary identifier for join tree node i.

the result set are outlined in Algorithm 2. CompressRow is a recursive procedure on the join tree. The algorithm takes as input a node in the join tree, the set of dictionaries, the row to be compressed, and returns the compressed representation of the input row encoded according to the input join tree and the associated dictionaries. Let the input row or tuple r be a sequence of values and let $r[t]$ denote the sub-sequence of r associated with the join tree node t. Similarly, let $r[c]$ denote the sub-sequence of r associated with the column c (of some leaf node). The base case of the recursion is when the input join tree node is a leaf node (Line 1-8). The algorithm attempts to compress the input row column by column using the column dictionaries. This column compressed row is further compressed using the dictionary $D(t)$ for the current input node t (Line 13-18). In general dictionary-based encoding, whenever a new value that is not in the dictionary is encountered, the new value is added to the dictionary that also assigns a code word for that new value. The decompressor needs to update its copy of the same dictionary in the same way; hence, the dictionary identifier and the new value is sent to the decompressor. As long as the dictionary in the decompressor assigns code words in the same deterministic way as the compressor, there is no need to send the code word itself.

The recursive case of the CompressRow algorithm occur when the input join tree node is a non-leaf node. Conceptually, the algorithm first compresses the sub-sequence of the input row corresponding to the left and the sub-sequence corresponding to the right child of the input join tree node t (denoted $t.left, t.right$ respectively) recursively. The algorithm then encodes the pair of compressed sub-sequences using the dictionary $D(t)$ associated with the current input join tree node (Line 13-18).

Example. Consider encoding the first row of the result set in Fig. 1(e) according to the join tree in Fig. 3(b). Logically, each field in the row is first encoded using the column dictionaries and the encoding tuple fragments proceeds bottom (depth-first search order). The fragment `(a1,b1)` that is associated with node R is first processed. Each of the values of the fragment is encoded using the column dictionaries. Assume for ease of exposition that integer codes are assigned starting from zero. The column-encoded pair `00` is then encoded using the dictionary $D(R)$ for node R. Since the dictionaries were initially empty, these encodings all result in new entries added to the dictionary. In principle, we should send the code returned by $D(R)$ for the fragment; however, we can make that implicit with the transmission

Algorithm 3 Decompress

Input: Receives join tree and compressed result set from compressor

Output: Returns the uncompressed result set

```
 1: $T \leftarrow$ receives join tree and schema of leaf nodes
 2: $\mathcal{D} \leftarrow \emptyset$
 3: for all non-root node $v \in T$ do
 4:    initialize dictionary $D(v)$
 5:    $\mathcal{D} \leftarrow \mathcal{D} \cup D(v)$
 6:    if $v$ is a leaf node then
 7:       for all columns $c$ of relation $v$ do
 8:          initialize dictionary $D(c)$
 9:          $\mathcal{D} \leftarrow \mathcal{D} \cup D(c)$
10: $W \leftarrow \emptyset$
11: while receives message $m$ from compressor do
12:    if $m.flag =$ DENTRY then
13:       DictAdd($D(m.nodeID), m.value$)
14:    else
15:       $W \leftarrow W \cup$ DecompressRow($T, \mathcal{D}, m.value$)
16: return $W$
```

of the dictionary entry for $D(R)$. The fragment `(c1)` is then encoded resulting in a new entry for $D(C)$ and $D(S)$, followed by the fragment for the join node j_1. The fragment to be encoded by $D(j_1)$ is constructed from the previously encoded fragments associated with the children of join tree node j_1. Again, since $D(j_1)$ is initially empty, a new entry is sent. Finally, the fragment for node j_2 is constructed from the previously encoded children fragments and transmitted without additional encoding. The sequence of (logical) codes are showed in the first column of Fig. 4. The second row of the result set is encoded similarly. The only difference in the second row is the value in the column for C. The values for other columns remain the same. Therefore, the only new dictionary entries we see are from the column dictionary for C and all of the dictionaries on the path from S to the root in the join tree, from the leaf up. So, we see a new dictionary entry for $D(C)$ for the new value c2. Then, we see an entry $D(S)$, which is the table dictionary entry for the new value c2. Lastly, we see an entry for $D(j_1)$. The new value is `(0,1)`, which indicates the entry for `0` in $D(R)$ and the entry for `1` in $D(S)$. Finally, we transmit the tuple fragment `(1,0)`, which denotes the entry for `1` in $D(j_1)$ and the entry for `0` in $D(Q')$.

3.3 Decompression

The decompression process is symmetric. The decompressor rebuilds the structure of the join tree it receives from the compressor, proceeds to populate dictionaries as it receives entries from the compressor, and uses them to decompress the encoded tuples.

Algorithm 3 outlines the pseudo-code for the decompression of a compressed result stream. The procedure for decompressing a single row within the results stream is given in Algorithm 4. The latter uses a method called ReverseDictLookup(D, i), which returns the value associated with codeword i in D.

Example. Suppose the compressed message given in Fig. 4 is received. When decoding a row, the decoder first receives all of the dictionary entries that it is missing. After pop-

Algorithm 4 DecompressRow($t, \mathcal{D}, e$)

Input: Join tree node t, collection of dictionaries $\mathcal{D}$, a tuple of dictionary codes e
Output: Returns the decompressed row

```
1: if $t$ is a leaf node then
2:     $row \leftarrow \emptyset$
3:     for all columns $c$ in relation $t$ do
4:         $row \leftarrow row \cup$ ReverseDictLookup($\mathcal{D}(c), e[c]$)
5:     return $row$
6: else
7:     $l \leftarrow$ ReverseDictLookup($\mathcal{D}(t.left), e[0]$)
8:     $r \leftarrow$ ReverseDictLookup($\mathcal{D}(t.right), e[1]$)
9:     $row \leftarrow$ DecompressRow($t.left$, $\mathcal{D}$, $l$)
10:    $row \leftarrow row \cup$ DecompressRow($t.right$, $\mathcal{D}$, $r$)
11:    return $row$
```

ulating its dictionaries, the entries are as shown in column 1 of Fig. 5. Subsequent to processing dictionary entries, the decoder receives the tuple fragment for the first row: `(0,0)`. Using the join tree, the decoder is able to determine that the tuple fragment `(0,0)` refers to the concatenation of the values for the 0-th entry of $D(j_1)$ and the 0-th entry of $D(Q')$. The rest of the decompression process is shown in detail below. We use $D(X)\!:\!n$ to denote the n-th entry in the dictionary $D(x)$, and $D(X)\!:\!n$ = x to denote that the value for the n-th entry in the dictionary $D(X)$ is "x".

1. TF $= (0, 0)$
2. $(D(j_1)\!:\!0 = (0, 0),\ D(Q')\!:\!0 = (0))$
3. $(D(R)\!:\!0 = (0, 0),\ D(S)\!:\!0 = (0),\ D(D)\!:\!0 =$ d1)
4. $(D(A)\!:\!0 =$ a1, $D(B)\!:\!0 =$ b1, $D(C)\!:\!0 =$ c1, d1)
5. (a1, b1, c1, d1)

In step 2, the decoder expands the tuple fragment $(0, 0)$ by doing dictionary lookups in $D(j_1)$ and $D(Q')$, finding that the values are $(0, 0)$ and (0), respectively. In step 3, the decoder determines from the join tree that the value $D(j_1)\!:\!0 = (0, 0)$ corresponds to the concatenation of the 0-th entry from $D(R)$, and the 0-th entry from $D(S)$. Similarly, it finds that $D(Q')\!:\!0 = (0)$ specifies the 0-th entry from $D(D)$, which is d1. The remaining steps follow the same process to arrive at the first row: (a1, b1, c1, d1).

When receiving the commands for the decompression of the second row, the decoder is only missing three of the needed dictionary entries. This is because the only new value in the second row is in the C column. Thus, $D(C)$ and every dictionary on the path to the root of the join tree must be updated with a new value. Other than $D(C)$, these dictionaries are $D(S)$, and $D(j_1)$. The new entries are shown in column 2 of Fig. 5. Decompression follows exactly the same steps that were taken for the first row:

1. TF $= (1, 0)$
2. $(D(j_1)\!:\!1 = (0, 1),\ D(Q')\!:\!0 = (0))$
3. $(D(R)\!:\!0 = (0, 0),\ D(S)\!:\!1 = (1),\ D(D)\!:\!0 =$ a1)
4. $(D(A)\!:\!0 =$ a1, $D(B)\!:\!0 =$ b2, $D(C)\!:\!1 =$ c2, d1)
5. (a1, b1, c2, d1)

3.4 Space Limited Dictionaries

So far we have described the proposed QA compression algorithm assuming that there is no limit on the size of the dictionaries. For large result sets with large number of distinct values, the size of each dictionary can be significantly large and put a strain on the client's resources. Conceptually, a dictionary is a mapping of tuple fragments to their codes. Consider a result set from a simple two-way join $R \bowtie S$. If all rows of R are distinct, the dictionary $D(R)$ could be as large as the number of rows in R if the join operator did not filter any rows of R. For very large relations, the dictionary may not even fit in memory! For the proposed QA compression scheme to be practical, the space required by each dictionary cannot be allowed to grow without restraint.

We propose a simple solution – space limited dictionaries. A space limited dictionary uses a fixed amount of space which can either be specified in bytes or in the number of entries. For ease of exposition, we will use number of entries. A space limited dictionary with n slots would be able to hold at most n entries. Each entry would be associated with a code between 0 and $n - 1$. We use integer code for ease of exposition, in practice Huffman codes can be used. It is possible to make use of Huffman codes because the compressor and decompressor are symmetric and maintaining the same dictionaries. As long as the decompressor is made aware that the compressor has decided to optimize its codes, it can mimic the same optimization. Initially, the n slots are empty. As new (previously unseen) entries arrive, they fill up the n slots. Thereafter the dictionary behaves somewhat like a cache. The arrival of a new entry would trigger the eviction of an existing entry and the new entry would use the code associated with the evicted entry. As long as the dictionary at the decompressor behaves in exactly the same way, decoding is possible and correct.

As with caching, several eviction policies are possible including least recently used (LRU), least frequently used (LFU), least recently added (LRA) etc. The LRA policy would evict the oldest dictionary entry (see Fig. 6) and LFU would evict the least frequently used entry. Both of these policies would require maintaining a timestamp or frequency count for each entry. In addition to an eviction policy, space limited dictionaries can also have a code assignment policy that determines when and how to reassign codes to the n entries. For example, in conjunction with LFU, a code assignment policy can reassign shorter codes to the more frequently used entries at every other eviction.

	Row 1	Row 2
$D(A)$	0 = a1	
$D(B)$	0 = b1	
$D(C)$	0 = c1	1 = c2
$D(D)$	0 = d1	
$D(R)$	0 = `(0,0)`	
$D(S)$	0 = `(0)`	1 = `(1)`
$D(Q')$	0 = `(0)`	
$D(j_1)$	0 = `(0,0)`	1 = `(0,1)`

Figure 5: Dictionaries reconstructed during decompression. Values added after decoding the first row are in the column labeled "Row 1". Those added after decoding the second row are under the "Row 2" column.

3.5 Dynamic Allocation of Dictionary Space

Our QA compression algorithm relies on a hierarchy of

value	code	time
a1b1	0	11
a2b1	1	13

(a) After 3 rows

value	code	time
a1b2	0	15
a2b1	1	13

(b) After 5 rows

Figure 6: Example of encoding the fragment of the result set of Fig. 1(e) associated with join tree node R using a space limited dictionary with two slots. When encoding the 5-th row with value '(a1,b2)', the oldest entry with code 0 is evicted.

disjoint dictionaries to encode result tuple fragments. Each dictionary in the collection or hierarchy may have different demands for memory at different times giving rise to the space allocation problem: Given a memory budget of M bytes and a collection of N dictionaries, how should the M bytes of memory be allocated to each of the N dictionaries? In this section, we discuss several solutions for this problem.

Naïve Solution. Given a set of N dictionaries which is limited to M bytes of memory, we can allot each dictionary M/N bytes. This is often suboptimal, because the space demand distribution of the N dictionaries are often skewed. Using an uniform allocation results in high-demand dictionaries not getting enough space, and low-demand dictionaries not using all the space allocated to them.

Static Solution. DBMSs often collect cardinality statistics on relations for query optimization. We could exploit such statistics to compute a static allocation to the dictionaries. For example, the size of a dictionary that is associated with a base table (in the join tree) can be bounded from above by the distinct count of tuples of that base table. Hence we can estimate the relative demand each dictionary has for space by looking at count statistics for the relation, column, or partial result it corresponds to in the join tree. A dictionary corresponding to a table with tuple fragments of average size n bytes and k distinct tuples would get double the space of a dictionary for a table with tuple fragments of average size $\frac{1}{2}n$ and k distinct tuples.

Dynamic Solution. While the static solution is an improvement over the naïve solution, it is still possible for the allocated space to be "wasted". Consider the case when there are many distinct values, but at any one window of time only a small number of distinct values are accessed, i.e., the entry insertion and entry access pattern for a particular dictionary exhibit strong temporal locality. The dynamic solution attempts to exploit this temporal locality by observing and maintaining statistics on the access patterns and re-assigning space to the dictionaries periodically. This approach requires each dictionary to spend some of the space budget to maintain the insertion order for its entries, and the number of times each entry has been accessed since insertion. We first allow the dictionary set to grow to its initial capacity of M bytes. Each of its dictionaries are allowed to grow as long as the total memory consumption is less than M bytes. When the capacity is reached, we use the relative sizes of the dictionaries to measure demand, and use the access counts and insertion order to measure the degree of temporal locality.

In order to quantify temporal locality, we follow this intuition: if access counts for older entries are not significantly higher than those for more recent entries, then it is likely that the older entries are no longer being used. Fig. 7 depicts a dictionary having evidence of temporal locality.

Older $\longrightarrow$ Newer

Index	0	1	2	3	4	5
Value	a	b	c	d	e	f
Access Count	10	10	10	8	5	4

Figure 7: Visualization of a dictionary after several insertions and accesses. Notice that the three oldest entries are accessed the same number of times, indicating temporal locality.

More formally, suppose that for a dictionary D, e_i refers to its i-th oldest entry. Further, let $c(e_i)$ denote the number of times e_i has been accessed since its insertion. Given a parameter α, we find the largest n for which the relative distance between $c(e_0)$ and $c(e_n)$ is $< \alpha$. That is, $(c(e_0) - c(e_n))/c(e_0) < \alpha$. We then label all entries e_i with $i \leq n$ as waste. Finally, we compute S, the sum of all memory consumed by entries labeled as waste.

After computing S_j for each dictionary D_j having a total memory consumption of N_j, we compute its new capacity as:

$$M \cdot \frac{N_j - S_j}{\sum (N_i - S_i)},$$

where M is the total capacity available to the entire dictionary set. We find in the experiments to follow in Sec. 4.6, this approach makes better use of allotted space than the naïve approach.

4. EXPERIMENTS

Implementation. We implemented the query-aware compression algorithm in C++, using sqlite3 as the relational database backend. The stream produced by the implementation uses a few optimizations not specified by the algorithm. Most importantly, dictionary indexes are encoded in a variable byte length format to minimize overhead.

Dataset. In each of the experiments to follow, we use the TPC-H [8] dataset. We use the provided `dbgen` program, and vary the scale factor parameter to control dataset sizes. For example, a scale factor of 0.25 results in a dataset occupying approximately 344MB of disk with indexes, and 273MB without. The query with the most tuples in its result set occupies approximately 1GB of disk when executed on the aforementioned database.

Performance Metrics. We measure the compression ratio defined as

$$\mathit{Compression\ Ratio} = \frac{\mathit{Raw\ data\ size\ in\ bytes}}{\mathit{Compressed\ data\ size\ in\ bytes}}.$$

Raw sizes are collected by writing results to disk in CSV format and measuring the file size. We measure gzipped sizes using the `gzip` program included with most UNIX-based operating systems with the `--best` flag. When measuring execution time, we use wall-clock time, omitting any IO by writing to a null output device.

QA compression offers its users a tradeoff: space or bandwidth savings for extra CPU cycles and memory. Although we do not use an end-to-end metric in these experiments, we note that a boost in compression ratio will see a proportional drop in power consumed by, for example, a wireless devices. Unless we see significantly greater gains in execution time, improvements in compression ratio are strong evidence that our algorithm will improve more tangible measures such as latency or energy consumption. This is because the energy saved from spending n fewer seconds using a network device outweighs the extra energy consumed from spending $m \approx n$ extra seconds of CPU execution time[23, 4].

Algorithms and variants. Our experiments evaluate the following algorithms and variants:

- **gzip** : The raw result set in CSV is compressed using gzip.
- **query-aware-gzip**: The result set is compressed using the proposed query aware algorithm followed by gzip

We do not compare QA compression alone to gzip because it is meant as a supplement to, and not a replacement for existing compression algorithms. As previously mentioned, we find that QA compression is highly orthogonal to techniques unable to take into account the redundancy revealed by the join tree model.

Queries. We derived a suite of queries from those provided with the TPC-H dataset. We remove filtering conditions, aggregators, and ordering. The following join orders are used:

1. customer ⋈ (orders ⋈ lineitem)
2. (part ⋈ partsupp) ⋈ (supplier ⋈ nation)
3. supplier ⋈ lineitem *(Note: only suppkey is used.)*
4. customer ⋈ orders
5. ((customer ⋈ orders) ⋈ lineitem) ⋈ (supplier ⋈ (nation ⋈ region))
6. (part ⋈ partsupp) ⋈ (supplier ⋈ (nation ⋈ region))

We focus entirely on queries involving only joins because selections and projections do not affect compression ratios. Figure 8 shows how each of these queries grow as the scale factor increases.

In the experiments to follow, we measure how dictionary sizes and number of rows resulting from the queries above affect the compression ratios. The dictionary size gives the maximum number of entries in *each* of the dictionaries used in the algorithm. The number of rows in the query results is determined by the scale factor argument in the `dbgen` program provided with the TPC-H toolset.

4.1 Overall Compression Rate

We begin by examining the compression ratios for each of the queries using fixed values for both result and dictionary sizes. Each dictionary used in the algorithm is allotted 50K entries, and the scale factor is fixed at 0.21. Results are shown in Figure 9. The vertical axis specifies compression ratio, so a higher number is better.

Notice that for most queries, QA compression combined with gzip nearly doubles the compression ratio of gzip by itself. In the case of Query 5, QA+gzip reduces the raw result set from approximately 840MB to 68MB, which is a 12x reduction. With gzip by itself, we see a reduction of only 5x, resulting in a compressed result set of approximately 163MB.

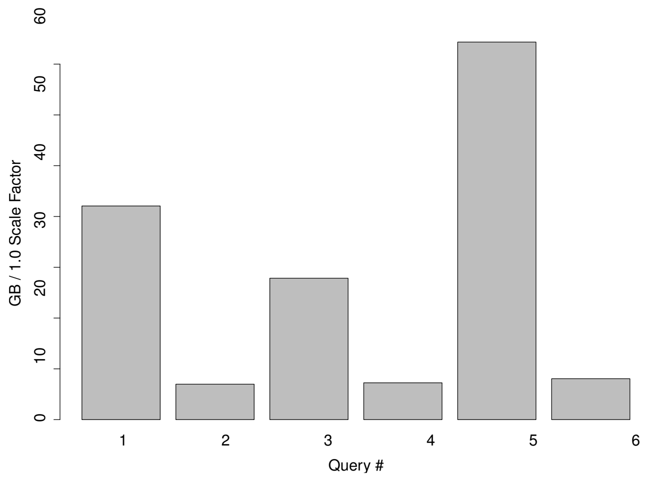


Figure 8: Depiction of query result growth rates, specified in terms of the size growth (in GB) when increasing dbgen's scale factor parameter by 1.0.

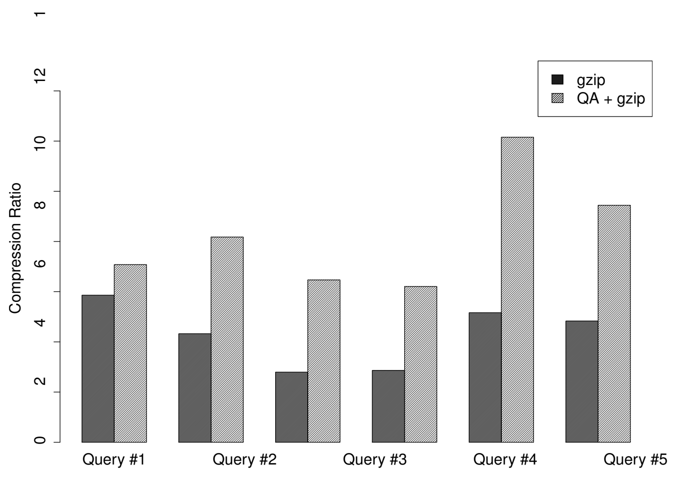


Figure 9: Compression ratios with dictionary sizes fixed at 50K and scale factor fixed at 0.21.

4.2 Varying Data Size

This experiment varies the number of rows in the query results while fixing the dictionary size. This shows us how well the algorithm scales with the size of the result set and the underlying database. Here, it is easy to observe the point at which dictionaries become full and older entries have to be cleared to make room for new ones. We refer to this as *dictionary saturation.* Figure 10 depicts dictionary saturation for dictionaries containing no more than 10K entries. Notice that Queries 1 and 4 exhibit the sharpest decline of compression ratio. We can attribute this to these being relatively simple queries, meaning there is less redundancy for the compression to exploit. Query 3 performs relatively well because we only use one of two available join keys, which introduces additional redundancy.

Figure 11 shows compression ratios when dictionaries are limited to 20K entries. Notice that this delays the occurrence of dictionary saturation in Query 4 by a 0.10 increase in scale factor. If we increase the maximum dictionary size to 30K, we delay dictionary saturation by another 0.10 increase in scale factor. This linear pattern is not surprising, as both the original tables and the join results are increasing linearly with scale factor.

4.3 Varying Dictionary Size

Finally, we vary the dictionary size and fix the query result size. This gives a rough idea how much memory the algorithm requires to perform well when run on large datasets. The dictionary size specifies the maximum number of entries in *every* dictionary used in the algorithm. Figure 12 shows compression ratios when the scale factor is fixed at 0.35. Notice that 10K entries is enough to prevent dictionary saturation for all except for Queries 1, 4, and 5. In the case of 5 we still see relatively good performance. In Queries 1 and 4, we see that 40K entries is enough to produce optimal performance.

In Query 1, notice that QA+gzip performs worse than gzip by itself when we limit dictionaries to a maximum of 10K entries. This is likely due to severe dictionary saturation. We expect this behavior if an entry is pushed onto the dictionary and evicted before it is referenced again. In this case, we only introduce extra bytes into the result stream and put extra space between text, which harms gzip's performance. We can see from this that it is important to allow dictionaries to be large enough to prevent this from happening.

4.4 Sensitivity to Join Order in gzip

When analyzing preliminary results, we noticed that the result sizes would vary when changing the order in which tables were joined, and by the left-right relationship between the nodes in the join tree. With the database engine we used, the left-right relationship is determined by the structure of the query. For example, `SELECT * FROM a, b WHERE a.c = b.c` produces different results than `SELECT * FROM b,a WHERE b.c = a.c`. In particular, the order in which the result tuples appear is affected.

Clearly, compression algorithms tend to be very sensitive to the order of the data they are acting on. If redundancy is spaced far apart in the result stream, compression algorithms generally have to use more memory to exploit it without suffering a loss in performance.

Figure 13 is the result of generating all possible ways to order the tables in Query 2 ($2^3 = 8$). Notice that there are two classes of results: one where gzip does only slightly worse than QA+gzip, and one where gzip's compression ratio is roughly half of QA+gzip. This is determined by the left-right relationship between the results in the highest level join. That is, which of (part $\bowtie$ partsupp) or (supplier $\bowtie$ nation) is on the left, and which is on the right. Note that applying query-aware compression before gzip in this case makes up for the lost performance in gzip. In particular, the compression ratio for QA+gzip in this case is relatively unaffected by the join order.

Figure 14 shows the same result when varying the join orders of Query 4. Notice that in the first ordering, QA+gzip performs slightly worse than gzip by itself. In examining the result stream, we discovered that tuples with redundancy exploitable by QA compression were all grouped together. This means that the redundancy is highly localized and exploitable by gzip. Using QA before gzip does very little beyond delocalizing coincidental redundancy that gzip could otherwise exploit. In the second result, however, similar tuples are spaced far away in the result set, and gzip suffers as a result of this. Applying QA compression beforehand helps to reorder the redundancy that gzip can exploit.

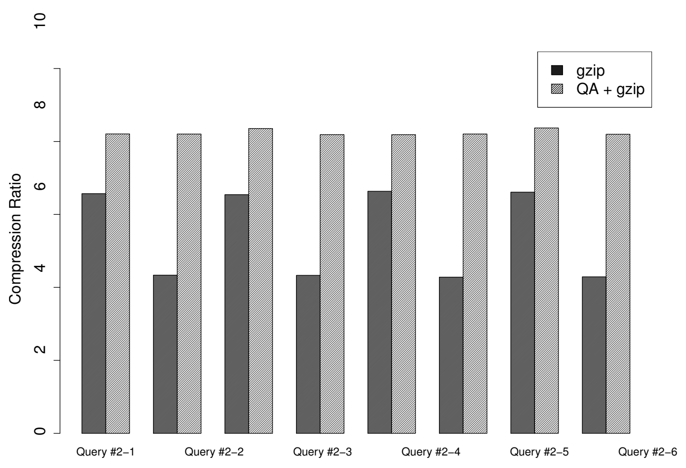


Figure 13: Compression ratios for each of the possible join orders for Query 2. A scale factor of 0.20 and dictionary sizes of 100K are used.

From this, we can see that with sufficiently large dictionaries, order does not not affect the performance of our algorithm. gzip, however, suffers significantly when similar tuples are spaced far apart in the result stream. If we apply query-aware compression with large enough dictionaries, followed by gzip to a result stream with poor ordering of redundant tuples, it tends to make up for the lost performance. Figure 13 evidences this especially well.

Ideally, a future implementation integrated with a database management system would allow for the query optimizer to predict whether tuples with redundancy will be adjacent to one another in the result stream. If so, our algorithm may offer only marginal improvement on top of gzip. The query planner could then intelligently decide to employ our algorithm rather than re-ordering the result stream.

4.5 Execution Time

In each of the previously discussed queries, we measure execution time for iterating through the result set without any additional operations, and for compressing the result set using query-aware compression. This gives us a good idea of how much overhead using query-aware compression introduces. The results were collected on a machine with a 2.2 GHz Intel® Core^{TM} i7 processor and 8 GB of 1333 MHz DDR3 RAM.

Results for Query 5 (the most complicated query we tried) are shown in Figure 15. Results for Query 2 are shown in Figure 16. For these measurements, we fixed dictionary sizes to a maximum of 100K entries and report the average over 10 runs. Here, all output operations are replaced with no-ops to allow for a more accurate representation of overhead.

These results suggest that overhead scales linearly with result size. This is what we expect, as this algorithm performs a nonzero amount of work for each tuple. So, in a

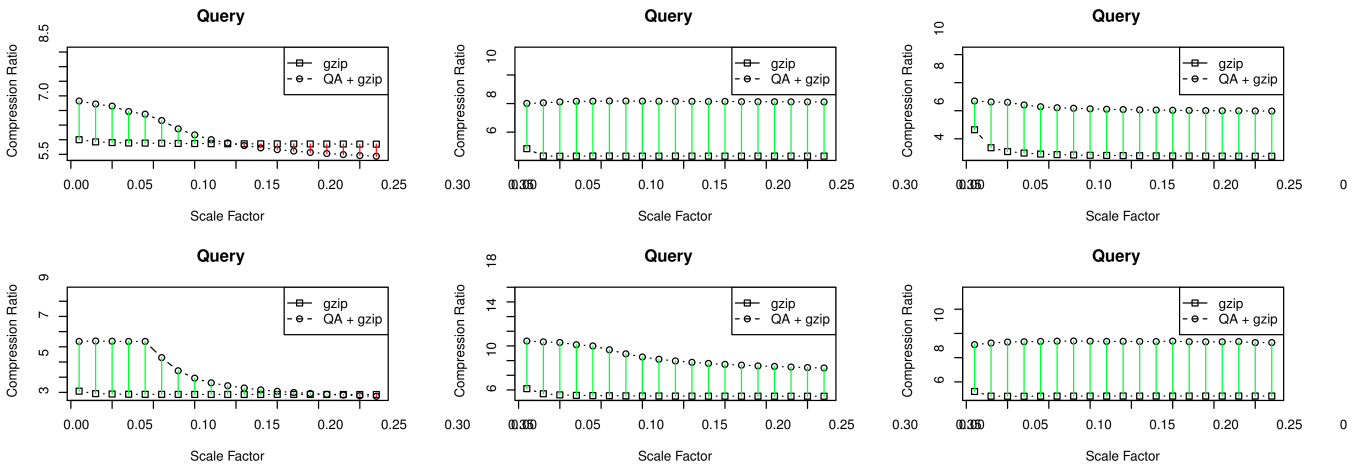


Figure 10: Compression ratios with dictionary sizes fixed at 10K entries.

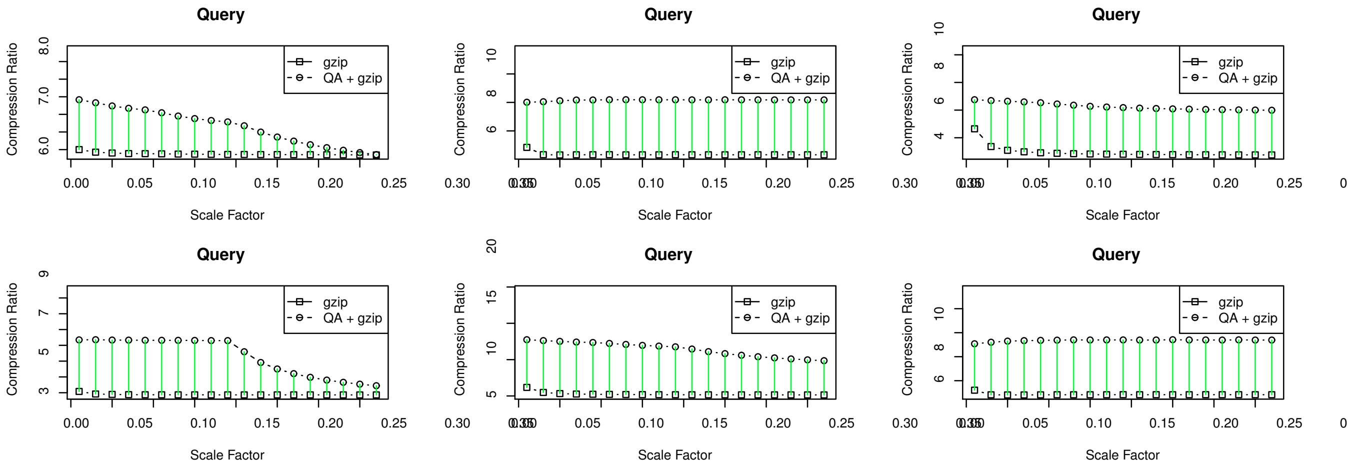


Figure 11: Compression ratios with dictionary sizes fixed at 20K entries.

stream containing n tuples, our algorithm performs $O(n)$ work. Therefore, should see an increase in the linear growth rate for execution time.

4.6 Dictionary Allocation Strategies

In this section, we compare the capacity allocation strategies introduced in Sec. 3.5. We run Query 5 from the previous experiments while varying the scale factor. We fix three different dictionary capacities: 50KB, 100KB, and 200KB. The results are shown in Fig. 17

Observe that the dynamic allocation approach makes better use of the available space. Notice that for a capacity of M bytes, if M/N bytes is enough to prevent the dictionaries from avoiding saturation, we see that the naïve and dynamic strategies have the same performance. However, if M/N bytes is *not* sufficient for some of the dictionaries, we see that the dynamic strategy outperforms the naïve strategy.

Finally, notice that the scale factor at which dynamic partitioning begins to outperform naïve partitioning increases as the available capacity increases. This is expected, as M/N bytes for each dictionary is sufficient for longer as M increases.

5. CONCLUSION

In this paper, we provide a recursive compression algorithm applicable to results of join queries. Section 3 describes our algorithm in detail, and includes examples of its application to a small dataset. We show that the associated decompression algorithm is symmetric. We find that simply evicting old dictionary entries is an effective means for limiting resource consumption without compromising improvements in compression ratio.

We find that in the best case, query-aware compression can improve compression ratios by more than a factor of two, as seen in Figure 9. In this case, raw query results occupy approximately 1GB of disk, while the stream compressed with QA+gzip occupies only 86MB.

As discussed in Section 4.4, we find that changing the left-right relationship of tables in a join query strongly influences the order of the result stream. Because compression algorithms tend to behave optimally when redundant data are close together in the result stream, the left-right relationship indirectly affects the achievable compression ratios of a result set. We find that with sufficiently large dictionaries, query-aware compression is able to alleviate all or most of

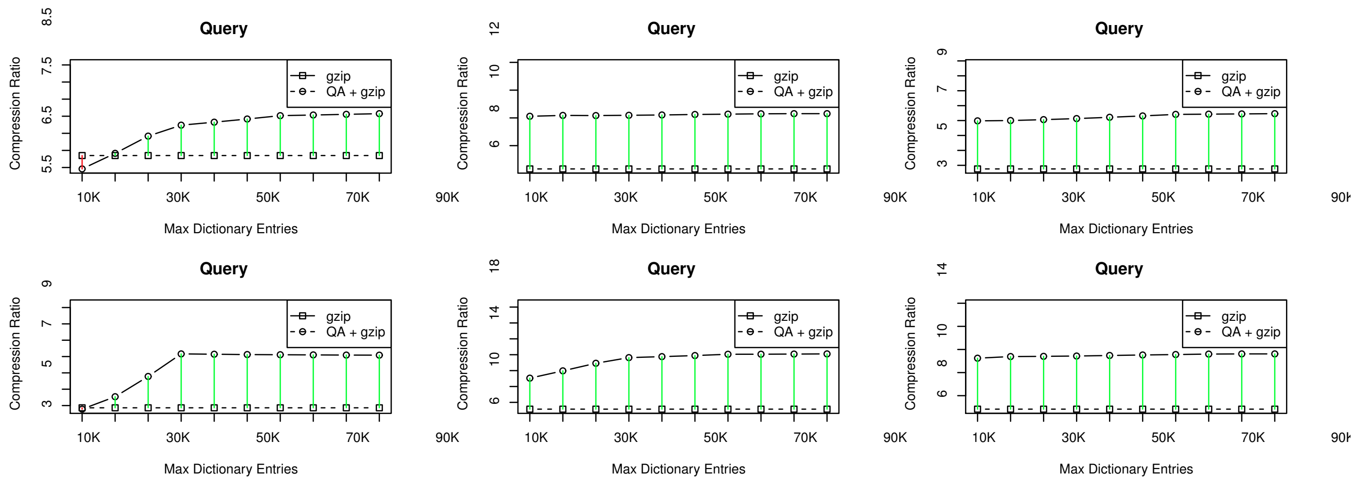
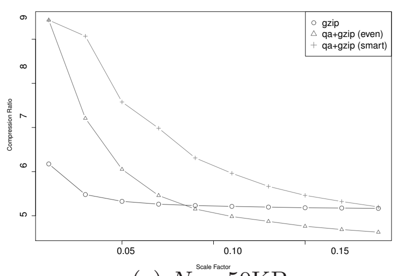
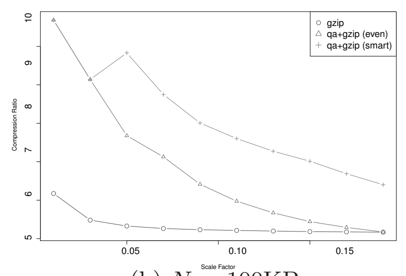


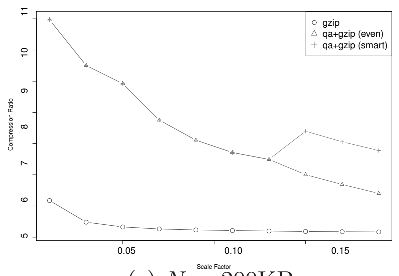
Figure 12: Compression ratios with scale factor fixed at 0.35.



(a) N = 50KB.



(b) N = 100KB.



(c) N = 200KB.

Figure 17: A comparison of compression ratios when using the naïve and dynamic (labeled "smart". gzip is also included as a baseline.) partitioning strategies

the lost compression performance in gzip due to disorder in the data.

Figures 15 and 16 show that overhead introduced by our algorithm grows linearly with the size of results stream.

In the future, our algorithm might be improved by using alternative dictionary eviction techniques. As mentioned in Section 3.4, there are a variety of alternative approaches that might be employed to improve performance. These include least frequently used (instead of least recently used), and the use of Huffman codes to reduce the size of index references in the result stream.

We conclude that query-aware compression is a promising technique to reduce the size of join results.

6. REFERENCES

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, New York, NY, USA, 2006. ACM.

[2] D. J. Abadi, P. A. Boncz, and S. Harizopoulos. Column-oriented database systems. *PVLDB.*, 2:1664–1665, August 2009.

[3] G. Antoshenkov, D. Lomet, and J. Murray. Order preserving string compression. *ICDE*, 0:655, 1996.

[4] M. Asplund, A. Thomasson, E. J. Vergara, and S. Nadjm-Tehrani. Software-related energy footprint of a wireless broadband module. In *Proceedings of the 9th ACM international symposium on Mobility management and wireless access*, MobiWac '11, pages 75–82, New York, NY, USA, 2011. ACM.

[5] B. Bhattacharjee, L. Lim, T. Malkemus, G. Mihaila, K. Ross, S. Lau, C. McArthur, Z. Toth, and R. Sherkat. Efficient index compression in DB2 LUW. *VLDB*, 2:1462–1473, August 2009.

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

[7] G. V. Cormack. Data compression on a database system. *Commun. ACM*, pages 1336–1342, 1985.

[8] T. P. P. Council. Tpc benchmark(tm) h, Feb. 2011.

[9] P. Deutsch. GZIP file format specification version 4.3, 1996.

[10] C.-L. Goh, K. Aisaka, M. Tsukamoto, and S. Nishio. Database compression with data mining methods. In K. Tanaka, S. Ghandeharizadeh, and Y. Kambayashi, editors, *Information Organization and Databases*, volume 579 of *The Kluwer International Series in Engineering and Computer Science*, pages 177–190.

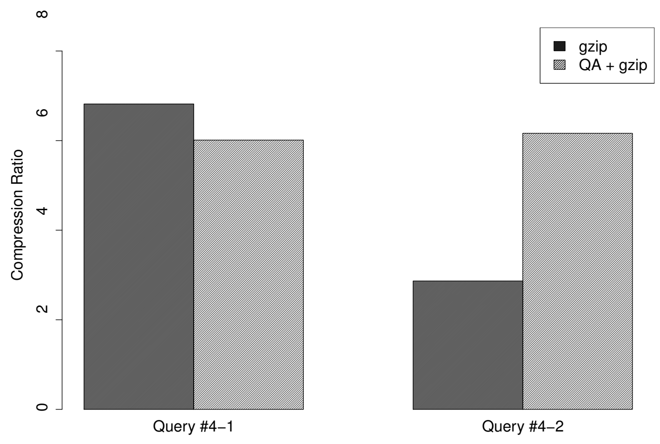


Figure 14: Compression ratios for each of the possible join orders for Query 4. A scale factor of 0.20 and dictionary sizes of 100K are used.

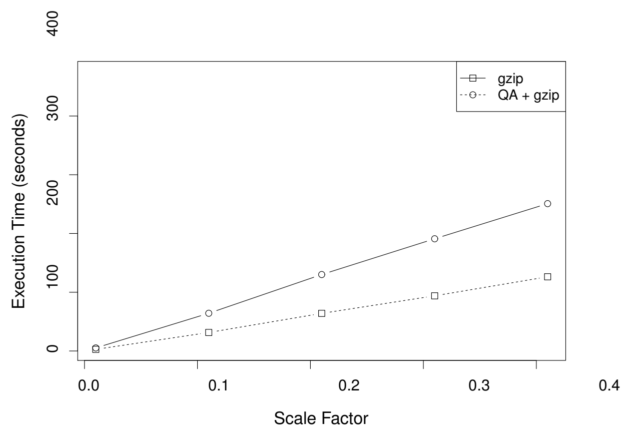


Figure 15: Execution time (in seconds) for Query 5.

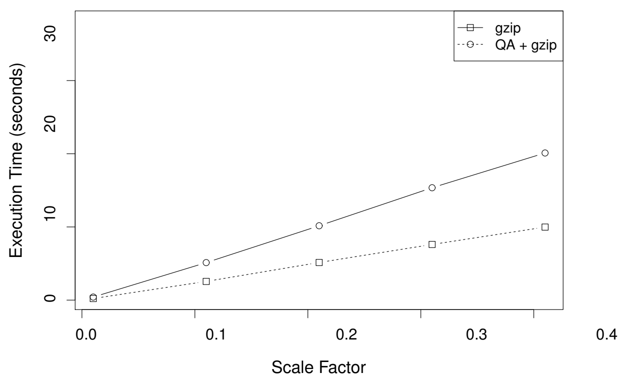


Figure 16: Execution time (in seconds) for Query 2.

Springer US, 2001.

[11] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. *ICDE*, 0:370, 1998.

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

[13] G. Hamilton, R. Cattell, and M. Fisher. *JDBC Database Access with Java: A Tutorial and Annotated Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1997.

[14] A. L. Holloway, V. Raman, G. Swart, and D. J. DeWitt. How to barter bits for chronons: compression and bandwidth trade offs for database scans. In *SIGMOD*, pages 389–400, New York, NY, USA, 2007. ACM.

[15] D. Huffman. A method for the construction of minimum redundant codes. In *Proc. IRE*, volume 40, pages 1098–1101, 1952.

[16] ISO. *ISO/IEC 11172-3:1993*, 1993 (accessed February 26, 2012). `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22412`.

[17] S. Maruyama, M. Takeda, M. Nakahara, and H. Sakamoto. An online algorithm for lightweight grammar-based compression. In *CCP*, pages 19–28, 2011.

[18] W. Ng and C. Ravishankar. Relational database compression using augmented vector quantization. In *ICDE*, pages 540 –549, mar 1995.

[19] V. Raman and G. Swart. How to wring a table dry: entropy compression of relations and querying of compressed relations. In *Very Large Data Bases*, pages 858–869. VLDB Endowment, 2006.

[20] G. Ray, J. R. Haritsa, and S. Seshadri. Database compression: A performance enhancement tool. In *Proc. of 7th Intl. Conf. on Management of Data (COMAD)*, 1995.

[21] M. A. Roth and S. J. Van Horn. Database compression. *SIGMOD Rec.*, 22:31–39, September 1993.

[22] H. Sakamoto, T. Kida, and S. Shimozono. A space-saving linear-time algorithm for grammar-based compression. In *SPIRE*, pages 218–229, 2004.

[23] J. Sharkey. Coding for life – battery life, that is. `http://dl.google.com/io/2009/pres/W_0300_CodingforLife-BatteryLifeThatIs.pdf`, May 2009.

[24] R. Signore, M. O. Stegman, and J. Creamer. *The ODBC Solution: Open Database Connectivity in Distributed Environments*. McGraw-Hill, Inc., New York, NY, USA, 1995.

[25] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30:520–540, June 1987.

[26] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[27] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.