QUERY-AWARE COMPRESSION OF JOIN RESULTS

A THESIS SUBMITTED TO THE
GRADUATE DIVISION OF THE
UNIVERSITY OF HAWAI‘I AT MĀNOA
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

DECEMBER 2012

By

Christopher M. Mullins

Thesis Committee:

Lipyeow Lim, Chairperson
Henri Casanova
Kyungim Baek
We certify that we have read this thesis and that, in our opinion, it is satisfactory in scope and quality as a thesis for the degree of Master of Science in Computer Science.

THESIS COMMITTEE

______________________________
Chairperson
Copyright 2012 by

Christopher M. Mullins
ABSTRACT

The proliferation of lightweight client devices such as iPhones, iPads, Android phones and tablets, has created an increased demand for cloud-based services. In many of these services, 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 any effort to reduce the amount of data transmitted across the network would not only conserve bandwidth but help with the battery life of the client devices. In this thesis 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. We evaluate 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 two times over gzip are possible.
# TABLE OF CONTENTS

Abstract ........................................................................ iv

List of Tables ................................................................... vii

List of Figures ................................................................... viii

1 Introduction ................................................................. 1

2 Previous Work ............................................................. 5

3 Query-Aware Compression ............................................ 8
   3.1 Algorithm ............................................................ 8
      3.1.1 Compression .................................................. 10
      3.1.2 Decompression ............................................... 14
   3.2 Experiments ......................................................... 15
      3.2.1 Experiment Setup ........................................... 15
      3.2.2 Compression Ratio ......................................... 17
      3.2.3 Execution Time ............................................. 18

4 Fixed Size Dictionaries ............................................... 21
   4.1 Formalization of Fixed-Size Dictionaries .................... 23
      4.1.1 Techniques For Maintaining Capacity ................ 23
      4.1.2 Sets of Fixed-Size Dictionaries ....................... 24
   4.2 Partitioning Algorithms ....................................... 25
      4.2.1 Naïve ........................................................... 25
      4.2.2 Pooled Capacity ............................................ 25
      4.2.3 Adaptive Repartitioning ................................. 26
4.2.4 Runtime Analysis ........................................ 28
4.2.5 Empirical Analysis ........................................ 29
4.3 Effects on Query-Aware Algorithm Performance .............. 32
  4.3.1 Performance Summary .................................... 33
  4.3.2 Varying Data Size ......................................... 34
  4.3.3 Varying Dictionary Size ................................... 34
  4.3.4 Execution Time ........................................... 35
4.4 Conclusions .................................................. 36

5 Join Order Optimization ......................................... 39
  5.1 Effects of Join Order on Compression Ratio .................... 39
  5.2 Estimating Compression Ratio with Unbounded Dictionaries ... 41
  5.3 Estimating Compression Ratio When Using Fixed-Size Dictionaries 46
  5.4 Choosing an Optimal Join Order ................................ 52
  5.5 Conclusions ................................................. 53

6 Functional Dependencies ......................................... 57
  6.1 Join Tree Augmentation ....................................... 58
  6.2 Constrained Functional Dependencies .......................... 61
    6.2.1 Algorithms for Exploiting CFDs ......................... 63
    6.2.2 Experimental Results .................................... 74
  6.3 Conclusions ................................................. 75

7 Conclusion ..................................................... 77

Bibliography ...................................................... 79
LIST OF TABLES
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The base relations, query and result set used in the running example.</td>
<td>2</td>
</tr>
<tr>
<td>3.1</td>
<td>Query-aware compression in a client-server framework.</td>
<td>8</td>
</tr>
<tr>
<td>3.2</td>
<td>Query execution plan for the SQL query in Fig. 1.1(d) and the corresponding join tree.</td>
<td>9</td>
</tr>
<tr>
<td>3.3</td>
<td>Logical encoding of the first two rows of the result set in Fig. 1.1(e) using the join tree in Fig. 3.2(b). The symbol ‘DE’ denotes the dictionary entry flag, ‘TF’ denotes a tuple fragment flag, ‘D(i)’ denotes the dictionary identifier for join tree node i.</td>
<td>13</td>
</tr>
<tr>
<td>3.4</td>
<td>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.</td>
<td>18</td>
</tr>
<tr>
<td>3.5</td>
<td>Compression ratios for various result set sizes.</td>
<td>19</td>
</tr>
<tr>
<td>3.6</td>
<td>The mean rows/second for each query at a fixed scale factor. ±1 standard deviation is shown with the whiskers centered at the mean.</td>
<td>20</td>
</tr>
<tr>
<td>4.1</td>
<td>Eviction of oldest entry in a fixed-size dictionary with capacity 5. The oldest entry is indicated with (*).</td>
<td>22</td>
</tr>
<tr>
<td>4.2</td>
<td>Example of encoding the fragment of the result set of Fig. 1.1(e) associated with join tree node R using a space limited dictionary with two slots. When encoding the 5th row with value ‘(a1,b2)’, the oldest entry with code 0 is evicted.</td>
<td>22</td>
</tr>
<tr>
<td>4.3</td>
<td>Visualization of a dictionary. Each entry $e = (value, index)$ is depicted showing its index in the first row, value in the second. $c(e)$ denotes the number of times $e$ has been accessed.</td>
<td>27</td>
</tr>
<tr>
<td>4.4</td>
<td>Compression ratio results</td>
<td>31</td>
</tr>
<tr>
<td>4.5</td>
<td>Execution time results</td>
<td>32</td>
</tr>
<tr>
<td>4.6</td>
<td>Compression ratios with dictionary sizes fixed at 50K and scale factor fixed at 0.21.</td>
<td>33</td>
</tr>
</tbody>
</table>
4.7 Compression ratios with dictionary sizes fixed at 10K entries. ............... 35
4.8 Compression ratios with dictionary sizes fixed at 20K entries. ............... 36
4.9 Compression ratios with scale factor fixed at 0.35. .............................. 37
4.10 Execution time (in seconds) for Query 5. ........................................ 38
4.11 Execution time (in seconds) for Query 2. ........................................ 38
5.1 The two possible join orders for the relations AB, BCD, and CE. ............ 41
5.2 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. ......................... 42
5.3 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. ......................... 43
5.4 Actual compressed size vs. predicted compressed size ......................... 46
5.5 Ratio of predicted compressed size and actual compressed size. ............... 47
5.6 Re-reference patterns for one join ordering of Query 5. Each of the above represents one equivalence class that applies to several related dictionaries. 54
5.7 The re-reference pattern for the same dictionary represented by Fig. 5.6(c) under a different join order ......................................................... 55
5.8 Re-reference patterns for one join ordering of Query 2. Each of the above represents one equivalence class that applies to several related dictionaries. 55
5.9 A demonstration that $\beta$ is positively correlated with $N$. .................... 56
6.1 A join tree augmented with artificial joins derived from the functional dependencies $X \rightarrow Y$ on $R$ and $A \rightarrow B$ on $Q'$. ................................. 57
6.2 Depiction of different outcomes based on which functional dependency is chosen first. Note that choosing $AB \rightarrow D$ first results in a schema with more joins. ......................................................... 59
6.3 Two join trees for the same result set. The first should be used when the constraint in the CFD is met. The second is for when the constraint is not met. ......................................................... 62
6.4 A depiction of a join tree where the relation BCDE has two CFDs. The figures above show the resulting join tree when one, both, or neither of the CFDs apply.

6.5 A depiction of the compressed results stream for $R(A,B,C)$ with the CFD $(A \leq 3) \Rightarrow (A \rightarrow B)$. Each block of rows represents the encoding of one tuple.

6.6 A CFD dependency graph. An outgoing edge indicates that the source should be evaluated before the target. A dashed edge corresponds to a dependency introduced by a constraint. Constraints are omitted where they do not introduce a dependency.

6.7 A cyclical dependency graph produced when the constraint of one CFD includes an attribute downstream of it. In this diagram, the dashed edge indicates a dependency introduced by a constraint.

6.8 Schema for the ORDERS table in the TPC-H dataset.

6.9 Join tree used for measuring compression ratio gains when using a CFD. The join corresponding to the CFD is shown with dotted lines.

6.10 CFD results for various fixed-size dictionary capacities.
Lightweight client devices such as iPhones, iPads, Android phones and tablets are fast becoming ubiquitous. The widespread adoption of these devices has fueled an increasing demand for cloud-based services for both the casual user as well as business users. Many of these services (e.g., business intelligence applications) follow the paradigm where queries are sent to cloud-based database servers for processing and the results relayed back to the client devices. These 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 help with the battery life, because data transmission via the network interface consumes significant amounts of energy.

In this thesis, we propose a novel query-aware 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. 1.1(e) and Fig. 1.1(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 values \((b1,c1)\) and \((b1,c2)\) are 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 generated 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 on 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 this issue, we use a novel 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. The proposed method also does not materialize the base relations in their entirety through the use of our memory-limited dictionary data structure.

Our query-aware 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 Open Database Connectivity (ODBC)/Java Database Connectivity (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. In fact, our method is sufficiently general to be applied on an arbitrary relation as long as a lossless-join decomposition of the relation is available.

**Contributions.** The contributions of this work are as follows.

- We design a novel query-aware 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 design a memory-limited dictionary data structure that bounds the memory requirement upfront thereby enabling a more space-efficient (de-)compression algorithms.

- We evaluate our compression method experimentally and demonstrate its effectiveness and efficiency.

- We provide an analytical model for estimating the compression ratio provided by the query-aware algorithm.

- We show how the same algorithm can be used to also exploit redundancy revealed by functional dependencies and conditional functional dependencies.

The rest of the thesis is organized as follows. Chapter 2 discusses previous work related to this thesis. Chapter 3 gives an overview of the query-aware compression and provides experimental data supporting its utility. Chapter 4 introduces techniques for running the query-aware algorithm using dictionaries with limited capacity, and covers experiments designed to highlight the effects of such. Chapter 5 develops an analytical model for the compression ratio provided by the query-aware algorithm, and introduces applications for
such a model. Chapter 6 shows how query-aware compression can exploit functional dependencies and conditional functional dependencies. Finally, Chapter 7 summarizes the findings of this thesis.
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 [20], arithmetic coding [33], and dictionary encoding such as the algorithms devised by Lempel and Ziv [34, 35]. 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. In the context of this work, 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. [30] 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 [10] proposes an algorithm based on Huffman codes designed for databases. Roth and Van Horn [29] propose a number of ways to apply compression algorithms in a relational database context. Ng and Ravishankar [26] provide an algorithm designed to work well with local decompression, allowing for common database operations to be performed on compressed data. Ray et. al [28] 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 [17] 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, 19, 27] 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. [16] and Bhattacharjee et al. [6] 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 [22] 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. [25] 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 [32] and JDBC [18] provide limited query result compression capabilities. They typically allow the user to specify one of the offline compression schemes (such as GZIP [12]) which is then applied to each record or a set of records before transmission. Chen and Seshadri [8] propose an algebraic compression framework for query results in the context of low-bandwidth 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. [15] 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.
CHAPTER 3
QUERY-AWARE COMPRESSION

This chapter introduces the query-aware compression algorithm. Section 3.1 gives an overview and examples of both compression and decompression of a sample result set. Section 3.2.1 defines and reports the results from experiments designed to test the performance of query-aware compression.

3.1 Algorithm

A schematic diagram of the compression and decompression process is outlined in Fig. 3.1. 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 a 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) op-
erators. 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 ($\sigma$ for selection, $\pi$ for projection, and $\Join$ for join). 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\footnote{A standard operator push-down rewriting algorithm can always be applied to ensure that condition.}.

A \textit{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. 3.2 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.
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. For the join tree in Fig. 3.2(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 they are added to a dictionary, new entries and their associated 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. 3.2(b), we will encode each row in the result set using the following sequence of dictionaries \( \langle D(A), D(B), D(R), D(C), D(S), D(j_1), D(D), D(Q') \rangle \).

### 3.1.1 Compression

The proposed query-aware 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.
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($T$), $\mathcal{D}$, $r$)

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.

**Compressing One Row.** The steps to compress one row of 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...
Algorithm 2 CompressRow(t, D, r)

Input: Join tree node t, collection of dictionaries D, a row r to be compressed
Output: Sends dictionary entries to decompressor, returns compressed row

1: rowcode ← ε
2: if t is a leaf then
3: for all columns c of relation t do
4: code ← ReverseLookup(D(c), r[c])
5: if code = ε then
6: code ← Push(D(c), r[c])
7: send DENTRY, c, r[c],
8: rowcode ← rowcode · code
9: else
10: leftcode ← CompressRow(t.left, D, r)
11: rightcode ← CompressRow(t.right, D, r)
12: rowcode ← leftcode · rightcode
13: if t is non-root then
14: code ← ReverseLookup(D(t), rowcode)
15: if code = ε then
16: code ← Push(D(t), rowcode)
17: send DENTRY, t, rowcode,
18: rowcode ← code
19: return rowcode

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
DE, D(A), a1,       DE, D(C), c2,
DE, D(B), b1,       DE, D(S), 1,
DE, D(R), 0, 0,     DE, D(j1), 0, 1,
DE, D(C), c1,       TF, 1, 0
DE, D(S), 0,
DE, D(j1), 0, 0,
DE, D(D), d1,
DE, D(Q'), 0,
TF, 0, 0

Figure 3.3: Logical encoding of the first two rows of the result set in Fig. 1.1(e) using the
join tree in Fig. 3.2(b). The symbol ‘DE’ denotes the dictionary entry flag, ‘TF’ denotes a
tuple fragment flag, ‘D(i)’ denotes the dictionary identifier for join tree node i.

with the current input join tree node (Line 13-18).

Example. Consider encoding (a1,b1,c1,d1), the first row of the result set in Fig. 1.1(e),
according to the join tree in Fig. 3.2(b). Logically, each field in the row is first encoded using
the column dictionaries and the encoding tuple fragments proceeds bottom-up (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 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 j1.
The fragment to be encoded by D(j1) is constructed from the previously encoded fragments
associated with the children of join tree node j1. Again, since D(j1) is initially empty, a new
entry is sent. Next, the fragment (d1) associated with node Q' is processed. As D(Q') is
empty, an entry is sent. Finally, the fragment for node j2 is constructed from the previously
encoded children fragments and transmitted without additional encoding. The sequence of
(logical) codes are shown in the first column of Fig. 3.3. The second row of the result set is
encoded similarly. The fragment (a1,b1) that is associated with node R is first processed
and D(R) already contains the required entry, nothing is transmitted. Next, the field ‘c2’
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 $m.flag = \text{DENTRY}$ then
13. PUSH($D(m.nodeID), m.value$)
14. else
15. $W \leftarrow W \cup \text{DECOMPRESSROW}(T, \mathcal{D}, m.value)$
16. return $W$

is encoded using $D(S)$ and since no entry for the value exists, the column dictionary $D(C)$ is first used to encode the value before using $D(S)$ resulting in two dictionary entries being sent. The fragment to be encoded at node $j_1$ is then constructed as ‘(0,1)’. Since the value is not currently in $D(j_1)$, a dictionary entry is sent. The next field to be encoded contains ‘d1’ which is already present in $D(Q')$ with code 0; hence no data is transmitted. Finally the entire row encoded as (1,0) at node $j_2$ is sent.

3.1.2 Decompression

The decoding process is symmetric. The decoder rebuilds the structure of the join tree from the stream, proceeds to populate dictionaries, and uses them to decode tuples.

Pseudo-code for the decompression of a result stream is shown in Algorithm 3. The procedure for decompressing a single row within the results stream is given in Algorithm 4. The latter uses a method called FORWARDLOOKUP($D, i$), which returns the value associated with codeword $i$ in $D$. 
Algorithm 4 \textsc{DecompressRow}(t, \mathcal{D}, e)

\textbf{Input:} Join tree node $t$, collection of dictionaries $\mathcal{D}$, a tuple of dictionary codes $e$

\textbf{Output:} Returns the decompressed row

1: \textbf{if} $t$ is a leaf node \textbf{then}
2: \hspace{1em} $row \leftarrow \emptyset$
3: \hspace{1em} \textbf{for all} columns $c$ in relation $t$ do
4: \hspace{2.5em} $row \leftarrow row \cup \text{\textsc{ForwardLookup}}(\mathcal{D}, e[c])$
5: \hspace{1em} \textbf{return} $row$
6: \textbf{else}
7: \hspace{1em} $l \leftarrow \text{\textsc{ForwardLookup}}(\mathcal{D}(t.left), e[0])$
8: \hspace{1em} $r \leftarrow \text{\textsc{ForwardLookup}}(\mathcal{D}(t.right), e[1])$
9: \hspace{1em} $row \leftarrow \text{\textsc{DecompressRow}}(t.left, \mathcal{D}, l)$
10: \hspace{1em} $row \leftarrow row \cup \text{\textsc{DecompressRow}}(t.right, \mathcal{D}, r)$
11: \textbf{return} $row$

3.2 Experiments

This section will present experiments designed to demonstrate the efficacy of query-aware compression. It is organized as follows. Section 3.2.1 defines metrics and describes the setup that will be used to produce the results in later sections. This experiment setup is used to produce the results discussed later in Sections 3.2.2 and 3.2.3, which report the compression ratios achieved and execution times, respectively.

3.2.1 Experiment Setup

\textbf{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.

\textbf{Dataset.} In each of the experiments to follow, we use the TPC-H [11] dataset. We use the provided \texttt{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

\[
\text{Compression Ratio} = \frac{\text{Raw data size in bytes}}{\text{Compressed data size in bytes}}
\]

Raw sizes are collected by writing results to disk in csv format and measuring the file size. Sizes reported for data compressed with query-aware compression include all necessary information to losslessly decompress it. 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 practical 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 [31, 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 by itself because our algorithm 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.

In the following experiments, dictionaries used by the QA compression algorithm are allowed to grow arbitrarily large. We measure the compression ratio for the results of each of the queries listed above across databases with varying sizes. The number of rows in each table is controlled using the scale factor argument for the dbgen program provided with the TPC-H toolset. Figure 3.4 shows how the result sets for each of these queries grow as the scale factor increases. For example, the results for Query 1 grow by approximately 40 GB when the scale factor is increased by one.

### 3.2.2 Compression Ratio

Compression ratios for each of the queries across multiple scale factors are shown in Figure 3.5. Notice that the compression ratio for QA+gzip remains constant as the scale factor grows. This is expected behavior. Each of the tables in the TPC-H dataset grow at a rate dictated by a linear multiplier applied to the scale factor, and any foreign key in a given table is repeated the same number of times as other values in the same column. As such, the ratio of exploitable redundancy with respect to the total size of the result set remains constant.
We can see that for most queries, gzip performs slightly better on smaller result sets. This due to the order in which the tuples appear in the result set. Pairs of tuples with significant amounts of redundancy do not necessarily appear close to one another. When the result set is small, this is not a problem as gzip uses enough memory to overcome the gap between these tuples. However, as the result set grows, the redundancy in these pairs of tuples becomes more difficult to exploit without using too much memory. So, when the result set becomes sufficiently large, we see the performance of gzip by itself drop.

In Chapter 4, we see that QA compression can suffer from this same effect if the capacities of its dictionaries are limited. However, since it is exploiting a deeper structure within the data, it can almost always maintain a compression ratio for longer given the same amount of memory.

### 3.2.3 Execution Time

To obtain the most accurate execution time metrics, we write results to a null device to avoid including overhead.
Figure 3.5: Compression ratios for various result set sizes.

Rather than measuring execution time at various scale factors, we first note that the number of rows processed per second remains roughly constant as the scale factor changes. This measurement for each of the six queries is shown in Fig. 3.6. The number reported is the mean across ten repetitions. We depict ±1 standard deviation with the whiskers centered at the mean for each bar.

Note that gzip by itself is faster than QA+gzip. This is not surprising. The execution time of QA is affected by things other than the size of the result stream it is processing. For example, a query involving five joins will take significantly more time to process than a query involving only two joins. In the case of transmitting the result stream over a network, we note that the process will still be IO-bound. This is apparent after noting that 15,000 rows/second with approximately 100 bytes per row would require an 11.4 Mbps transfer rate to keep up.
Figure 3.6: The mean rows/second for each query at a fixed scale factor. ±1 standard deviation is shown with the whiskers centered at the mean.
CHAPTER 4
FIXED SIZE DICTIONARIES

So far we have described the proposed query-aware compression algorithm assuming that there is no limit on the size of the dictionaries. For large result sets with many distinct values, the memory consumed by the query-aware algorithm can 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 $D(R)$ would be as large as the number of rows in $R$. For very large relations, the dictionary may not even fit in memory! For the proposed query-aware 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. Note that 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 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. This comes with the disadvantage of having to retransfer previously evicted entries each time they occur, but it allows the user to limit how much
memory the algorithm consumes. As long as the dictionary at the decompressor behaves
in exactly the same way, decoding is possible and correct. Figure 4.1 provides an example
of how an eviction might look. If the compressor and decompressor agree to, for example,
evict the oldest entry when pushing onto a full dictionary, they will stay in sync.

As with caching, several eviction policies are possible including least recently added
(LRA), least frequently used (LFU) etc. The LRA policy would evict the oldest dictionary
entry (see Fig. 4.2) and LFU would evict the least frequently used entry. Both of these poli-
cies 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.

The remainder of this chapter will formalize, analyze, and test different methods for
maintaining fixed-size dictionaries. The later sections are organized as follows: Sec. 4.1
formalizes fixed-size dictionaries, and sets of fixed-size dictionaries. In Sec. 4.2, we introduce
and analyze various techniques for maintaining sets of fixed-size dictionaries. Finally, in
Sec. 4.3, we analyze the effects of using fixed-size dictionaries in the query-aware algorithm.

4.1 Formalization of Fixed-Size Dictionaries

We first provide a more rigorous formalization of a dictionary in order to discuss various solutions. A dictionary is a data structure that maintains a mapping from index ⇔ value, and supports the following operations:

1. Push\((D, value)\): adds value to \(D\) and assigns it an index.
2. Contains\((D, value)\): checks if value is tracked by \(D\).
3. ForwardLookup\((D, i)\): returns the value associated with the index \(i\). This is also denoted with \(D[i]\).
4. ReverseLookup\((D, value)\): returns the index associated with value.

A **fixed-size dictionary** supports all of the above operations, but is also capable of keeping its size below an allotted capacity of \(N\) bytes. That is, the sum of the sizes of all of its values is \(\leq N\). In addition, it should also support the Resize\((D, N)\) operation, which does whatever is necessary to make its entries occupy less than \(N\) bytes of space.

4.1.1 Techniques For Maintaining Capacity

There are several obvious ways that allow a dictionary to keep its memory consumption under a given capacity.

**Refuse to add new entries.** When Push\((D, value)\) is called on a dictionary where adding value would cause it to exceed its capacity, it can simply refuse to add it. This is sometimes an acceptable solution. In many cases, however, it is necessary to reference the dictionary entries by index. It is also possible that the dictionary be filled with relatively uncommon values and later refuses to add more frequent values.

**Compress entries.** A dictionary can compress its contents in order to fit more entries while maintaining its capacity. This is clearly not a permanent solution, however. The
space savings which compression offer are limited, and in most cases, even a compressed dictionary will require more room to fit more values.

**Evict entries.** The dictionary can opt to evict one or more of its entries to make room for a new one. There are several obvious metrics that can be used to identify which entries should be evicted:

1. Least recently added (LRA): evict oldest dictionary entries
2. Least recently used (LRU): evict the entry that has been least recently accessed.
3. Least frequently used (LFU): evict the entries that are accessed the fewest times
4. Random
5. Largest: evict entries requiring the most space

An **eviction policy** is a set of rules that deterministically evicts one or more entries to make room for a new entry. In the case of the random eviction policy, a pseudo-random number generator with an agreed upon seed can be used. A dictionary’s eviction policy can use any combination of these rules, but it is often the case that LRA is best.

For the remainder of this chapter, the focus will be on fixed-size dictionaries that maintain their capacities by evicting entries.

### 4.1.2 Sets of Fixed-Size Dictionaries

Notice that query-aware compression makes use of a set of $M$ dictionaries. In order to address physical limitations, it is necessary to limit the memory consumption of the entire set to $N$ bytes. Often times, the $M$ dictionaries have differing space requirements. For example, one dictionary might use 12KB, while another dictionary grows to 2MB. For this reason, it is often not obvious how to optimally divide a capacity of $N$ bytes across the $M$ dictionaries without application-specific knowledge.

In the following section, we introduce three application-agnostic algorithms for partitioning a shared capacity across a set of dictionaries.
4.2 Partitioning Algorithms

We discuss three techniques for partitioning a capacity of $N$ bytes across a set of $M$ dictionaries. The first evenly divides the capacity across all of the dictionaries. The second allows dictionaries to behave normally until the capacity is reached. The final algorithm attempts to further optimize the partitions by detecting and reallocating wasted space.

4.2.1 Naïve

The naïve algorithm, when given a capacity of $N$ bytes, assigns each of the $M$ dictionaries $N/M$ bytes. This has the advantage of being simple and fast. However, it clearly has potential to introduce waste. If some dictionaries require less than $N/M$ space, and some require more than $N/M$ space, space allocated to the smaller dictionaries will go to waste.

In addition, some dictionaries that appear to require significantly more space might not need it. Namely, if values entered into the dictionary have good temporal locality, a large dictionary is unnecessary, even if it sees many unique values. For example, if a value is pushed onto a dictionary, referenced a few times soon after, but never again, the dictionary does not need to retain values for very long. It can evict even relatively recent values without incurring any penalty to compression ratio due to one value being entered more than once.

4.2.2 Pooled Capacity

Rather than naively assuming that each dictionary will have equal demand for capacity, we can allow all dictionaries to grow until their total memory consumption approaches a capacity of $N$ bytes. At this point, we limit each dictionary to its current size.

This approach is better than the naïve algorithm in that it prevents very small dictionaries from reserving capacity that they don’t need. However, it still fails to account for temporal locality.
4.2.3 Adaptive Repartitioning

This approach allows us to look for signs of temporal locality at the cost of maintaining additional statistics for each dictionary entry. It uses the same technique outlined in the previous section, but when the capacity is reached, it attempts to find and reallocate waste within each dictionary. We define waste to be any entry that will not or is unlikely to be referenced later in the result stream. For example, dictionaries that maintain many distinct values, but whose entries have good temporal locality, are likely to have significant waste.

**Detecting Waste.** In the case of streaming algorithms, it is impossible to determine whether or not a given value will be needed again later in the stream. As such, we must resort to using heuristics. The waste detection heuristic described in this section is application-agnostic, and requires $O(N/\beta)$ memory, where $\beta$ is the average size of the values the dictionaries maintain. It is capable of detecting temporal locality within the data contained in a dictionary.

This heuristic requires that dictionaries maintain the following statistics for each entry:

1. **Order of insertion.** This can be achieved by using, for example, a queue. Entries are pushed onto the queue when they are introduced into the dictionary, and popped when they are evicted.

2. **Access counts.** Each time an entry $e = (value, index)$ is referenced, either by \textsc{ForwardLookup}(D, index), or by \textsc{ReverseLookup}(D, value), the access count for $e$, denoted $c(e)$, is incremented by one.

We use these statistics to detect temporal locality as follows: if the access counts for older entries are not significantly higher than the access counts for more recent entries, then it is likely that the older entries are no longer being used. We can be even more certain of this if the $k$ oldest values have roughly the same access counts.

Figure 4.3 shows a visualization of a dictionary that appears to have waste. Notice that the access count for $D[0]$ is the same as the access counts for entries $D[1]$ and $D[2]$. If
Algorithm 5 \textsc{Find-Waste}(D, \alpha, \gamma)

\textbf{Input:} Dictionary \(D\), max rel-distance \(\alpha\), max fraction of entries considered waste \(\gamma\)

\textbf{Output:} Index of most recent index classified as waste

\begin{verbatim}
a ← c(\text{Oldest}(D, 0))
i ← 1
while true do
  b ← c(\text{Oldest}(D, i))
  if \((a - b)/a > \alpha\) or \(i \geq \gamma \cdot \text{Num-Entries}(D)\) then
    return \(i\)
  end
  i ← i + 1
end
\end{verbatim}

Reallocating Waste. After finding it, we must reallocate the space occupied by waste. We now formalize an algorithm used to reallocate capacity used by entries classified as waste.

In the worst case, for a dictionary of capacity \(K\), this algorithm is \(\Theta(K)\). However, since it already assumes access counts will be roughly increasing, we can substitute binary search for the linear search. This makes the algorithm \(\Theta(\log K)\).

Figure 4.3: Visualization of a dictionary. Each entry \(e = (\text{value}, \text{index})\) is depicted showing its index in the first row, value in the second. \(c(e)\) denotes the number of times \(e\) has been accessed.

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
\text{e.index} & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
\text{e.value} & a & b & c & d & e & f \\
\hline
\text{c(e)} & 10 & 10 & 10 & 8 & 5 & 4 \\
\hline
\end{tabular}
\end{center}

the structure of the data stream is relatively constant over time, this fact suggests that \(D\) probably needs no more than four entries.

Algorithm 5 details the simple waste detection strategy detailed above. Given a threshold \(\alpha\), it finds \(e_i\), the newest entry having relative-distance\((e_0, e_i) < \alpha\). An additional parameter \(\gamma\) limits the maximum number of entries classified as waste. We denote the \(n\)th oldest entry in a dictionary \(D\) with \text{Oldest}(D, n).

In the worst case, for a dictionary of capacity \(K\), this algorithm is \(\Theta(K)\). However, since it already assumes access counts will be roughly increasing, we can substitute binary search for the linear search. This makes the algorithm \(\Theta(\log K)\).
allotment it actually needs. In particular, a dictionary \( D_i \)'s bid, \( b(D_i) \), should be:

\[
b(D_i) = \text{used capacity within } D_i - \# \text{ bytes wasted in } D_i.
\]

The number of bytes wasted can be found by summing the size of each entry classified as waste.

After each dictionary makes its bid, it is assigned a new capacity proportional to the fraction of its bid over the sum of all bids. In particular, if \( b(D_i) \) denotes \( D_i \)'s bid, its new capacity will be:

\[
D_i \text{'s new capacity} = N \cdot \frac{b(D_i)}{\sum_{j=1}^{M} b(D_j)},
\]

where \( N \) denotes the total capacity allotted to the \( M \) dictionaries. Any dictionary exceeding its new capacity is forced to evict entries until it no longer does.

4.2.4 Runtime Analysis

The naive and pooled capacity algorithms are both \( \Theta(M) \), for a set containing \( M \) dictionaries.

Alg. 5 depicts a \( \Theta(K) \) algorithm is given to find the number of wasted bytes in a dictionary with capacity \( K \). Since this is the only \( \omega(1) \) process for computing \( b(D_i) \), it follows that collecting the bid for a single dictionary \( D_i \) is \( \Theta(K_i) \), and that collecting all of the bids is \( \Theta(N) \), since \( N = \sum K_i \).

After the bids are collected, it is possible that some dictionaries are resized. If \( \text{Resize}(D, N) \) grows the capacity of \( D \), it is \( \Theta(1) \) since the only operation to be done is to update the capacity for \( D \). However, if it shrinks the capacity of \( D \), it is possible that evictions will be necessary. The maximum number of evictions for any given dictionary is \( \gamma K \). The amount of time a single eviction takes depends on implementation, but if an array is used to facilitate \( \text{ForwardLookup} \), removing an entry from the forward dictionary takes \( \Theta(1) \)
time. If a hash table is used for calls to ReverseLookup, there are $\Theta(1)$ operations in the average case. Therefore, shrinking the dictionary is $\Theta(\gamma K)$ in the average case. Since the total capacities of the dictionaries cannot exceed $N$, repartitioning all of the dictionaries is $\Theta(\gamma N)$ in the average case. To alleviate this cost, the user can repartition the dictionaries long before their total use reaches $N$ bytes. If this is done when the dictionaries have a total use of $C$ bytes, the average case will be $\Theta(M \cdot \gamma C)$. Since collecting the bids is $\Theta(N)$ and applying the partitions using Resize is $\Theta(\gamma N)$, we can conclude that this algorithm is $\Theta(N)$ in the average case.

4.2.5 Empirical Analysis

In this section, we explore the efficacy of this approach by comparing naïve partitioning to adaptive repartitioning. When applying adaptive repartitioning to QA compression, there are two relevant measurements considered:

1. the amount of overhead adaptive repartitioning introduces to the execution time
2. the compression ratio

The expectation is a slight increase in execution time, but an improvement in the compression ratio.

Experiment Setup

QA compression is applied to a five-way join on the TPC-H [11] dataset. As before, controlling the size of the result set is achieved by varying the scale factor from 0.01 (resulting in a 13MB database and a 40MB result set), to 0.19 (resulting in a 260MB database and a 760MB result set).

In four separate experiments, the set of fixed-size dictionaries are assigned capacities of
50KB to 200KB in increments of 50KB. As before, compression ratio is measured as:

\[
\frac{\text{# of bytes in original result set}}{\text{# of bytes in compressed result set}}
\]

Three measurements are taken for each set of parameters:

1. Size of result set when compressed using gzip
2. Size of result set when compressed using QA compression using naïve partitioning, followed by gzip (on the plots, “qa + gzip (even)”)  
3. Size of result set when compressed using QA compression using adaptive repartitioning, followed by gzip (on the plots, “qa + gzip (smart)”)  

Execution times are collected on a machine with an Intel® Core™ i7 processor and 8GB of RAM. In order to smooth out noise, each experiment is repeated five times. The reported execution times are the means of the wall clock execution times for each run.

Adaptive repartitioning is run once, after the dictionaries fill roughly 50% of the available capacity.

Results

**Compression Ratio.** Compression ratio results are reported in Figure 4.4. Using the definition of compression ratio above, higher is better. Notice that in each of these, naïve partitioning does exactly as well as adaptive repartitioning for smaller values of the scale factor. This is because \(N/M\) bytes per dictionary is enough to fit all of the unique entries for each dictionary across the entire data stream. Once the result set is large enough to exceed this demand, however, adaptive repartitioning helps to borrow space from small dictionaries and assign it to larger dictionaries. As this results in fewer re-reference misses for the larger dictionaries, we see an improvement to the compression ratio.

**Execution Time.** Results are shown in Figure 4.5. Because the adaptive repartitioning
Figure 4.4: Compression ratio results

algorithm is only run once, the execution time difference between it and the naïve partitioning is very minimal. Adaptive repartitioning takes marginally more time – on the order of a second or two – regardless of the size of the capacities tested in these experiments. This suggests that very few of the worst case $\Theta(K)$ evictions are being done for each dictionary. Most of what is being done is taking completely unused (rather than wasted) capacity from small dictionaries and assigning it to larger dictionaries, which is $\Theta(1)$. The small amount of overhead is due to evicting wasted entries in large dictionaries with good temporal locality.
4.3 Effects on Query-Aware Algorithm Performance

In Sec. 3.2, we saw how query-aware compression performed when the size of its dictionary set was not constrained. In this section, we execute the same experiments with dictionary sets of limited size. We use the same data and methodology described in Sec. 3.2.1, but add a new dimension: the size of the dictionary set. In addition, we use the adaptive repartitioning algorithm introduced in Sec. 4.2.3.

For simplicity, dictionaries are limited by number of items they contain rather than by space occupied by their contents. The same effects are observed regardless of which technique is used, with, for example, 50K entries generally being slightly less limiting than a total capacity limit of 50KB.

Figure 4.5: Execution time results
Figure 4.6: Compression ratios with dictionary sizes fixed at 50K and scale factor fixed at 0.21.

4.3.1 Performance Summary

We begin by examining the compression ratios for each of the queries using fixed values for both result and dictionary sizes. The dictionary set used by QA compression is limited to 50K entries, and the scale factor is fixed at 0.21. Results are shown in Figure 4.6. 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.

QA compression for Query 1 exhibits relatively poor performance, but still outperforms gzip by itself. Notice that Query 1 exclusively involves tables with relatively large growth factors (as outlined in the TPC-H schema). Most of the remaining queries involve smaller tables such as Nation, which QA compression is better at exploiting. This is because each tuple in the result set of one of these queries references one of a very small number of rows. This is not the case when joining exclusively on large tables.
4.3.2 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 4.7 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 4.8 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.

4.3.3 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 4.9 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
Figure 4.7: Compression ratios with dictionary sizes fixed at 10K entries.

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.

4.3.4 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™ i7 processor and 8 GB of 1333 MHz DDR3 RAM.

Results for Query 5 (the most complicated query we tried) are shown in Figure 4.10. Results for Query 2 are shown in Figure 4.11. 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 constant amount of work for each tuple for a given join tree. So, in a stream containing $n$ tuples, our algorithm performs $O(n)$ work. Therefore, should see an increase in the linear growth rate for execution time.

## 4.4 Conclusions

In this chapter, we introduced fixed-size dictionaries, which are used by the query-aware algorithm to limit its memory consumption. We discussed the challenge of optimally dividing a memory limitation of $N$ bytes across $M$ dictionaries having different space requirements.
In an effort to address this problem, we provided several application-agnostic techniques for partitioning a given capacity across a set of dictionaries. We showed how the query-aware algorithm can use fixed-size dictionaries by evicting older entries to make room for newer ones. Empirical analysis showed mostly expected results, with larger dictionaries generally providing better compression ratios than smaller ones. In addition, we noticed that dictionaries large enough to avoid evicting entries that will be used later in the result stream usually provide the best compression ratio. We see that even for result sets in excess of 1GB in size, dictionary sets restricted to as few as 50K entries are large enough to produce optimal compression ratios.
Figure 4.10: Execution time (in seconds) for Query 5.

Figure 4.11: Execution time (in seconds) for Query 2.
When analyzing preliminary results, we noticed that the compressed result sizes would vary when changing the order in which the tables were joined, and by the left-right relationship between nodes of the join tree. With the database engine we used, the left-right relationship is determined by the structure of the query. For example, \texttt{SELECT * FROM a, b WHERE a.c = b.c} produces different results than \texttt{SELECT * FROM b, a WHERE b.c = a.c}. The only difference in the raw results was the ordering of the tuples. Compression algorithms tend to be 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.

In this chapter, we investigate the effects join order has on compression performance, and introduce a method for approximating the compression ratio provided by the QA algorithm given a join tree. Section 5.1 summarizes the effects join order has on compression ratios provided by the QA algorithm when applied to the TPC-H dataset. Sections 5.2 and 5.3 introduce methods for estimating the performance of the QA algorithm when using unbounded and fixed-size dictionaries, respectively. Finally, Section 5.4 shows how to use these models to select an optimal join order.

### 5.1 Effects of Join Order on Compression Ratio

In this section, we examine the effect join order has on compression ratios provided by the query-aware algorithm when applied to the TPC-H dataset. Figure 5.2 shows the result of generating all possible ways to order the tables in Query 2. 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 ✕ partsupp) or (supplier ✕ nation) is on the left, and which is on the right. Applying query-aware compression before gzip in this case makes up for the lost performance in gzip. In particular, join order has nearly no effect on the compression ratio achieved by QA+gzip. The cause for this is subtle: one favors an ordering of tuples having redundant fragments in close proximity. The other join order has this redundancy spaced far enough apart in the result set that gzip is unable to exploit it. Query-aware compression is able to pick up the slack by removing the redundancy through use of dictionaries.

Figure 5.3 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 any 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 serendipitous 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.

From this, we can see that with sufficiently large dictionaries, order does not have a significant effect on 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 5.2 evidences this especially well.

We note, however, that it is possible for join order to affect the compression ratio even when the size of a dictionary set is unbounded. For example, consider the relations $AB = R(A, B)$, $BCD = R(B, C, D)$, and $CE = R(C, E)$. There are two possible join trees producing the same relation $R(A, B, C, D, E)$. These join trees are determined by the join order. The join trees associated with each ordering are shown in Fig. 5.1. If we know that
while \(|AB| \ll |CE|\) while \(|BCDE| \approx |ABCD|\), then the compression ratio provided by the QA algorithm applied to the join tree in Fig. 5.1(a) will be better than that when applied to the join tree in Fig. 5.1(b). This is the case because the dictionary maintained for \(AB\) is smaller than the one maintained for \(CE\), meaning the sizes of the indices are also smaller. Since we do not maintain a dictionary for the root of the join tree, it is in our best interest to minimize the size of indices used by its children’s dictionaries.

Although performance does depend on join order even when dictionaries are unbounded, it is important to note that the extent to which it does is far greater when using fixed-size dictionaries.

![Join tree diagrams](image)

Figure 5.1: The two possible join orders for the relations \(AB\), \(BCD\), and \(CE\).

In the next section, we introduce a means for estimating the compression ratio provided by the QA algorithm given a join tree and statistics about the tables represented by its nodes. Using this, we can choose the best join order.

### 5.2 Estimating Compression Ratio with Unbounded Dictionaries

Using count statistics for each node in the tree and each of the attributes in the resulting result set, we can produce an approximation for the number of bits it takes to transfer the compressed data. The following assumptions are made:

1. Indices for a dictionary containing \(n\) entries are represented with \([\log n]\) bits on average. Note that this assumption is very likely invalid when using Huffman codes.
2. When pushing an item onto the dictionary, a dictionary identifier is transmitted along with the value being pushed, occupying \( \log N \) bits, where \( N \) is the total number of dictionaries used.

3. The number of entries for each dictionary is unbounded.

To start with, the average size of an index for a dictionary representing some collection of values \( X \) is given in Equation 5.1.

\[
\text{IndexSize}(X) = \lceil \log(\# \text{ distinct elements in } X) \rceil
\]  

(5.1)

Using this, the average size of a value contained within a dictionary used by a join tree node \( n \) is given in Equation 5.2.

\[
\text{ValueSize}(n) = \begin{cases} 
\text{IndexSize}(n.left.relation) \\
+ \text{IndexSize}(n.right.relation) & : \text{if } n \text{ is an inner node} \\
\sum_{a \in n.relation.attributes} \text{IndexSize}(n.relation[a]) & : \text{if } n \text{ is a leaf}
\end{cases}
\]

(5.2)
Using these two equations, it is possible to measure the total overhead introduced by transferring dictionary information for a join tree node and its entire subtree. This includes indices and dictionary identifiers. Equation 5.3 shows how to compute this value. Here, $K$ refers to the number of dictionaries used by the algorithm, and $n$.distinctCount refers to the distinct number of entries for the dictionary representing a join tree node $n$.

\[
\text{DictionarySize}(n) = \begin{cases} 
  n.\text{distinctCount} \cdot (\text{IndexSize}(K) + \text{ValueSize}(n)) \\
  + \text{DictionarySize}(n.\text{left}) \\
  + \text{DictionarySize}(n.\text{right}) & : \text{if } n \text{ is inner} \\
  n.\text{distinctCount} \cdot (\text{IndexSize}(K) + \text{ValueSize}(n)) & : \text{if } n \text{ is leaf} 
\end{cases}
\]

In one last prerequisite to approximate the total size of compressed data, Equation 5.4 shows how to compute the number of bytes required to transmit dictionary data for each of the attributes in the result set. This takes as input an attribute $A$ from the result set, and requires access to the number of distinct values for this attribute, as well as the average size.
across the distinct values.

\[
\text{AttributeSize}(A) = \text{IndexSize}(A) \cdot A.\text{distinctCount} + A.\text{distinctCount} \cdot A.\text{avgDistinctSize}
\] (5.4)

Finally, Equation 5.5 shows how to compute an approximation for the total number of bits in the compressed data. Recall that each leaf node maintains a separate dictionary for each one of its attributes. Since the recursion in Equation 5.3 ends at the leaves, we must add the bits transmitted due to attribute dictionaries in the last term of CompressedSize.

\[
\text{CompressedSize}(n) = n.\text{count} \cdot \text{ValueSize}(n) + \text{DictionarySize}(n.\text{left}) + \text{DictionarySize}(n.\text{right}) + \sum_{a \in n.\text{relation.attributes}} \text{AttributeSize}(a)
\] (5.5)

Notice that since we do not maintain a dictionary for the root of the join tree, we transmit a value for each tuple, resulting in \(\text{root.\text{count}} \cdot \text{ValueSize}(\text{root})\) bits.

This function requires detailed information about the schema it acts on. Although these statistics are expensive to calculate, they can be cached and maintained cheaply. Computing it exactly with a single pass over the data takes \(O(nk)\) time, where \(n\) is the number of tuples in the result set, and \(k\) is the number of nodes in the join tree. This would also require \(O(nk)\) memory if an exact answer is required. If the required statistics can be accessed in \(O(1)\) time, then this computation becomes \(O(k)\).

If the user is willing to sacrifice some accuracy to improve memory requirements, HyperLogLog [14] or a similar technique can be used to approximate distinct counts using very little memory. This approach in particular allows the estimation of cardinalities of more than \(10^9\) using as little as 1.5 KB memory with standard error of less than 2%.
To measure the utility of query-aware compression, we consider Equation 5.6, which gives the total number of bits in the uncompressed result set $r$.

$$\text{UncompressedSize}(r) = \sum_{a \in r.\text{attributes}} (a.\text{avgSize} \cdot a.\text{count}) \quad (5.6)$$

Using equations 5.6 and 5.5, we can approximate the compression ratio provided by query-aware compression given the root of a join tree:

$$\text{CompressionRatio}(n) = \frac{\text{UncompressedSize}(n.\text{relation})}{\text{CompressedSize}(n)} \quad (5.7)$$

**Verifying Correctness of Size Approximation.** In this section, we empirically verify that the previously introduced model is accurate. For scale factors varying from 0.01 to 0.41, we first estimate the compressed size for the result set of Query 6 using the analytic, and then measure the actual compressed result size. At each measurement, dictionaries are allowed to grow arbitrarily large. The results are shown in Figure 5.4.

Notice that the predicted and actual sizes are not exactly the same. This is due to differences in the ideal implementation assumed in the approximation and the suboptimal implementation used to measure the actual compressed size. In particular, the ideal implementation could use exactly $\lceil \log N \rceil$ bits to represent an index for a dictionary with $N$ items. The implementation used to measure the actual compressed size uses variable-length integer encoding (VLE), which introduces a fair amount of waste for the following reasons:

1. The first bit from each byte is used to indicate whether or not another byte will follow, so $1/8$th of index bits are overhead.

2. Only supports byte-length indexes. So an index that requires 8 bits needs two full bytes, since the first bit in each byte is overhead, as mentioned previously.

As the number of tuples in the result set grows, the amount of space saved by not having the overhead of VLE grows. We see that in Figure 5.5, this is the general trend. The upward
spikes occur when a frequently used index needs to introduce another bit to account for all of its entries. VLE allows for indices representable with $7n$ bits to fit into $n$ bytes, regardless of how many entries are in the dictionary. For example, the indices for the first $2^7$ entries only require one byte. VLE pays for the introduction of new index bytes slowly since its older entries can still be referenced using the same number of bytes. In contrast, the ideal implementation assumed by the approximation requires that all of the indices in a dictionary of size $N$ be expressed using $\lceil \log N \rceil$ bits.

### 5.3 Estimating Compression Ratio When Using Fixed-Size Dictionaries

When a dictionary’s size is fixed, many of its entries might be evicted and re-entered multiple times. This can result in the model set forth previously to severely overestimate compression
Figure 5.5: Ratio of predicted compressed size and actual compressed size.

ratio. To address this, it is necessary to estimate the cost incurred by re-entering values that were previously evicted at least once.

Notice that fixed-size dictionaries behave like caches. Encountering an entry that is not yet in the dictionary is analogous to a cache miss. Cache behavior is well studied. Liu et al. [24] developed a model for the expected number of cache misses over time. We apply their analysis here to create model for compression performance when using fixed-size dictionaries.

For convenience, we refer to the event occurring when a value must be re-entered into its dictionary after previously being evicted as a re-reference miss. If we can measure the expected number of re-reference misses, as well as the average cost of a re-reference miss, we can update the model developed in the previous section to generalize for fixed-size dictionaries.
In order to approximate the number of re-reference misses, we need to find the following two functions. With each of these, the time dimension refers to the number of dictionary accesses. That is, \( t = 1 \) occurs after one dictionary access, \( t = 11 \) occurs after 11 dictionary accesses, and so on.

1. \( P(t) \) – the probability that a particular slot in the dictionary has been evicted by time \( t \).

2. \( R(t) \) – the average number of times an item in the dictionary is re-referenced at \( t \) accesses since its last occurrence.

Given these two things, we can calculate \( M(t) = P(t) \cdot R(t) \), the expected number of re-reference misses at time \( t \). If \( R(t) \) and \( P(t) \) are continuous, we can find the expected number of re-reference misses:

\[
M = \int_{0}^{\infty} M(t) \, dt.
\]

We first consider \( P(t) \), which is fairly easy to model discretely. Suppose that the average number of accesses between two misses is \( \beta \). If we assume that the usage of the dictionary is uniform, then the probability that a particular slot has its entry evicted after one access is:

\[
P = \frac{1}{\beta N},
\]

where \( N \) is the number of slots in the dictionary. Then the probability that it is not evicted after a single access is

\[
\overline{P} = 1 - \frac{1}{\beta N}.
\]

This allows us to estimate the probability that a given slot experiences an eviction after \( t \)
accesses:

\[
P(t) = 1 - (\bar{P})^t = 1 - \left(1 - \frac{1}{\beta N}\right)^t \\ \approx 1 - e^{-\frac{t}{\beta N}}.
\]

We use the approximation in Eqn. 5.10 because the integration is much more feasible if we do. This follows from the fact that

\[
e^{-1} = \lim_{n \to \infty} (1 - 1/n)^n,
\]

which means that when \( n \) is sufficiently large,

\[
e^{-k} \approx (1 - 1/n)^{kn}.
\]

\( R(t) \) is data-dependent, and since query-aware compression makes use of many dictionaries, it is likely that we will observe many different behaviors. We collect empirical data in order to test this hypothesis and hopefully fit \( R(t) \) to a model.

We find that whenever there is a pattern in the re-reference behavior, the data fit a negative exponential model. That is, \( R(t) = e^{at+b} \), with \( a < 0 \). An Example is shown in Fig. 5.6. Notice that in the case of Fig. 5.6(b), the re-reference rate is flat across all time. This indicates that the tuples are ordered randomly with respect to the values represented by these dictionaries. In contrast, the three remaining models indicate that the values represented by their dictionaries tend to occur close together in the result stream.

Note that in each of these re-reference plots, we observe equivalent behavior for some subsets of dictionaries. That is, there are often several dictionaries that have exactly the same re-reference patterns. These equivalence classes are to be expected, as, for example, a new value in a column dictionary implies a new value in a leaf dictionary, which implies
a new value in an index dictionary, and so on. We note that the re-reference patterns for a particular dictionary are often heavily influenced by join order, as this changes the order of the tuples in the result stream. Fig. 5.7 depicts the re-reference pattern for the same dictionary shown in Fig. 5.6(c) under a different join order.

Some queries and join orders yield relatively chaotic re-reference patterns. An example is shown in Fig. 5.8(a). Notice that in this case, the data do not even fit an exponential model. Even worse, it does not appear to have a consistent pattern. Small chunks of these data appear to fit linear models, but no single function will be a good model for this data.

It is important to note that even among the well behaved data, there are some dictionaries that have random re-reference patterns. This is to be expected. If this were the case that all dictionaries had exponential re-reference models, it would mean that the data are ordered in some way across all dimensions represented by each of the dictionaries, which is almost always impossible.

If we assume that $R(t)$ fits a negative exponential model, then we have that $R(t) = e^{ax+b}$, where $a < 0$. Given this, we can find $M(t)$, the function giving the expected number of re-reference misses at time $t$:

$$M(t) = P(t) \cdot R(t) = (1 - e^{-\frac{t}{\beta N}}) \cdot e^{at+b} = e^{at+b} - e^{at+b-\frac{t}{\beta N}}.$$  

This allows us to integrate over all time, giving us the expected number of re-reference
misses:

\[ M = \int_0^\infty M(t) \, dt \]

\[ = \int_0^\infty e^{at+b} \, dt - \int_0^\infty e^{at+b-\frac{t}{\beta N}} \, dt \]

\[ = \left[ \frac{e^{at+b}}{a} \right]_0^\infty - \left[ \frac{\beta Ne^{at+b-\frac{t}{\beta N}}}{a\beta N - 1} \right]_0^\infty \]

\[ = \left[ 0 - \frac{e^b}{a} \right] - \left[ 0 - \frac{e^b}{a - \frac{1}{\beta N}} \right] \]

\[ = e^b \left( \frac{1}{a - \frac{1}{\beta N}} - \frac{1}{a} \right) \]

**Incorporating into compression ratio approximation.** Since each dictionary maintained by the algorithm can have different behavior, we must consider each one separately. Suppose that \( M_i \) indicates the number of re-reference misses for dictionary \( D_i \), and \( C_i \) indicates the average cost of a re-reference miss. Then the total re-reference cost across each of \( k \) dictionaries is:

\[ \sum_{i=1}^k (C_i \cdot M_i) . \]

\( C_i \) depends on the type of dictionary \( D_i \) is. If \( D_i \) is a inner or leaf dictionary, the cost is ValueSize(\( n_i \)), where \( n_i \) is the join tree node associated with \( D_i \). The value of \( C_i \) if \( D_i \) is a column dictionary is just the average size of a value in that column. Given this, we can develop an additional function:

\[ \text{ReRefCost}(n) = \begin{cases} 
\text{\( n\).attribute.avgSize } & \text{if } D_{n.index} \text{ is a column dictionary} \\
\text{ValueSize}(n) & \text{otherwise} 
\end{cases} \]

We can then modify DictionarySize(\( n \)) to include this, which will complete our modifications.
Problems with this model. In order for this model to be used, a number of parameters must be known. In particular:

1. $a$ and $b$, model parameters for the re-reference patterns.
2. $\beta$, the average time between misses
3. $N$, the dictionary capacity. This is usually given a priori.

Unfortunately, all of these parameters are heavily data-dependent, and do not generalize well across different datasets, or even different join orders. In addition, it is almost always the case that some dictionary re-reference patterns for a given result set will not fit an exponential model, which makes the above analysis inapplicable. Even worse, these parameters are not independent. For example, $\beta$ is positively correlated with $N$, but not in a predictable way. An example of this for a particular dictionary with varying values of $N$ is shown in Fig. 5.9. The degree to which $N$ affects $\beta$ is entirely dependent on data and its ordering.

Furthermore, in order to collect these parameters, it is necessary to – at the very least – sample the result set. The parameters $a$ and $b$ can be found by collecting re-reference data, which requires us to pass over the data with dictionaries large enough to capture the shape of the exponential function, assuming the exponential model fits. The parameter $\beta$ can be found in the same way, but since it is dependent on $N$, the value of $N$ must be known a priori.

5.4 Choosing an Optimal Join Order

In the previous sections, we saw models for predicting the compression ratio provided by the QA algorithm given a join tree. Using these models, we can choose a join tree from a
set of possible join trees that optimizes the performance approximation provided by these models.

Recall from the previous section that we either have to assume the size of the dictionary set used by the QA algorithm is unbounded, or that we have all of the parameters necessary for the model proposed for fixed-size dictionaries. In the case that neither of these assumptions can be met, we cannot make an informed decision on which join tree is optimal.

5.5 Conclusions

In this chapter, we show that the performance of the QA algorithm is sometimes dependent on the order in which we join the relations in a given query. We find that this is due to join order affecting the order in which tuples appear in the result set, which, in turn, affects the temporal locality of the entries maintained by the algorithm’s dictionaries. Sometimes, this can mean that similar tuples can be placed close together under some join orders, but far apart under others. When using fixed-size dictionaries, this means that values will have to stay in dictionaries longer before they are referenced again. We also note that even when using unbounded dictionaries, join order can have a small impact on compression ratio.

In an effort to estimate the performance of query-aware compression given a join order, we introduced analytical models for both unbounded and fixed-size dictionaries. Although these models allow us to evaluate join orders, we note that they both require access to detailed statistics about the result set to be compressed. These statistics might not be readily available. In the case of the parameters for the fixed-size model, they often require that we sample the result set. We conclude that it is in most cases impractical to approximate the performance provided by query-aware compression, or to use these models to choose an optimal join order.
Figure 5.6: Re-reference patterns for one join ordering of Query 5. Each of the above represents one equivalence class that applies to several related dictionaries.
Figure 5.7: The re-reference pattern for the same dictionary represented by Fig. 5.6(c) under a different join order.

Figure 5.8: Re-reference patterns for one join ordering of Query 2. Each of the above represents one equivalence class that applies to several related dictionaries.
Figure 5.9: A demonstration that $\beta$ is positively correlated with $N$. 
CHAPTER 6
FUNCTIONAL DEPENDENCIES

Functional dependencies (FDs) describe functional relationships between two sets of attributes in a relation $R$. In particular, a set of attributes $X$ functionally determines another set of attributes $Y$ if and only if each value of $X$ is associated with exactly one value of $Y$. This is usually denoted as $X \rightarrow Y$. For the sake of convenience, we refer to $X$ as the left-hand side (LHS), and $Y$ as the right-hand side (RHS) of the FD. For example, in a table detailing employee salaries at a large corporation, the attribute for social security number functionally determines the employee’s salary. The reverse is not true, however, since there may be multiple people who earn the same salary.

Functional dependencies can be used to further improve the performance of query-aware compression. In this chapter, we discuss how to modify the QA algorithm to exploit the redundancy revealed by functional dependencies. In Section 6.1, we discuss an algorithm for augmenting a join tree using FDs that apply to its leaves. In Section 6.2, we introduce a weaker form of functional dependencies that can be used to exploit further redundancy.

![Figure 6.1: A join tree augmented with artificial joins derived from the functional dependencies $X \rightarrow Y$ on $R$ and $A \rightarrow B$ on $Q'$.]
6.1 Join Tree Augmentation

Consider a functional dependency $X \rightarrow Y$ on a relation $R$. Then specifying the values for both $X$ and $Y$ in every row introduces avoidable redundancy, as the values for $X$ determine the values for $Y$. Instead, notice that we can treat the relation $R$ as the result of a join. If $R_{XY}$ is the relation formed by taking only the attributes in the sets $X$ and $Y$ from $R$, and $R_{Y}$ is the relation with all attributes but those in $Y$, then we can say $R = R_{Y} \Join_X R_{XY}$. Now that this relation is expressed in terms of a join, we can apply query-aware compression.

If we are given sets of FDs for each of the base relations represented by the leaves of a join tree, we can augment the provided join tree with artificial joins corresponding to each of the FDs. Consider the join tree depicted in Fig. 3.2(b). Suppose that we are given the FDs $X \rightarrow Y$ for the relation $R$, and $A \rightarrow B$ for $Q'$. The join tree augmented with these FDs is shown in Fig. 6.1. Given the FDs, the computational costs incurred by expanding the join tree are minimal. We simply interpret the result stream as if it were the product of two further joins – one to form $R$, and the other to form $Q'$.

In order to best exploit all information given by FDs, the augmented join tree we produce should have a join for each non-trivial FD we are aware of. For example, when given only a single FD for each leaf, we augment the join tree with one artificial join per leaf. However, if provided with a set of FDs applying to a single leaf, the choice of which FD to expand first can have a significant effect on the resulting join tree.

For example, consider the relation $R(A, B, C, D, E)$ with FDs $AB \rightarrow D$ and $C \rightarrow DE$. The join trees resulting from picking each one first are shown in Figure 6.2. Notice that if $AB \rightarrow D$ is chosen first, we can still expand $C \rightarrow DE$ because $C \rightarrow E$ still holds. However, if we reverse the order, then $AB \rightarrow D$ is no longer usable. This example demonstrates that the choice of which FDs to expand first can have a significant effect on the resulting join tree, and, by extension, the compression ratio provided by the QA algorithm.

In cases where the sets of attributes in the RHS of each pair of functional dependencies
The result of choosing $AB \to D$ first

The result of choosing $C \to DE$ first

Figure 6.2: Depiction of different outcomes based on which functional dependency is chosen first. Note that choosing $AB \to D$ first results in a schema with more joins.

are disjoint, the leaves and the shapes of the resulting join tree remain constant. Choosing a functional dependency $f$ results in the leaves associated with $f$ appearing closer to the root of the join tree.

We first consider a naïve algorithm which does not take this into account. A schema is in Boyce-Codd normal form (BCNF) if each of its relations $R_i$ satisfy the following property: when a FD $f : X \to Y$ holds on $R_i$, then either:

1. $f$ is trivial – that is, $Y \subseteq X$, or

2. $X$ functionally determines all attributes in $R_i$, that is, $X$ is a superkey of $R_i$.

A schema that is in BCNF is free of any redundancy due to functional dependencies. The BCNF decomposition algorithm [9], shown in Alg. 6, takes as input a relation, and outputs a list of sub-relations that all satisfy BCNF. This is easily adaptable to an algorithm that produces a join tree rather than a list of relations. Assume that we have an oracle function 

\textsc{ChooseBestFD} that returns an FD that, when expanded, does not inhibit the production of the optimal join tree. Alg. 7 shows how to use this oracle function to produce a join tree for a relation given as input.

Finally, we can use Alg. 7 to augment a provided join tree. We do this by expanding each leaf into its own sub-join tree. This procedure is summarized in Alg. 8. The question

\footnote{$X^+$ is the \textit{closure} of $X$, meaning the set of all attributes determined by $X$. For example, if $A \to B$, $B \to C$, then $(A)^+ = \{A, B, C\}$.}
Algorithm 6 BCNFDecompose($R$)

Note: `NEWRELATION(S)` returns a new relation with attributes $S$.

**Input:** relation $R$

**Output:** set of sub-relations satisfying BCNF

1. $S \leftarrow \{R\}$
2. while there exists a $R_i \in S$ not in BCNF do
3.   if there is a nontrivial $f : X \rightarrow B$ on $R_i$ then
4.     $R_1 \leftarrow \text{NEWRELATION}(X^+) \quad$\footnote{1}
5.     $R_2 \leftarrow \text{NEWRELATION}((R_i\text{.attributes} - (X^+ - X)))$
6.     $S \leftarrow (S - R_i) \cup \{R_1, R_2\}$
7. return $S$

Algorithm 7 BuildBCNFJoinTree($R$)

**Input:** relation $R$

**Output:** root of join tree formed by BCNF decomposition of $R$

1. $f \leftarrow \text{CHOOSEBESTFD}(R)$
2. if $f \neq \text{nil}$ then
   1. $X \leftarrow f.lhs$, $B \leftarrow f.rhs$
   2. $n \leftarrow \text{NEWJOINNODE}(R)$
   3. $R_1 \leftarrow \text{NEWRELATION}(X^+)$
   4. $R_2 \leftarrow \text{NEWRELATION}((R - (X^+ - X)))$
   5. $n.left \leftarrow \text{BUILDBCNFJOINTREE}(R_1)$
   6. $n.right \leftarrow \text{BUILDBCNFJOINTREE}(R_2)$
   7. return $n$
else
   1. return $\text{NEW-JOIN-TREE-NODE}(R)$

of how to implement CHOOSE-BEST-FD still remains. We can use the following heuristics as guides:

1. The number of nodes in the join tree that will result is positively correlated with the compression ratio it provides. A join tree with $n$ nodes will likely provide a worse compression ratio than one with $m > n$ nodes. A greedy variant on this would be to maximize the number of available nontrivial FDs available in the children relations produced by an expansion.

2. We can use the analytical model for unbounded dictionaries developed in Sec. 5.2 if the number of possible join trees is small. Even when not using unbounded dictionaries, a join tree with a superior compression performance when using unbounded dictionaries
Algorithm 8: AugmentJoinTree($T, W$)

**Input:** join tree $T$, result set $W$

**Output:** join tree augmented with FD joins

```
for each leaf $t \in T$ do
  $t' \leftarrow$ BUILDBCNFJOINTree($W$.relation)
  $t$.left $\leftarrow$ $t'$.left
  $t$.right $\leftarrow$ $t'$.right
return $T$
```

is very likely to produce a superior compression ratio when using fixed-size dictionaries.

Now that the join tree has been adapted to exploit redundancy revealed by FDs, we can pass it to the QA algorithm without any additional work.

### 6.2 Constrained Functional Dependencies

Recall that a functional dependency $X \rightarrow Y$ exists on a relation $R$ if whenever values of $X$ in $R$ agree, values of $Y$ also agree. This is a strong condition, and will only arise in relations that are not in a normal form. In this section, we discuss a weaker form of FDs known as constrained functional dependencies (CFDs) [13].

CFDs are a generalization of FDs. In particular, we are allowed to impose constraints on values in the relation. When this constraint is met, the CFD behaves exactly like a regular FD. The behavior is undefined if the constraint is not met. In effect, a CFD is a FD that applies to the subset of a relation satisfying the constraint of the CFD. A CFD on a relation $R$ is denoted as follows:

\[
c \Rightarrow (X \rightarrow Y),
\]

where $c$ is some constraint on the values of tuples in $R$. When a tuple satisfies $c$, the functional dependency $X \rightarrow Y$ applies. For notational convenience, we refer to $c$ as the constraint of a CFD. As with functional dependencies, we respectively refer to $X$ and $Y$ as the *left-hand side* (LHS) and *right-hand side* (RHS) of a CFD.
(a) Join tree for when the CFD holds. The dotted edges indicate a join corresponding to the CFD.

(b) Join tree for when the CFD does not hold.

Figure 6.3: Two join trees for the same result set. The first should be used when the constraint in the CFD is met. The second is for when the constraint is not met.

For example, consider a relation containing customer information area code, phone number, and state. The state does not always determine the area code. Most states have multiple area codes to accommodate their large populations. Some states with smaller populations, however, have a single area code. A CFD describing this relationship might look like:

\[(\text{Population} < 1,000,000) \Rightarrow (\text{State} \rightarrow \text{AreaCode})\] (6.1)

Consider the case when we are given a single CFD. Then we can build two separate join trees: one for when the constraint applies, and one when it does not. In the former, the node corresponding to the CFD will have two children. In the later, the node will be a leaf. An example is shown in Fig. 6.3. Here, the result of the query is \(\text{R(Person, State, Population, AreaCode)}\), and we assume that the CFD shown in Eqn. 6.1 holds.

Notice that the two join trees only diverge in structure at the node \(\text{R(State, Population, AC)}\). In this case, we can use the same dictionaries for each of the join trees because it is possible to distinguish which dictionaries the values refer to without any further information. In particular, values in the dictionary represented by the node \(\text{R(State, Population, AC)}\) will have either two or three indices. If a value has three indices, we can assume it is from Fig. 6.3(b), as these will refer to values in the column dictionaries for State, Population,
and AC. However, if the value has only two indices, we can assume that it is from the join tree in Fig. 6.3(a), as they will refer to values in the dictionaries for R(State, Population) and R(State, AC). In this way, we can proceed with zero overhead.

**Definitions.** For notational convenience, we refer to a join tree without any augmentation from CFDs as a *base join tree*. A base join tree leaf node having one or more CFDs that apply to the relation it represents is called a *CFD node*. Furthermore, a child of a base node in a join tree augmented with joins induced by CFDs is called an *ancillary node*. We refer to a dictionary representing a CFD node or an ancillary node as a *mixed dictionary*. In a mixed dictionary, we refer to a value corresponding to a join tree where no CFDs apply, as in Fig. 6.3(b), as a *base value*. A value in a dictionary corresponding to an ancillary node is referred to as an *ancillary value*. Notice that base values will be \( n \)-tuples where \( n \) is the number of columns in the relation. In contrast, ancillary values will always be 2-tuples, where each of the indices in the tuple refer to entries in dictionaries corresponding to ancillary nodes.

Distinguishing which join tree a value in a mixed dictionary refers to by examining the number of elements in the tuple does not always work. Take, for example, a base join tree where one of its CFD nodes represents a relation having two CFDs. Based on which of the CFDs apply, we will use one of the four join trees shown in Fig. 6.4. While we will be able to distinguish base values from ancillary values, we will not be able to distinguish which join tree ancillary values correspond to. In particular, ancillary values corresponding to the join trees shown in Figures 6.4 (a), (b), and (c) will all have values in the dictionary for BCDE having two indices.

### 6.2.1 Algorithms for Exploiting CFDs

In this section, we propose three distinct approaches to exploiting CFDs. Given \( n \) CFDs, the first algorithm partitions the entire result set into \( 2^n \) sets and uses the appropriate join tree for each of them. The second introduces labels in mixed dictionaries in order to specify
Figure 6.4: A depiction of a join tree where the relation BCDE has two CFDs. The figures above show the resulting join tree when one, both, or neither of the CFDs apply which join tree the decompressor should use. The last algorithm introduces a new data structure and is usually capable of avoiding overhead.

**Partitioning Solution.** Consider a list of CFDs that apply to a result set $W$. We begin by partitioning $W$ into disjoint sets where exactly the same conditions apply to each tuple in the partition. For example, the first partition might contain all tuples where none of the conditions apply. The second might contain tuples where $c_0$ applies, but none of the rest do, and so on. Note that the same join tree can be used for each tuple in a given partition. Algorithm 9 details a simple way to assign a row to a partition based on which CFDs hold on it. Pseudocode for compression using CFDs is shown in Algorithm 10. For ease of exposition, we send the join tree to be used for each partition. In practice, we can send each of the CFDs at the beginning of the algorithm and instead send the identifiers for each partition. The decompressor can then reconstruct the join tree.

Note that this approach sacrifices the online nature of the QA algorithm. That is, using this algorithm makes QA compression a blocking algorithm. The results can no longer be streamed to the user. It also introduces a significant amount of overhead because if there are $n$ distinct CFDs, there are $O(2^n)$ partitions, with each partition having a distinct join tree. In addition, the raw result set must be buffered in memory or on disk, and the original
Algorithm 9 \textsc{GetPartitionID}(r, CFD_0, CFD_1, \ldots, CFD_n)

\textbf{Input:} Row from result set \( r \), list of CFDs that might apply to \( r \) CFD_i

\textbf{Output:} A string that identifies the partition \( r \) belongs in

1: \( id \leftarrow \epsilon \)
2: \textbf{for} \( i := 0 \) to \( n \) \textbf{do}
3: \quad \textbf{if} constraint \( c \) for CFD_i holds on \( r \) \textbf{then}
4: \quad \quad \( id \leftarrow id + '1' \)
5: \quad \textbf{else}
6: \quad \quad \( id \leftarrow id + '0' \)
7: \textbf{return} \( id \)

Algorithm 10 \textsc{OfflineCFDEncode}(T, W, CFD_0, CFD_1, \ldots, CFD_n)

\textbf{Input:} Base join tree \( T \), Result set \( W \), List of CFDs applying to \( W \) CFD_i

\textbf{Output:} Sends compressed result to decompressor

1: \( \mathcal{P} \leftarrow \text{InitMultiMap()} \)
2: \textbf{for all} row \( r \in W \) \textbf{do}
3: \quad \textsc{Push}(\( \mathcal{P}, \textsc{GetPartitionID}(r, \text{CFD}_0, \ldots, \text{CFD}_n), r \))
4: \textbf{for all} partition \( p \in \mathcal{P}.\text{keys} \) \textbf{do}
5: \quad \( T \leftarrow \textsc{AugmentJoinTree}(T, \mathcal{P}[p]) \)
6: \quad \text{send} \( |\mathcal{P}[p]| \), the number of rows in this partition
7: \text{Compress}(T, \mathcal{P}[p])

order of the tuples is lost.

\textbf{Naïve Labeling Solution.} Instead of expecting the decompressor to infer which dictionaries values refer to, we can provide labels that explicitly identify them. Each of the values in a dictionary associated with a CFD node are augmented with a label specifying which CFDs apply. This label can be omitted in the case where the appropriate join tree can be unambiguously be inferred by the decompressor. For example, if the relation represented by the leaf in question has four attributes, then any dictionary value with four indices will correspond to the join tree where none of the CFDs apply.

As an example, typical values in the dictionary corresponding with the node BCDE in the join trees depicted in Fig. 6.4 might appear as follows:
Here, a * next to a value denotes a join tree identifier. When we see 0*, for example, we assume that the indices to follow refer to dictionary entries for the nodes BC and CDE, respectively. Note that the first row, an ancillary value, is a 4-tuple. Without augmenting it with a join tree identifier, we would not be able to distinguish base values from ancillary values.

Because this approach is suboptimal and offers no benefits over either the previously or soon to be discussed algorithms, we omit pseudocode.

**CFD Maps and Dependency Resolving.** We propose a more efficient solution. Consider a base join tree with a CFD node. Rather than attempting to distinguish between base and ancillary values in a mixed dictionary, we opt to always use the base join tree, meaning mixed dictionaries will always contain base values. However, we maintain a new map data structure for each CFD to be used when the constraint holds. These maps will be keyed on column dictionary indices from the LHS of their corresponding CFD, and the values will be indices from the RHS of the CFD.

When encoding the value tuple for a CFD node, we first check if the constraints corresponding to its CFD holds. If it does, and we have not seen the elements in the LHS of the CFD, we push a key-value pair onto the CFD map, with the key being the LHS tuple, and the value being the RHS tuple. We can expect the decompressor to do the same thing. After this is done, we proceed normally with the encoding of the tuple. If the constraint holds and we have already seen the elements in the LHS, then we omit the RHS elements. We can do this because the CFD map maintains the relationship between the LHS and RHS elements. Before this approach is useful in the general case, there are several complications
that must be dealt with.

Before showing an example, we discuss the additional complications introduced when using fixed-size dictionaries. If we evict one of the column dictionary entries corresponding to either or LHS or a RHS value in a CFD map, then the entry in the CFD map is no longer valid. Worse, without additional computational overhead, there is no way of fixing this. To address this issue, we can take one of two approaches:

1. As it has been presented, the keys and values in a CFD map refer to indices in column dictionaries. If the indices in the column dictionary are removed or changed, the entries in the CFD map will be invalidated. Instead of using indices, we can just have the CFD map hold the values. This will eliminate this problem, but require significantly more memory.

2. Each column dictionary entry can maintain a list of pointers to CFD map entries that reference it. When an entry in a column dictionary is evicted, we also evict any entries in the CFD map that reference it. Before evicting the entry in the CFD map, we remove all references to it. This is complicated, but has significantly less overhead memory cost.

Although the first approach is more straightforward, it requires significantly more memory than the second approach, and, as a result, will also cause more evictions when using fixed-size dictionaries. This will negatively affect the compression ratio. In addition, since the values are larger, we also incur computational overhead due to longer lookup times in the CFD map. To avoid these problems, we take the second approach.

An example of the compression process for the result stream of a relation $R(A, B, C)$ with the CFD $(A \leq 3) \Rightarrow (A \rightarrow B)$ is shown in Fig. 6.5. When a value $x$ is pushed onto a dictionary, we use the notation $\Rightarrow \#n$ to indicate that the dictionary code assigned to $x$ is $n$. Note that each time an entry in the column dictionary for $A$ is evicted, we also evict any CFD entry that references it. This occurs at times $t = 9$ and $t = 11$. 
This approach works very cleanly when dealing with a single CFD. Additional complications can arise when more than one CFD applies to the same CFD node. In particular, if the attributes involved with the CFDs have non-trivial intersections, we will need to determine an ordering on the tuple elements that respects CFD dependencies. For example, if we have the CFDs $c_1 \Rightarrow (B \rightarrow A)$ and $c_2 \Rightarrow (A \rightarrow C)$, then we should consider values from attribute $B$ before considering values of attribute $A$. This is because if $c_1$ is satisfied, then an entry in the CFD dictionary will determine the value of $A$ and we will not need to specify it in the tuple fragment.

To address this, we can build a dependency graph for CFDs. Each CFD will be represented by a node in the graph, and an edge $u \rightarrow v$ indicates that the CFD represented by $u$ should be evaluated before $v$. In particular, values for the attributes contained in the LHS and RHS of the CFD represented by $u$ should be considered before attributes contained in the CFD represented by $v$. The edge $u \rightarrow v$ will exist if and only if the RHS of $u$’s CFD involves attributes from the LHS of $v$’s CFD, or if the constraint for $v$’s CFD involves attributes in the RHS of $u$’s CFD. Given this dependency graph, we can determine
Figure 6.6: A CFD dependency graph. An outgoing edge indicates that the source should be evaluated before the target. A dashed edge corresponds to a dependency introduced by a constraint. Constraints are omitted where they do not introduce a dependency.

an ordering on the attributes that satisfies it.

Consider, for example, a relation $R(A, B, C, D, E, F)$, which has the following CFDs:

1. $(A < 10) \Rightarrow (A \rightarrow B)$
2. $(B \geq 5) \Rightarrow (BC \rightarrow D)$
3. $(A = 3) \Rightarrow (D \rightarrow F)$
4. $(E \leq 12) \Rightarrow (E \rightarrow C)$

Given these, we should evaluate the first and last CFDs before considering the second CFD. This is because if $A \rightarrow B$ or $E \rightarrow C$ apply, then the values for attributes for one or both of $B$ and $C$ will be in CFD dictionaries, and not in the tuple fragment. As such, the values for the LHS of the second CFD will not be determined until the first and last CFDs are considered. Fig. 6.6 depicts a dependency graph for the CFDs shown above.

We can use topological sort on this dependency graph to determine the order in which we should evaluate the CFDs. Note that this will affect the ordering of the elements in the rows that the CFDs apply to. For example, with the CFDs and their dependency graph shown above, we might choose to order the elements as follows:

$$(A, E, B^*, C^*, D^*, F^*).$$

Here, a $(^*)$ next to an attribute indicates that it will be omitted from result tuples if a particular CFD applies.

Unfortunately, these dependency graphs are not necessary acyclical. For example, con-
Figure 6.7: A cyclical dependency graph produced when the constraint of one CFD includes an attribute downstream of it. In this diagram, the dashed edge indicates a dependency introduced by a constraint.

Consider a relation R(A, B, C, D, E) for which the following CFDs apply:

1. \((C \leq 1) \Rightarrow (A \rightarrow B)\)
2. \((E = 4) \Rightarrow (B \rightarrow C)\)

While the FD component of the first CFD requires it be upstream of the second, its constraint requires that values of \(C\) be provided before it can be known whether it applies. This suggests that first should be downstream of the second, creating a cycle, as shown in Fig. 6.7. In order to know if we should apply the first CFD, we must know the value of \(C\). To know the value of \(C\), we must either include it in the results stream, or rely on it being in a CFD map for corresponding to the second CFD. In either of these cases, we are forced to introduce extra information in the result stream to disambiguate.

This issue can be addressed in one of two ways:

1. For each CFD cycle, we can introduce a bit in each tuple fragment of the result stream indicating whether or not the first CFD in the cycle can be applied. That is, whether or not the constraint for this CFD is met. This, in effect, removes one of the edges in the cycle, breaking it.

2. We can ignore one of the CFDs in the cycle. This also has the effect of breaking the cycle.

Note that this effect can also arise without the constraints coming into play. Consider, for example, the two CFDs \(c_1 \Rightarrow A \rightarrow BC\) and \(c_2 \Rightarrow BD \rightarrow A\). We can address cycles created in this way in a similar fashion. We can either ignore one of the CFDs, or always include the attribute \(B\) in the tuple fragment. Note that the problem of finding the smallest set of
Algorithm 11 COMPRESSCOLUMN($t$, $D$, $r$, $c$)

**Input:** join tree node $t$, dictionary set $D$, row $r$, column $c$

**Output:** Encodes a single column $c$ for a given row $r$

1. $code \leftarrow \text{FORWARDLOOKUP}(D(c), r[c])$
2. \textbf{if} $code = \epsilon$ \textbf{then}
3. \hspace{1em} $code \leftarrow \text{Push}(D(c), r[c])$
4. \hspace{1em} send $ENTRY, c, r[c]$
5. \textbf{return} $code$

Edges or vertices to remove from a graph to leave it acyclical are NP-complete problems [23]. However, there are known fast approximation algorithms for both of these problems [7, 5].

Note that the changes to COMPRESSROW (shown in Algorithm 2) required by this approach are constrained to the block beginning with line 2 where the join tree node $t$ is a leaf. We define COMPRESSLEAFNODE in Algorithm 12, which includes a sketch of the approach outlined in this section. Line 3, computes the dependency graph of a set of CFDs and uses topological sort to determine an order in which the CFDs should be evaluated. Finally, Algorithm 14 ties all of these procedures together to encode a result set given a join tree and a list of CFDs that apply to it.
Algorithm 12 CompressLeafNode(t, D, r, C, M)

Input: join tree leaf node t,
dictionary set D,
row r,
C, list of CFDs applying to t,
CFD map set M

Output: generates row code for r

1: rowcode ← ε
2: A ← ∅ { used to keep track of columns that have been sent }
3: σ ← topological sort of dependency graph generated from C
4: for i := 1 to |σ| do
5: let z = (c ⇒ (X → B)) be the CFD represented by σi
6: if constraint c passes on row r then
7: cfdkey ← ε
8: for all column c ∈ X do
9: if c /∈ A then
10: code ← CompressColumn(t, D, r, c)
11: cfdkey ← cfdkey · code
12: rowcode ← rowcode · code
13: A ← A ∪ {c}
14: cfdval ← MAPLOOKUP(M(z), cfdkey)
15: if cfdval = ε then
16: for all column c ∈ B do
17: if c /∈ A then
18: code ← CompressColumn(t, D, r, c)
19: cfdval ← cfdval · code
20: rowcode ← rowcode · code
21: A ← A ∪ {c}
22: MAPPUT(M(z), cfdkey, cfdval)
23: for all column c of relation t do
24: if c /∈ A then
25: code ← CompressColumn(t, D, r, c)
26: rowcode ← rowcode · code
27: return rowcode
Algorithm 13 \texttt{CompressRow}(t, D, r, C, M)

\textbf{Input:} join tree leaf node $t$, 
dictionary set $D$, 
row $r$, 
$C$, list of all CFDs applying to any node in the join tree 
CFD map set $M$ 
\textbf{Output:} sends compressed row using CFDs

\begin{enumerate}
\item $rowcode \leftarrow \epsilon$
\item \textbf{if} $t$ is a leaf \textbf{then}
\item $C \leftarrow$ all CFDs from $C$ applying to $t$
\item $rowcode \leftarrow \texttt{CompressLeafNode}(t, D, r, C, M)$
\item \textbf{else}
\item $leftcode \leftarrow \texttt{CompressRow}(t.left, D, r)$
\item $rightcode \leftarrow \texttt{CompressRow}(t.right, D, r)$
\item $rowcode \leftarrow leftcode \cdot rightcode$
\item \textbf{if} $t$ is non-root \textbf{then}
\item $code \leftarrow \texttt{ForwardLookup}(D(t), rowcode)$
\item \textbf{if} $code = \epsilon$ \textbf{then}
\item $code \leftarrow \texttt{Push}(D(t), rowcode)$
\item send $DENTRY, t, rowcode$
\item $rowcode \leftarrow code$
\item return $rowcode$
\end{enumerate}

Algorithm 14 \texttt{Compress}(T, W, C)

\textbf{Input:} Join tree $T$, Result set $W$, Set of CFDs $C$ that apply to $W$

\textbf{Output:} Sends compressed result set to decompressor

\begin{enumerate}
\item send join tree $T$ and schema of leaf nodes
\item send CFDs $C$
\item $D \leftarrow \emptyset$
\item $M \leftarrow \emptyset$
\item \textbf{for all} non-root node $v \in T$ \textbf{do}
\item initialize dictionary $D(v)$
\item $D \leftarrow D \cup \{D(v)\}$
\item \textbf{if} $v$ is a leaf node \textbf{then}
\item \textbf{for all} columns $c$ of relation $v$ \textbf{do}
\item initialize dictionary $D(c)$
\item $D \leftarrow D \cup \{D(c)\}$
\item \textbf{for all} CFDs $z$ applying to relation $v$ \textbf{do}
\item initialize map $M(z)$
\item $M \leftarrow M \cup \{M(z)\}$
\item \textbf{for all} row $r \in W$ \textbf{do}
\item send $\texttt{CompressRow}(\text{root}(T), D, r, C, M)$
\end{enumerate}

73
6.2.2 Experimental Results

In this section, we empirically measure the compression ratio improvements observed when using CFDs. Consider the schema of the ORDERS table of the TPC-H dataset, shown in Fig. 6.8. We impose the following CFD on this table:

\[(O\_CUSTKEY < x) \Rightarrow (\{O\_CUSTKEY\} \rightarrow \{O\_ORDERPRIORITY, O\_SHIPPRIORITY, O\_CLERK\})\],

where \(x\) is a parameter that determines how many tuples the CFD applies to. We manipulate the data to ensure that the FD component of this CFD applies to all tuples satisfying the constraint \(O\_CUSTKEY < x\). Rather than fixing a value \(x\) across databases generated from various scale factors, we choose a CFD ratio \(r\) used to compute \(x\) for each database. This ratio determines the percentage of tuples in the ORDERS table that the CFD applies to. A value for \(x\) corresponding to \(r = 0.1\), for example, has 10% of the tuples in ORDERS satisfying the constraint \(O\_CUSTKEY < x\).

To measure relative compression performance, we run the QA algorithm on Query 5 from previous sections. We partition the ORDERS relation into two parts: one where the constraint applies, and one where it does not. We express the first of these partitions as the join of two relations: ORDERS\_NO\_PRIORITIES and PRIORITIES. For the portion of the ORDERS table where the CFD applies, we augment the join tree corresponding to Query 5 as shown in Fig. 6.9. For the remaining tuples, we use the base join tree. These compressed results are concatenated to form the entire compressed result set. As a baseline, we also
measure the compression ratio when using the base join tree for the entire ORDERS table, including those tuples satisfying the constraint. We vary the scale factor from 0.01 to 0.31 and measure the compression ratio provided using the base join tree and the augmented join tree using CFD ratios 0.1, 0.3, 0.5, and 0.7. The dictionary capacities are also varied from 10MB to 30MB.

Compression ratio results are shown in Fig. 6.10. As we might expect, compression ratio is positively correlated with the CFD ratio. We note that the compression ratio improvements observed when making use of the CFD are relatively constant across the scale factor for a given CFD ratio. This is what we expect to see, as the amount of redundancy attributable to the CFD remains constant with respect to the size of the result set. In addition, we see that using larger dictionaries improves the compression ratio by a constant amount regardless of scale factor or CFD ratio.

6.3 Conclusions

In this chapter, we introduced functional dependencies and showed how to exploit them to improve the compression ratio provided by the query-aware algorithm. We saw that when a functional dependency applies to a relation $R$, we can interpret $R$ as if it were the result of a join. We use this insight to augment join trees involving relations that have functional dependencies with additional joins, thereby improving the model for redundancy in the join
Figure 6.10: CFD results for various fixed-size dictionary capacities.

Later, we introduced several techniques to exploit redundancy revealed by constrained functional dependencies. Each of these algorithms takes a different approach to solving difficulties due to CFDs applying to only a subset of a relation. Empirical analysis confirmed that the query-aware algorithm can make use of CFDs to improve the compression ratio it provides.
CHAPTER 7
CONCLUSION

In this thesis, we provide a recursive compression algorithm applicable to results of join queries. We find that because it is able to better model the data it is acting on, it often provides significant improvements over gzip alone. In empirical analysis, we notice that our implementation of the query-aware algorithm introduces a significant computational overhead. While this may render QA compression impractical in some contexts, we suggest that in situations where bandwidth or energy consumption is sufficiently limited, it will still offer benefits over gzip.

In order to limit memory consumption, we introduced fixed-size dictionaries, and showed how they can be used in the QA algorithm by evicting old entries to make room for newer ones. As the algorithm makes use of a set of $M$ dictionaries, we introduced three techniques for partitioning a memory constraint of $N$ bytes across each of the $M$ dictionaries. In experimental analysis, we saw that query-aware compression can still perform well when its memory consumption is limited.

Experimental results showed that the compression ratio provided by the QA algorithm is often sensitive to the order in which the relations it is acting on are joined. In an effort to choose an optimal join order, provide analytical models to approximate the compression ratio given a join tree. However, we note that these models require access to detailed statistics about the data, some of which can only be obtained by sampling.

We also discuss how functional and constrained functional dependencies can be used to better model the redundancy in a result set. With the former, we show how to augment a provided join tree with artificial joins corresponding to functional dependencies applying to join tree leaves. Several techniques are introduced to overcome difficulties associated with CFDs applying to only a portion of the result set. Empirical analysis showed that CFDs can be used by the query-aware algorithm to improve the compression ratio it provides.
Future work involves efforts to further improve the performance of the QA algorithm. Our implementation used a simple form of variable-length codes for the dictionary indices. However, using Huffman codes and dynamically optimizing code lengths based on access count statistics would likely improve compression ratios. We also expect that it is possible to modify the QA algorithm to exploit soft functional dependencies [21], which are a generalization of CFDs.

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


