Joins on Encoded and Partitioned Data

Jae-Gil Lee[◇*] Gopi Attaluri[‡] Ronald Barber[†] Naresh Chainani[‡] Oliver Draese[‡]
Frederick Ho[#] Stratos Idreos[•*] Min-Soo Kim[◁*] Sam Lightstone[‡] Guy Lohman[†]
Konstantinos Morfonios[△*] Keshava Murthy[#] Ippokratis Pandis[▷*] Lin Qiao[○*]
Vijayshankar Raman[†] Vincent Kulandai Samy[†] Richard Sidle[†] Knut Stolze[‡] Liping Zhang[‡]

[†]IBM Almaden Research Center [◇]KAIST, Korea [‡]IBM Software Group [•]Harvard University
[#]IBM Informix [◁]DGIST, Korea [▷]Cloudera [△]Oracle [○]LinkedIn

ABSTRACT

Compression has historically been used to reduce the cost of storage, I/Os from that storage, and buffer pool utilization, at the expense of the CPU required to decompress data every time it is queried. However, significant additional CPU efficiencies can be achieved by deferring decompression as late in query processing as possible and performing query processing operations *directly* on the still-compressed data. In this paper, we investigate the benefits and challenges of performing joins on compressed (or encoded) data. We demonstrate the benefit of *independently* optimizing the compression scheme of each join column, even though join predicates relating values from multiple columns may require translation of the encoding of one join column into the encoding of the other. We also show the benefit of compressing "payload" data other than the join columns "on the fly," to minimize the size of hash tables used in the join. By partitioning the domain of each column and defining separate dictionaries for each partition, we can achieve even better overall compression as well as increased flexibility in dealing with new values introduced by updates. Instead of decompressing both join columns participating in a join to resolve their different compression schemes, our system performs a light-weight mapping of only qualifying rows from one of the join columns to the encoding space of the other at run time. Consequently, join predicates can be applied directly on the compressed data. We call this procedure *encoding translation*. Two alternatives of encoding translation are developed and compared in the paper. We provide a comprehensive evaluation of these alternatives using product implementations of each on the TPC-H data set, and demonstrate that performing joins on encoded and partitioned data achieves both superior performance and excellent compression.

1. INTRODUCTION

Most commercial database management systems (DBMSs) currently provide some form of compression to reduce the volume of data stored, saving both on the cost of the storage medium (usually disk) and on the time to access data [1, 23]. Many of these systems compress the data via dictionary encoding [4], in which each value, or portion of a value, of a column that would normally consume many bytes is replaced by an encoded value requiring only a few bits. For example, the 10-byte value "California" or its 2-byte abbreviation "CA" can be encoded in a 6-bit value "000101," because all 50 U.S. states can be encoded in 6 bits, and California is the 5th state alphabetically. Unlike text-oriented compression schemes such as gzip [6], which require de-compressing an entire page to access any part of it, encoding column values via dictionaries preserves the column and row boundaries, so that individual rows can be decompressed. Dictionaries may draw their values from either an entire table [8], an individual column [14, 17, 18], or a storage unit such as a page or extent [8, 13, 17].

Today's dictionary-encoded systems typically de-compress the data on a page as soon as it has been read into main memory, so that query processing can be readily performed on the unencoded data. For example, the encoded 6-bit value of "000101" would be expanded back to the original 10-byte value "California" or perhaps the 2-byte value "CA," so that predicates could be applied to it, sorting and/or grouping could be performed, and expressions such as concatenation or substring could be computed. Consequently, the space savings of compression enjoyed by the I/O system are unfortunately not exploited by the internal data structures in main memory, cache, and registers that perform query processing.

Recently, however, a few DBMSs have begun to realize substantial performance improvement by frugally keeping values encoded and performing query processing on the encoded values [1, 7, 10, 15, 18, 22]. In the above-mentioned example, a predicate such as State = "California" can be applied on the encoded value as State = 000101. Moreover, the compact encoding of the values permits loading multiple values for a column into a register, so that a predicate can be applied simultaneously to all those values in a vector comparison [9, 12, 18, 22].

Intuitively, join predicates also benefit from operating on encoded data. For example, hash joins find matching values of join columns via a hash table. Join column values from the "build" table of the join are hashed to find a location in which they are stored, and similarly the join column values from the "probe" side are hashed to see if there is a match from the build side. So long as the columns to be joined have the same encoding, the original, unencoded values of the join columns will find a match if and only if the encoded values match. Performance of a hash join is dominated by the random accesses needed to probe the hash table. Storing encoded values in this hash table reduces its size and allows it to fit better in the memory hierarchy, thus improving probe performance, reducing the need to spill the hash table to disk, etc.

For each join key (join column(s)), a hash table typically also stores a "payload," which is composed of so-called "auxiliary"

[*]Work done while the author was at IBM

Proceedings of the VLDB Endowment, Vol. 7, No. 13

columns that are used later in the query. Clearly these auxiliary columns in the payload will consume less space in the hash table when they too are retained in encoded form. Even the results of expressions may benefit from encoding data "on the fly," i.e., by re-encoding the results of expressions as they are evaluated.

1.1 Contributions

This paper describes how joins are processed on encoded data in the Informix Warehouse Accelerator (IWA), a main-memory accelerator to the disk-based Informix database server product, packaged together as the Informix Ultimate Warehouse Edition (IUWE).[1] IWA reads selected row-organized data that is stored on disk by the Informix database server, encodes each column independently, and stores that data in the memory of IWA using a PAX-like page format [2]. A query posed to Informix is automatically routed to IWA if the data referenced in that query is replicated in IWA; otherwise the Informix data server executes on the original data. See [3] for an overview of the IWA system.

During query processing, joins are performed on the encoded columns without having to have a common dictionary between those columns. Instead we *translate* the encoded value of one column to the encoded value of the other before we scan the second table. Moreover, this translation supports a mix of compressed and uncompressed values within a column. This approach provides us with better compression and more flexible reorganization as the data is incrementally updated, while still executing the join with excellent performance.

Specifically, the contributions of this paper include:

1. We justify the benefit of performing joins on encoded columns.
2. We introduce a novel "on-the-fly" encoding scheme for *payload* columns, using a new hash table data structure that assigns stable bucket positions to inserted values.
3. We explore the problem of adding values not found in the original dictionary, and the benefit of partitioning the domain to deal with this problem.
4. We describe a technique for performing hash joins on encoded columns, each of which may have its own dictionary. For this purpose, we propose a notion of *encoding translation* and develop two variants: *eager* and *deferred* translations.
5. We conduct extensive experiments using IWA on the TPC-H data set to demonstrate the advantages of our approach.

IWA has been generally available since March 2011 on the Linux operating system on Intel processor-based servers; the IBM AIX operating system on IBM POWER processor-based servers; the HP-UX operating system on Intel Itanium processor-based servers; and the Oracle Solaris operating system on Oracle SPARC servers. When running on Linux, the database server and the accelerator can be installed on the same or different computers, communicating via TCP/IP. During query processing, each query is executed in parallel by worker processes on IWA, and merged and returned to Informix by a coordinator process. When both Informix and IWA are running on the same machine, the coordinator and worker nodes simply become processes on the same machine that communicate via loopback. Informix and IWA can be on distinct SMP hardware, with IWA running both the coordinator and worker processes on the same hardware. IWA can also be deployed on a blade server for increased capacity and performance, such as Intel Xeon-based IBM servers supporting up to 80 cores and 6 TB of DRAM. Nodes on IBM blade servers support up to 4 sockets and up to 640 GB of DRAM. Since both the number of cores and memory capacity of the hardware is increasing rapidly, the IUWE software has been packaged flexibly enough to run directly on hardware or in a virtualized/cloud environment. Each database server can have zero, one, or more IWAs attached to it.

[1]These techniques were also used in the IBM Smart Analytics Optimizer for DB2 for z/OS V1.1, a predecessor product to today's IBM DB2 Analytics Accelerator for DB2 for z/OS.

1.2 Outline

The rest of this paper is organized as follows. Section 2 explains the basics of hash joins. Sections 3 and 4 then go into encoding, for join columns and payload columns respectively. Sections 5 and 6 present our approach of partitioning a join column and two variants of encoding translation. In Section 7, we present the results and analysis of our experiments. We survey related work in Section 8 and conclude in Section 9.

2. JOINS

IWA uses hash joins as its primary join method. A hash join is composed of a *build phase* and a *probe phase*. Below we describe the high-level procedure of joining one or more build tables with one probe table.

During the build phase, one or more tables are each scanned, applying predicates local to that table. Qualifying rows add their join-column values to a hash table for that table. Optionally, *payload columns* that will be used later in the query may also be inserted into the same bucket of the hash table with its join-column values. This procedure is repeated for each build table to which the probe table is joined.

During the probe phase, the probe table is scanned, applying any predicates local to that table and then evaluating each join predicate by probing the corresponding hash table that was built for each build table, to check if each join-column value exists. If it does, the payload columns will be fetched from the hash table to do subsequent grouping and aggregation.

Although the encoding and partitioning techniques presented in this paper are applicable generally, it is convenient and clearer to present our encoded join technique using the terminology of *star schemas* common in OLAP systems [16], in which one or more *dimension tables* act as the build tables to be joined with a very large *fact table* as the probe table. Similarly, the techniques of this paper are not limited to joins between *primary key (PK)* and *foreign key (FK)* columns, although we will find it clearer to reference the join columns that way.

Example *1.* Figure 1 shows an example of processing a simple join query between `LINEITEM` and `ORDERS`. Local predicates are specified on `O_OrderDate` and `L_ShipDate`, and grouping is done by the values of `O_OrderDate`. Thus, the hash table from `ORDERS` contains the values of the PK column `O_OrderKey` and the payload column `O_OrderDate` of the qualifying rows. This hash table is passed to the fact-table scan. □

For *snowflake schemas* having more than one level of dimension tables, this technique may be applied recursively, starting from the outermost dimension tables and joining inward. In such cases, a table which is neither outer-most nor inner-most in the schema acts as both a fact table and a dimension table in successive joins.

3. ENCODING JOIN COLUMNS

Past work on join processing over compressed data either focused on nested loop joins [1], which are not suitable for business intelligence (BI) queries, or used simple approaches such as encoding both join sides in exactly the same way, which sacrifice

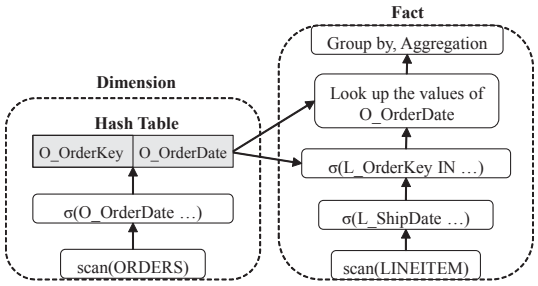


Figure 1: Example of star-join processing.

compression performance [7]. In this section, we will delve more deeply into performing hash joins on compressed values, exploring both the advantages and challenges, as well as practical techniques for dealing with unencoded values and different compression schemes between the different join columns.

3.1 Why Join on Compressed Values?

Since joins consume the bulk of query processing time in most BI queries, it seems almost obvious that keeping values compressed while performing joins will significantly improve query performance. By occupying less space, compressed values consume fewer resources, such as space in hash tables, thereby reducing cache-line misses. This applies to both join columns as well as *payload columns*. We will address these two types of columns separately. Additionally, keeping values highly compressed permits packing multiple values into a register and comparing them with a single instruction, achieving increased parallelism through single instruction, multiple data (*SIMD*) operations such as value comparisons [18].

3.2 Types of Compression

Columns in IWA are compressed by one of a few encoding schemes. Dictionary encoding, in which each value is replaced by its code, is used for columns with low cardinality and at least some repetitions (and preferably some skew in those repetitions), such as FK columns. Columns having no repetitions, such as a PK column, will gain nothing from dictionary encoding and are usually compressed with *minus encoding*, which stores the *difference* of a value from a given entry in the dictionary. For string fields, we use a variant that applies XOR instead of difference, called *prefix encoding*. Minus encoding still requires a dictionary with at least one element. For example, the values 1543, 1540, 1550, 1547, and 1539 can be more compactly represented (in just 4 bits) as the differences 4, 1, 11, 8 and 0 from a single dictionary value, 1539. Prefix encoding removes a common leading portion and works well on URLs, integers with leading zeros, or other application-specific commonality such as 'CUST0004', 'CUST0047', 'CUST0101', etc.

We considered but discarded many other compression methods. Run-length encoding (RLE) counts the number of successive values that are identical, so is most effective when instances are ordered, as in C-Store's projections [20]. Abadi's "Null Compression" [1] is a special case of RLE. Bit-vector encoding is effective only for extremely low-cardinality columns. Lempel-Ziv (LZ) compression works only when rows need not be accessed individually, e.g., for compressing entire pages.

3.3 Matching Encoded Values

The encoding schemes described above assign a unique code to each encoded value. Thus we can apply the join predicate on codes instead of on values. But the join columns on both sides of the join must be encoded identically. The basic idea for performing joins on encoded values is quite simple: if the same compression mapping M is applied to two values V_1 and V_2 that are equal, then their encoded values are equal: $M(V_1) = M(V_2)$. When the mapping M is an isomorphism, then the converse is also true. That is, if the encoded values $M(V_1)$ and $M(V_2)$ are equal, then the unencoded values are equal: $V_1 = V_2$.

There are three options for realizing this identical encoding:

1. **Encoding join columns identically on disk (per-domain encoding)**: It seems natural that two values V_1 and V_2 being tested for equality, which are drawn from different columns, should be encoded with the same mapping M. Abadi et al. [1] and many previous authors have assumed this simple *per-domain encoding*. Identically encoding both columns to be joined is ideal for join speed, because the two columns can be compared directly. But we found this approach to be unworkable. First of all, would we know *a prioi* all the columns a user would join on? DBAs do have a best practice of specifying referential integrity (RI) as constraints or as hints. Say that we have RIs from the columns `OrderDate` and `ShipDate` of a `PRODUCT` table to a `DATE` dimension. The RIs are suggestive of a join, but which column's distribution should we pick for deriving the common encoding? The "hub" of the RIs will be a primary key, effectively making all the codes fixed length, since the key likely has a uniform distribution. Section 5.2 describes many reasons why IWA uses variable encoding lengths.
2. **Translating both join columns to a new common encoding at runtime**: This is the most flexible option, because we can choose an encoding that is best for the subset of values that actually participate in the join, potentially compressing even better than the original encoding of those columns. But we do pay for the CPU cost of decoding and re-encoding values on both columns. For numerical columns, we usually employ minus encoding, and it is fairly cheap to decode and re-encode since we do not need to access the dictionary for every single code we are decoding. With dictionary encoding, decode involves a random access and re-encode involves a hash table lookup, so are quite expensive to do on both sides of the join.
3. **Encoding join columns independently and translating one join column to the encoding of the other at runtime (per-column encoding)**: As part of join processing, we can unify the encoding of the join columns. In a hash join, we can either convert the build side to the encoding used in the probe side, or vice-versa. We chose this option for IWA, which enables us to independently encode columns, because the values present in each column are typically very different, even though they are drawn from the same domain. We call this *per-column encoding*. The downside to per-column encoding is that we must perform *encoding translation*. In IWA, we translate from the encoding of the build side to the encoding of the probe side, which gives reduced translation cost for the usual case where the build side is smaller. Translation is formally defined in Definition 1, i.e., $M_{FK}(V_{FK}) == \mathbf{M_{FK}}(\mathbf{M_{PK}}^{-1}(M_{PK}(V_{PK})))$.

Definition *1.* Consider two encoded values $E_i = M_i(V_i)$ and $E_j = M_j(V_j)$ from different columns compressed with the mappings M_i and M_j, respectively. M^{-1} denotes de-compression using the mapping. *Encoding translation* is performing either $M_j(M_i^{-1}(E_i))$ or $M_i(M_j^{-1}(E_j))$ to be able to compare V_i and V_j in the same encoding space. □

In this paper, we will show that it is possible to achieve lightweight encoding translation while exploiting the benefits of per-column encoding. Before we discuss encoding translation in

detail, however, we first introduce "on-the-fly" encoding of payload columns and the need for partitioning each column's domain.

4. ENCODING PAYLOAD DATA: ON-THE-FLY ENCODING

Need for On-the-Fly Encoding

Our focus so far has been on encoding of join columns. But what about the *payload columns*, which are the columns used later in the query, such as for grouping? The payload of a join typically has columns from the dimension tables that are used in grouping. For example, a query to find the average order price grouped by the customer city and the product brand will pick those columns from joins with the `CUSTOMER` and `PRODUCT` dimensions. The join key is usually just an integer, whereas the payloads are often wider strings. Keeping them compressed reduces the size of the hash table for both join and group-by: only after applying the HAVING predicates we need to decompress them.

The original encoding of a payload column is insufficient to achieve this goal, for the reasons listed below. Thus, IWA *re*-encodes the payload values *on-the-fly* (*OTF*).

1. **Updates**: It is unrealistic to assume that all values will be encodable with a fixed dictionary. Over time, new values for a column get inserted, and they will be stored unencoded. So each payload column has at least two representations—encoded and unencoded—and potentially more if the distribution has skew. But hash table data structures do not directly handle variable-length payloads[2]. So we either have to pad them all to unencoded values or have 2^N different hash tables where N is the number of payload columns. This issue arises for the join keys too, but it is easier to handle since N is small (often 1).
2. **Expressions**: OTF encoding can be applied not just to columns, but also expressions. For example, grouping is often performed on a coarsified form of a dimension column, such as `MONTH(Shipdate)`. Normally this expression result would be left unencoded, but with OTF encoding we can exploit the small cardinality of months and encode it very compactly.
3. **Correlation**: OTF encoding can also exploit correlations. It is common to group on correlated columns, such as `City`, `State`, `ZIPCode`, and `Country`. During load, these columns will be encoded individually. But at query time, we can exploit the correlation among them to produce a tighter code.
4. **Predicates**: Local and/or join predicates will likely reduce the cardinality of each column, allowing a more compact representation in a reduced dictionary. For example, predicates on the month and year would likely reduce the cardinality of all remaining date columns by an order of magnitude or more.

OTF Mapping Table with Stable Hash Positions

OTF encoding is a mapping from a value to a fixed-length code, and we need to construct this mapping in a *single pass* over the input (i.e., as part of the operator pipeline for the build side of the join). IWA uses a novel *OTF mapping table* to construct this mapping. As we encounter unencoded values, they are inserted into this mapping table. If a new value (one not seen until then) is encountered, an insert call adds it to the mapping table and outputs an OTF code, which is actually an *index* into the bucket where the value was inserted. If an existing value is encountered, an insert call returns the index of the existing value.

[2]We did not use a pointer to a heap object because that costs another pointer as well as more fragmentation in the memory pool and incurs latch contention in the memory allocator.

We cannot implement this mapping table as a straight hash table, because the OTF code assigned to a value can never be changed, *even if the hash table is resized*. So the OTF mapping table is implemented as a list of hash tables, each double the capacity of the previous. Each hash table is a standard (linear probing) hash table holding the unencoded values. The OTF code assigned to a value is calculated from the hash bucket it falls into, cumulatively adding the capacity of all earlier hash tables and the size of the original dictionary itself.

Example *2.* Consider a mapping table made up of three hash tables of capacity 1024, 2048, and 4096 entries. Suppose that the original dictionary has 600 entries. Then, a value that goes into the bucket 40 in the third hash table will get an OTF code $600+1024+2048+40=3712$. □

The hash tables within the mapping table are implemented as lock-free data structures. The list of hash tables is a lock-free array of pointers, and concurrent inserts to the hash tables use compare-and-swap on flags (one flag per bucket) for synchronization.

Compaction of the OTF Code

This OTF code is initially sized to the log of the maximum number of unencoded values we might encounter. IWA maintains this upper bound as unencoded values are inserted. For OTF encoding over expressions, we use a 64-bit number. After the build side of the join has been fully scanned and all possible unencoded values have been seen, we know exactly how wide the OTF code needs to be. So we can revisit the values and "compact" the OTF codes further.

DB2 with BLU acceleration [18] also employs OTF encoding. Here, the build side of hash joins uses partitioning, so we have an opportunity, after the input has been partitioned, to do this compaction as part of entering the payloads into the join hash table itself. Each OTF hash table is just a bitmap (there is no payload), so this compaction requires only a simple linear scan of the bitmap to compute a prefix population count for each occupied bucket, and then a scan over the partition with lookup into the OTF hash table to reassign OTF codes.

Performance Improvement by OTF Encoding

To investigate the impact of the OTF encoding on query performance, we ran queries obtained from real customer workloads that reference both join and payload columns. The queries were executed with the OTF encoding enabled and disabled, respectively, on a 100GB TPC-DS data set. Figure 2 shows that OTF encoding improved the performance of all queries—by 17% on average and up to 52%.

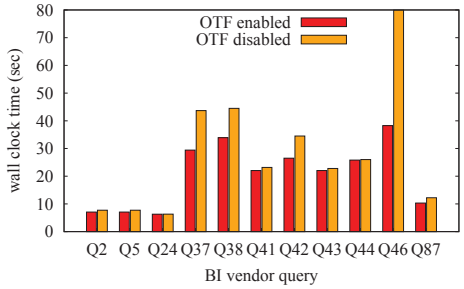


Figure 2: Impact of the OTF encoding.

5. PARTITIONING COLUMN DOMAINS

This section describes why and how IWA partitions data before encoding and storing it.

5.1 Problem of Uncompressed Values

Any compression scheme using a dictionary has a fundamental challenge: how to deal with new values not present when the dictionary was created. Two common solutions are:

1. **Leave Room**: One common approach to dealing with new values is to leave sufficient room in the initial dictionary for future values. However, this has a number of problems associated with it. How much space is enough? Overestimating the number of future values means that bit combinations will be wasted. For example, doubling the dictionary size from N values to $2N$ values adds one bit to all values, which would always be 0 for values in the initial dictionary and would be unnecessary if no new values ever occurred. Conversely, once N values have been added, there is no way to add additional values without re-encoding every value. Furthermore, if the initial dictionary is defined to be order-preserving, so that range predicates can be applied to compressed values [9], then adding new values after the initial values destroys this order-preserving property, as there's usually no assurance that values will be added to the database in any particular order, nor any way to predict where new values will occur.
2. **Partition the Encoded Domain**: A second approach to dealing with new values is to partition the domain and create separate dictionaries for each partition. In this way, the impact of adding new values can be isolated from the dictionary(s) of any existing partition(s). The initial dictionary D_0 for a partition P_0 can be optimized for the values present at the time of its creation, and new values can simply be added to a partition P_1 using a completely separate dictionary D_1 that will be created on the fly, as values arrive. The characteristics of the two dictionaries, and thus the encoding schemes, can be completely different, allowing the compression of any domain to adapt to the characteristics of the new values. For example, P_0 could use dictionary encoding, and P_1 delta encoding. Partitioning also isolates and limits the impact of changing the compression scheme. For example, since the values added to P_1 arrived in no particular order, at some point (e.g., when it is full) we might want to re-assign the values in dictionary D_1 to make it order-preserving. Doing so would affect only the values stored in P_1. Alternatively, we could leave the values in the second partition P_1 unencoded to avoid the cost of encoding them twice—once when they arrived and a second time when the partition has filled. This unencoded partition would of course have to be processed differently than the encoded partition(s), increasing code complexity. We will discuss this type of partitioning more in subsequent sections.

   Note that partitioning a domain is very different from block- or page-based compression, which merely compresses whatever values happen to occur within a physical chunk. Unlike block compression, domain partitioning, by construction, guarantees that a particular value can occur in at most one dictionary, which can significantly simplify searching for a single value.

5.2 Better Compression

Although providing a home for new values is our primary reason for partitioning the domain of each column, there can be other benefits to domain partitioning, most notably better compression. Raman et al. [19] described a domain partitioning scheme called *frequency partitioning* that exploits skew in a domain to encode more frequent values in fewer bits and less frequent values in more bits, as in Huffman encoding, while still defining and operating on *fixed-length* codes *within* each partition. Frequency partitioning is automatically done by IWA without intervention of the DBA.

Example *3.* Consider a fact table of the sales of products having various countries of origin in Figure 3. At load time, for each column, frequency partitioning independently builds a histogram of the value occurrences and partitions the histogram into column partitions according to the frequencies of those occurrences. Then, the values in each partition are encoded using the same number of bits. In the `Origin` column, China and the USA are the most frequent and need only one bit to represent them. The European Union countries are next most frequent and are representable using 5 bits. The remaining 196 (or so) nations would require 8 bits. If there were 1,000,000 sales originating from China and the USA, 100,000 from the EU countries, and 10,000 from others, all values could be represented by only $1 \times 1,000,000 + 5 \times 100,000 + 8 \times 10,000 =$ 1.58 M bits, over 5.6 times better compression than the 8.88 M bits required if every nation was represented by eight bits needed to encode all possible nations. Similarly, we can partition the `Product` column into the top-64 products, representable using six bits, and all the rest. □

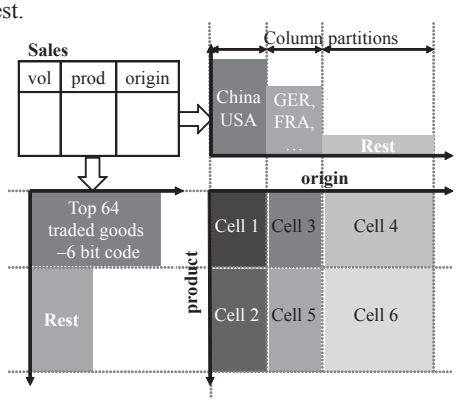


Figure 3: Example of frequency partitioning.

The compression efficiency achievable by frequency partitioning becomes more prominent in skewed data, as we will show in the following two examples.

Example *4.* Suppose that a retail store is running a data warehouse whose schema is similar to the TPC-DS benchmark. In this data warehouse, the fact tables store the dates when the products were shipped, sold, or returned; and the `DATE` dimension table stores the detailed information for each calendar day. Obviously, the amounts of shipments, sales, or returns are not distributed uniformly: these amounts are usually higher on the weekend than during the week and are especially high on holidays such as Thanksgiving Day and Christmas.

Table 1 shows the column sizes of a 1GB TPC-DS data set when compressed by per-column encoding and per-domain encoding. Since these columns are FK columns in the fact tables, the dates in the columns are skewed and have many repetitions. For compression, we use Huffman encoding to measure the lower bound of frequency partitioning on the number of bits. In *per-column encoding*, the encoding of a column is done by using its own dictionary to take advantage of data skew. On the other hand, in *per-domain encoding*, the encoding of a column is done by the dictionary for its PK column where there is no repetition. The gains in compression achieved by exploiting data skew (the differences between the third and fourth columns) are shown to be significant (33%∼50%) in the TPC-DS data set. □

Example *5.* Let's consider a data warehouse that stores the entire population of the U.S. According to the U.S. census[3], the top-1000 frequently occurring last names occupy 40.6% of the entire

[3]http://www.census.gov/genealogy/www/data/2000surnames/

Table 1: Benefit of frequency partitioning in skewed data.

Column	Original Size (bits)	Compressed *with* Using Skew (bits)	Compressed *without* Using Skew (bits)
TPC-DS Data Set			
ss_sold_date_sk	92,172,928	29,335,719	44,520,246
sr_returned_date_sk	9,200,448	3,016,765	4,492,594
cs_sold_date_sk	46,129,536	15,327,881	23,219,100
cs_ship_date_sk	46,129,536	15,460,990	23,221,450
cr_returned_date_sk	4,610,144	1,568,780	2,332,695
ws_sold_date_sk	23,020,288	7,653,072	11,643,858
ws_ship_date_sk	23,020,288	7,774,634	11,645,361
wr_returned_date_sk	2,296,416	750,204	1,111,444
inv_date_sk	375,840,000	94,410,000	190,260,000
U.S. Census Data Set			
last_name	7,747,874,336	3,299,326,711	4,175,255,724

population. Thus, last names are skewed and have many repetitions. The sizes of the column compressed in the same way as in Example 4 are reported in Table 1, and the gain achieved by exploiting data skew is also quite significant (21%). □

We note that Table 1 indeed shows the advantages of *per-column encoding* over *per-domain encoding*. That is, with an existence of data skew, per-column encoding, which uses its own dictionary and thus can exploit data skew, allows us to improve compression ratio significantly than per-domain encoding, which could not exploit data skew. Examples 4 and 5 indicate that skewed data are common in practice.

5.3 Challenges for Partitioning

When the partitioned columns in Example 3 are stored in a row store or are stored together as a *column group* because they are frequently accessed together (such as `Street`, `City`, `State`, `ZIPCode`, and `Country`), then the intersection of these partitions defines *cells*. These cells contain the rows having one of the values from each corresponding partition, in which each row is formed by concatenating the fixed-length code for each of its columns. In Figure 3, Cell 1 contains all rows having either China or the USA as its origin, represented using only one bit, and one of the top-64 products, denoted by a six-bit code. Note that code lengths are fixed within cells, but would vary from cell to cell.

When each of these columns in a column group or row is partitioned, the number of cells is the product of the number of partitions for each of its columns. In Figure 3, the 3 partitions for `Origin` and the 2 partitions for `Product` induce $3 \times 2 = 6$ cells. This quickly gets out of hand: even if each column has only 2 partitions, one for all encoded values and one for unencoded values, a column group having C columns would have 2^C cells, whose contents would likely be very skewed and sparse. This is because 2^{C-1} of those cells would contain rows having one or more unencoded values.

One way to limit this proliferation of cells in column groups or row stores is to have just one cell, called the *catch-all cell*, to which we will assign *any* row that has at least one unencoded value in any column. Example 6 shows an example of the catch-all cell. This scheme minimizes the number of cells needed for unencoded values, but complicates join processing because now we always have to consult the catch-all cell, in addition to its encoded cell, as Example 6 illustrates.

Example *6.* Consider a portion of the TPC-H schema in Figure 4. Only two columns of the `LINEITEM` table are shown in the figure. Suppose the dictionary of `LINEITEM` has two partitions for `L_OrderKey` and one partition for `L_ShipDate`, owing to frequency partitioning. Thus, two (2×1) encoded cells and the catch-all cell are created. The encoded cells have the data in the form of codes. In contrast, for uniformity the catch-all cell stores the entire row unencoded, even if some column values are encodable. For example, although the value *100* of `L_OrderKey` of the fifth row is encodable, the entire row is stored unencoded in the catch-all cell, since the value *5/1/2010* of `L_ShipDate` is not encodable due to a missing dictionary entry. Thus, the value exists in two forms: *0* in encoded form and *100* in unencoded form. □

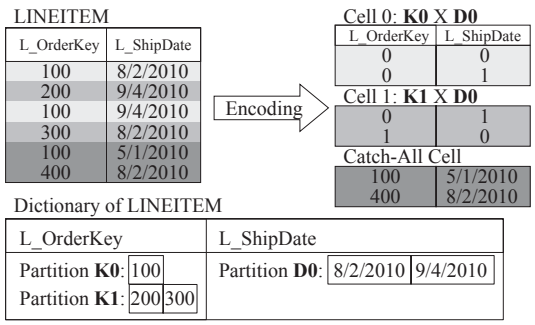


Figure 4: Example of data encoding with the catch-all cell.

6. ENCODING TRANSLATION

Recall from Section 3 that join columns are better encoded individually ("per-column encoding"), necessitating *encoding translation*, defined in Definition 1. Since hash joins typically build the hash table from the smaller table to minimize the hash table's size, particularly if its join column is a PK, it is usually cheaper to re-encode the qualifying rows of the build table(s), using the encoding of the larger probe table. In a star schema, this means that each PK of each dimension table is decoded and then re-encoded using the dictionary of the corresponding FK in the fact table, and again we will use this terminology without loss of generality.

For row stores or column stores having column groups, the presence of a catch-all cell, introduced in Section 5.3, complicates this translation somewhat, because a particular value can occur in the fact table both in its encoded form (in encoded cells) and in its unencoded form in the catch-all cell (e.g., the two representations for the FK value *100* in Example 6). These multiple representations for the same value induce two alternative approaches for encoding translation: `DTRANS` (Dimension TRANSlation), which resolves the multiple representations during the dimension-table scan; and `FTRANS` (Fact TRANSlation), which resolves them during the fact-table scan. Each has its pros and cons, and is applicable in different situations. Note that `DTRANS` puts more weight on reducing the overhead of processing fact tables, whereas `FTRANS` stresses reducing that of processing dimension tables.

6.1 The DTRANS Approach

In this approach, multiple representations of the same FK value are resolved at the stage of the dimension-table scan.

6.1.1 Hash Table Construction

One hash table is built for (all cells in) each partition of the FK, plus one (the last) for the catch-all cell. The hash tables for encoded partitions are likely to be very compact because: (i) the values are compressed, and (ii) each hash table is responsible for only one partition, not the entire table. This compact size is helpful for improving the probing performance in the probe stage. The hash table for the catch-all cell contains *all* qualifying key values in unencoded form. An encodable value therefore must be put into two

hash tables because, in the dimension-table scan, we do not know which FK values will have multiple representations.

Algorithm 1 shows the pseudocode for building the hash tables. For each qualifying row of the dimension table, the PK[4] is decoded using its dictionary in the dimension table, and then re-encoded using the dictionary of the fact table (Lines 3∼6). If this PK value is encodable with the FK dictionary, its code is added to the hash table corresponding to that FK partition (Lines 7∼11). The PK is always added to the hash table for the catch-all cell in unencoded form (Line 12). This is because *any* FK value can exist in the catch-all cell of the fact table.

Algorithm 1 Building the hash tables in DTRANS.

INPUT: a dimension table, $dict_{fact}$, $dict_{dim}$ /* $dict_{fact}$ (or $dict_{dim}$) is a dictionary of a fact (or dimension) table */
OUTPUT: a set of hash tables $HT[0 \sim num_{part}]$
1: $num_{part} \Leftarrow$ # of encoded partitions in $dict_{fact}$;
2: **for** each qualifying row r in the dimension table **do**
3: $pk \Leftarrow loadColumn(r,\ primaryKey)$;
4: $val_{pk} \Leftarrow decode(pk,\ dict_{dim})$;
5: /* If encoding fails, $partId$ is set to num_{part} */
6: $\langle code_{pk}, partId \rangle \Leftarrow encode(val_{pk},\ dict_{fact})$;
7: /* If val_{pk} is encodable using $dict_{fact}$ */
8: **if** $partId < num_{part}$ **then**
9: /* $HT[n]$ indicates the n-th hash table */
10: $addKey(code_{pk},\ HT[partId])$; /* encoded */
11: **end if**
12: $addKey(val_{pk},\ HT[num_{part}])$; /* catch-all */
13: **end for**

Example *7.* Figure 5 shows the output of processing a scan on the ORDERS table. Only relevant columns are shown in the figure. Suppose a predicate is specified to select the rows with O_Orderstatus = "S". According to the dictionary of the fact table in Figure 4, two hash tables are built for the encoded partitions, and one for the catch-all cell. Here, *0* in HT[0] represents *100*, whereas *0* and *1* in HT[1] represent *200* and *300*, respectively. The last hash table contains all—four in this example—qualifying key values. □

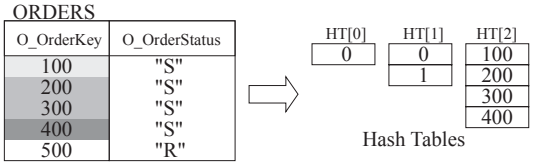


Figure 5: Example of a dimension table scan using DTRANS.

If the catch-all cell of the fact table is ever completely empty, this means that all values of all rows of the fact table are encoded, guaranteeing that every FK value has only one representation. As a result, the full duplication of the PKs is unnecessary, and Line 12 in Algorithm 1 can be bypassed.

6.1.2 Hash Table Probe

During the fact-table scan, the hash tables constructed during the dimension-table scan will be probed. In the DTRANS approach, which hash table to probe for each partition is pre-determined, because there exists a one-to-one correspondence between each partition of the FK and its hash table. The corresponding hash table is directly probed without decoding or re-encoding the FK value. For encoded partitions, this probing can be done efficiently, since the hash table is likely to fit in the L2 or L3 cache.

Algorithm 2 shows the pseudocode for probing the hash tables. The algorithm first derives the FK partition of the cell currently being scanned (Line 2) and thus the hash table to probe for each qualifying row to check whether its FK satisfies the join condition (Lines 3∼8).

Algorithm 2 Probing the hash tables in DTRANS.

INPUT: a fact table, a set of hash tables $HT[0 \sim num_{part}]$
OUTPUT: a bit vector $Vec[\,]$ indicating matching rows
1: **for** each cell c in the fact table **do**
2: $partId \Leftarrow$ the partition which c belongs to;
3: **for** each qualifying row r in c **do**
4: $fk \Leftarrow loadColumn(r,\ foreignKey)$;
5: /* If a key is found, "true" is returned */
6: $Vec[i] \Leftarrow lookupKey(fk,\ HT[partId])$;
7: $i \Leftarrow i + 1$;
8: **end for**
9: **end for**

Example *8.* Figure 6 shows the process of executing a scan on the LINEITEM table using the set of hash tables in Figure 5. Note that the FK value *100* has multiple representations as the code *0* in the first partition and as the value *100* in the catch-all cell. These multiple representations are handled by the duplication of the PKs into the hash table HT[2] for the catch-all cell. □

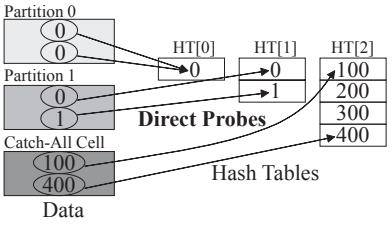


Figure 6: Example of a fact table scan using DTRANS.

6.2 The FTRANS Approach

The main drawback of DTRANS is the high cost of the full duplication of PKs. This cost becomes prohibitive when dimension tables are very large. In contrast, the FTRANS approach aims at avoiding that duplication by paying an additional cost in the fact-table scan.

6.2.1 Hash Table Construction

The FTRANS approach constructs the hash tables in the same way as those in the DTRANS approach *except the last one containing unencoded values*. Whereas DTRANS keeps *all* PK values—even though many of them are encodable—in the catch-all hash table, the FTRANS approach inserts *only unencodable* PK values into that hash table. In FTRANS, the encoded values and the unencoded values are *disjoint*, unlike those in Section 6.1. Since no duplication is required, the hash tables are smaller and faster to construct.

Algorithm 3 shows the pseudocode for building the hash tables. Until Line 6, it is identical to Algorithm 1. In Lines 7∼8, if $partId = num_{part}$, the PK value is unencodable using the dictionary of the fact table; otherwise, it is encodable. Thus, $HT[0 \sim num_{part} - 1]$ keeps encodable keys in the same way as in Algorithm 1, but $HT[num_{part}]$ contains only unencodable keys (Line 8).

[4]We assume that a PK consists of a single column, for ease of presentation. For a composite key, the step of concatenating codes or values should precede, which has been fully implemented in IWA.

Algorithm 3 Building the hash tables in FTRANS.

INPUT: a dimension table, $dict_{fact}$, $dict_{dim}$ /* $dict_{fact}$ (or $dict_{dim}$) is a dictionary of a fact (or dimension) table */
OUTPUT: a set of hash tables $HT[0 \sim num_{part}]$

```
1: num_part ⇐ # of encoded partitions in dict_fact;
2: for each qualifying row r in the dimension table do
3:    pk ⇐ loadColumn(r, primaryKey);
4:    val_pk ⇐ decode(pk, dict_dim);
5:    /* If encoding fails, partId is set to num_part */
6:    ⟨code_pk, partId⟩ ⇐ encode(val_pk, dict_fact);
7:    /* HT[0 ∼ num_part − 1] hold encodable keys,
         but HT[num_part] unencodable keys */
8:    addKey(code_pk, HT[partId]);
9: end for
```

Example *9.* The hash tables built by the FTRANS approach for Example 7 is the same as Figure 5 except that $HT[2]$ has only the value 400. □

6.2.2 Hash Table Probe

The FTRANS approach resolves the multiple representations of the same FK value during the fact-table scan. To find the FK values that are *actually encodable but were left unencoded*, we attempt to encode each (unencoded) FK value in the catch-all cell. If the encoding succeeds, the query engine needs to probe the appropriate encoded hash table for that *encoded* FK value. Otherwise, the query engine probes the catch-all hash table for the *unencoded* key.

Algorithm 4 shows the pseudocode for probing the hash tables. For encoded partitions (until Line 8), it is identical to Algorithm 2. Lines 9∼23 are added to process the catch-all cell differently from Algorithm 2. For each qualifying row, we check if its FK value can be encoded using the dictionary of the fact table (Lines 13∼14). Depending on the encodability, which hash table to probe is determined to be either one of the encoded ones $HT[0 \sim num_{part}-1]$ or the unencoded one $HT[num_{part}]$ (Lines 15∼20).

Example *10.* Figure 7 shows the process of executing a scan on the LINEITEM table in the previous example. The FK values in the encoded partitions are treated in the same way as in Example 8. Then, in the catch-all cell, the FK value *100* is actually encodable and is converted to the code *0* of the first partition. So the encoded hash table HT[0] is probed for the code. In contrast, since the FK value *400* is not encodable at all, the unencoded hash table HT[2] is probed for the value. □

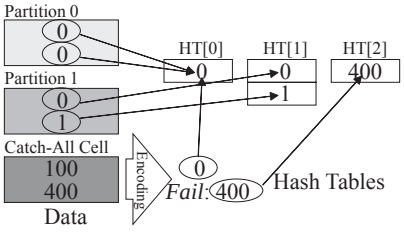


Figure 7: Example of a fact table scan using FTRANS.

7. EVALUATION

Using the IWA product code, we now verify that joining encoded data is beneficial (Section 7.2.1). We then contrast the performance of per-column versus per-domain encoding (Section 7.2.2). Finally, for column groups and row stores that use per-column encoding and partitioning to isolate new values in a "catch-all" cell, we compare our two approaches to encoding translation, DTRANS and FTRANS (Section 7.2.3).

Algorithm 4 Probing the hash tables in FTRANS.

INPUT: a fact table, a set of hash tables $HT[0 \sim num_{part}]$
OUTPUT: a bit vector $Vec[\,]$ indicating matching rows

```
1: for each cell c in the fact table do
2:    if c is an encoded cell then
3:       partId ⇐ the partition which c belongs to;
4:       for each qualifying row r in c do
5:          fk ⇐ loadColumn(t, foreignKey);
6:          Vec[i] ⇐ lookupKey(fk, HT[partId]);
7:          i ⇐ i + 1;
8:       end for
9:    else
10:      /* c is the catch-all cell */
11:      for each qualifying row r in c do
12:         fk ⇐ loadColumn(r, foreignKey);
13:         /* If encoding fails, partId is num_part */
14:         ⟨code_fk, partId⟩ ⇐ encode(fk, dict_fact);
15:         /* If fk is encodable using dict_fact */
16:         if partId < num_part then
17:            Vec[i] ⇐ lookupKey(code_fk, HT[partId]);
18:         else
19:            Vec[i] ⇐ lookupKey(fk, HT[num_part]);
20:         end if
21:         i ⇐ i + 1;
22:      end for
23:   end if
24: end for
```

7.1 Experimental Setting

To measure the performance on a real system, we implemented five alternative configurations, shown in Table 2, in an early version of IWA.[5] DTRANS and FTRANS have already been explained in detail in Section 6. DECODE simulates traditional query processing by decoding the column values immediately before join processing. 1DICT uses the same encoding for both of the join columns as in [7], so does not require encoding translation. Finally, UNENCODE does not encode values at all.

Table 2: Five configurations used for the experiments.

Name	Description
DTRANS	Encoding translation during dimension query processing (Section 6.1)
FTRANS	Encoding translation during fact query processing (Section 6.2)
DECODE	Run-time decoding *before joining*
1DICT	Per-domain encoding, i.e., using only one dictionary without encoding translation
UNENCODE	No encoding at all

Two versions of the standard TPC-H data set were used for our experiments.

1. To more readily vary parameters and observe their effects, we initially constructed a *simplified* TPC-H data set composed of only one "fact" table LINEITEM and one "dimension" table ORDERS, each containing only the two or three columns referenced in the queries we ran. To focus on join performance and exclude other expensive operations such as GROUP BY and ORDER BY, we used the SQL query below. It simply joins O_Orderkey, which has a uniqueness constraint, with L_Orderkey, which has a (roughly) uniform distribution drawn from O_Orderkey. The local predicate on

[5]Only DTRANS is included in the released product.

L_Orderkey was omitted whenever we set its selectivity to 1. To test the effects of scaling, we generated two sizes of the fact table—100M and 500M rows—and five sizes of the dimension table—1K, 10K, 100K, 1M, and 10M rows. The results are presented in Section 7.2.

```
SELECT   COUNT(*)
FROM     LINEITEM, ORDERS
WHERE    L_Orderkey = O_Orderkey
         AND NOT ISNULL O_OrderKey
         [AND L_Orderkey < constant]
```

2. To verify that our simplified data set didn't bias our results, we also tested a wider set of queries on a *standard* TPC-H scale factor 10 data set, whose results are presented in Section 7.3.

The experiments were designed to vary four parameters: (i) the number of rows in the dimension table ($size_{dim}$), (ii) the number of rows in the fact table ($size_{fact}$), (iii) the ratio of the number of unencoded rows to the total number of rows ($ratio_{unenc}$), and (iv) the selectivity of a local predicate on the fact table (sel_{fact}). The parameters $size_{dim}$ and $size_{fact}$ were discussed earlier. We varied $ratio_{unenc}$ among 0, 1/16, 1/8, 1/4, and 1/2, re-loading the entire data set for each. We controlled sel_{fact} by varying the constant in the SQL query above.

Each experiment ran the same query seven times. To omit possibly spurious outliers, we removed the minimum and the maximum of the seven runs, and averaged the rest. Recall that IWA is a main-memory accelerator, so there is no disk I/O in any of these results. This execution time is decomposed into three component times: t_{probe}, t_{build}, and t_{base}. t_{probe} measures the time for probing the hash tables, and t_{build} that for building the hash tables. t_{base} is the base cost of just scanning the fact table without performing joins, which is the same for all alternatives except UNENCODE. It is measured by running a single table query—not a join query—that contains the same predicates on the fact table. t_{base} is useful for visualizing variable portions of the total execution time. In Figures 9∼12, these three component times are distinguished by the fill types: the dark-solid portion indicates t_{probe}, the light-solid portion t_{build}, and the cross-hatched portion t_{base}, as below.

All experiments were conducted on an IBM System x equipped with two quad-core Xeon CPUs and 48 GB of main memory. The CPU has 8 MB of shared L3 cache. The server runs on SUSE Linux Enterprise 10. Hyper-threading is disabled. Our system is implemented in C++ using GCC 4.2.

7.2 Simplified TPC-H Results

Until Section 7.2.2, all the rows are encoded ($ratio_{unenc} = 0$) to concentrate on the benefits of compression. After that, some rows remain unencoded ($ratio_{unenc} > 0$) to dig into the differences between DTRANS and FTRANS. sel_{fact} is always set to be 1, to avoid unnecessary predicates, except when varying the number of qualifying rows in the fact table.

7.2.1 Joining Encoded vs. Unencoded Data

Increasing Join Predicates

First, we show the benefit of joining encoded versus unencoded join columns as the number of those join predicates increases.[6] Figure 8 contrasts the wall clock time of the join with encoded columns (DTRANS) versus unencoded columns (UNENCODE) as the number of join columns varies from 1 to 4, where $size_{fact}$ = 100M, $size_{dim}$ = 100K, $ratio_{unenc} = 0$, and $sel_{fact} = 1.0$.

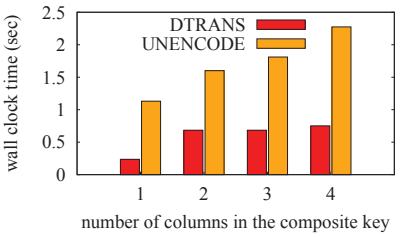


Figure 8: Benefit of encoding data.

Clearly, the advantage of joining encoded data (DTRANS) versus unencoded data (UNENCODE) increases with the number of join predicates. The initial jump in the execution time of DTRANS from 1 to 2 is caused by the triggering of the logic that concatenates multiple join-column values into chunks (each 16, 32, or 64 bits). Thereafter, its performance remains constant in the range between 2 and 4 join predicates, because the compression of the join-column values keeps the composite key within only one chunk. In contrast, the time for UNENCODE increases steadily as join columns are added, because the unencoded values increase the number of chunks.

Increasing Dimension Table Size

Next, we demonstrate the benefit of joining encoded versus unencoded data as the size of the dimension table increases from 1K to 10M rows, as shown in Figure 9. The fact table remains fixed at 500M rows ($size_{fact}$ = 500M and $sel_{fact} = 1.0$). Each bar, denoting the time for a particular configuration, is composed of t_{probe}, t_{build}, and t_{base} in the order of appearance from top to bottom. Joins on encoded data (DTRANS and FTRANS, the first and second bars) outperform traditional joins (DECODE, the third bar) that decode the data before a join by up to 40%. The reasons are twofold. First, DTRANS or FTRANS defer decoding until after the join is done. Second, the compressed values of DTRANS or FTRANS result in relatively smaller hash tables than for DECODE, reducing the number of cache misses when probing the hash tables. Further investigation reveals that the latter reason is far more significant than the former, since decoding is a simple look-up in the dictionary, so is quite fast. In this figure, DTRANS and FTRANS are identical because all data is encoded ($ratio_{unenc} = 0$).

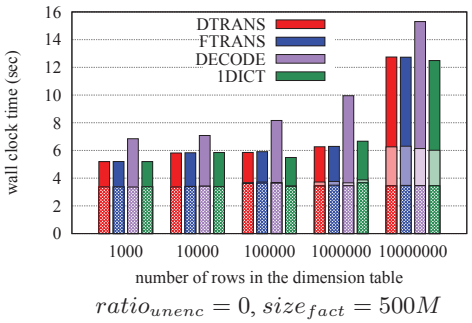
The relative improvement in t_{probe} becomes particularly dramatic when the hash tables of DTRANS fit in the cache, but those of DECODE do not, e.g., when $size_{dim}$ = 1M. Here, the size of the encoded join column is below 32 bits, so the hash tables of DTRANS or FTRANS can fit in the L3 cache, because 32 bits × 1M × 2 = 8 MB[7] (≈ the size of the L3 cache). In contrast, the hash tables of DECODE are about twice as big, and hence will not fit in L3. For dimension tables smaller than 1M rows, both the encoded and decoded hash tables fit in the L3 cache. For larger ones, however, neither of them fit. This is the reason why the margin reaches its maximum at $size_{dim}$ = 1M.

7.2.2 Per-Domain vs. Per-Column Encoding

Next, we compare the encoding of the join columns using the same dictionary (per-domain) versus separate dictionaries (per-column). Table 3 summarizes the consequences to both the compression ratio of the 10M-row dimension table ORDERS and the

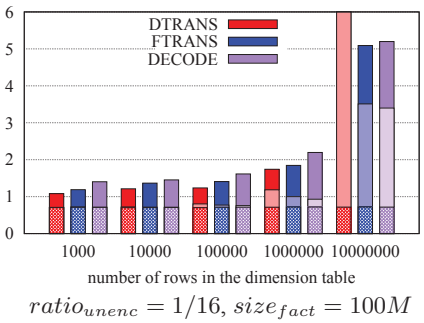
[6] This experiment required slightly varying both the data to add join columns and the query to add join predicates.

[7] The total number of buckets is by construction about two times the actual number of PK values, to minimize collisions.



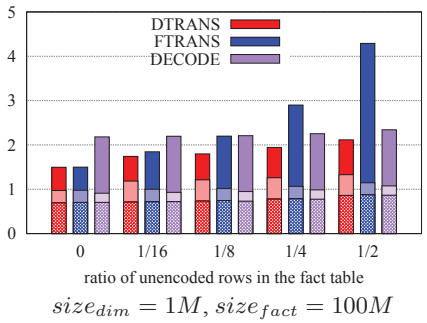
$ratio_{unenc} = 0, size_{fact} = 500M$

Figure 9: Encoded vs. decoded joins, and per-column vs. per-domain dictionaries.



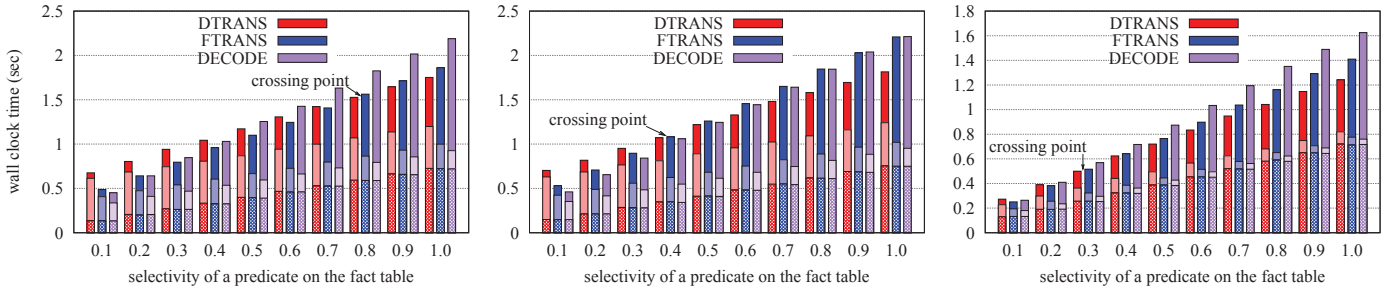
$ratio_{unenc} = 1/16, size_{fact} = 100M$

Figure 10: Effects of the size of the dimension table on encoding translation.



$size_{dim} = 1M, size_{fact} = 100M$

Figure 11: Effects of the ratio of unencoded rows on encoding translation.



(a) $ratio_{unenc} = 1/16, dim_{size} = 1M$ (b) $ratio_{unenc} = 1/8, size_{dim} = 1M$ (c) $ratio_{unenc} = 1/16, size_{dim} = 100K$

Figure 12: Effects of the size of the fact table on encoding translation.

join performance of both encoding schemes. The original and compressed sizes of the table ORDERS are shown in parentheses in the second column. These sizes roughly indicate the sizes of the PK column O_OrderKey since the two other columns occupy just 11% of the table in this simplified TPC-H data set.

As expected from our earlier explanation, DTRANS shows a compression ratio 16% better than 1DICT, because applying the encoding scheme of the FK to the PK column creates unnecessary partitions for the PK, thereby wasting space. We expect that this difference in compression ratio will only increase for a skewed data set, compared with the TPC-H data set, which has a (roughly) uniform distribution. Recall that Table 1 reported much larger differences in compression ratio.

Table 3: Effects of different encoding schemes across tables.

Alternative	Compression Ratio	Performance (sec)
DTRANS (per-column)	1.80 (158.89 / 88.09 MB)	12.75 ($t_{build} = 6.26$)
1DICT (per-domain)	1.55 (158.89 / 102.77 MB)	12.49 ($t_{build} = 6.01$)

With regard to performance, even though 1DICT omits the decoding and re-encoding that DTRANS does while scanning dimension tables, the third column in Table 3 shows little difference between the two, and then only in the t_{build} portion, which is shown in parentheses. Not surprisingly, we found that t_{build} was dominated by the cost of adding keys into hash tables and occasionally resizing those tables. These results come from the $dim_{size} =$ 10M point in Figure 9, but we get similar results for other values of dim_{size}.

These results demonstrate that we can achieve the compression and flexibility benefits of having different encoding schemes for the join columns, with only a tiny penalty in execution time.

7.2.3 DTRANS vs. FTRANS

We now compare our two variants of encoding translation, DTRANS and FTRANS, as a function of three key parameters: the size of the dimension table, the ratio of unencoded values, and the size of the fact table.

Increasing Dimension Table Size

We first investigate the effect that the size of a dimension table has on the relative performance of DTRANS and FTRANS. Figure 10 shows the wall clock time as $size_{dim}$ varies from 1K to 10M rows, holding $size_{fact}$ constant at 100M rows with $sel_{fact} = 1.0$. To exaggerate the differences between DTRANS and FTRANS, we use a nonzero fraction of unencoded rows ($ratio_{unenc} = 1/16$) for the first time in our evaluation. As expected, t_{build} grows as $size_{dim}$ increases, since more keys are inserted into hash tables, and t_{probe} also increases, since more cache misses may occur because of the larger hash tables. However, note that t_{build} increases more rapidly in DTRANS than in FTRANS as $size_{dim}$ increases, because DTRANS inserts *all* qualifying PK values into the catch-all hash table, if the catch-all cell of the fact table is not empty, whereas FTRANS does not. This overhead of DTRANS is affordable when the dimension table is small, but becomes prohibitive when $size_{dim} \geq$ 10M. Thus, DTRANS is preferred if the dimension table is small enough.

Increasing Unencoded Values in Fact Table

What happens to the relative performance of DTRANS versus FTRANS as more unencoded values are inserted into the catch-all cell of the fact table? FTRANS is shown to increasingly lose to DTRANS in Figure 11, for $size_{dim} =$ 1M rows, $size_{fact} =$ 100M rows, and $sel_{fact} = 1.0$. As the ratio of unencoded rows in the fact table, $ratio_{unenc}$, increases from 0 to 1/2, the wall clock time of FTRANS—and especially the t_{probe} portion—increases steeply. Recall that this is because FTRANS tries to encode each unencoded FK value in the catch-all cell and determines the hash table to

probe, depending on whether it is encodable or not, adding a branch as well as overhead. In short, the number of rows in the catch-all cell is the most crucial parameter for FTRANS. Thus, DTRANS is preferred if a large fraction (e.g., > 0.5) of rows are unencoded. In fact, Figure 11 shows that FTRANS is even worse than DECODE, once $ratio_{unenc} > 1/8$, because DECODE eliminates these overheads and branches in FTRANS. But DTRANS totally dominates the other two approaches in this experiment.

Note also in Figure 11 that t_{build} of DTRANS (the light-solid portion of the first bar) jumps from 0.28 at $ratio_{unenc} = 0$ to 0.47 at $ratio_{unenc} = 1/16$. This is due to the cost of inserting all PK values into the catch-all hash table, which was affordable with $size_{dim} =$ 1M in this figure.

Increasing Fact Table Size

We next investigate the effect of the size of the fact table on the relative performance of DTRANS and FTRANS, by varying the selectivity of the local predicate (L_Orderkey $<$ *constant*) on the fact table to simulate fact tables of various sizes. Figures 12(a) to 12(c) show the wall clock time as the portion sel_{fact} of $size_{fact} =$ 100M rows varies from 0.1 to 1.0, where $ratio_{unenc}$ is either 1/16 or 1/8, and $size_{dim}$ is either 1M or 100K. In these figures, both t_{base} and t_{probe} increase as sel_{fact} increases, which are expected because more rows satisfy the selection predicate and more rows need to be checked for the join, respectively.

However, note that t_{probe} increases more rapidly in FTRANS than in DTRANS as sel_{fact} increases. When unencoded rows of the fact table are processed, FTRANS requires additional operations for encoding translation, whereas DTRANS does not. Here, a larger number of unencoded rows are included in query processing as sel_{fact} increases. For this reason, FTRANS outperforms DTRANS for low selectivity, and the opposite is true for high selectivity. Thus, FTRANS is preferred if the fact table is small enough or if any local predicates on the fact table are sufficiently selective.

The point at which the performance of the two alternatives cross depends on several parameters. The crossing point appears at 0.4 in Figure 12(b), earlier than at 0.8 in Figure 12(a) owing to a higher value of $ratio_{unenc}$. The higher $ratio_{unenc}$ is, the higher the overhead of encoding translation in FTRANS is. The crossing point appears at 0.3 in Figure 12(c), owing to a smaller value of $size_{dim}$ in comparison with Figure 12(a). The smaller $size_{dim}$ is, the lower the overhead of encoding translation in DTRANS is.

7.3 Standard TPC-H Results

We now show the results for the standard TPC-H data set, on a wider range of queries. Since the first version of IWA was not yet able to handle nested queries, we limited our experiments to those six TPC-H queries not having nested query blocks, then eliminated GROUP BY, ORDER BY, and other expensive clauses (e.g., CASE) from these queries to concentrate on join performance. Tables 4 and 5 report the results when $ratio_{unenc}$ is 0 and 0.01, respectively; i.e., in Table 5 we force 1% of the rows to be unencoded at initial loading to see the difference between the two alternatives. The first number of each entry indicates the time for processing the dimension tables, and the second that for processing the fact table.

In Table 4, DTRANS and FTRANS (in bold) are identical because all rows are encoded, and both outperform DECODE. However, the difference isn't very significant here, since the hash tables from the dimension-table scan are either much smaller or larger than the L3 cache. Hence, either both the encoded and unencoded hash tables fit in the L3 cache, or neither of them does.

In Table 5, FTRANS still outperforms DECODE. However the performance of DTRANS becomes poor, with the cost of the build phase (the first number for each query) consistently being about twice that of the other two methods. This is because the first four queries—Q3, Q5, Q10, and Q12—reference the dimension ORDERS, which contains 15M rows, and the last two queries—Q14 and Q19—reference the dimension PART, containing 2M rows. These two dimension tables are large enough that the cost of duplicating their PK values into the hash tables becomes the dominating cost in DTRANS.

Table 4: Times, in seconds, to process the dimension tables and the fact table in TPC-H queries, when $ratio_{unenc} = 0$.

	DTRANS	FTRANS	DECODE
Q3	**4.559, 1.168**	**4.530, 1.163**	4.635, 1.394
Q5	**4.572, 1.472**	**4.592, 1.459**	4.688, 1.899
Q10	**4.533, 1.166**	**4.561, 1.170**	4.650, 1.385
Q12	**4.063, 1.165**	**4.076, 1.157**	4.076, 1.388
Q14	**0.418, 1.086**	**0.419, 1.097**	0.421, 1.352
Q19	**0.417, 1.083**	**0.419, 1.075**	0.426, 1.358

Table 5: Times, in seconds, to process the dimension tables and the fact table in TPC-H queries, when $ratio_{unenc} = 0.01$.

	DTRANS	FTRANS	DECODE
Q3	9.293, 1.175	**4.562, 1.173**	4.638, 1.398
Q5	9.396, 1.474	**4.583, 1.465**	4.682, 1.906
Q10	9.316, 1.180	**4.567, 1.169**	4.618, 1.384
Q12	8.500, 1.167	**4.063, 1.159**	4.093, 1.387
Q14	0.840, 1.077	**0.416, 1.064**	0.419, 1.339
Q19	0.839, 1.077	**0.417, 1.069**	0.419, 1.335

7.4 Summary of the Results

Our results can be summarized as follows:

1. Our DTRANS or FTRANS outperforms traditional DECODE for most cases by up to 40%.
2. DTRANS or FTRANS improves the compression ratio by at least 16% (or up to 50% in Table 1), with negligible overhead in query processing, in comparison with having one dictionary for both join columns (1DICT).
3. DTRANS is preferred when the dimension tables are small.
4. FTRANS is preferred when the fact tables are small or local predicates on the fact tables are very selective.
5. DTRANS is preferred when a large fraction of rows are left unencoded.

8. RELATED WORK

Compression in DBMSs has been actively studied in both the database industry and academia. There are two main research directions. One direction is to develop compression schemes that achieve better (de)compression speed and/or a better compression ratio. The other direction is to develop query processing methods that take advantage of compressed data. Our work represents a blend of both directions.

Existing methods for joins on compressed data can be classified as in Table 6. The horizontal axis represents whether data is compressed at load time or not, and the vertical one whether the data is decompressed at run time or not. Traditional database systems that do not support compression belong to the lower-right category. Our framework falls into the upper-left category, and it also contributes to the upper-right category through our on-the-fly encoding.

Most commercial DBMSs store data compressed, but they usually decompress the data before it is processed, so belong to

Table 6: Classification of query processing on compressed data.

Loading / Run time	Compressed	Uncompressed
Compressed	Decompress *after* query processing	Compress *before or during* query processing; decompress *after*
Uncompressed	Decompress *before* query processing	No compression

the lower-left category. Storing the data compressed reduces the amount of disk I/O's, but decompressing immediately means that even non-qualifying rows incur the cost of decompression and the uncompressed data consumes more memory and cache throughout query processing. Chen et al. [4] proposed a query optimization method that takes a decompression cost into account. Westmann et al. [21] discussed how compression can be integrated into a relational query engine. Specifically, they suggested an extension of the iterator model to avoid repeated decompressions.

Motivated by the high decompression cost, Graefe and Shapiro [7] claimed that query processing on compressed data can improve query performance. To simplify the system, they use only a single compression scheme for each domain. By requiring all values from the same domain to be compressed with the same dictionary, join columns are compressed in exactly the same way, and compressed key values can be directly compared with each other to find matches during join processing. This approach eliminates the need for any decompression during a join, but it can result in suboptimal compression decisions and encodings for individual columns, as shown earlier.

Abadi et al. [1] proposed an architecture of a column-oriented database system that supports direct operations on compressed data. The architecture is prototyped in C-Store [20], whose ordered "projections" primarily favor run-length encoding. Their query processing algorithms therefore focus on how to exploit compression schemes that represent multiple values in a single field. In such schemes, multiple results can be generated simultaneously by a single operation. This work, however, does not address the problem of how to handle *different* compression schemes on the columns being joined. Thus, this work, like that of Graefe and Shapiro, corresponds to our `1DICT` scheme.

In SAP HANA [5], every column is compressed using a sorted dictionary. That is, each value is mapped to an integer value (code). These codes are further compressed by several techniques such as run-length encoding (RLE), prefix coding, sparse coding, and cluster coding. The standard query operators are directly applied to the compressed data structures. Lemke et al. [11] proposed scan and aggregation operators that are designed to operate on top of compressed data. However, it is not known how joins are processed for independently compressed columns. In addition, HANA does not support a mix of compressed and uncompressed values.

9. CONCLUSIONS

In this paper, we proposed a technique for processing joins on encoded and partitioned data. First, we demonstrated the benefit of independently optimizing the compression scheme of each column and the benefit of partitioning a column to deal with incremental updates. Then, two variants of encoding translation were developed to execute the join under this environment. Next, an on-the-fly encoding scheme was developed to encode non-join columns as well. Last, by conducting extensive experiments with our product system, we showed that our technique achieves both superior performance and excellent compression.

10. REFERENCES

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

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, pages 169–180, 2001.

[3] R. Barber et al. Blink: Not your father's database! In *BIRTE*, pages 1–22, 2011.

[4] Z. Chen, J. Gehrke, and F. Korn. Query optimization in compressed database systems. In *SIGMOD*, pages 271–282, 2001.

[5] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[6] J.-L. Gailly. The gzip home page. `http://www.gzip.org`. Accessed: 2013-12-05.

[7] G. Graefe and L. D. Shapiro. Data compression for database performance. In *Symp. on Applied Computing*, 1991.

[8] IBM. DB2 data compression and storage optimization. `http://www-01.ibm.com/software/data/db2/linux-unix-windows/storage-compression.html`. Accessed: 2013-12-05.

[9] R. Johnson, V. Raman, R. Sidle, and G. Swart. Row-wise parallel predicate evaluation. *PVLDB*, 1(1):622–634, 2008.

[10] P.-Å. Larson et al. Enhancements to SQL server column stores. In *SIGMOD*, pages 1159–1168, 2013.

[11] C. Lemke, K.-U. Sattler, F. Faerber, and A. Zeier. Speeding up queries in column stores – a case for compression. In *DaWak*, pages 117–129, 2010.

[12] Y. Li and J. M. Patel. BitWeaving: Fast scans for main memory data processing. In *SIGMOD*, pages 289–300, 2013.

[13] Microsoft. Page compression implementation, SQL Server 2012. `http://technet.microsoft.com/en-us/library/cc280464.aspx`. Accessed: 2013-12-05.

[14] Netezza. The Netezza fast engines framework. White Paper, 2008.

[15] S. J. O'Connell and N. Winterbottom. Performing joins without decompression in a compressed database system. *ACM SIGMOD Record*, 32(1):6–11, 2003.

[16] P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *ACM SIGMOD Record*, 24(3):8–11, 1995.

[17] Oracle. Advanced compression with Oracle Database 11g. White Paper, 2012.

[18] V. Raman et al. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[19] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. In *ICDE*, pages 60–69, 2008.

[20] M. Stonebraker et al. C-Store: A column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[21] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *ACM SIGMOD Record*, 29(3):8–11, 2000.

[22] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-Scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, 2009.

[23] M. Zukowski, S. Heman, N. Nes, and P. A. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, pages 59–59, 2006.