Data Compression in Oracle

Meikel Poess
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065
USA
meikel.poess@oracle.com

Dmitry Potapov
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065
USA
dmitry.potapov@oracle.com

Abstract

The Oracle RDBMS recently introduced an innovative compression technique for reducing the size of relational tables. By using a compression algorithm specifically designed for relational data, Oracle is able to compress data much more effectively than standard compression techniques. More significantly, unlike other compression techniques, Oracle incurs virtually no performance penalty for SQL queries accessing compressed tables. In fact, Oracle's compression may provide performance gains for queries accessing large amounts of data, as well as for certain data management operations like backup and recovery. Oracle's compression algorithm is particularly well-suited for data warehouses: environments, which contains large volumes of historical data, with heavy query workloads. Compression can enable a data warehouse to store several times more raw data without increasing the total disk storage or impacting query performance.

1. Introduction

The amount of data businesses are retaining for data warehouse applications is exploding at a record rate because. For comprehensive mining purposes data warehouses not only keep vast amounts of detailed data (e.g. call and click-stream data) but also store this data over an extended period of time. In the past commercially available database systems have not heavily utilized compression techniques on data stored in relational tables. A standard compression technique may offer space savings, but only at a cost of much increased query elapsed time. Hence, this trade-off has made compression not always attractive for relational databases.

In this paper we introduce an innovative table compression technique, recently introduced in the Oracle RDBMS [8] that is very attractive for large relational data warehouses. It can be used to compress tables, table partitions and materialized views (these database objects are essentially implemented as tables). The status of a table can be changed from compressed to non-compressed at any time by simply adding the keyword `COMPRESS` to the table's meta-data. Changing the status of a table does not compress its existing contents. Only newly loaded rows are compressed allowing for a mixture of compressed and non-compressed rows to coexist. On the other hand Oracle's RDBMS offers the possibility to compress an already existing table in its entirety without reloading it.

The reduction of disk space using Oracle table compression can be significantly higher than standard compression algorithms, because it is optimized for relational data. It has virtually no negative impact on the performance of queries against compressed data; in fact, it may have a significant positive impact on queries accessing large amounts of data, as well as on data management operations like backup and recovery. These benefits of compression come at the cost of increased load and update times. However, used in conjunction with other features of the Oracle RDBMS these performance degradations can be compensated for. For instance, since data in a typical data warehouse application is organized chronologically, it can be partitioned by day, month or year using Oracle's partition feature. As new data from the operational database arrives only the most current partitions are updated. In order to avoid performance degradation during updates the most current partitions can be kept non-compressed until no more updates occur on them. Then they can be compressed.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment

Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003

1.1 Other Compression Techniques

Data compression techniques adopted in other RDBMS for Data Warehouse Applications [6,7] also implement a lossless dictionary-based approach. However, they use static, table-wide dictionaries instead of dictionaries optimised on a block level. Due to its global optimality of compression a table-wide dictionary approach can result in high compression factors. This is beneficial for uniformly distributed, static data. Furthermore, a global dictionary may use less disk space compared to a block level approach, which potentially repeats dictionary entries across blocks. However, compared to a global dictionary approach Oracle's block based dictionary implementation shows many benefits. In comparison to [6] Oracle's dictionary entries are created by the system rather than requiring the user to specify them manually. A 20-column table using the compression technique proposed in [6], requires the user to specify up to 20x255=5100 dictionary entries. On the other hand, since data warehouse tables are periodically refreshed, the list of the most frequent column values can change over time, making it necessary to recreate the entire table. Secondly, Oracle's algorithm dynamically adapts to changes in data distribution without compromising the compression factor. While [6] utilizes a table wide list of most frequent column values, [7] exploits the column value frequency of the first rows to populate the dictionary. This can lead to sub-optimal compression when data distributions change over time. For instance, as a data warehouse is refreshed new dates, which are not present in the dictionary, enter the system while old dates, which are present in the dictionary, are purged out of the system. Thirdly, in order to access a row, in Oracle only one block needs to be accessed, as opposed to multiple blocks in case of a global symbol table, greatly increasing buffer cache efficiency and reducing memory cache misses. If compressed column values are accessed, using a global dictionary it is necessary to access multiple blocks. Even if dictionary blocks are pinned into the buffer cache, accessing multiple blocks to uncompress a column value increases memory cache misses and, therefore, adversely increases CPU time.

The remainder of this paper is organized as follows. In Section 2 we demonstrate how compressed blocks are stored within the Oracle RDBMS and how load, update and query operations operate on compressed blocks. In Section 3, we analyze how well data can be compressed utilizing data extracted from a life customer data warehouse. In section 4, we conduct experiments to evaluate the performance impact of table compression on common data warehouse operations. In section 5, utilizing the industry standard data warehouse benchmarks, TPC-H, we demonstrate the impact of table compression to load, query and update performance of a comprehensive data warehouse environment. We draw our conclusions in Section 6.

2. Table Compression Implementation

The compression algorithm used in Oracle for large data warehouse tables compresses data by eliminating duplicate values in a database block (aka. database page). The algorithm is a lossless dictionary-based compression technique. The compression window for which a dictionary (symbol table) is created consists of one database block. Therefore, compressed data stored in a database block is self-contained. That is, all the information needed to recreate the uncompressed data in a block is available within that block.

2.1 Compressed vs. Non-Compressed Blocks

Figure 1 illustrates the differences between storing data in a compressed versus non-compressed block. With the exception of a symbol table in the beginning, compressed database blocks look very much like regular database blocks. Code modifications done in the Oracle RDBMS server to allow for compression are very localized. Only the portions of the code dealing with formatting blocks, and accessing rows and columns were modified. As a result, accessing a compressed block is completely transparent to the database user or any application, and all database features and functions that work on regular database blocks also work on compressed database blocks with the exception of dropping columns. The top part of Figure 1 shows a typical data warehouse like fact table with rowid, invoice id, customer first name, customer last name and sales amount. There are entries for five customers showing six purchases. For data warehouse fact table it is very common to have this highly denormalized structure.

The bottom left part of Figure 1 shows how a non-compressed block stores the data of the fact table: all the redundant information is stored. The bottom right part shows how the same data is stored in a compressed block: instead of storing all data, redundant information is replaced by links to a common reference in the symbol

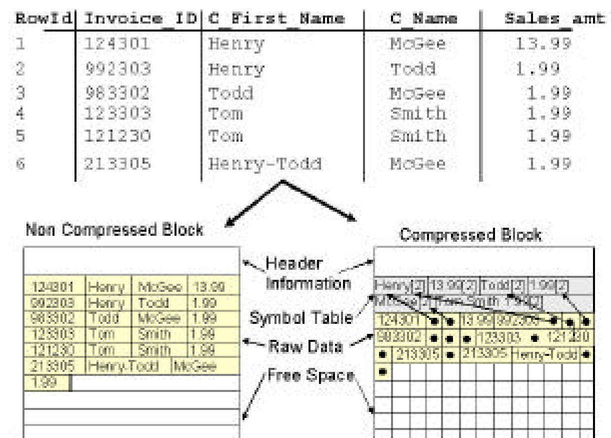

Figure 1: Compressed Block vs. not Compressed Block

table, indicated by the black dots (for readability not all references are illustrated with arrows). For each column value in all columns, based on length and number of occurrences in one block, the algorithm decides whether to create an entry into the symbol table for this column value. If column values from different columns have the same values, they share the same symbol table entry. This is referred to as cross-column compression. Only entire column values or sequences are compressed. Sequences of columns are compressed as one entity if a sequence of column values occurs multiple times in many rows. This is referred to as multi-column compression. This optimisation is particularly beneficial for OLAP type materialized views using grouping sets and cube operators. For instance a cube of a table often repeats the same values along dimensions creating many potential multi-column values. Multi-column compression can significantly increase the compression factor and query performance. In order to increase multi-column compression, columns might be reordered within one block. For short column values and those with few occurrences no symbol table entry is created limiting the overhead of the symbol table and ensuring that compressing a table never increases its size. However, this is transparent to any application.

For instance the name "Henry" occurs twice as first name (rows 1 and 2). Consequently, column values for "Henry" are replaced with a link into the symbol table. The name "Todd" appears twice, once as a first name (row 3) and once as a last name (row 2). In this case, the compression algorithm references both, first and last name to the same symbol table entry (cross-column compression). For rows 4 and 5 the compression algorithm uses multi-column compression. Instead of compressing "Tom", "Smith" and "1.99" as separate entities, it combines all three into one symbol table entry "Tom|Smith|1.99" reducing the number of references in rows 4 and 5 to only one. Unique columns, for instance the invoice id, are not compressed. Also, the current implementation does not allow for partial column compression. Hence, the first name in row 6, "Henry-Todd", although both name parts exists as references in the symbol table does not get compressed.

2.2 Compression of Database Blocks

Figure 2 outlines the steps of the compression process. During load operations data is first loaded into a block in its uncompressed format (Step 1). The result is a block with a compression factor of 1 (= no compression), where rows occupy the entire block (NC=block size). After one block is fully loaded, the compression process of this block starts (Step 2). This process tries to convert the non-compressed block into its compressed format as described in the previous section. The result is a modified block where rows occupy a fraction of the block (C). A

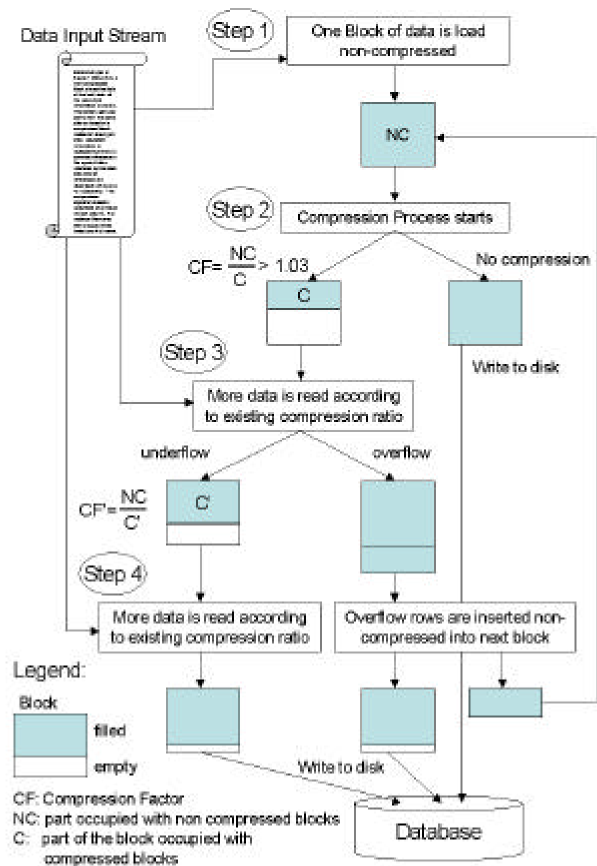

Figure 2: Process of Compressing a Block

local compression ratio (CF[1]) smaller than 1.03[2] indicates that this block is not worthwhile compressing. This block is written to disk and the next block is filled in Step 1. A local compression ratio larger than 1.03 triggers to investigate whether to further compress this block can increase its compression ratio. Its local compression ratio is used to determine how many compressed rows are likely to fit into an entire block via extrapolation. That is, assuming the local compression ratio CF, the average row length and the remaining free block space the algorithm calculates the number of rows it can further fit into this block. This number of rows is read into a buffer and compressed (Step 3). If not all buffered rows fit into a compressed block, an overflow occurs. That is, the extrapolation calculation based on CF was wrong, due to changes in data distribution and this block is written to disk. The remaining rows (overflow rows) are inserted non-compressed into the next block (Step 4). If all buffered rows fit into a compressed block, the previously calculated compression factor (CF) might be too conservative (possible underflow situation). Considering

[1] $CF = \frac{NC}{C}$

[2] This fudge factor was empirically measured to reflect the overhead of compression

the compression factor off all rows just loaded (CF'[3]), the algorithm again extrapolates how many more rows are likely to fit into this block. If the remaining space allows for at least one more row to be inserted, the algorithm considers an underflow situation and buffers as many rows as it extrapolates fit into this block (Step 5). The resulting compressed block is written to disk and the next block is loaded in Step 1.

This semi-offline compression technique has been chosen to achieve local optimality of compression ratio. The algorithm is greedy, meaning that it tries to load as many rows as possible into each block. It does not change row ordering and does not attempt to achieve any form of global compression ratio optimality. The problem of global compression ratio optimality is similar to global space optimality, which is very computationally intensive, as it is closely related to the problem of single-dimensional bin packing. If global compression ratio optimality is desired, the entire set of rows to be compressed need to be buffered before blocks can be populated. For large data warehouses this is not feasible because it would potentially require to buffer terabytes of data, which is not practical. Therefore, the algorithm processes the row set to be loaded in a streaming fashion with some offline characteristics for better compression quality. Section "Space Savings" shows that this compression implementation achieves very high compression ratios on typical data warehouse data. As discussion with customers show load time is very important as some only have a limited time window available for refreshing their data warehouse.

2.3 Query access

Since Oracle's compression implementation works via duplication elimination in each block at column level, column values are present in the block's symbol table in their non-compressed format (see Figure 1). This has many advantages for read only data access of column values. Firstly, a single row access only touches one block preserving locality of reference. Secondly, no expensive decompression operations need to be performed for read access. Decompression essentially means to follow short references to columns into the symbol table and locating the right column or column sequence.

To take further advantage of compression, a couple of optimizations are implemented. The first one optimizes predicate evaluation for sequential row access. Using this optimization predicates on compressed column, values are evaluated at most once per block instead of once per row as in the non-compressed case, giving queries accessing compressed columns a very substantial benefit. This optimisation takes advantage of the fact that compressed blocks contain information about data duplication in a block. Assuming a query applies the predicate `color = 'green'` too all rows of a table. The query engine reads one block at a time, accesses the field `color` for all rows and evaluates the above predicate. For compressed column values, after the predicate has been evaluated, the result is kept. If a subsequent compressed column value is accessed for the same value, this predicate does not need to be evaluated since the result is already available. The reference into the symbol table in this case also refers to the predicate result.

The second optimization use multi-column compression to increase column access. In Oracle columns are chained in order of creation time. In order to access column C_k of a N-column table $(C_1,C_2,...C_k,...C_N)$ k-1 references in a linked list need to be traversed. Multi-column compression can reduce the number of references that need to be traversed, because multiple columns are replaced with one reference into the symbol table (see Section 2.1). However, in case a multi-column compressed column is accessed, a minor overhead occurs because the multi-column compression is more expensive to decompress.

2.4 Updates and Deletes

Update and Delete operations on compressed data work very similar to update and delete operations on uncompressed data. The only difference is that if a compressed column is updated or an entire row is deleted the symbol table needs to be maintained. As part of each symbol table entry a reference counter is maintained. When a column is updated the algorithm checks whether a symbol table entry for the new value exists. If it exists, the reference of the updated column is modified to the new symbol table entry and its reference count is increased by one. At the same time the reference count of the old value is decreased by one. On the other hand, if no symbol table entry exists for the new column value, that value is inserted non-compressed into the row. During delete operations all references counters of the deleted rows are decreased by one. Once a reference counter becomes zero, the corresponding symbol table entry is purged from the symbol table. A symbol table is never deleted from a block even if no reference into it exists because the overhead of an empty symbol table is only 4 bytes.

Some update operations can take significant advantage of compression. For instance, operations that set a column to the same value in all rows of one table as in:

```
UPDATE TABLE item
SET i_color = 'green'
WHERE i_color ='blue'.
```

In this case a new symbol table entry for the new value is created in the symbol table and all rows are updated to reference this entry. If the old column value

[3] $CF' = \frac{NC}{C'}$

(in the above example 'green') was also compressed (i.e. a symbol table existed for it) and its reference count after the update operation became zero, the old symbol table entry is replaced with a new symbol table entry without touching all rows of one block. This is clearly much more space and time efficient than updating all rows of one block.

3. Space Savings

Table compression can significantly reduce disk and buffer cache requirements for database tables. Since the compression algorithm utilizes data redundancy to compress data at a block level, the higher the data redundancy is within one block, the larger the benefits of compression are. Although there might be data redundancy across data blocks, that data cannot be used to further compress data.

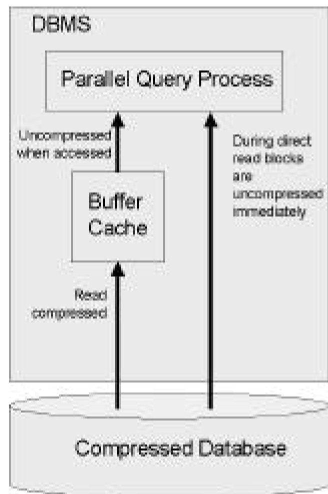

Figure 3: Data Access Path with Compression

If a table is defined "compressed" it will use fewer data blocks on disk, thereby, reducing disk space requirements. Data from a compressed table is read and cached in its compressed format and it is decompressed only at data access time. Because data is cached in its compressed form, significantly more data can fit into the same amount of buffer cache (see Figure 3).

In order to avoid any confusion about how compression and space savings is measured, we define compression factor and space savings: Compression factor (CF) of a table is defined as the ratio between the number of blocks required to store the non-compressed table compared to the number of blocks needed for the compressed version:

$$CF = \frac{\#non_compressed_blocks}{\#compressed_blocks}$$

The space savings (SS) are therefore defined as:

$$SS = \frac{\#non_compressed_blocks - \#compressed_blocks}{\#non_compressed_blocks} \times 100$$

Unique fields, (fields with a high cardinality) such as primary keys cannot be compressed, whereas fields with a very low cardinality can be compressed very well. On the other hand, longer fields yield a larger compression factor since the space saving is larger than for shorter fields. Additionally, if a sequence of columns contains the same content, the compression algorithm can apply multi-column compression. In most cases, larger block sizes increase the compression factor for a database table as more column values can be linked to the same symbol table. Sorting data before loading can further increase the compression factor. The more fields of the same content that are concentrated in each block the more efficiently the compression algorithm works. If one knows that one or multiple fields of a database object have similar values - indicated by a low number of unique values - sorting the data on those fields is likely to increase the compression factor. However, sorting on fields with very low cardinality does not necessarily yield a large compression factor increase. Due to the low cardinality of this field, rows with the same value can be found already at a high concentration in each block. Therefore, best results can be achieved by sorting on a field that is both long and has a medium cardinality.

3.1 Space Savings on Customer Data

As mentioned in earlier sections compression factors depend on many parameters such as the frequency of values, its distribution in the input stream etc. This section demonstrates compression factors that were measured on customer data. The data is modeled utilizing a star schema. The fact table `DAILY_SALES` is the center surrounded by the dimensions `TIME`, `CUSTOMER`, `SALES REGION`, `ITEM` and `PROMOTION`. In addtition to the regular table there are two summary tables defined on `SALES`: `WEEKLY_SALES` and `WEEKLY_AGGR`. `WEEKLY_SALES` aggregates `SALES` for each item and customer to weekly numbers. `WEEKLY_AGGR` builds on `WEEKLY_SALES` by aggregating further on postal codes.

Fact and summary tables are usually the largest tables in a star schema representing 70% or even more of the total database size. In contrast dimension tables are very small. Hence, compressing dimension tables does not yield an overall large disk savings and should only be considered when dealing with very large dimensions. We therefore only compress the fact table and materialized views of our test schema configuration.

Figure 4 illustrates how well data of the customer star schema compresses. The size for `SALES` decreases from 27GB to 8.6GB yielding a compression factor of 3.1. The two materialized views compress at compression factors of 2.9 and 4.0. `WEEKLY_SALES` shrinks from 18.8GB to 6.5GB while `WEEKLY_AGGR` shrinks from 7.5GB to 1.9GB yielding a space savings of 67 to 75 percent. That is, the compressed version of `WEEKLY_SALES` requires only 25% disk and buffer cache space than their uncompressed counterpart while the compressed versions of `DAILY_SALES` and `WEEKLY_SALES/ WEEKLY_AGGR` require only 33% of the resources that their uncompressed counterparts use. The overall database size reduces from 55GB to 18GB. The space saving, compressing only the fact tables and their materialized views, achieved on the customer's entire star

schema is about 67% at a compression factor of about 3.1. This shows the key benefit of compression: reduction of database space by 2/3.

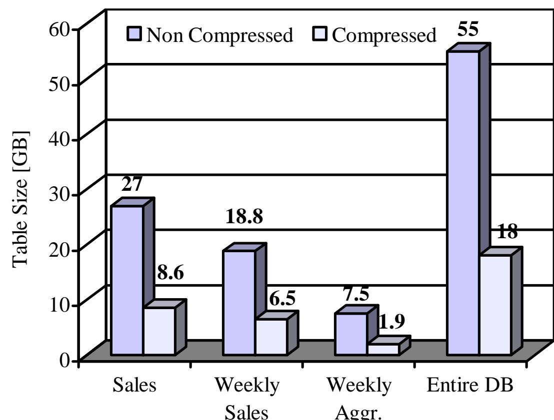

Figure 4: Compression Results of Customer Star Schema Tables

4. Performance Analysis

This section investigates the performance impact of compressed tables on typical data warehouse operations, such as the creation (loading), updating and querying of compressed tables. Compressed tables can be created in multiple ways, via loading from external media or other tables. The load experiment reads data from an external media. Updating compressed tables include UPDATE and DELETE operations. INSERT operations are omitted since the code path for inserting new data is already tested as part of the load experiment. Although the execution plans of data warehouse queries can be very complex, possibly involving multiple data access methods, multiple joins, sort and aggregation operations, only the data access methods are affected by compressed tables. The following query experiments focus on full table scans and row access by row identifier (rowid). They test the two ways compressed rows are accessed in Oracle, namely directly from disk or via the buffer cache.

The analysis focuses on the CPU and IO performance during the above described operations. In order to compare two performance tests the following measures are used:

$$CPUC = \sum_{t=1}^{T} c_i(t_i - t_{i-1}) \text{, where } c_i \in \{c_1, c_2, ..., c_T\}$$

is the CPU utilization for sample i in the measurement interval T; where $t_i \in \{t_0 = 0, t_1, ..., t_T\}$, measured in CPUU (CPU seconds), and

$$IOC = \sum_{t=1}^{T} io_i(t_i - t_{i-1}) \text{, where } io_i \in \{io_1, io_2, ..., io_T\}$$

is the IO utilization for sample i in the measurement interval T; t_i defined as above, measured in MB.

4.1 Experiment Setup

For the experiments conducted in this section we use one table (`SALES`), which mimics a star schema like fact table. It has about 2.5 Million rows and 19 columns. Loaded into a non-compressed table, its data resides in 152472 data blocks and 28354 data blocks when loaded into a compressed table. With a data block size of 8 KB, this is about 1.2GB non-compressed and 221MB compressed. The compression factor, therefore, is about 5.4. As described in Figure 5 the number of distinct values in columns of the SALES fact table varies between 1 and 2.5 Million (unique). There is a mixture of character and number columns.

Column Name	Number of distinct values
SS_SOLD_DATE_SK	1707
SS_SOLD_ITEM_SK	10989
SS_SOLD_CUSTOMER_S	9990
SS_SOLD_ADDR_SK	5000
SS_SOLD_STORE_SK	5
SS_TICKET_NUMBER	2488060
SS_QUANTITY	100
SS_WHOLESALE_COST	9888
SS_LIST_PRICE	15570
SS_SALES_PRICE	15570
SS_EXT_DISCOUNT_AM	1
SS_EXT_SALES_PRICE	104442
SS_EXT_WHOLESALE_C	117296
SS_EXT_LIST_PRICE	104442
SS_EXT_TAX	27639
SS_COUPON_AMT	1
SS_NET_PAID	104442
SS_NET_PAID_INC_TAX	148121
SS_NET_PROFIT	30359

Figure 5: Table Characteristics

The system used is a 24 CPU Sun server with 400 MHz ultra sparc processors and 8 GB of main memory. The maximum IO throughput exceeds 300MB/s. Hence, neither IO, main memory nor CPU are bottlenecks during all tests. However, the degree of parallelism, the operations are executed, was limited to four, two or one, making this the bottleneck of the execution.

4.2 Load Test

Typically a data warehouse is loaded initially and then periodically refreshed in a data maintenance phase during which data is sometimes transformed to suit the format and semantics of the data warehouse. Regardless of the phases and whether transformations occur inside or outside the database, data needs to be loaded into the data warehouse in a timely fashion. Oracle's preferred tool to load data into data warehouses is its "External Tables" feature [5].

The load tests consist of loading 2.5 Million rows (1.2 GB) with a parallel degree of four utilizing external tables into the SALES tables both compressed and non-compressed. Each load is done with direct insert, bypassing the buffer cache and any row level logging. However, full ACID requirements are fulfilled during the

load tests. The following database operations share the same compression code path as parallel load: "`move table compress`"[4], "`move partition compress`"[5], "`create table as select`"[6]. Since they show the same performance behavior as parallel load, they are not analyzed in separate experiments.

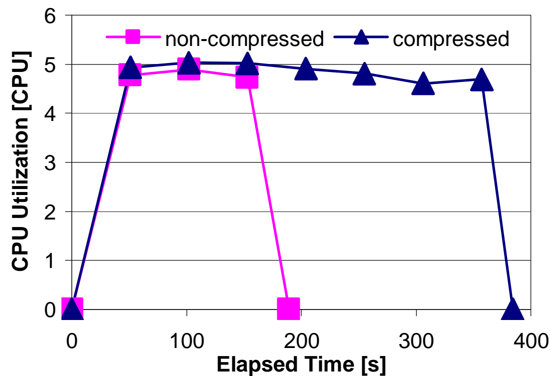

Figure 6: Parallel Load Performance (CPU)

The non-compressed test completes in 181s while the compressed test takes twice as long (364s). The compression factor of 5.4 results in 221 MB to be written by the compressed test and 1191 MB to be written by the non-compressed test. Figure 6 shows the CPU utilization during the execution of both loads. During their runs both tests utilize about 5 CPUs. However, the load into a compressed table indicated by the triangles takes 1950 CPUU compared to only 897 CPUU for the non-compressed run; an increase of 117%.

Figure 7 shows the IO utilization of the system during

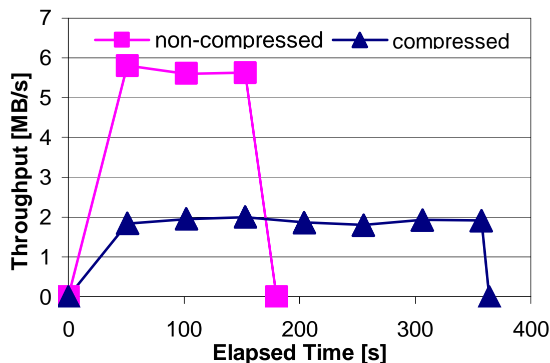

Figure 7: Parallel Load Performance (IO)

the two loads. During the non-compressed load the IO throughput is about 5 MB/s and only about 2 MB/s during the compressed. The ratio between the total number of bytes written in both cases corresponds to the compression ratio of 5.4.

This is not surprising as the process of transforming an uncompressed block into its compressed representation is computationally expensive. Please refer to Section "Compression Implementation" for a description of the algorithm. As will be shown in later sections, this approach to compression benefits the retrieval of information not its loading. Considering that the majority of the accesses to a data warehouse are read only this has been a design goal of table compression.

4.3 Delete and Update (DML) Operations

In data warehouses DML operations, such as delete and update, are not very common. They are mostly used to clean erroneously inserted records or to refresh materialized views. Sometimes delete operations are used to clean up old records so that the amount of data kept in the data warehouse is constant, as new rows from the operational system are inserted. However, this can be done more efficiently with drop partition operations. Compared to DML operations in OLTP environments, where only a few records are purged or updated, DML operations in data warehouse environments purge or update many records.

The delete experiments described in this section consist of purging rows (transaction records) from a non-

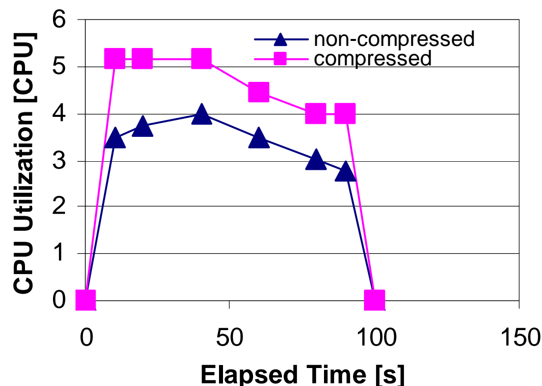

Figure 8:Delete Operation CPU Utilization

compressed and a compressed `SALES` fact table, which occurred in three stores (number 1,2 and 3). It deletes 1.5 million rows (60%) of the `SALES` table with a parallelism degree of four. Before each test the `SALES` table is loaded into the buffer cache to minimize IO interference (warm cache).

```
DELETE FROM TABLE sales
WHERE ss_store_id IN (1,2,3)
```

Figure 8 shows the CPU utilization during the delete operations of the non-compressed table and compressed

[4] compresses an existing table without copying its content
[5] compresses an existing partition without copying its content
[6] creates a table by querying another table

table. The delete operations take about the same amount of time (100 second) to complete. The total CPU consumption during the non-compressed delete is about 31 CPUU and 42 CPUU during the compressed delete. This is an overhead of about 27%.

The update tests update one field for 20% (about 500,000 rows) of all rows in a compressed and non-compressed version of the SALES fact table with parallelism degree of four. Before each test, the SALES fact table is cached into the buffer cache.

```
UPDATE store_sales_c_t
SET ss_quantity = ss_quantity+1
WHERE ss_sold_store_sk =1
```

The non-compressed update test completes in about 87s. With 121s the compressed test takes about 40% longer to complete. The CPU utilization of the non-compressed run is slightly higher than that of the compressed run (see Figure 9). However, since, the elapsed time of the compressed update test is 20% longer, total CPU consumption is 40 CPUU, 50% higher than the non-compressed update test.

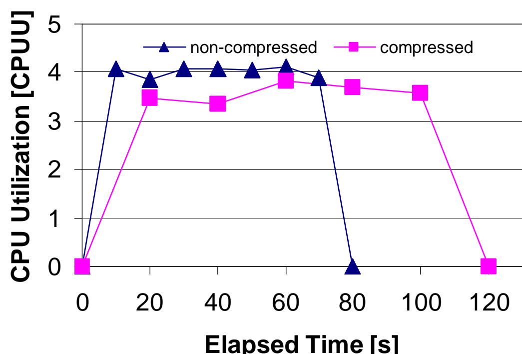

Figure 9:Update Operation CPU Utilization

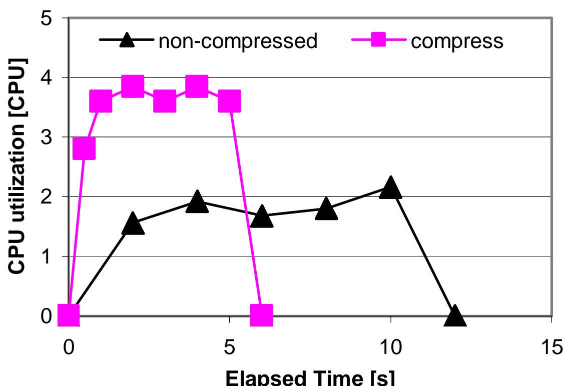

Figure 10: Parallel Full Table Scan CPU Utilization

Similar to load operations, compression imposes a slight overhead on delete and update operations because of their added cost to maintaining the symbol table as described in Section "Compression Implementation".

4.4 Full Table Scan

Full table scans (fts) are very often performed as part of a larger query execution to access tables with no suitable index defined, for instance during hash join operations. Otherwise an index scan or regular index access is performed. The same code path that is used in fts is also used during partition scans and rowid range scans. The full table scan test used in this experiment consists of scanning the entire SALES table in parallel degree 4 aggregating on all fields:

```
SELECT COUNT(ss_sold_date_sk),
       COUNT(ss_sold_date_item_sk),
       …
       COUNT(ss_net_paid_inc_tax),
       COUNT(ss_net_profitt)
FROM   store_sales_c;
```

The fts of the non-compressed table takes about 12s while the fts of the compressed table takes only about 6s. As shown in Figure 10 the compressed test utilizes about 4 CPUs while the non-compressed test utilizes 3 CPUs. The total CPU used in the non-compressed test is about the same as in the compressed test (20 CPUU compared to 21 CPUU). As described in Section "Compression Implementation" accessing a compressed field adds some CPU overhead since for each compressed field an additional pointer needs to be traversed. This test does not show a significant increase in total CPU consumption because the CPU savings due to many fewer blocks processed compensates for the compression overhead. However, the peak CPU consumption of the compressed run is higher than the non-compressed run indicating a higher per block CPU consumption (see Figure 10).

As shown in Figure 11 the maximum IO read performance of the compressed run is significant lower than that of the non-compressed run (35MB/s vs. 90MB/s). However, with a total of 1.2GB read in the

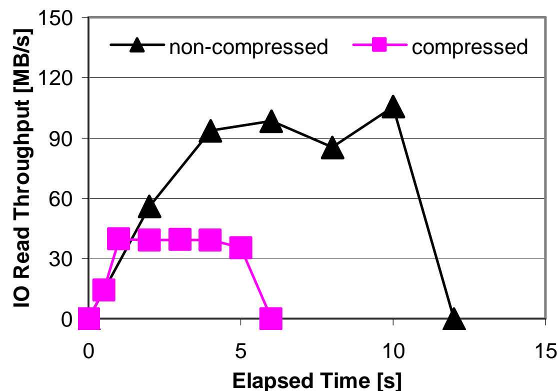

Figure 11:Parallel Full Table Scan IO Performance

non-compressed tests and 221MB in the compressed tests the ratio of data read in both tests corresponds to the compression factor of 5.4.

4.5 Table Access by Rowid

Table access by rowid is a very common operation performed as part of many data warehouse operations such as index range scans and star transformed queries. During a table access by rowid all rows are accessed by rowid. A rowid contains file, segment, block and block offset information of the row to be retrieved, which makes direct access to rows possible. Hence, before rows can be accessed their rowids need to be collected. This is usually done through indexes (conventional or bitmap).

The *Table Access by Rowid* tests consist of accessing a non-compressed and a compressed `SALES` table by rowid. Rowids are identified by performing a range-scan on an index on ss_quantity. Finally, two columns of the `SALES` table are accessed and aggregated. This query runs in serial:

```
SELECT MAX(SS_WHOLESALE_COST),
MAX(SS_QUANTITY*
    SS_TICKET_NUMBER)
FROM SALES
WHERE SS_QUANTITY between 1 and 3;
```

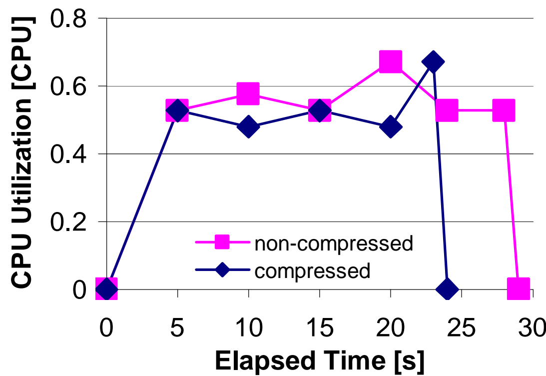

Figure 12: Table Access by ROWID

The query against the non-compressed table finishes in about 29 seconds, while the query against the compressed table finishes in 24 seconds, a savings of about 17%. Figure 12 indicates that the CPU utilization of both query runs is identical at about 0.7 CPUs. Considering that the compressed test finishes 5 seconds earlier, the total CPU utilization in the compressed case is abut 17% less than in the non-compressed case. These queries utilize the buffer cache for both index and data blocks. The buffer cache is about 5% of the compressed table and 1.4% of the non-compressed table. This tests shows that the compressed table utilizes the buffer cache much more effectively as the non-compressed table.

5. Compression with TPC-H

In the previous section we have systematically analyzed basic operations that are widely used in queries on data warehouses. However, usually queries used in data warehouses are more complex. They use multiple join methods, aggregations and sort operations. To demonstrate the outstanding performance characteristics of table compression in a large data warehouse environment Oracle has employed it in a scale factor 100 TPC-H benchmark publication [1]. The benchmark configuration used is a 4 node Compaq (DEC) AS ES45 68/1000 with 16 Alpha EV 68/1000 MHz CPUs with 8 MB cache running True64 Unix. Please note that the elapsed times for the compressed runs are taken from the published TPC-H benchmark, while the non-compressed times are obtained during the benchmark tuning phase and have not been published.

TPC-H has been widely accepted as the industry standard benchmark for data warehouse applications [3,4]. As of January 2003 there are 35 different results by 7 hardware and 4 database vendors [2]. Its schema consists of eight base tables modelling the data warehouse of a typical retail environment (see Figure 13). Tables such as `PART` (P), `SUPPLIER` (S), `PARTSUPP` (PS) and `CUSTOMER` (C) contain relatively static information about items typical retail companies buy from their supplier and sells to their customer, while nation and region are very small tables containing only a few rows. These tables amount to about 15% of the total database. The two largest tables, `LINEITEM` (L) and `ORDERS` (O) contribute to the remaining 85% of the total database size. They contain numerical measurements similar to a fact table in our star schema example, while the remaining tables contain detailed data, further describing the numerical measurements.

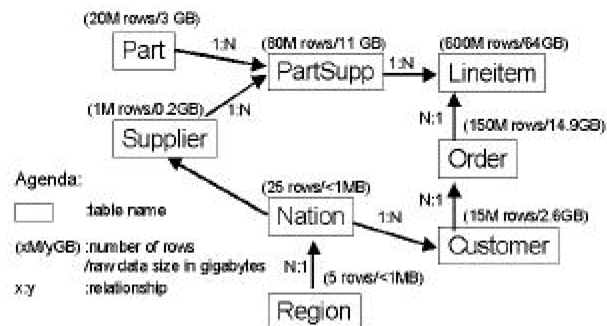

Figure 13: TPC-H and TPC-R Schemas

TPC-H's performance test is comprised of a set of 22 business queries and two update functions, designed to exercise system functionalities in a manner representative of complex business analysis applications. Queries are run in a power and throughput mode: the power tests measures the raw query execution power of the system when connected with a single active user; the throughput test measures the ability of the system to process the most

queries in the least amount of time. TPC's primary metric is a combination of the power and throughput tests. Discussing all 22 queries would be far beyond the scope of this paper. Therefore, we will limit our discussion to a representative subset (Query 1, 6 and 15) and the two update functions.

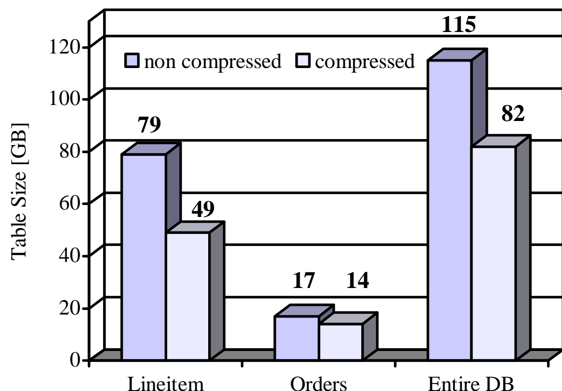

Figure 14: Compression Factor TPC-H Tables

5.1 Discussion of TPC-H Compression Factor

When loaded with the compression feature, compression for LINEITEM is the highest at a compression factor of about 1.6, while ORDER compresses at a compression factor of about 1.2. (see Figure 14). Therefore, the compressed versions of LINEITEM and ORDERS consume only about 60% to 80% of the uncompressed tables, LINEITEM shrinks from 79 GB to 49 GB while ORDERS shrinks from 17 GB to 14 GB. The overall compression factor for the TPC-H database is about 1.4 resulting in a space saving of about 29%.

The compression factors of TPC-H tables are significantly lower than those of the customer example described in Section 3.1. This has many reasons. Firstly, since TPC-H data is programmatically generated, the data distribution is arbitrary. Most columns have a uniform distribution. That is, over a given range there are no clusters with the same values. Analysis of multiple data from multiple customers shows that real world data is not uniformly distributed, but clustered. This is especially common for data warehouse fact and summary tables; in almost every Data Warehouse environment periodical data maintenance applies some kind of grouping or sorting to the new data. For instance, an ETL process, consolidating new fact table information from sources, has to compare and aggregate, thus sort, the various sources prior to the insertion into the fact table. Similar data clustering occurs naturally in summary tables that perform group by or advanced OLAP operations such as rollup and cube. As a matter of fact, TPC-H has been widely criticized for having mostly uniformly distributed data. Secondly, in order to increase the table size, TPC-H tables contain "fill columns" such as COMMENT. These fields are generated with a very long unique content making it almost impossible to compress.

5.2 Discussion TPC-H Query Performance

Figure 15 shows the elapsed times and speedups[7] achieved with compression of the three TPC-H queries 1, 6, and 15. This shows that Query 1 has a relatively small slowdown of about 2 percent, which might well be in the measurement error range, when run against the compressed table LINEITEM. On the other hand Query 6 and Query 15 show a significant elapsed time speedup. Query 6 shows a speedup of 35% while Query 15 shows a 38% speedup. The overall speedup[8] of all 22 TPC-H queries run against the compressed database and against the non-compressed database is about 10%. Elapsed time for the insert test (rf1) increased by about 3.9% while the elapsed time for the delete test (rf2) decreased by about 17%. The primary metric for TPC-H, QphH@100GB, improved by about 10%. For some of the queries we observed a slight CPU overhead. As demonstrated in the section about Space Savings, TPC-H data yield a compression factor of only 1.2 for ORDERS and 1.6 for LINEITEM. Queries against the compressed database perform on average 27% fewer disk accesses.

Elapsed Time of Query 1 increases by about 2%. This can be explained by the slight increase in CPU utilization and the fact that this query is CPU bound. The total CPU consumption in the compressed case increases by about 2%. Since there is no CPU left for query execution, elapsed time increases by about the same amount as the total CPU consumption increases. The large decrease in disk utilization of about 38% has very little impact on this query because the disk subsystem is not the bottleneck during this query.

Query 6 performance improves by about 35%. This query being IO bound leaves much of the available CPUs unused. Consequently, the increase of CPU utilization can be easily compensated for by the system without degrading performance.

Similar to Query 6, Query 15 benefits from table compression showing a 38% performance increase. This query performs multiple join operations. Similarly to query 6, this query leaves some of the available CPU unused decreasing query elapsed time by about 8%. This

---

7

$$Speedup = \frac{ElapsedTimeNonCompressed - ElapsedTimeCompressed}{ElapsedTimeNonCompressed}$$

8

$$OverallSpeedup = \sum_{i=1}^{i=22} \frac{ElapsedTimeNonCompressedQi - ElapsedTimeCompressedQi}{ElapsedTimeNonCompressedQi}$$

query benefits greatly from compression reading about 42% less data in case of the compressed database.

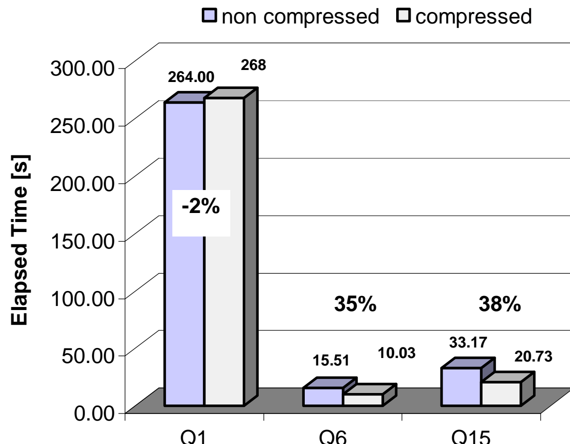

Figure 15: Load Performance using External Table

Using compressed tables in TPC-H reduces the elapsed time for most queries in this system setup, but increases the elapsed time for some. IO bound queries directly benefit from compression because it reduces the average consumption of the bottlenecking resource, namely the disk subsystem. CPU bound queries also benefit from fewer disk subsystem consumption. However, the overhead in accessing compressed tables can hurt elapsed time of these queries.

6. Conclusions

In this paper we presented an innovative compression technique, recently introduced in the Oracle RDBMS for reducing the size of relational tables. By using a compression algorithm specifically designed for relational data, Oracle is able to compress data much more effectively than standard compression techniques. In our example the compression factors vary, depending on the table content, between 2.9 and 4, yielding a space savings of 67% to 75%. However, internal analysis of real customer data during the development of Table Compression showed compression factors up to 12. The vast majority of compression factors of production systems are between 3 and 6.

Unlike other compression techniques, queries against compressed tables incur virtually no query performance penalty. Our full table scan experiments show that compression decreases elapsed time by 50% while increasing total CPU time slightly (5%). The buffer cache experiments indicate a 17% performance gain in case of compressed tables with no significant change in CPU utilization. Both experiments show how small the actual overhead of uncompressing blocks is and how large the performance gain of compression can be, both for direct and buffer cache reads.

There is very little impact of compression to the elapsed time of delete operations. However, a CPU overhead of 27% was observed. Update operations are slightly more expensive, showing an increase of 20% in elapsed and 50% in CPU time. Elapsed and CPU for load operations increases by about 2 times. Additionally, we demonstrated how data compression was successfully employed in TPC-H, the industry standard data warehouse benchmark, to increase Oracle's performance by 10%.

Since the target applications for data compression are mostly read-only systems, e.g. large data warehouses and OLAP systems, the penalty shift from query to load and DML operations is a design goal in our implementation. It originated directly from our customer's requirements to have maximum space savings with the least impact on query performance.

7. Acknowledgements

We would like to thank Ray Glasstone for his encouragement to write this paper and George Lumpkin, Cetin Ozbutun, Susy Fan, Hermann Baer and Alexander Tsukerman for their many useful suggestions.

References

[1] TPC-H 100 GB published 07/15/02 by HP/Oracle on Alpha Server ES45 and Oracle 9iR2 Executive Summary: http://www.tpc.org/results/individual_results/HP/es45_5578_es.pdf FDR http://www.tpc.org/results/FDR/tpch/es45_5578_fdr.pdf

[2] TPC web site: www.tpc.org

[3] Levine, C., Stephens Jr., J.M., DeWitt, D. (Chair), "Standard Benchmarks for Database Systems", ACM SIGMOD 1997 Industrial Session 5 (http://www.tpc.org/information/sessions/sigmod/indexc.htm)

[4] Poess, M. and Floyd, C., "New TPC Benchmarks for Decision Support and Web Commerce". ACM SIGMOD REORD, Vol 29, No 4 (Dec 2000)

[5] Oracle Data Warehouse Guide: otn.oracle.com

[6] Mark Morri, "Teradata Multi-Value Compression V2R5.0", Teradata Whitepaper, July 2002

[7] DB2 V3 Performance Topics, GG24-4284-00, http://www.frc.utn.edu.ar/campus/ibm/abstract/gg244284.htm

[8] Table Compression in Oracle 9i: A Performance Analysis, An Oracle Whitepaper http://otn.oracle.com/products/bi/pdf/o9ir2_compression_performance_twp.pdf