High speed compression algorithm for columnar data storage

György Balogh

Abstract—**Lossless compression algorithms with very high compression and decompression speed are widely used in data warehouses today. Even small improvement of these algorithms can have high impact on storage space but more importantly on data access speed which effects response time of data analysis systems. We present a generic column storage compression algorithm (DictComp) with very fast compression and decompression speed. At worst the performance of the algorithm falls back to the LZ4 [4] compression algorithm but on data dominated by few values (which is very frequent in unnormalized database tables) over 5 GB/sec/CPU core decompression speed can be achieved.**

Index Terms—**Big Data, Hadoop, Impala, log analysis, cloud computing, decompression, algorithm**

I. INTRODUCTION

Storage capacity and data access speed of storage devices evolve exponentially albeit with very different exponent: data access speed is getting exponentially slower relative to storage capacity. In 1991 a typical disk had 40 MB storage which could be scanned and processed in around a minute while today the typical capacity is 2 TB yet the full scan time is more than 4 hours! Yet in cloud based log analysis services huge amount of log data have to be stored and analysed. The presented method provides an efficient storage for log data achieving compression ratio of 10-30 and similar data access speedup which translates to query response time speedup.

Distributed storage and processing (e.g.: Hadoop) can mitigate this problem: more disks can work parallel so data access speed can be scaled up linearly with the number of disks. Another possibility to boost data access speed is high speed lossless data compression. CPU speed is also evolving much faster than data access speed so there is expanding opportunities for the implementation of more and more clever compression algorithms. A new generation of so called real-time compression algorithms has developed in the last couple of years. There is nothing new in these algorithms theoretically, but the very efficient implementations and ratio of CPU speed and data access speed makes the decompression time almost negligible today. Examples of such algorithms are LZO, Snappy and LZ4. LZ4 can achieve around 1 GB/sec/core decompression speed on log data with a compression ratio of 10 boosting the 100 MB/sec sequential disk I/O limit with an order of magnitude.

LZO, Snappy and LZ4 are generic compression algorithms, however in database storage engines other structural information such as field and record boundaries are also available, which can further boost the speed of compression and decompression.

In analytical databases data is typically stored in columnar way: values of one field for a larger set of records are stored in a compressed block. Columnar storage has many advantages for analytical workloads. Columns that are not participating in a query don't have to be read, thereby sparing significant disk I/O. A list of values for the same field typically can be compressed better. More and more data is stored in an unnormalized way with many repeating values. Log files are a typical example of this: the same value (e.g.: server address) is stored over and over again even if it is the same in each case. Unnormalized storage has the advantage of data locality: all information for a record is available locally; there is no need for indirections for potentially remote data.

In this publication we present a compression algorithm for columnar data. The algorithm falls back to an LZ4 compression in worst case, but can achieve decompression speed over 5 GB/sec/core for column data dominated by a few values.

The presented compression algorithm can be applied in analytical database storage engines. One of our goals is to integrate this result to the Parquet storage format to further speed up analytical queries over data stored in Parquet format. This would reduce query response time of queries over "big data" data sets. In practice this effect BI tool performance on all kind of data sets (financial, click stream, sensor data etc.).

II. THE ALGORITHM

In the case of columnar compression, the input for the compression algorithm is a sequence of string values and the output is a byte sequence. This compression schema can be applied to table columns with boolean, text and enum types. Numerical data types needs different class of algorithms.

In case of decompression, the input is the coded byte sequence and the output is the original string sequence. However, for many query operations the original string sequence does not have to be fully materialized. These operations can be performed on compressed or 'half compressed' data.

Compression is performed in blocks with the size of typically 1000-10000 items. Each block independently

Manuscript received June 06, revised June 6. György Balogh, CTO of Logdrill Kft, a subsidiary of Kürt Zrt. Contact: gyorgy.balogh@kurt.hu

compressed contains all information for decompression. The main idea of the compression algorithm is to split the data into three parts: literals, literal lengths and dictionary indexes as shown in Figure 1. Unique literals are concatenated without string terminator and compressed with LZ4. Length of the unique literals are collected separately and compressed with a very efficient integer compression algorithm described later. The same integer compression algorithm is used to compress the dictionary indexes.

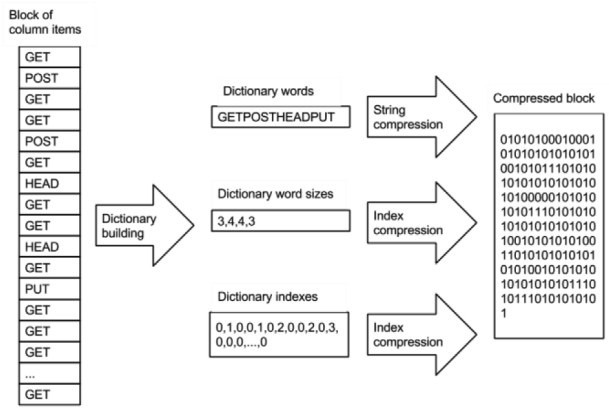

Fig. 1. The block compression process

Fig. 2. shows the compression algorithm. The dictionary index starts with 0 and increased for every new item. A fast hash function is used to recognize repeating items.

```
    virtual void addFromString(const char * str, size_t len)
    {
     Index h = murmurHash64A(str, len) % BUCKETS;
     Index ind = m_indexesByHash[h];
     if ( ind != m_unknownItemIndex && len ==
m_sizes[ind] &&
          strncmp(str, m_strings.data() + m_offsets[ind],
len) == 0)
      {
       // existing item
       m_indexes.push_back(ind);
      }
      else
      {
       // new item
       m_offsets.push_back(m_strings.size());
       m_strings.insert(m_strings.end(), str, str + len);
       m_sizes.push_back(len);
       m_indexes.push_back(0);
       m_indexesByHash[h] = ++m_nextId;
      }
    }
```

Fig. 2. Dictionary building algorithm.

III. INDEX COMPRESSION

Daniel Lemire and Leonid Boytsov have recently presented a comprehensive evaluation of different integer compression algorithms [8]. We evaluated the algorithms based on decompression speed including the disk I/O. The best candidate was the index compression algorithm (Simple8b) published by [7].

Key idea of the algorithm is to use 64 bit words to code input sequences. 4 bits are used to select the encoding schema the rest 60 bits holds the data. Example encoding schemes: 60 pieces of 1-bit numbers, 30 pieces of 2-bit numbers, 20 pieces of 3-bit numbers ... 1 pieces of 60 bit number. During decoding the 64 bit code hold a register and by repeatedly applying shift in conjunction with bitwise AND operations, this method can produce the output numbers at an extremely fast decoding speed.

```
    template<size_t LENGTH, uint8_t BITS, size_t K =
LENGTH>
    struct DecodeLiteral
    {
     static void decode(uint16_t *& out, uint64_t code)
     {
      *out++ = (code >> (60 - BITS * (LENGTH - K +
1))) & ((1U << BITS) - 1);
      DecodeLiteral<LENGTH, BITS, K -
1>::decode(out, code);
     }
    };
    template<size_t LENGTH, uint8_t BITS>
    struct DecodeLiteral<LENGTH, BITS, 1>
    {
     static void decode(uint16_t *& out, uint64_t code)
     {
      *out++ = (code >> (60 - BITS * LENGTH)) & ((1U
<< BITS) - 1);
     }
    };
```

Fig. 3. Fast integer decoding routines. To decode a number, shift and bitwise AND operations are needed. Compared to this the loop administration cost gets too high. With the help of template meta-programming, loop unrolling can be enforced leading to 100% speed up compared to a simple loop.

```
    template<size_t MAX_LENGTH, uint8_t BITS,
uint8_t SCHEMA_ID>
    static size_t encodeLiteral(const uint16_t * data, size_t
len, uint64_t * out)
    {
     const uint64_t max = (1U << BITS) - 1;
     const uint32_t n = std::min(len, MAX_LENGTH);
     *out = SCHEMA_ID;
     for (size_t i = 0; i < n; ++i)
     {
      if (data[i] > max) // termination condition
       return 0;
      *out = (*out << BITS) | data[i]; // coding
     }
     *out <<= 60 - BITS * n;
     return n;
    }
```

Fig. 4. Fast integer encoding routine. Doing the test and encoding in one loop is the key idea to speed up the coding.

Compared to the benchmark implementation of Daniel Lemire and Leonid Boytsov, we achieved significant speed up

both in compression and decompression speed. With loop unrolling (with template meta-programming) we achieved 100% decompression speedup. Decompression speed can get above 2 billion integers per second which translates to 1-2 CPU cycles per integer. In case of 32 bit numbers this translates to 8 GB/sec decompression speed.

In the baseline implementation, the test of coding schema and the actual coding are done in two steps. First it finds the best schema in a greedy manner then performs the coding in another pass. We modified the coding algorithm to do the schema test and coding in one step. The termination condition and the coding are independent and can be executed in parallel. Presumably as an effect of the superscalar execution, the added coding does not slow down the test which gives us an almost 100% coding speed up in compression speed. If a coding schema terminates due to a too large number, then the next coding will simply overwrite the invalid code generated by the previous schema.

The original index coding algorithm has only two schemas for encoding long 0 runs (run length with 120 and 240). We extended the algorithm with a new schema that can encode 6 runs in the 60 bit data part. 7 bits are used to encode the run length and 3 bits encode the data. This schema significantly increased the compression ratio (50-100%) in case of zero dominated distributions.

IV. Evaluation

First let's consider two special input distributions: all items are different and only a few items are different. In the first case dictionary index will always be zero (zero index means new item). In this case the all zero indexes will compress extremely well (with run length encoding) with negligible decompression time overhead so the compression will be an LZ4 compression basically. In the other case the dictionary indexes will be small and again compress very well. In this case the literal string will be short so the decompression time will be dominated by the index decompression speed. Depending on the distribution, the decompression will be a mixture of LZ4 decompression and index decompression with decompression speed ranging from around 1 GB/sec (LZ4 dominated) to over 5 GB/sec (Index decompression dominated).

We tested the compression algorithm on two realistic big data datasets:

- US domestic flight statistical database [6]. (10 million records, 29 fields, 1.6 GB in CSV).
- Web server logs of the 1998 World Cup [1]. (500 million records, 1 fields, 1.9 GB in CSV).

Measurements were performed on a $1000 class laptop with 8GB RAM, Intel Core I7 processor having 4 cores running at 2GHz. Operating system was Ubuntu 11.04. Measured sequential disk read speed is 70MB/sec.

We measured compression performance (bulk load) and some simple SQL queries. Query plans are hand coded over our column compression storage. We selected three columnar database engines for comparison: InfoBright [3], MonetDB [5] and Cloudera Impala [2]. We tested the InfoBright Community Edition (version 4.0.7), MonetDB v1.0 Jul2012-SP2 and Impala 1.1. All are 64-bit versions.

Table 1 shows the measured execution times.

	Flight29			*Web500M*		
	Bulk load	Cold query	Warm query	Bulk load	Cold query	Warm query
MonetDB	59.21	0.92	**0.04**	167.32	7.12	0.95
DictComp	**17.31**	**0.61**	0.05	**29.51**	**2.34**	**0.70**
InfoBright	78.13	1.67	0.92	130.34	70.96	70.94
Cloudera Impala	57.35	4.90	0.46	97.79	47.7	15.82

Table 1. Bulk load and query execution times (in seconds) of the engines on the two datasets. In case of bulk load the input file was always in the file cache. For queries we tested the query when the file and database caches were cleared (cold run) and after multiple runs of the same query (caches are warmed up).

In query performance DictComp and MonetDB are close. DictComp gets significantly better in I/O bound cold queries, for warm queries they almost exactly match. InfoBright simply fell short with the *Web500M* dataset, even a grep can perform better (6 seconds from file cache) on the original uncompressed dataset. The main reason why InfoBright cannot handle record numbers of this magnitude comes from its old architecture, which is inherited from the MySQL framework. The MySQL storage interface forces the InfoBright storage engine to copy every single item to the MySQL plan executor. So the whole dataset have to be decompressed and copied item by item. This makes the InfoBright query execution CPU bound on the *Web500M* dataset, which then results in the same execution time for cold and warm queries.

V. Acknowledgement

This publication/research has been supported by the European Union and Hungary and co-financed by the European Social Fund through the project TÁMOP-4.2.2.C-11/1/KONV-2012-0004 - National Research Center for Development and Market Introduction of Advanced Information and Communication Technologies.

VI. References

[1] 1998 world cup web site access logs. http://ita.ee.lbl.gov/html/contrib/WorldCup.html

[2] Cloudera Impala. http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html

[3] Infobright. http://www.infobright.com

[4] The LZ4 lossless compression algorithm. http://code.google.com/p/lz4

[5] Monetdb. http://www.monetdb.org

[6] USA domestic flights info data. http://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=On-Time

[7] Vo Ngoc Anh and Alistair Moffat. Index compression using 64-bit words. Softw., Pract. Exper., 40(2):131–147, 2010

[8] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. CoRR, abs/1209.2137, 2012.

György Balogh is the CTO of LogDrill Kft. György received his computer science degree from the University of Szeged and has 20 years of data mining and machine learning experience. György spent 6 years at Vanderbilt University in Tennessee as a researcher and developed the sensor fusion algorithms of the first distributed shooter localization system. Currently György Balogh is working on the LogDrill product family specialized for log and big data analytics.