How Achaeans Would Construct Columns in Troy

Alekh Jindal, Felix Martin Schuhknecht, Jens Dittrich,
Karen Khachatryan, Alexander Bunte

Information Systems Group, Saarland University
http://infosys.cs.uni-saarland.de

ABSTRACT

Column stores are becoming popular with data analytics in modern enterprises. However, traditionally, database vendors offer column stores as a different database product all together. As a result there is an all-or-none situation for column store features. To bridge the gap, a recent effort introduced column store functionality in SQL server (a row store) by making deep seated changes in the database system. However, this approach is expensive in terms of time and effort. In addition, it is limited to SQL server. In this paper, we present Trojan Columns, a novel technique for injecting column store functionality into a given closed source row-oriented commercial database system. Trojan Columns does not need access to the source code of the database system. Instead, it uses UDFs as a pluggable storage layer to write and read data. Furthermore, Trojan Columns is transparent to users, i.e. users do not need to change their schema and their queries remain almost unchanged. We demonstrate Trojan Columns on a row-oriented commercial database DBMS-X, a closed source top notch database system. We show experimental results from TPC-H benchmarks. Our results show that Trojan Columns can improve the performance of DBMS-X by a factor of up to 9 for TPC-H queries and up to factor 17 for micro-benchmarks — without changing the source code and with minimal user effort.

1. INTRODUCTION

1.1 Background

Row stores (e.g. Oracle and DB2) as well as column stores (e.g. Vertica, MonetDB), have, in recent times, emerged as two major technologies in the commercial database market. However, database vendors typically offer different database products for row and column stores respectively. This is a huge problem for customers since they have to make a strategic decision on which (one or more) database product to use. Alternatively, in recent times, several people have argued for the superiority of column stores over row stores [12, 7, 2, 1]. As a result, several enterprise customers might choose to migrate to a column store. However, this is an expensive process. It involves new licensing costs, additional training for the administrators and developers, and migration effort from the old to the new database product. For a customer, such change of faith from one database product to another is a once-in-while process. Rather, he would be interested in having both row and column technologies in the same existing system.

6th Biennial Conference on Innovative Data Systems Research (CIDR '13) January 6-9, 2013, Asilomar, California, USA.

1.2 Problem

Nicolas Bruno proposed C-Tables to mimic column stores in row-oriented databases [3]. The main idea of C-Tables is *"to extend the vertical partition approach to explicitly enable the RLE encoding of tuple values"*. For a given relation, we need to first sort its attributes and then apply RLE encoding over each attribute. [3] introduced the idea of emulating column stores in row stores and works well with simpler datasets and queries. However, [3] does not work well with more complex datasets and queries. To illustrate, Table 1 shows the query times of C-Tables[1] over four *unmodified* TPC-H queries in a row-oriented commercial database DBMS-X.

Query	Standard Row	C-Table
Q_1	8.20	211.39
Q_6	8.23	96.17
Q_{12}	9.80	5457.37
Q_{14}	8.61	335.55

Table 1: TPC-H query times (in seconds) for scale factor 1.

We see that for all four queries C-Tables perform worse than standard row layout in DBMS-X. Furthermore, C-Tables could be up to 500 times slower (for Q_{12}) than standard row layout. This is because C-Tables are loosely integrated within the database system and they incur redundant tuple reconstruction joins at query time. For example, C-Tables need to perform 6 tuple reconstruction joins for TPC-H query 1. Hence, C-Tables do not work well with TPC-H like datasets and queries.

Another recent approach integrated column store indexes into SQL Server [11]. In that approach, column store indexes store segments of columns as blobs in the standard row store table. This, however, requires deep changes in all the layers of the database system, including query processing and data storage enhancements. As a result, this is a considerable effort for the vendors, and meanwhile the users have to wait for the next product release. The above approach certainly helps SQL Server users, since they can still use the rich DBMS features as well as the sophisticated query optimizer. But what about the users of other database products, e.g. IBM DB2, Oracle? It is not clear whether they can or are

[1]We tried to implement C-Tables as closely to the description in the paper as possible except that we do *not* create any pre-joined materialized views, as suggested by the C-Table authors, since we want to see the overall query costs, including possibly costly joins.

willing to emulate column stores as well. Overall, is it possible to have a generic approach which works across all systems? Users of these database products will certainly ask this question.

In summary, the research problem we explore in this paper is as follows. For any proprietary closed source row-oriented database product, is it possible to introduce column store technology in it to support analytical workloads efficiently? All this *without* having access to the source code of the database product.

1.3 Our Idea

Our idea is to use User Defined Functions (UDFs) as an access layer for data storage and retrieval. To do so, we create and install certain UDFs within the database system and exploit them whenever we need to store or access data. The data is actually stored in a compressed column-oriented fashion on disk. But the UDFs translate it into the row layout for the query processor. This means we trick the database into believing that the data is still stored in row layout, even though it is not. Finally, note that our approach is very different from the two extremes of data stores: either having a different product for different stores or doing deep seated changes in the database product. We do neither of these, but still gain performance significantly. The major benefits of our approach are as follows:

(1.) Injects column store functionality into existing closed source database products e.g. IBM DB2, Oracle.

(2.) Logical view of data remains unchanged for the outside user; instead, the changes are transparently injected inside the DBMS.

(3.) Does not invade or make heavy changes in the system; rather, uses lightweight UDFs to store and access the data.

(4.) Just fixes the storage layer (row, column, or even column-grouped layouts) by inserting appropriate UDFs.

(5.) Reuses the query optimizer (at least partially) and therefore no need to re-implement state-of-the-art database technologies.

(6.) User queries remain (almost) unchanged.

The novel use of UDFs as an access layer raises several interesting questions. We answer some of them below.

Why UDF? UDFs have been there since long, not just in data managing systems, but also in operating systems, middleware, and programming runtime environments. However, in this paper, we exploit the UDFs in a novel way to enable column store functionality in an existing row store, without making heavy untenable changes into the system. Although the database UDFs were primarily designed to extend the application specific functionality of the database. However, in this paper, we also consider UDFs as tools which can be exploited by the vendors (or middlewares) to integrate core database features into the system. With this, the business model of database distributors is not just limited to shipping new releases but can also include supplying core functionality patches, as UDFs, in order to: (i) meet the customer requirements quickly, and (ii) get quick customer feedback on the functionality before actually coming out with a new product release.

What about UDF portability? UDF interfaces may differ slightly from one database product to another. However, the core concepts remain the same. This is similar to different implementations of SQL, having the same core concepts. In our work, we consider three major commercial database systems and one major open source database system. We found the UDF interfaces of all these systems to be very similar. Thus, we believe that we can abstract the majority of the functionality into a common code base for these systems.

What about query optimization? One might think that the use of UDFs for core database features rules out automatic query optimization. However, this is not true. Several researchers have proposed techniques to inspect the UDF code and use it for automatic query optimization. For example, Manimal [4] and Hadoop-ToSQL [9] analyse UDFs in MapReduce for automatic query optimization. [5] considers query optimization, query rewriting and view maintenance for queries with user defined aggregate functions, i.e. optimizing UDFs similar to built-in aggregate functions (min, max, count, sum, avg). More recently, researchers have proposed to build a query optimizer which performs a fully automatic static code analysis pass over the UDFs (or *black boxes* as they call it), and enable several query optimizations, including selection and join reordering, as well as limited forms of aggregation push-down [8]. Thus, we believe our framework could be extended to leverage these prior works for automatic query optimization.

1.4 Contributions

Our main contributions are as follows:

(1.) We present *Trojan Columns*: a radically different technique to inject column store functionality into a given database system. Trojan Columns masks the database storage layer and translates back and forth from users' row-view to the physical column-view of data. All the while, users' view remains unchanged. (Section 2)

(2.) We present techniques to query Trojan Columns. We show how to push down one or more operators in the query tree to the UDFs. We describe how to rewrite the user queries in order to use the UDFs for data access. (Section 3)

(3.) We present experimental results from DBMS-X over TPC-H datasets. We evaluate Trojan Columns over three different benchmarks including (1) unmodified TPC-H queries, (2) simplified TPC-H queries as proposed in the C-Store paper [12], as well as (3) micro-benchmarks to investigate the pros and cons of Trojan Columns by varying several query parameters. (Section 4)

2. TROJAN COLUMNS

In this section, we present Trojan Columns: a novel way of *injecting* column store functionality in a given row-oriented database system. Trojan Columns uses table returning UDFs[2] to translate the user's logical row view to the physical column view on disk. The core philosophy of Trojan Columns is very similar to Trojan techniques in Hadoop++ [6, 10]: effect the changes from inside without changing the source code of the system. In the following, we describe how to create Trojan Columns.

Trojan Columns maps logical relations to physical tables as follows. First, we horizontally partition a given relation T into segments. Then, we store each attribute in a given segment as a separate BLOB (binary large object) in a physical table T_trojan(segment_ID, attribute_ID, blob_data). To illustrate, consider the following entries of a Customer relation.

Assuming a segment size of 4, i.e. 4 entries of each column are mapped to a different row, Trojan Columns converts this to the following Customer_trojan table:

We store each entry in the blob_data column of the above table as a BLOB, thus mimicking a column-oriented storage. Experimentally, we found bigger segment sizes, e.g. $\sim$10M, to be more suitable. The data storage idea for Trojan Columns is inspired

[2]Typically database systems support three kinds of UDFs based on their return types: (i) scalar value returning UDFs, (ii) row returning UDFs, and (iii) table (of rows) returning UDFs.

Customer		
name	phone	market_segment
smith	2134	automobile
john	3425	household
kim	6756	furniture
joe	9878	building
mark	4312	building
steve	2435	automobile
jim	5766	household
ian	8789	household

Customer_trojan		
segment_ID	attribute_ID	blob_data
1	name	smith, john, kim, joe
1	phone	2134, 3425, 6756, 9878
1	market_segment	automobile, household, furniture, building
2	name	mark, steve, jim, ian
2	phone	4312, 2435, 5766, 8789
2	market_segment	building, automobile, household, household

by SQL Server Column Indexes [11]. However, in practice, Trojan Columns is radically different from Column Indexes in several ways.

First, Trojan Columns uses UDFs to store the blob data instead of native SQL support in case of Column Indexes. This not only makes Trojan Columns plug-and-play, but also allows us to customize the blob storage to user applications. For example, we might prefer light weight compression (e.g. RLE) for read-intensive application and higher compression ratio (e.g. Huffman) for archive applications. Additionally, we could simply let the database system apply the default compression method for blobs, e.g. TOAST compression in PostgreSQL. Likewise, we might choose to sort the data within or even across the blobs in any way, e.g. sort on attribute IDs. Essentially, we have full control and flexibility to decide how the blobs must reside on disk.

Second, Trojan Columns is quite different from SQL Server Column Indexes not only in data storage, but also in data access. Similar to data storage, the access method is plugged into a given database, instead of making deep changes in the data access layer itself. Again, based on user applications, we have full control to decide how to access the data. For example, for an airline company, we may choose to look for all possible connections only for premium customers.

Third, Trojan Columns uses standard database tables to store the blob, instead of a new index type in case of Column Indexes. The database uses its own physical storage mechanism to persist the blob table on disk. In other words, Trojan Columns simply provides a mechanism to decouple logical representation of relations from their physical implementation, a.k.a. *physical data independence*, which, unfortunately, still remains a myth in several modern databases [14]. Finally, with Trojan Columns the database system is entirely agnostic of the column store functionality injected within, i.e. no new SQL keywords, or data types, or any entries in the system catalog have to be added. The consequence is that Trojan Columns is *database product independent*, i.e. they can be *plugged* into any existing database product.

3. QUERYING TROJAN COLUMNS

In the previous section, we described how to create Trojan Columns. In this section, we describe how we process queries using Trojan Columns. Since Trojan Columns internally store data in column-oriented fashion, we need to translate the data back to row layout before passing it to the query processor, i.e. use a UDF to scan the table. Additionally, we may also push down other operators to the UDF in order to boost performance. Below, we first describe operator pushdown as a technique to process Trojan

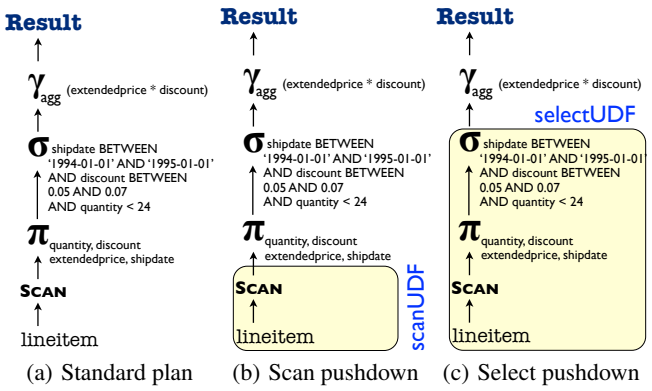


Figure 1: Standard and UDF query plans for TPC-H Query 6.

Columns, and then we describe how to rewrite user queries.

3.1 Operator Pushdown

The core idea of querying Trojan Columns is to push a part of the query tree down to the UDF. This means that a part of the query is processed by the UDF while the remaining query is still processed by the standard database query executor. Let's consider query 6 from the TPC-H benchmark [13] as a running example below. Figure 1(a) shows the logical query plan for query 6. Below, let's see how we can push down one or more operators in query 6 to a UDF.

Scan Pushdown. First of all, we need to push down the scan operator to the UDF. This is because we need to interpret Trojan Columns correctly (and differently) at the leaf level. Suppose that `lineitem` table in query 6 is stored as Trojan Columns. Figure 1(b) shows the query plan with the UDF. As shown in the figure, the UDF now figures out which physical table to read (the blob and not the row representation) for `lineitem` table. Also, the UDF is responsible for interpreting the physical table, reconstructing the logical `lineitem` tuples, and passing them on to the upper part of the query tree.

Projection Pushdown. Along with the scan, we can also push down the projection operator to the UDF, i.e. pass the projected attributes as parameters to the UDF. The UDF now returns only the projected attributes. Since the UDF return type is still the complete row, all other attribute values are set to NULL. A consequence of pushing projection down to the UDF is that the UDF now needs to fetch the blobs of only the projected attributes. This saves considerable I/O cost and improves query performance.

Selection Pushdown. To push the selection down, we simply pass the selection predicate to the UDF, as shown in Figure 1(c). The UDF is now responsible for evaluating the select predicate on each of the incoming tuple. To do so, the UDF now only fetches the selection attributes first. Then, before returning the tuple, the UDF evaluates the selection predicate. If the predicates hold true then the UDF fetches the projection attribute blobs, if needed, and returns a tuple of the projected attributes. If the selection predicates do not hold true, then the UDF inspects the next selection attribute values. This continues until either a qualifying tuple if found or end of data is reached. Pushing down the selection to the UDF has two advantages: (1) the number of UDF output tuples, and consequently the number of UDF calls are reduced, and (2) we can perform late materialization by fetching projection attributes only for segments having at least one tuple qualifying the selection predicates. The first advantage saves the overhead in each UDF

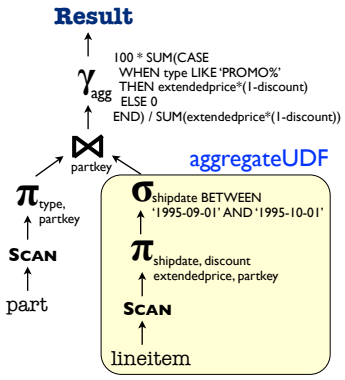


Figure 2: Example UDF query plan for TPC-H query 14.

call, while the second advantage saves I/O for projection attributes.

Aggregation Pushdown. We can even push down the aggregates (and group by) to the UDF. The UDF must now do the grouping and aggregation *before* outputting any of the tuples. This means that the UDF must precompute the results when initializing and then simply return the aggregated result subsequently. The major benefit of pushing aggregation down the UDF is to dramatically reduce the number of UDF calls.

Dealing with Join Queries. So far we have considered single table queries, i.e. no join conditions. Now let us see how joins are processed in the presence of Trojan Columns. For queries having join conditions, we simply push down the scan, selection, and projection operators to the UDF and let the database do the join. This works well because the output of UDF can be processed by the database query executor. Figure 2 shows the UDF query plan for TPC-H query 14. From the figure we see that the `lineitem` leaf is pushed inside the UDF, while the join is still performed outside. Also note that the query plan in Figure 2 accesses `part` table using the standard database access method. This is because `part` is a much smaller table and it does not pay off to use a UDF for it. Thus, we see that UDFs can be seamlessly integrated into the query pipeline. This holds true even for nested queries, e.g. TPC-H query 8. Alternatively, instead of letting the database executor process the join, one could think of even pushing down the join to the UDF. The UDF would then have to access two physical tables and join them based on the join condition. The advantage would be that we could have even lesser output tuples (depending on join selectivity). However, the problem is that we will need to recode the physical join operators as well as the optimizer logic to pick the physical join operator. Thus, we see the pros and cons of pushing too many operators down the UDF. Exploring these in more detail will be part of a future work.

Where does operator pushdown lead to? In the extreme case, we can push down the entire SQL query, i.e. all query operators, down to the UDF. However, this means that the UDF is now responsible for deciding how to execute a given query. In other words, the UDF must take care of query optimization as well as execution, making it a micro-kernel for processing SQL queries. The consequence is that the user must now recode all physical operators, cost models, as well as the optimization logic. Obviously, this is very hard to do. Therefore, it is important to strike the right balance when pushing down to the UDF. While too much is nasty, too little kills performance. A general practice could be to push only the leaves to the UDF and let the database handle the joins, unless they could be rewritten to selections.

3.2 Query Rewriting

Typically, there are two extremes of query rewriting with column layout: (1) complete query rewriting, due a complete change in schema, e.g. C-Table [3] and standard vertical partitioning, and (2) no query rewriting, if the column stores are natively implemented, i.e. no schema change at all. Our approach finds the middle ground. At bare minimum, we only rewrite the data access paths in the query, while keeping the rest of the query unchanged. This means that we can simply specify a data access UDF, for accessing Trojan Columns, in the `FROM` clause of the SQL statement. Note that we can use Trojan Columns for any subquery, thereby having layouts (row or column) on a per-table basis. In the following, let us see how to rewrite the SQL statements when using Trojan Columns. Consider TPC-H query 6 from Figure 1(a):

```
SELECT
  SUM(l_extendedprice*l_discount) AS revenue
FROM
  lineitem
WHERE
  l_shipdate >= '1994-01-01' AND l_shipdate < '1995-01-01'
  AND l_discount BETWEEN 0.05 AND 0.07
  AND l_quantity < 24;
```

Assume we have a scan UDF `scanUDF`(*table_name*) to read the blob data and convert them into logical tuples of table *table_name*. The query with scan pushdown is as shown in Figure 1(b):

```
SELECT
  SUM(l_extendedprice*l_discount) AS revenue
FROM
   scanUDF('lineitem')
WHERE
  l_shipdate >= '1994-01-01' AND l_shipdate < '1995-01-01'
  AND l_discount BETWEEN 0.05 AND 0.07
  AND l_quantity < 24;
```

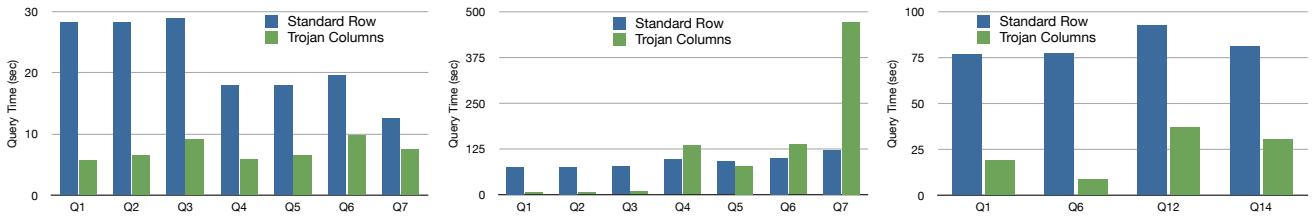
If we further push down the selection and projections, using UDF `selectUDF`, query 6 is now as shown in Figure 1(c):

```
SELECT
  SUM(l_extendedprice*l_discount) AS revenue
FROM
   selectUDF(
      'lineitem',
      'quantity,discount,extended price,price',
      'l_discount in (0.05,0.07)
         AND  l_shipdate in (1994-01-01,1995-01-01)
         AND l_quantity < 24'
      )
```

In the extreme case, if we push everything inside the UDF, the query will simply be: `SELECT * FROM everythingInUDF(....);` In future, we hope to leverage the view mechanism in standard databases to automatically rewrite the incoming user queries with the UDF access paths. Further study on this will be a part of future work.

3.3 Handling Inserts and Updates

Trojan Columns is primarily a read-only approach. This is a reasonable assumption since by design the column store functionality is suited for analytical workloads, which are typically read-only. Therefore, similar to SQL Server Column Indexes [11], Trojan Columns does not support direct inserts and updates to the underlying column-oriented representation. Still, Trojan Columns handles inserts as follows: (1) maintain a temporal row table to store the newly inserted records, (2) create an insert trigger to keep a count of the number of rows inserted in the temporal row store, and (3) use write-UDF to create and insert blobs into the physical table,



(a) Simplified queries, Simplified dataset (b) Simplified queries, Unmodified dataset (c) Unmodified queries, Unmodified dataset

Figure 3: Comparing TPC-H Query Runtimes of Trojan Columns with Standard Row in DBMS-X.

once sufficient number of rows which can fill a segment have been inserted into the standard row table. At query time, the read-UDF must also read the temporal row table for newly inserted rows.

Note that the above strategy of a trigger and function call for every insert might be too expensive for insert intensive applications. Alternate strategies could be to periodically bulk load Trojan Columns from the base table or to partition the base table into multiple tables and create Trojan Columns for each partition independently. To handle updates, the update-UDF first needs to determine the segment in which the update must be applied. Thereafter, the update-UDF must read the affected blobs in that segment and write them back.

4. EXPERIMENTS

We implemented Trojan Columns in DBMS-X, a closed source commercial database system. We ran experiments to see the performance improvements due to Trojan Columns in DBMS-X. We ran all experiments on a single node with 3.3 GHz Dual Core i3 running 64-bit platform Linux openSuse 12.1 OS, 4x4 GB main memory, 2 TB 5,400 rpm SATA hard disk. We use cold file system caches for all our experiments and restart the database, in order to clear database buffers, before running each query. We repeat each measurement 3 times and report the average.

In the following, we proceed as follows. First, we evaluate Trojan Columns on TPC-H queries and see the impact. Then, we study the pros and cons of Trojan Columns using single table micro-benchmarks. Finally, we see how far are Trojan Columns from materialized views as well as from a column store database system.

4.1 Trojan Columns on TPC-H queries

4.1.1 Simplified queries, Simplified dataset

In this experiment, we use the *simplified* TPC-H queries as proposed in the C-Store paper [12], and also used by other researchers [3]. In addition, we apply the same dataset settings to Trojan Columns as applied to C-Store in the simplified benchmark, i.e. we (1) simplify the schema of the tables, (2) exploit prematerialized joins (D-tables), and (3) presort the tables to allow for efficient sort-based grouping and compression. Figure 3(a) shows the results for scale factor 10. We can see that Trojan Columns improve over standard row for all queries, with improvements of almost 5 times for Q_1 and 4.4 times for Q_2. Even in the worst case Trojan Columns improve query Q_7 by 70%. All this in the same closed source commercial database DBMS-X.

4.1.2 Simplified queries, Unmodified dataset

Let us now see how the Trojan Columns behave if we apply them over the above 7 simplified TPC-H queries without making any dataset/schema changes, i.e. we neither use pre-materialized joins nor simplify the table schemas or pre-sort the data. Figure 3(b) shows the results for scale factor 10. From the table we can see that the improvement factor of Trojan Columns over standard row goes up to 13.3, 11.84, and 8.47 for Q_1, Q_2, and Q_3 respectively. This means that Trojan Columns work even better for these queries on unmodified datasets. However, on the other hand, for queries Q_4, Q_6, and Q_7, Trojan Columns performs worse than standard row. Thus, indeed the results change if we do not modify the dataset. With unmodified datasets, we see that Trojan Columns do not work very well for low selectivity queries (Q_4, Q_6, Q_7). Before investigating this further in Section 4.2, let us first see the query performances with unmodified dataset and *unmodified* TPC-H queries below.

4.1.3 Unmodified queries, Unmodified dataset

Let us now take four non-nested, high selectivity, (and *unmodified*) TPC-H benchmark queries: Q_1, Q_6, Q_{12}, and Q_{14}. Figure 3(c) shows the query times for standard row and Trojan Columns for these four queries on scale factor 10. We can see that Trojan Columns outperforms standard row over all these queries. The maximum improvement is by factor 9 for Q_6, followed by factor 4 for Q_1, factor 2.6 for Q_{14}, and factor 2.5 for Q_{12}. All this in the same system (DBMS-X) and without touching the source code.

Next, let us see the query times for non-nested and low selectivity (unmodified) TPC-H queries — Q_3, Q_5, Q_{10}, and Q_{19}. Tables 2 show the results.

Query	Standard Row	Trojan Columns
Q_3	111.88	809.38
Q_5	99.73	169.34
Q_{10}	110.94	119.46
Q_{19}	79.14	43.12

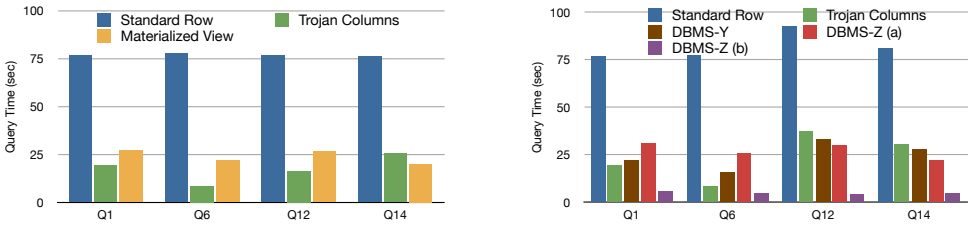
Table 2: TPC-H query Set 2 runtimes (in seconds).

We can see that, apart from Q_{19}, Trojan Columns does not perform very well with low selectivity queries, similar as in the previous section. We also tried nested (and unmodified) TPC-H queries. However, query nesting reduces the benefits of using Trojan Columns. This is because Trojan Columns only improves the I/O costs, which is just a fraction of the overall query costs. Apart from I/O, the remaining query processing costs are still the same as those for standard row.

4.2 Trojan Columns on Micro-benchmarks

In this section, we evaluate Trojan Columns on a micro-benchmark. The idea is to understand the pros and cons of Trojan Columns using simpler single table queries. Our micro-benchmark consists of queries of the following form over the `lineitem` table.

```
SELECT attr_1,attr_2,..,attr_r FROM lineitem
  WHERE l_partkey >= lowKey AND l_partkey < highKey;
```



(a) Trojan Columns versus Materialized Views

(b) Trojan Columns versus Column Store

Figure 4: Comparing TPC-H Query Runtimes with Materialized Views and Column Stores.

We vary the selectivity of the above query (by adjusting lowKey and highKey) as well as the number of projected attributes. Figure 5 shows the improvement factor of Trojan Columns over standard row when varying the number of referenced attributes from 1 to 16, and selectivity from 10^{-6} to 1. From the figure, we see that Trojan Columns has a maximum improvement factor of over 17 (lower left region). Also, we see that for low selectivities (≥ 0.1) Trojan Columns performs worse than standard row. To investigate this, we break down the query runtime into data access, data processing (decompression, operator evaluation etc.), and data output costs. Our results showed that data output costs dominate (as high as $60 - 80\%$) the query runtime for low selectivity queries. This is because each call to the UDF interface has some overhead: the lower the selectivity, the more function calls, the higher the overhead. These function call overheads overshadow the performance improvements of Trojan Columns for low selectivities. In principal, this overhead could be removed if the database storage interface were available in LLVM bitcode. Then the UDF query could at runtime be dynamically recompiled *together with the DBMS storage layer* to remove that boundary and bake the UDF into the kernel. This remains an interesting avenue for future work.

However, overall even for medium sized selectivites the performance gains of Trojan Columns are tremendous.

# referenced attributes (r)	1E-06	1E-05	1E-04	1E-03	1E-02	1E-01	1E+00
16	2.13	2.13	2.11	2.06	1.55	0.47	0.06
15	4.64	4.62	4.55	4.27	2.57	0.55	0.06
13	5.00	5.00	4.94	4.61	2.70	0.57	0.06
11	5.79	5.82	5.75	5.24	2.87	0.56	0.06
9	6.39	6.38	6.25	5.79	3.11	0.54	0.06
7	7.00	6.96	6.80	6.23	3.17	0.56	0.06
5	10.96	10.94	10.55	9.27	3.75	0.57	0.06
3	12.86	13.57	13.22	11.03	4.16	0.56	0.06
1	17.43	17.61	16.61	13.57	4.39	0.57	0.06

selectivity (fraction of tuples accessed)

Figure 5: Trojan Columns improvement factor in DBMS X.

4.3 How far are Trojan Columns?

4.3.1 *Comparison with Materialized Views*

The focus of Trojan Columns in this paper is to improve query I/O cost. However, as mentioned before, I/O is just a fraction of the total query costs. Since the database system is unaware of the column store inside, the query processing costs remain the same *outside* the UDF. To better understand the impact of Trojan Columns, let us now see the query times *inside* the subquery. To do so, we measure just the time to compute the subquery computed by the read-UDF using (1) Standard Row, (2) Trojan Columns, and (3) a Materialized View perfectly matching the query expression.

Figure 4(a) shows the results. We can see that Trojan Columns is significantly better (factor 5 on average) than standard row. Furthermore, we also see that except for Q_{14} Trojan Columns actually outperforms Materialized Views by a factor of up to 2.5. This is because Trojan Columns benefits from efficient column-oriented compression. Query Q_{14} has the lowest selectivity (1.25%) among these four queries, and therefore Trojan Columns does not perform as well as Materialized Views. This is a very good result considering that Materialized Views require ∼12GB of storage in this experiment, whereas Trojan Columns only requires ∼5GB. Still, the performance of Trojan Columns is very close to Materialized Views for Q_{14}. We conclude that Trojan Columns provides considerable improvements in terms of I/O costs. Furthermore, we see Trojan Columns as a method that improves over Materialised Views, i.e. a better way of storing and accessing query subexpressions.

4.3.2 *Comparison with Column Stores*

Trojan Columns allows users to use their existing row-oriented database system for efficiently supporting analytical workloads as well, i.e. bridge the huge gap between row stores and column stores. Thus, it would be interesting to see how far are Trojan Columns from a database system with column store technology as well as from a full blown column store. To do so, we run unmodified TPC-H queries on Trojan Columns as well as on two other systems: (i) a top notch commercial row-oriented database system DBMS-Y, with vendor support for column store technology, and (ii) a top notch commercial column-oriented database system DBMS-Z.

Figure 4(b) shows the results. We can see that while Trojan Columns are slower than DBMS-Y for Q_{12} and Q_{14} (by around 10%), Trojan Columns are in fact faster than DBMS-Y for Q_1 and Q_6 (by 10% and 80% respectively). This is even though DBMS-Y is deeply modified in order to support column functionality, whereas Trojan Columns does not even have access to the source code. The better performance of Trojan Columns for Q_1 and Q_6 is because Trojan Columns push down even the aggregation operator to the data access layer. DBMS-Z (a) in Figure 4(b) denotes DBMS-Z with the same (default) table schemas as Trojan Columns. From the figure, we see that Trojan Columns are quite competitive to a full blown column-oriented database system and can achieve comparable query performance in the same row-oriented database system. On the other hand, Trojan Columns are still far off from DBMS-Z, if the table schemas in DBMS-Z are optimized to achieve the best possible compression ratios — denoted as DBMS-Z (b) in Figure 4(b). Trojan Columns are up to 3.2 times slower than DBMS-Z (b) for single table queries and as high as 8.7 times slower for multi-table queries.

Overall, we see that Trojan Columns bridge the huge gap between the performances of row store and column store. In fact,

Trojan Columns match or even outperform the performance of a database system with column functionality, even without touching the source code of underlying database system. However, Trojan Columns still have some ground to cover before matching the performance of a full blown column-oriented database system.

5. DISCUSSION

Trojan Column Benefits. From the above experiments, we see that Trojan Columns significantly improves the performance of DBMS-X. This is because Trojan Columns can successfully emulate a column store without any overhead that typically exists in full vertical partitioning or other schema level approaches. The main advantage of Trojan Columns comes from improved I/O performance: we access only the referenced attributes. In addition, we apply lightweight column-oriented compression schemes. Furthermore, we push one or more SQL operators down to the UDF and evaluate them directly on BLOB data.

In contrast, C-Tables work only for up to 3 attributes, unless the input datasets are pre-joined. For a higher number of referenced attributes, the tuple reconstruction joins kill the C-Table performance. This is not the case for Trojan Columns. In fact, as we saw in the experiments, Trojan Columns is not at all affected by the number of referenced attributes.

Trojan Column Limitations. We found that the major performance problem in using table UDFs, for accessing Trojan Columns, is the additional overhead of UDF-function calls. These function calls lead to a significant decrease in performance if many rows are returned. For each row that is returned to the outside, DBMS-X invokes one call of the UDF and passes a large number of arguments to it. For a table like `lineitem` with 16 attributes, this means passing already 32 return variables (16 return variables + 16 indicators) to the function in each call, additional to the arguments passed by the user. Thus low selectivity queries are a problem for Trojan Columns.

The UDFs which perform only projection and selection are very flexible and need only the table schema to be generated. For example, we have one UDF for `lineitem`, and it can perform all kinds of projections and selections on the table. This is possible since the result schema of the query is always a subset of the `lineitem` schema. However, for queries which also perform grouping and aggregation, we need to generate the UDFs for each individual query. This is because the schema of the query result might not be a subset of the original table schema. Though manual, these adaptations can still be done easily and quickly.

As future work, a main goal is to increase the flexibility of the approach. We are planning to build generators and compilers (also in the form of UDFs), which create and install all necessary functions for a given query or table and database product. This is possible since only small adaptations are needed to tweak a function towards a query. Another main problem we face at the moment is the additional overhead at the leaf level caused by too many result rows. We could eliminate this problem in many cases, if we could push the join operator into the UDF. This would also allow for performing grouping/aggregation after the join and would lead to a significant decrease of function call overhead.

Query Optimization Considerations

(a) Selectivity. At the moment, we decide whether or not to use Trojan Columns manually. Ideally, however, we would like to hide this decision using a view, which is then used in the query invoked by the user. The view should be able to switch between Trojan Columns and the standard row store, depending on the query selectivity. Note that the view needs to pass the projection and selection operators down to the UDF, in case it chooses Trojan Columns.

(b) UDF cost estimates. DBMS-X supports a mechanism to adjust the estimated cost of a UDF in terms of the expected cardinality, i.e. it is possible to specify the number of rows that the UDF might return. Unfortunately, this cardinality is static and has to be set for each individual query. This cardinality information is then used in the access plan calculation to find the best plan.

(c) Intermediate results. If a query materializes intermediate results on disk, then the optimizer could consider using Trojan Columns for them, thus improving performance higher up in the query tree.

6. CONCLUSION

In this paper, we presented Trojan Columns, a radically different approach for supporting analytical workloads efficiently in a closed source commercial row-oriented database system. Trojan Columns does not make any changes to the source code of the database system, but rather use UDFs as a pluggable storage layer for data read and write. Trojan Columns can be easily integrated into an existing database system environment (without even restarting the DBMS). As a result, Trojan Columns is transparent to the user, i.e. the user continues using his existing database product with minimal changes to his queries. We implemented Trojan Columns in DBMS-X and show query runtimes from unmodified TPC-H benchmark, simplified TPC-H queries (as proposed by other researchers), as well as from single table micro-benchmarks. Our results show that Trojan Columns improves the performance of DBMS-X by up to a factor 9 on the unmodified TPC-H benchmark, by up to a factor 13 on simplified TPC-H queries, and by up to a factor 17 on single table micro-benchmarks. All this without touching the source code of the database system.

Acknowledgments. Work partially supported by BMBF.

7. REFERENCES

[1] D. J. Abadi et al. Column-Stores vs. Row-Stores: How Different Are They Really? In *SIGMOD*, pages 967–980, 2008.

[2] D. J. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *SIGMOD*, pages 671–682, 2006.

[3] N. Bruno. Teaching an Old Elephant New Tricks. In *CIDR*, 2009.

[4] M. J. Cafarella and C. Ré. Manimal: Relational Optimization for Data-Intensive Programs. In *WebDB*, 2010.

[5] S. Cohen. User-Defined Aggregate Functions: Bridging Theory and Practice. In *SIGMOD*, pages 49–60, 2006.

[6] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *PVLDB*, 3(1), 2010.

[7] S. Harizopoulos et al. Performance Tradeoffs in Read-Optimized Databases. In *VLDB*, pages 487–498, 2006.

[8] F. Hueske, M. Peters, M. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. Opening the Black Boxes in Data Flow Optimization. *PVLDB*, 5(11), 2012.

[9] M.-Y. Iu and W. Zwaenepoel. HadoopToSQL: A MapReduce Query Optimizer. In *EuroSys*, 2010.

[10] A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich. Trojan Data Layouts: Right Shoes for a Running Elephant. In *SOCC*, 2011.

[11] P.-Å. Larson, C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, A. Surna, and Q. Zhou. SQL Server Column Store Indexes. In *SIGMOD*, pages 1177–1184, 2011.

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

[13] TPC-H, `http://www.tpc.org/tpch/`.

[14] O. G. Tsalos, M. H. Solomon, and Y. E. Ioannidis. The GMAP: A Versatile Tool for Physical Data Independence. In *VLDB*, 1994.

APPENDIX

A. IMPLEMENTATION DETAILS

In the paper, we presented the general idea of Trojan Columns. In this section, we discuss the implementation details for Trojan Columns in DBMS-X. We considered two interfaces to implement Trojan Columns in DBMS-X: (i) UDF interface, and (ii) CLI interface. In the following, we describe how we implemented Trojan Columns using these two interfaces.

A.1 DBMS-X Table UDF Interface

Interface. The table UDF interface in DBMS-X is a C interface to write UDFs. In general, the signature of UDF to query the Trojan Columns (with n attributes) looks as follows.

```
void SQL_API_FN udf_name (
    SQLUDF_CHAR* tableName,
    SQLUDF_CHAR* projectionStr,
    SQLUDF_CHAR* selectionStr,
    SQLUDF_CHAR* groupbyStr,
    SQLUDF_CHAR* logFile,
    SQLUDF_RESULTTYPE1* RESULT1,
    SQLUDF_RESULTTYPE2* RESULT2,
    .
    .
    SQLUDF_RESULTTYPEn* RESULTn,
    SQLUDF_SMALLINT* RESULT1_IND,
    SQLUDF_SMALLINT* RESULT2_IND,
    .
    .
    SQLUDF_SMALLINT* RESULTn_IND,
    SQLUDF_TRAIL_ARGS_ALL          // current UDF state
)
```

The first 5 arguments are actually passed to the function by the user, when invoking the UDF. The remaining arguments are not passed to the function by the user. They are the result variables to which we write the result row. For each call, DBMS-X maps the memory of the result table to these pointers.

Scratch Pad. There is a specific memory area provided by DBMS-X, which is called the scratchpad. This memory area is kept alive during the individual calls to the UDF and can be used to maintain a state between the calls. In our approach, all main data structures are held inside of the scratchpad, since they are used in several phases.

Multi-threading. Apart from the main (output) thread, we maintain separate threads for I/O and processing. This means that as soon as one of the buffers is free, the next segment is already loaded into it. For each loaded attribute that is compressed, we decompress it in a separate processing thread. The processing thread also performs the selection and produces a selection vector, indicating which tuples qualify. Until the processing thread performs the decompression and selection, the main (output) thread waits. When the processing thread is finished, the main thread inspects the selection vector. If a selected row is found in the vector, then the row data is immediately returned. We keep the current position of the selection vector in the scratchpad to output the next row in the subsequent UDF call. There are several wait/notify constructs necessary to coordinate the segment loading with double buffering, selection and outputting. In between the function calls, we keep alive the processing thread (together with the decompression threads) in the scratchpad.

Communication with DBMS-X. To communicate with DBMS-X from inside the UDFs, we use embedded SQL in C (SQC). As a result, the UDF program is not written in pure C, but in a mixture of C and SQL. For example, it is possible to place statements like "EXEC SQL FETCH ... INTO ..." within the functions. These statements are used to query the Trojan Columns blobs from the database. Anytime we query the database with embedded SQL statements, we have to store the result of this query inside host variables, which act as a connection bridge between DBMS-X and our program. Host variables have to be declared in a separate area and they support special datatypes, corresponding to SQL types. Note that it is not possible to modify the database using UDFs in DBMS X; only querying is allowed. Before installing, the SQC file must be precompiled to create a standard pure C file. This file contains calls to internal DBMS-X functions that represents the SQL equivalents. We can then compile this file using a standard C compiler and link it to the database.

Installation. Finally, we install the UDF to query Trojan Columns (with n attributes) in DBMS-X as follows.

```
CREATE FUNCTION udf_name(
    tableName VARCHAR(128),
    attributeStr VARCHAR(1000),
    selectionStr VARCHAR(1000),
    groupbyStr VARCHAR(1000),
    timeFile VARCHAR(1000)
  )
  RETURNS TABLE (
    attribute_1 TYPE1,
    attribute_2 TYPE2,
    .
    .
    attribute_n TYPEn,
  )
  SPECIFIC udf_name               // internal UDF name
  EXTERNAL NAME 'ext_udf_name'    // UDF program name
  LANGUAGE C
  PARAMETER STYLE DBMS_X_SQL
  NOT DETERMINISTIC
  FENCED NOT THREADSAFE     // different address space
  READS SQL DATA
  NO EXTERNAL ACTION
  SCRATCHPAD 10000          // size in bytes
  FINAL CALL                // final call phase executed
  DISALLOW PARALLEL;        // single database partition
```

A.2 DBMS-X Call Level Interface (CLI)

The table UDF interface in DBMS-X has the limitation that the exact schema of rows to return must be fixed at compile time. To overcome this, we developed a second approach based on the DBMS-X Call Level Interface (CLI) and Stored Procedures (SP). CLI is a C/C++ interface that translates queries and data between an application and a database. It allows us to create the queries for accessing the data dynamically at runtime. Furthermore, as a CLI exists for many DBMSs, the routine is easily portable. In DBMS-X, the entry function for stored procedures looks as follows.

```
SQL_API_RC SQL_API_FN udf_name(
    CHAR *tableName,
    CHAR *projectionStr,
    CHAR *selectionStr,
    CHAR *logFile,
    SQLINT16 *tableName_IND,
    SQLINT16 *projectionStr_IND,
    SQLINT16 *selectionStr_IND,
    SQLINT16 *logFile_IND,
    SQLUDF_TRAIL_ARGS          // stored procedure state
  )
```

Note that in contrast to table UDF interface, we do not need to specify the return type for stored procedures. However, they are not able to return their results directly. Instead, in contrast to UDFs, they allow write accesses to the database. We exploit this to store the results of a query into a temporary table in the database. There it can be queried by normal SQL or used as an intermediate result for further computation. This means an existing user query cannot be translated one to one, but it must be split into two calls: (1) a

stored procedure call, and (2) a query for post-processing and returning the results to the user. For example, for TPC-H query 14 the rewritten query first calls the stored procedure to project and select the relevant data from `lineitem`. The final query then joins `part` table with the result table, which is several orders of magnitude smaller than the original `lineitem` table. So the final query is extremely cheap and the overall costs are dominated by the stored procedure. The main advantage of CLI -SP is that because of its highly dynamic interfaces, we don't need to recompile the routine for every query or table. We only have a single stored procedure that can be used in arbitrary queries without more effort than rewriting the query. We install the stored procedure for querying Trojan Columns as follows.

```
CREATE PROCEDURE sp_name(
    IN name VARCHAR(128),
    IN attr VARCHAR(512),
    IN selPred VARCHAR(512),
    IN logfile VARCHAR(128)
  )
  SPECIFIC sp_name
  DYNAMIC RESULT SETS 0          // do not return results
  NOT DETERMINISTIC
  LANGUAGE C
  PARAMETER STYLE SQL
  FENCED NOT THREADSAFE
  MODIFIES SQL DATA
  PROGRAM TYPE SUB               // used as library
  EXTERNAL NAME 'ext_sp_name'    // procedure program name
```

B. EFFECT OF COMPRESSION

Now let us see the impact of compression on Trojan Columns. We play around with four light weight compression techniques for column stores: delta, 7-bit, dictionary, and run length encoding. We look at the cardinalities of each of the attributes, estimate the expected compression ratio, and pick the compression method which gives the maximum compression ratio. For dictionary compression, we also consider the expected dictionary size. Since Trojan Columns works very well on query set 1, we focus on that in this section. Also note that Trojan Columns is used only for `lineitem` table in query set 1, since other tables have no selection predicates and hence have very high function calls overhead. We successively turn on four levels of compression on `lineitem` table as follows:

Compression Level 0. no compression applied.
Compression Level 1. delta encoding enabled.
Compression Level 2. Level 1 + 7-bit encoding enabled.
Compression Level 3. Level 2 + dictionary compression enabled.
Compression Level 4. Level 3 + run length encoding enabled.

Table 3 shows the data load times for different compression levels.

Measurement	Level 0	Level 1	Level 2	Level 3	Level 4
Upload Time (sec)	1636.44	1642.73	1610.59	1550.63	1451.39
Table Size (GB)	11.83	11.57	10.89	5.19	5.07

Table 3: Upload times and tables sizes with compression.

We can see that the upload time decreases with higher levels of compression. From compression level 0, i.e. no compression, to compression level 4, the improvement in upload time is 185 seconds. The reason for this is that while we spend more CPU cycle to compress the data, we save on I/O when writing the data to disk. Figure 6 shows the query times of query set 1 for different compression levels. From Figure 6, we can see that the query time improves up to compression level 4. For instance, for query 6, the improvement in runtime from compression level 0 to compression level 4 is 49%. However, the improvement is not dramatic. The reason for this is that the overall query costs as well as the UDF overheads dominate the improvements in I/O due to compression.

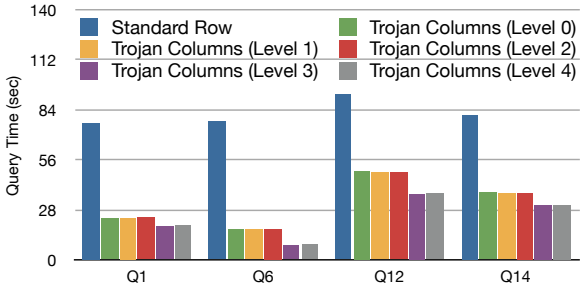


Figure 6: Effect of compression on query set 1.

Table 4 shows the compression methods used for each of the `lineitem` attribute. Except `ExtendedPrice` (too small compression ratio) and `Comment` (VARCHAR), we apply compression on all `lineitem` attributes.

Attribute	Compression Method	Expected Compression Ratio
OrderKey	Delta	4
PartKey	7-Bit	1.4
SuppKey	7-Bit	2
Linenumber	7-Bit	4
Quantity	Dictionary	8
ExtendedPrice	None	-
Discount	Dictionary	8
Tax	Dictionary	8
ReturnFlag	Run Length Encoding	1.42
LineStatus	Run Length Encoding	3.63
ShipDate	Dictionary	4.98
CommitDate	Dictionary	4.98
ReceiptDate	Dictionary	4.98
ShipInstruct	Dictionary	25
ShipMode	Dictionary	10
Comment	None	-

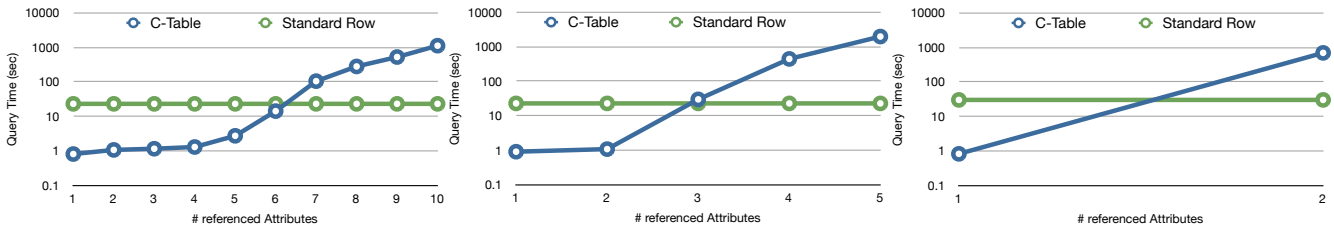
Table 4: Compression methods and the expected compression ratio for each `Lineitem` attribute.

C. SEGMENT SIZES

In all our previous experiments on Trojan Columns, we used a segment size of 10M rows, i.e. 10M rows of the relation were stored as one data blob. Now let us see the impact of segment size on data upload and query times. Note that higher segment sizes produce fewer number of data blobs and hence have lesser random I/Os. Furthermore, higher segments sizes allow us to compress the data better and hence the I/O performance improves even further. However, higher segment sizes consume more database resources (memory, CPU) each time a data blob is processed. Additionally, higher segment sizes limit our capability to parallelize data loading and decompression[3]. Finally, the segment size should be small enough so that the blob data does not exceed its maximum allowed size (e.g. PostgreSQL allows maximum blob size of 1GB).

Table 5 shows the upload times when varying the segment size. We can see that the upload time drops to almost half when varying the segment size from 100K to 10M. This is because higher segment sizes result in more compressed bobs and hence better I/O performance. Figure 8 shows the query runtimes with different segment sizes. For all queries in Figure 8, the query runtimes improve

[3]Ideally, a blob should be just as big such that the time to load and the time to decompress the blob are nearly equal.



(a) Cardinality = 10 (b) Cardinality = 100 (c) Cardinality = 1000

Figure 7: Comparing query times of CTable and standard row for different attribute cardinalities.

with larger segment sizes. However, the improvement is more when changing segment size from 100K to 1M than when changing the segment size from 1M to 10M.

Segment Size	100K	1M	10M
Upload Time (sec)	2824.79	1681.26	1451.39

Table 5: Upload times with varying segment size.

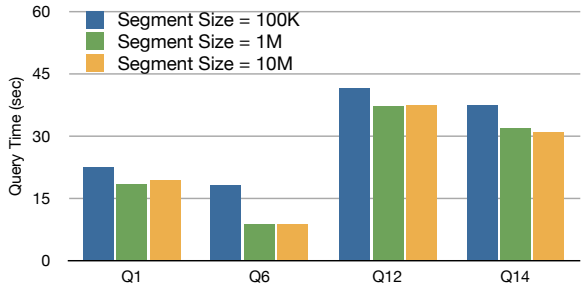
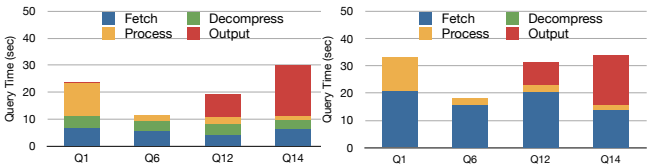


Figure 8: TPC-H query runtimes with varying segment size.

D. QUERY COST BREAK-DOWN

In order to understand where we can further improve the UDFs, we need to see its cost breakdown. Figure 9(a) shows the breakdown of UDF processing time into four costs: fetching data, decompressing data, processing (selections, grouping/aggregation), and outputting the results. From the figure we can see that processing costs dominate in query Q_1 while outputting costs dominate in query Q_{14}. However, fetching and decompression are at the major costs for Q_6 and Q_{12}. To contrast the effect of compression, Figure 9(b) shows the cost breakdown for uncompressed data. We can see that there are no decompression costs now, however the fetching costs go up significantly and dominate most queries.



(a) With compression (b) Without compression

Figure 9: Query cost breakdown (in seconds) for Trojan Columns over TPC-H query set 1.

E. STORED PROCEDURES

As discussed in Section A, we considered two interfaces to implement Trojan Columns in DBMS-X. In the experiments, we showed results from the Table UDF Interface implementation in DBMS-X. This was because the UDFs thus created can be easily nested in SQL queries without much changed (we just need to change the FROM clause). In contrast, the Call Level Interface (CLI) in DBMS-X needs a CALL statement to invoke the UDF and store the results in a temporary table. This temporary table must be then used by the remainder of the query. For the sake of completeness, we also present the results from the CLI implementation of Trojan Columns.

Figure 10 shows the runtimes of Trojan Columns using stored procedures (SP) for TPC-H query set 1. We can see that except for query Q_{14}, Trojan Columns using stored procedures are very close to those Trojan Columns using UDFs. Stored procedures are slow for Query Q_{14} because it produces a large number of output tuples. Since stored procedures cannot return the results, they must write these output tuples into another table.

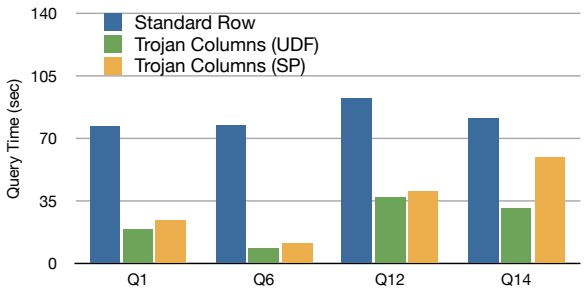


Figure 10: Query times (in seconds) with stored procedures.

F. C-TABLE EVALUATION

In order to investigate C-Table in more detail, we ran some micro-benchmarks on them. We take three synthetic datasets. Each dataset contains integer attributes with the same cardinality (10, 100, and 1000 respectively). For each dataset, we create C-Tables over its attributes and vary the number of referenced attributes.

Figure 7 shows the results. We can see from the figure that for lower cardinalities C-Tables work very well compared to standard row. For instance, for cardinality 10, C-Tables are better than row for up to 6 referenced attributes. However, for higher cardinalities e.g. 1000, C-Tables do not work so well. This is because the tuple reconstruction costs overshadow the benefits of RLE encoding in C-Tables.