In Column-Oriented Database Systems Query Execution

Dr. Veeranjaneyulu G
Department of Computer Science and Engineering
Shadan Women's College of Engineering & Technology, Khairatabad, Hyderabad, Telangana-4

Medapati Lalitha
Department of Computer Science and Engineering
Shadan Women's College of Engineering & Technology, Khairatabad, Hyderabad, Telangana-4

Kampe Shilpa
Department of Computer Science and Engineering
Shadan Women's College of Engineering & Technology, Khairatabad, Hyderabad, Telangana-4

Abstract- There are two different ways to map a two-dimension relational database table onto a one-dimensional storage interface: to store the information in the table in table row-by-row or column-by-column. Database system implementations and research have focused on the row-by row data, since it performs best on the most common application for database systems: business transactional data processing. There are a set of rising applications for database systems for which the row-by-row layout performs badly. These applications are more critical in nature, whose goal is to read through the data to gain new insight and use it to drive decision making and planning. In this paper, we study the problem of poor performance of row-by-row data describe for these emerging applications, and estimate the column-by-column data design chance as a solution to this problem. There has been a diversity of proposal in the literature for how to build a database system on top of column-by-column layout. These proposals have different level of completion effort, and have different performance characteristics. If anyone wanted to create a new database system that utilizes the column-by-column data design, it is ambiguous which proposal to follow. This paper provides (to the best of our knowledge) the only detailed study of multiple implementation approaches of systems and categorizes the different approaches into three wide categories, and evaluates the tradeoffs between approaches. We finish off that building a query executer particularly designed for the column-by-column query layout is essential to accomplish excellent performance. thus, we describe the performance of C-Store, a new database system with a storage layer and query executer built for column-by-column data design. We introduce three new query execution techniques that significantly improve performance. First look at the problem of integrate density and execution so that the query executer is capable of directly operating on squeezed together data. This improves performance by improving I/O (less data needs to be read off disk), and CPU (the data need not be decompressed). We explain our solution to the problem of executer extensibility – how can new compression techniques be added to the system without having to rewrite the operator code? Second, analyze the problem of Tuple construction.

Keywords – C-Store, C-Store optimizer, row-by-row storage, DNA, SenSage.

I. INTRODUCTION

The world of relational database systems is a two dimensional world. Data is stored in the form of table where rows correspond to distinct real-world entities or associations, and columns are attributes of those entities. For example, a business might store information about its customers in a database table where each row contains information about a different customer and each column stores a particular customer attribute.

There is a distinction between the conceptual and physical properties of database tables. The two dimensional property exists only at the conceptual level. On a physical level, the database tables need to be mapped onto one dimensional structure before being stored.

1.1 ROWS VS. COLUMNS

There are two obvious ways to map database tables onto a one dimensional interface: store the table row-by-row or store the table column-by-column. The row-by-row approaches keep all data about an entity together. In the customer example it will store all information about the first customer and then all information about the 2nd customer and so on. The column-by-column approach keep all attribute information together: the entire customer names will be stored consecutively, then all of the customer addresses, etc. Both approaches are reasonable designs and typically a choice is made based on performance expectations. If the expected workload tends to access data on the granularity of an entity (e.g., find a customer, add a customer, delete a customer), then the row-by-row storage is

preferable since all of the needed information will be stored together. On the other hand, if the expected workload tends to read per query only a few attributes from many records , then column-by-column storage is preferable since irrelevant attributes for a particular query do not have to be accessed (current storage devices cannot be read with fine enough granularity to read only one attribute from a row; this is explained further in Section 3.2).The vast majority of commercial database systems, including the three most popular database software systems (Oracle, IBM DB2, and Microsoft SQL Server), choose the row-by-row storage layout. The design implemented by these products descended from research developed in the 1970s. The design was optimized for the most common database application at the time: business transactional data processing. The goal of these applications was to automate mission-critical business tasks. For example, a bank might want to use a database to store information about its branches and its customers and its accounts. Typical uses of this database might be to find the balance of a particular customer's account or to transfer $100 from customer A to customer B in one single atomic transaction. These queries commonly access data on the granularity an entity (find a customer, or an account, or branch information; add a new customer, account, or branch). Given this workload, the row-by-row storage layout was chosen for these systems. Starting in around the 1990s, however, businesses started to use their databases to ask more detailed analytical queries.

1.2 PROPERTIES OF ANALYTIC APPLICATIONS

The natures of the queries to data warehouses are different from the queries to transactional databases. Queries tend to be:

• **Less Predictable**. In the transactional world, since databases are used to automate business tasks, queries tend to be initiated by a specific set of predefined actions..

• **Longer Lasting**. Transactional queries tend to be short, simple queries ("add a customer", "find a balance", "transfer $50 from account A to account B").

• **More Read-Oriented Than Write-Oriented**. Analysis is naturally a read-oriented endeavor. Typically data is written to the data warehouse in batches (for example, data collected during the day can be sent to the data warehouse from the enterprise transactional databases and batch-written over-night), followed by many read only queries.

• **Attribute-Focused Rather Than Entity-Focused**. Data warehouse queries typically do not query individual entities; rather they tend to read multiple entities and summarize or aggregate them.

1.3. Implications on Data Management.

As a consequence of these query characteristics, storing data row-by-row is no longer the obvious choice; in fact, especially as a result of the latter two characteristics, the column-by-column storage layout can be better. The third query characteristic favors a column-oriented layout since it alleviates the oft-cited disadvantage of storing data in columns: poor write performance. In particular, individual write queries can perform poorly if data is 18 laid out column-by-column, since, for example, if a new record is inserted into the database, the new record must be partitioned into its component attributes and each attribute written independently. However, batch-writes do not perform as poorly since attributes from multiple records can be written together in a single action. On the other hand, read queries (especially attribute-focused queries from the fourth characteristic above) tend to favor the column-oriented layout since only those attributes accessed by a query need to be read, and thus this layout tends to be more I/O efficient. Thus, since data warehouses tend to have more read queries than write queries, the read queries are attribute focused, and the write queries can be done in batch, the column-oriented layout is favored. Surprisingly, the major players in the data warehouse commercial arena (Oracle, DB2, SQL Server, and Teradata) store data row-by-row.

1.4 Dissertation Goals, Challenges, and Contributions

The overarching goal of this dissertation is to further the research into column-oriented databases, tying together ideas from previous attempts to build column-stores, propose new ideas for performance optimizations, and building an understanding of when they should be used. In spirit, this dissertation serves as a guide for building a modern column-oriented database system. There are three sub-goals. First, we look at past approaches to building a column-store, investigative the tradeoffs between approaches. Second, we suggest techniques for improving the performance of column-stores. Third, we standard column-stores on applications both inside and outside their traditional sweet-spot.

II. CONCLUSIONS AND FUTURE WORK

Column-oriented database systems, due to their I/O efficiency on read-mostly, attribute-oriented queries are being increasingly adopted in the commercial marketplace. This trend has resulted in a variety of venture-backed column-oriented database start-ups bursting onto the scene, including Vertical, Parcel, Cal Pont, and in the increasing popularity of column-oriented databases that have been around for a while (including Sybase IQ, Sand DNA Ana-lyrics, and SenSage). It is clear that column-stores are well suited for analytical workloads like those found in data warehouses that serve customer relationship management and business intelligence applications. Both in the commercial marketplace and in the research literature, these different variants of column-stores make different architectural decisions and make different claims on their performance relative to standard row-store tech-
nology. In this dissertation, we classified these different column-store variants into three primary implementation approaches, and showed that fundamental differences between these approaches result in significantly different performance. In essence, we found that if the DBMS is designed from scratch for column-oriented data layout, building a storage layer and a query executor around this data layout, a significant performance improvement can be obtained over less aggressive (but easier to implement) approaches. As a result of this finding, we set out to build such a column-store, with a specially designed storage layer and query executor. We presented the architecture and implementations of this new column-store: C-Store.

III. FUTURE WORK

Although the C-Store query executor achieves impressive performance on its own, there is still very little interaction between the executor and the rest of the database system. Most importantly, many of the techniques described in this dissertation have not been incorporated into the C-Store optimizer. The optimizer is not aware of the decompression costs of the various compression algorithms, nor able to use the analytical model presented in this dissertation to decide on a materialization strategy. Further, since column-stores are very I/O efficient. we have found that most queries are CPU (and not disk) limited. Thus, good CPU cost models for each step in a query execution plan need to be 129 incorporated into a cost-based optimizer. Hence, the design of an optimizer designed for a column-oriented database remains an interesting area for future work. This dissertation also has implications on database design. The chapter on compression showed that many narrow materialized views are preferable to fewer wider materialized views since a higher percentage of columns can be sorted (or secondarily sorted) in narrow tables. The chapter on the invisible join showed that in many cases, it does not help to include dimension table columns in fact table materialized views. The design of an automatic database designer that decides what materialized views to create and how data should be sorted /compressed in column-oriented databases/data warehouses remains another interesting area for future work. Column-stores can be thought of as an extreme opposite of row-stores. This observation leads to the question of whether there are situations where a hybrid store might be able to do better than a column- or row-store could do individually. For example, one might want to store sorted attributes and variable-width attributes in a columns, and fixed-width attributes in (dense-packed) rows. Or one might want to store the less frequently accessed attributes in rows and the more frequently accessed attributes in columns. Goldilocks-stores could be an interesting area of research. Load times into column-stores are another avenue for future work. Certainly, it is expected that loading a column-oriented database will be slower than loading a row-oriented database, but it would be interesting to evaluate the extent of this disadvantage look at algorithms to alleviate it. Finally, the chapter on processing Semantic Web data also only focused on query execution. The design of a complete database for Semantic Web applications, including parsing from SPARQL to SQL, query rewriting to a vertically partitioned schema, and path-expression optimization, would be an exciting system to build.

IV. CONCLUSION

In this paper, we described three approaches to building a column-store. Each approach requires more modifications to the DBMS than the last. We implemented each approach and showed, through performance results, that these extra modifications to the DBMS result in significant performance gains. The third approach, which requires that the storage layer and the query execution engine be designed for the column-orientation of the data, performs almost a factor of 3 faster than the second approach and at least a factor of 5 faster than the first approach. Further, this approach opens up possibilities for further optimizations that, as we will show in later chapters, result in significant performance gains. Thus, we recommend the third approach for building column-oriented database systems. Given that this is the case, we describe the details of how we built our column-store, “C-Store”, by using this approach.

REFERENCES

[1] Library catalog data.http://simile.mit.edu/rdf-test-data/barton/.
[2] Longwell website.http://simile.mit.edu/longwell/.
[3] LZOP compression code.http://www.lzop.org.
[4] Redland RDF Application Framework.http://librdf.org/.
[5] Simile website.http://simile.mit.edu/.
[6] Swoogle.http://swoogle.umbc.edu/.
[7] TPC-H.http://www.tpc.org/tpch/.
[8] TPC-H Result Highlights Scale 1000GB. http://www.tpc.org/tpch/results/tpch_result_detail.asp?id=107102903.
[9] UniProt RDF Dataset.http://dev.isb-sib.ch/projects/uniprot-rdf/.
[10] Vertica. http://www.vertica.com/.
[11] WordNet RDF Dataset.http://www.cogsci.princeton.edu/˜wn/.
[12] World Wide Web Consortium (W3C).http://www.w3.org/.
[13] RDF Primer. W3C Recommendation.http://www.w3.org/TR/rdf-primer, 2004.
[14] RDQL - A Query Language for RDF. W3C Member Submission 9 January 2004. http://www.w3.org/Submission/RDQL/, 2004.
[15] C-Store code release under BSD license.http://db.csail.mit.edu/projects/cstore/, 2005.
[16] SPARQL Query Language for RDF. W3C Working Draft 4 October 2006.
[17] http://www.w3.org/TR/rdf-sparql-query/, 2006.
[18] Daniel J. Abadi. Column stores for wide and sparse data. In CIDR, Asilomar, CA, USA, 2007.
[19] Daniel J. Abadi, Samuel R. Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In SIGMOD, pages 671–682, Chicago, IL, USA, 2006.
[20] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel R. Madden. aterialization strategies in a column-oriented DBMS. In ICDE , pages 466–475, Istanbul, Turkey, 2007.
[21] Rakesh Agrawal, Amit Somani, and Yirong Xu. Storage and Querying of E-Commerce Data. In VLDB, 2001.
[22] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance. In VLDB , pages 169–180, 2001.