An Intuition of the Necessitate of Column-Oriented Database Systems

Sanil.S.Nair [1]
Lecturer, Prof. Ram Meghe Institute of Technology & Research, Badnera- Amravati.

Vishwajit.S.Patil [2]
Lecturer, Prof. Ram Meghe Institute of Technology & Research, Badnera- Amravati.

Bhruthari.G.Pund [3]
Lecturer, Prof. Ram Meghe Institute of Technology & Research, Badnera- Amravati.

ABSTRACT
In this paper, we are paying attention to the problem of poor performance of row-by-row data layout for the emerging applications, and evaluate the column-by-column data layout opportunity as a solution to this problem. There have been a variety of proposals for how to build a database system on top of column-by-column layout. These proposals have different levels of implementation effort, and have different performance characteristics. If one wanted to build a new database system that utilizes the column-by-column data layout, it is unclear which proposal to follow. This paper provides (to the best of our knowledge) the detailed study of multiple implementation approaches of such systems, categorizing the different approaches into three broad categories, and evaluating the tradeoffs between approaches.

Keywords
Column oriented database, c-store, Query execution plans, and row stores

1. INTRODUCTION
We ask that authors follow some simple guidelines. In essence, we ask you to make your paper look exactly like this document. The easiest way to do this is simply to download the template, and replace the content with your own material.

The world of relational database systems is a two dimensional world. Data is stored in tabular data structures where rows correspond to distinct real-world entities or relationships, 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 (name, address, e-mail, etc.). There is, however, a distinction between the conceptual and physical properties of database tables. This afore-mentioned two dimensional property exists only at the conceptual level. At a physical level, database tables need to be mapped onto one dimensional structure before being stored. This is because common computer storage media (e.g. magnetic disks or RAM), despite ostensibly being multi-dimensional, provide only a one dimensional interface (read and write from a given linear offset).

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 approach keeps all information about an entity together. In the customer example above, it will store all information about the first customer, and then all information about the second customer, etc. The column-by-column approach keeps 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 (e.g., a query that finds the most common e-mail address domain), then column-by-column storage is preferable since irrelevant attributes for a particular query do not have to be accessed

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.

2. NEED FOR CHANGE
Starting in around the 1990s, however, businesses started to use their databases to ask more detailed analytical queries. For example, the bank might want to analyze all of the data to find associations between customer attributes and heightened loan risks. Or they might want to search through the data to find customers who should receive VIP treatment. Thus, on top of using databases to automate their business processes, businesses started to want to use databases to help with some of the decision making and planning. However, these new uses for databases posed two problems. First, these analytical queries tended to be longer running queries, and the shorter transactional

write queries would have to block until the analytical queries finished (to avoid different queries reading an inconsistent database state). Second, these analytical queries did not generally process the same data as the transactional queries, since both operational and historical data (from perhaps multiple applications within the enterprise) are relevant for decision making. Thus, businesses tended to create two databases (rather than a single one); the transactional queries would go to the transactional database and the analytical queries would go to what are now called data warehouses. This business practice of creating a separate data warehouse for analytical queries is becoming increasingly common; in fact today data warehouses comprise $3.98 billion [10] of the $14.6 billion database market [7] (27%) and is growing at a rate of 10.3% annually [10].

3. PROPERTIES OF ANALYTIC APPLICATIONS

The nature of the queries to data warehouses is 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. As a result, the basic structures of the queries used to implement these predefined actions are coded in advance, with variables filled in at run-time. In contrast, queries in the data warehouse tend to be more exploratory in nature. They can be initiated by analysts who create queries in an ad-hoc, iterative fashion.

• Longer Lasting: Transactional queries tend to be short, simple queries ("add a customer", "find a balance", "transfer $50 from account A to account B"). In contrast, data warehouse queries, since they are more analytical in nature, tend to have to read more data to yield information about data in aggregate rather than individual records. For example, a query that tries to find correlations between customer attributes and loan risks needs to search though many records of customer and loan history in order to produce meaningful correlations.

• More Read-Oriented than Write-Oriented: Analysis is naturally a read-oriented endeavour. 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. Occasionally data will be temporarily written for "what-if" analyses, but on the whole, most queries will be read-only.

• 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 (for example, queries like "what is the average customer balance" are more common than "what is the balance of customer A's account"). Further, they tend to focus on only a few attributes at a time (in the previous example, the balance attribute) rather than all attributes.

4. 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 favours 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 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 favours 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 favoured. Surprisingly, the major players in the data warehouse commercial arena (Oracle, DB2, SQL Server, and Teradata) store data row-by-row (in this paper, they will be referred to as "row-stores"). Although speculation as to why this is the case is beyond the scope of this paper, this is likely due to the fact that these databases have historically focused on the larger transactional database market and wish to maintain a single line of code for all of their database software [9]. Similarly, database research has tended to focus on the row-by-row data layout, again due to the field being historically transactional focused. Consequently, relatively little research has been performed on the column-by-column storage layout ("column-stores").

5. EXPLORING COLUMN-STORE DESIGN APPROACHES

Due to the recent increase in the use of database technology for business analysis, planning, and intelligence, there has been some recent work that experimentally and analytically compares the performance of column-stores and row-stores [2, 3, 4, 5, 64, 9, 1]. In general, this work validates the prediction that column-stores should outperform row-stores on data warehouse workloads. However, this body of work does not agree on the magnitude of relative performance. This magnitude ranges from only small differences in performance [4], to less than an order of magnitude difference [3, 5], to an order of a magnitude difference [9, 1], to, in one case, a factor of 120 performance difference [9].

One major reason for this disagreement in performance difference is that there are multiple approaches to building a column-store.

6. APPROACHES IN BUILDING A COLUMN-STORE

One approach is to vertically partition a row-store database. Tables in the row-store are broken up into multiple two column tables consisting of (table key, attribute) pairs [6]. There is one two-column table for each attribute in the original table. When a query is issued, only those thin attribute-tables relevant for a particular query need to be accessed—the other tables can be ignored. These tables are joined on table key to create a projection of the original table containing only those columns necessary to answer a query, and then execution proceeds as normal. The smaller the percentage of columns from a table that needs to be accessed to answer a query, the better the relative performance with a row-store will be (i.e., wide tables or narrow queries will have a larger performance difference). Note that for this approach, none of the DBMS code needs to be modified — the approach is a simple modification of the schema.

Another approach is to modify the storage layer of the DBMS to store data in columns rather than rows. At the logical level the schema looks no different; however, at the physical level, instead of storing a table row-by-row, the table is stored column-by-column. The key difference relative to the previous approach is that table keys need not be repeated with each attribute; the ith value in each column matches up with the ith value in all of the other columns (i.e., they belong to the same tuple). Similarly with the previous approach, only those columns that are relevant or a particular query need to be accessed and merged together. Once this merging has taken place, the normal (row-store) query executor can process the query as normal. This is the approach taken in the studies performed by Harizopoulos et. al. [5] and Halverson et. al. [4]. This approach is particularly appealing for studies comparing row-store and column-store performance since it allows for the examination of the relative advantages of systems in isolation. They only vary whether data is stored by columns or rows on disk; data is converted to a common format for query processing and can be processed by an identical executor. A third approach is to modify both the storage layer and the query executor of the DBMS [8, 9, 1]. Thus, not only is data stored in columns rather than rows, but the query executor has the option of keeping the data in columns for processing. This approach can lead to a variety of performance enhancements. For example, if a predicate is applied to a column, that column can be sent in isolation to the CPU for predicate application, alleviating the memory-CPU bandwidth bottleneck. Further, iterating through a fixed width column is generally faster than iterating through variable-width rows (and if any attribute in a row is variable-width, then the whole row becomes variable width).Finally, selection and aggregation operations in a query plan might reduce the number of rows that need to be created (and output as a result of the query), reducing the cost of merging columns together. Consequently, keeping data in columns and waiting to the end of a query plan to create rows can reduce the row construction cost. Thus, one goal of this paper is to explore multiple approaches to building a column-oriented database system, and to understand the performance differences between these approaches (and the reasons behind these differences).

7. CONCLUSION

Surprisingly, the major players in the data warehouse commercial arena (Oracle, DB2, SQL Server, and Teradata) store data row-by-row (in this paper, they will be referred to as "row-stores"). Although speculation as to why this is the case is beyond the scope of this paper, this is likely due to the fact that these databases have historically focused on the larger transactional database market and wish to maintain a single line of code for all of their database software [9]. Similarly, database research has tended to focus on the row-by-row data layout, again due to the field being historically transactional focused. Consequently, relatively little research has been performed on the column-by-column storage layout ("column-stores").

8. REFERENCES

[1] Peter Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-pipelining query execution. In CIDR, 2005.

[2] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. Query optimization in compressed database systems. In SIGMOD '01, pages 271–282, 2001.

[3] George Copeland and Setrag Khoshafian. A decomposition storage model. In SIGMOD, pages 268–279, 1985.

[4] Alan Halverson, Jennifer L. Beckmann, Jeffrey F. Naughton, and David J. Dewitt. A Comparison of C-Store and Row-Store in a Common Framework. Technical Report TR1570, University ofWisconsin-Madison, 2006.

[5] Stavros Harizopoulos, Velen Liang, Daniel J. Abadi, and Samuel R. Madden. Performance tradeoffs in readoptimized databases. In VLDB, pages 487–498, Seoul, Korea, 2006.

[6] Setrag Khoshafian, George Copeland, Thomas Jagodis, Haran Boral, and Patrick Valduriez. A query processing strategy for the decomposed storage model. In ICDE, pages 636–643, 1987.

[7] Carl Olofson. Worldwide RDBMS 2005 vendor shares. Technical Report 201692, IDC, May 2006.

[8] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, EdmondLau, Amerson Lin, Samuel R. Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alexander Rasin, Nga Tran, and Stan B. Zdonik. C-Store: A Column-Oriented DBMS. In VLDB, pages 553–564, Trondheim, Norway, 2005.

[9] Michael Stonebraker, Chuck Bear, Ugur Cetintemel, Mitch Cherniack, Tingjian Ge, Nabil Hachem, Stavros Harizopoulos, John Lifter, Jennie Rogers, and Stan Zdonik.

One size fits all? - Part 2: Benchmarking results.In Proceedings of the Third International Conference on Innovative Data Systems Research (CIDR), January 2007.

[10] Dan Vesset. Worldwide data warehousing tools 2005 vendor shares. Technical Report 203229, IDC, August 2006.