International Journal of Computer Science and Informatics

Volume 3 | Issue 4 Article 13

April 2014

THE MATHEMATICAL ANALYSIS OF COLUMN ORIENTED DATABASE

AMIT KUMAR DWIVEDI
Department of Computer Science and Engineering, Rajasthan Institute of Engineering and Technology, Jaipur, India, amitdwi@gmail.com

VIJAY KUMAR SHARMA
Department of Computer Science and Engineering, Rajasthan Institute of Engineering and Technology, Jaipur, India, vijaymayankmudgal2008@gmail.com

THE MATHEMATICAL ANALYSIS OF COLUMN ORIENTED DATABASE

AMIT KUMAR DWIVEDI[1], VIJAY KUMAR SHARMA[2]

[1,2]Department of Computer Science and Engineering, Rajasthan Institute of Engineering and Technology, Jaipur, India
Email: amitdwi@gmail.com, vijaymayankmudgal2008@gmail.com

Abstract— There are two obvious ways to map a two-dimension relational database table onto a one-dimensional storage interface: store the table row-by-row, or store the table column-by-column. Historically, database system implementations and research have focused on the row-by row data layout, since it performs best on the most common application for database systems: business transactional data processing. However, there are a set of emerging applications for database systems for which the row-by-row layout performs poorly. These applications are more analytical 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 about the facts responsible for making Column Oriented database when traditional Row-oriented databases are already present, analysis of generation of Column Oriented databases, etc. Finally It will be conclude by giving a probabilistic analysis of Column Oriented database.

Keywords-*Data warehouses, Database systems, Databases, Performance analysis*

I. INTRODUCTION

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 College might store information about its students in a database table where each row contains information about a different student and each column stores a particular student attribute (name, address, e-mail, etc.).

A. 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 approach keeps all information about an entity together. In the College example above, it will store all information about the first student, and then all information about the second student, etc. The column-by-column approach keeps all attribute information together: all of the student names will be stored consecutively, then all of the student 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 student, add a student, delete a student), 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 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. 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 [3] of the $14.6 billion database market [2] (27%) and is growing at a rate of 10.3% annually [3]the various table text styles are provided. The formatter will need to create these components, incorporating the applicable criteria that follow.

B. Properties of analytic applications

[5]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. 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 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. 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.

C. 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 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. Although speculation as to why this is the case is beyond the scope of this dissertation, 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 [6]. Similarly, database research has tended to focus on the row-by-row data layout, again due to the field being historically transaction ally focused. Consequently, relatively little research has been performed on the column-by-column storage layout ("column-stores").

The overarching goal of this paper is to further the research into column oriented databases, How Column Oriented data base can be generated by traditional row oriented databases. An approach of generating column oriented database is Vertical Partitioning. Vertical partitioning subdivides attributes into groups and assigns each group to a physical object**.** In our context [1]Vertical Partitioning will be defined as breaking apart the columns of high cardinality (number of columns) tables into distinct smaller tables based on frequency of usage of each column or set of columns. But here the question is about the parameters responsible for generation of Column Oriented Database? On what basis we perform vertical partitioning.

II. RELATED WORK

[4] Vertical partitioning is used during the design of a database to improve the performance of transactions: fragments consist of smaller records, and therefore fewer pages in secondary memory are accessed to process a transaction. When allocating data to a memory hierarchy, vertical partitioning is used to store the attributes that are most heavily accessed in the fastest memory. In the design of a multiple-site distributed database, fragments are allocated, and possibly replicated, at the various sites.

Vertical partitioning algorithm can be understand by the following figure

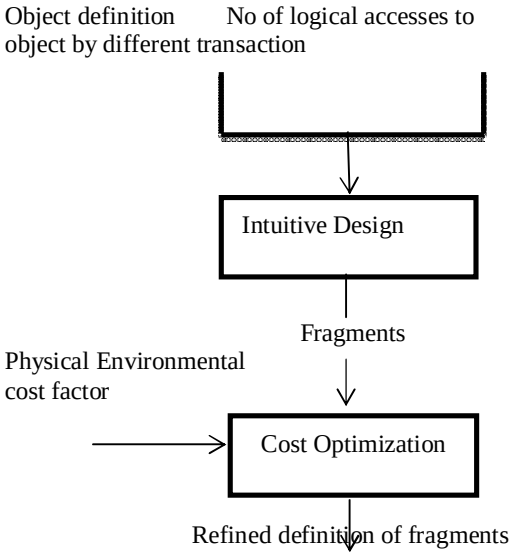

Fig.1

Vertical fragments are ultimately stored in the database by using some physical file structure. In order to obtain improved performance, fragments must be "closely matched" to the requirements of the transactions. The ideal case occurs when each transaction "matches" a fragment, because the transaction has to access that fragment only. If certain sets of attributes are always processed together by transactions, the design process is trivial. But in real-life applications one rarely comes across such trivial examples; hence, for objects with tens of attributes, we need to develop a systematic approach to vertical partitioning. As pointed out in [ll], an object with m attributes can be partitioned into B(m) different ways, where B(m) is the m^{th} Bell number; for large m, B(m) approaches m^m; for m =15, it is $\approx 10^9$, for m = 30, it is $\approx 10^{23}$.

The first step in the design of a vertical partition is to construct an ATTRIBUTE AFFINITY (AA) matrix. Figure 1 shows an example of an attribute affinity matrix derived from the attribute usage in Table 1. AA is a symmetric square matrix which records the affinity among the attributes $a_{i'}$ and $a_{i''}$ as a single number, $aff_{i'i''}$ (= $aff_{i''i'}$), defined below.

Let the following parameters be defined for each transaction k:

U_{ki} = 1 if transaction k uses attribute a_i

= 0 otherwise (U_{ki} is an element of ATTRIBUTE USAGE(AU) matrix).

TABLE 1

	Attribute usage matrix										Type	Single site accesses per time period
Attribute	1	2	3	4	5	6	7	8	9	10		
T 1	1	0	0	0	1	0	1	0	0	0	Type 1 = R	Acc 1 = 25
T 2	0	1	1	0	0	0	0	1	1	0	Type 2 = R	Acc 2 = 50
T 3	0	0	0	1	0	1	0	0	0	1	Type 3 = R	Acc 3 = 25
T 4	0	1	0	0	0	0	1	1	0	0	Type 4 = R	Acc 4 = 35
T 5	1	1	1	0	1	0	1	1	1	0	Type 5 = U	Acc 5 = 25
T 6	1	0	0	0	1	0	0	0	0	0	Type 6 = U	Acc 6 = 25
T 7	0	0	1	0	0	0	0	0	1	0	Type 7 = U	Acc 7 = 25
T 8	0	0	1	1	0	1	0	0	1	1	Type 8 = U	Acc 8 = 15
	Attribute length vector											
	10	8	4	6	15	14	3	5	9	12		

III. PROPOSED PROBABILISTIC ANALYSIS FOR COLUMN ORIENTED DATABASE

A. Probabilistic relations between Row Oriented database and Column Oriented database

Suppose there is a traditional row-oriented database having a Relation R.

The relation R contains tuples (S, C_i)

Where S is the Serial no, C_i represents the set of columns.

$1 \leq i \leq n$

Now if we want to think about Column-Oriented database development then we first have to think why we need Column-Oriented database and how to derive Column-Oriented database from traditional Row-Oriented database

Suppose the relation R having two million rows and several columns but we have to access data only from few columns. But as per the nature of traditional databases unnecessary extra data from other columns will also be read when in query is fired.

Column oriented database are generated by analyzing the probability of hit (Affinity) that occurs on the columns of the Relation R(S,C_i). So on some columns there will be less hit and some of them there will be more hits. The columns that will be more hit are "vertically partitioned" with the serial no.

For Example – Suppose Column C_1 and C_5 are generally accessed by query. So we can make two separate relations R1(S, C_1) and R2(S, C_5).The approach that will be used for separation is called "Vertical Partitioning". According to the above partition of relation we can say that column C_1 and C_5 are of higher probability of hit.

If $R_1(S,C_1)$ is separated first from R(S,Ci) where $1 \leq i \leq n$. We can represent this separation by the following probability equation

$$P_{RH}(C_1) = P_{RHMAX}(C_i) \quad (1)$$

where $1 \leq i \leq n$

$R_2(S,C_5)$ is separated after $R_1(S,C_1)$ from R(S,Ci). We can represent this separation by the following probability inequality

$$P_{RH}(C_1) \geq P_{RH}(C_5)) \quad (2)$$

Where $P_{RH}(C_1)$ represents probability of hit on columns C_1, $P_{RHMAX}(C_i)$ represents probability of maximum hit on columns of relation R(S, Ci) and $P_{RH}(C_5)$ represents probability of maximum hit on columns C_5.

Both column C_1 and C_5 must be higher probability of hit than other columns because of their separation so this complete probability equation can be represented by following equation.

$$P_{RH}(C_1) \geq P_{RH}(C_5) > P_{RH}(\text{Other Columns in the relation R(S,Ci)}) \quad (3)$$

B. Total Probability Calculation for Column Oriented database

Now the Column Oriented database has been generated by the two columns C_1 and C_5. If column C_1 and C_5 are accessed separately in a mutually exclusive manner then total probability of hit on Column Oriented database can be calculated by the following equation

$$P_{HC} = P_H(C_1)+P_H(C_5) \quad (4)$$

If column C_1 and C_5 are accessed together then the total probability of hit can be calculated by the following equation

$$P_{HC} = P_H(C_1)*P_H(C_5) \quad (5)$$

Where P_{HC} represents total probability of hit on Column Oriented database, $P_H(C_1)$ & $P_H(C_5)$ represents probability of hit on columns C_1 and C_5 respectively.

C. Generalized Probability Calculation

Suppose there are n columns out of which k columns are accessed together and rest of the columns are accessed separately in a mutually exclusive manner then on the basis of equation (4) and (5) we may generalized the total probability of hit by equation below

$$P_{HC} = \prod_{i=1}^{k} Ci + \sum_{i=k+1}^{n} Ci \quad (6)$$

IV. FUTURE WORK

Vertical partitioning adds some redundancy in our database system because it requires that the position attribute to be stored in every column, which wastes space and bandwidth. This approach also doesn't support logical data independence. At physical level both relations whether it is column-store or row-store needs to be updated at the same time to maintain consistency. So there is a need of designing storage layer and the query execution engine for the column orientation of the data then it will certainly perform better than the current approach.

V. CONCLUSION

The probabilistic analysis helps to analyze the column oriented database development mathematically. Vertical Partitioning is a very efficient approach in Column Oriented database design. We don't have to change or modify our Database software also we don't need to buy any new software. Using this approach we can make our Row Oriented database acts as a Column Oriented Database

REFERENCES

[1] Fadi Chalfoun, Aphelion Web Development Blog "Database Optimization:Vertical Partitioning in MySQL" on February 11, 2010

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

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

[4] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie dou "Vertical Partitioning Algorithms for Database Design" Stanford University

[5] Daniel J. Abadi "Query Execution in Column-Oriented Database Systems" Massachusetts Institute of Technology,February 2008

[6] 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.

◈◈◈