Pivoting approaches for bulk extraction of Entity–Attribute–Value data

Valentin Dinu*, Prakash Nadkarni, Cynthia Brandt

Center for Medical Informatics, Yale University School of Medicine, PO Box 206009, New Haven, CT 06520-8009, United States

Abstract

Entity-Attribute-Value (EAV) data, as present in repositories of clinical patient data, must be transformed (pivoted) into one-column-per-parameter format before it can be used by a variety of analytical programs. Pivoting approaches have not been described in depth in the literature, and existing descriptions are dated. We describe and benchmark three alternative algorithms to perform pivoting of clinical data in the context of a clinical study data management system. We conclude that when the number of attributes to be returned is not too large, it is feasible to use static SQL as the basis for views on the data. An alternative but more complex approach that utilizes hash tables and the presence of abundant random-access-memory can achieve improved performance by reducing the load on the database server.

© 2006 Published by Elsevier Ireland Ltd.

1. Introduction

The "generic" or Entity-Attribute-Value (EAV) data modeling approach [1] is used in database design when a potentially vast number of parameters can describe something, but relatively few apply to a given instance. This model is especially appropriate for physical representation of the clinical data sub-schema of clinical data repositories (CDRs), where the total number of possible parameters across all specialties of medicine ranges in the hundreds of thousands. Large-scale systems that utilize EAV design for clinical data are the HELP CDR [2,3] and its commercial version, the 3M CDR [4], the Columbia-Presbyterian CDR, the Cerner PowerChart Enterprise CDR [5], and the clinical study data management systems (CSDMS) Oracle Clinical [6] and Phase Forward’s ClinTrial [7].

Conceptually, an EAV table consists of triplets, an Entity (the thing being described, e.g., a patient’s clinical encounter), an Attribute (a parameter of interest) and a Value (for that attribute). Because all types of facts reside in the same "value" column, EAV data needs transforming (pivoting) into one-column-per-parameter structure for use by applications such as graphing and many statistical analyses. This operation lacks built-in support in the current versions of relational database engines. (Note: SQL Server 2005 includes a newly introduced PIVOT command. This command, however, requires an aggregate function to be specified, so that its output includes transformed values such as the mean, sum, etc. rather than the original values that are desired in the basic EAV-pivoting operation.) The issues of pivoting clinical EAV data have not been previously explored in depth.

A pivoting operation extracts a subset of data from the repository, e.g., all results for a test panel or questionnaire gathered over a period, or for a given clinical study, yielding a table with as many columns as parameters of interest, with additional columns identifying individual rows, e.g., patient ID and time stamps. The table is often written to disk as a delimited text file for importing by other applications. The most detailed existing description of pivoting, Johnson et al.
describes the use of data access modules (DAMs) in the Columbia Clinical Repository—procedural code to implement the equivalent of pivoted "views". While noting that DAMs are "complex and hard to modify to meet the needs of application developers in a timely manner", the authors identify several limitations of alternative approaches such as static SQL views, e.g., a static SQL view needs as many joins as attributes of interest. (Note: In some highly normalized EAV schemas, one would need twice or thrice this number.) In 1994, most database engines limited the number of joins per statement to a relatively small number, e.g., 16. These limits are now more generous (e.g., 256 joins in SQL Server 2000).

A well-known SQL tuning text [9] mentions optimization of a statement to a relatively small number, e.g., 16. These limits are now more generous (e.g., 256 joins in SQL Server 2000).

Below we discuss three methods that can generate the same pivoted output table using different approaches: full outer join, left outer join, and hash tables performing in memory the equivalent of multiple joins.

2.1. Method A: using full outer joins

Algorithm. Any given statement generates one strip of data per attribute of interest, and then combines the strips using a series of FULL OUTER JOIN operations. Each strip is created through an inner join between the Entities table and the EAV data table—the former being filtered on Study ID and CRF ID, since the same CRF can be used across multiple studies, and the latter filtered on Attribute ID. In addition, for N attributes, N – 1 full outer joins are needed. The total number of join operations per statement is therefore 2 × N – 1.

2.2. Method B: using left outer joins

Algorithm. We determine essential Entity information (Encounter ID, patient ID, time stamp) on the total number of clinical encounters (641) by filtering the Entities table alone on Study ID and CRF ID. We join this information with each strip of data, generated as above. However, we use LEFT OUTER JOIN operations, where complete Entity information merges with whatever matches for each attribute. The total number of joins per statement is N inner joins (to generate each attribute's data), plus N outer joins for merging = 2 × N.

2.3. Method C: using hash tables and memory to perform the equivalent of multiple joins

Algorithm. We formulate the general pivoting problem: For individual attributes conceptually requires full outer join operations, where non-matching rows on either side of a join are preserved, and missing values recorded as "nulls". Currently, most mainstream database engines support "full outer joins" natively in SQL.

Benchmarking was written in Java and ran against three DBMSs: Oracle 9i, SQL Server 2000 SP4 and SQL Server 2005 Beta 2. (While two of the systems are by the same vendor, our results indicate that the query execution engines of the two are quite different.) The benchmarks for each database used the same schema and data, and utilized the same indexes. The databases and application ran on the same dedicated machine (single-CPU 1.8 GHz Pentium 4 with 1 GB RAM), to eliminate the factor of network bandwidth. At the time of benchmarking, no applications were running on the computer except the Java code and the database being tested.

We ran each test at least three times and averaged the results. The test database schema, test data set, Java code, generated queries and detailed benchmarks are available via ftp://custard.med.yale.edu/pivot/Benchmarks.zip.

Formulating the general pivoting problem: For individual attributes conceptually requires full outer join operations, where non-matching rows on either side of a join are preserved, and missing values recorded as "nulls". Currently, most mainstream database engines support "full outer joins" natively in SQL.
Step 4a. A query fetches all EAV triplets (Encounter ID, DBMS method, while Fig. 2 illustrates the same results grouped by Beta 2. Server 2000 SP4 and from 87 to 18961 ms on SQL Server 2005 times for 1–42 attributes ranged from 206 to 8055 ms on SQL with the number of attributes (experiments were halted. The Java process crashed with a SQL exception on attempting a 10-attribute merge. Inserting a variety of attributes = 0.955.) The query run (Fig. 1A) time was spent executing the query and then iterating where the EAV data is stored across multiple data-type-specific tables (e.g., strings, integers, decimal numbers, as in TriaDB), one would repeat the second query for each necessary EAV table, as determined by metadata that indicated how many attributes of each data-type existed for the desired set.

The ordering information is used to make the basic algorithm more scalable if needed, as described later. We capture this data from the database and use it to create a second hash table, with Entity/Encounter ID as key and row number in the array as value—e.g., Encounter 14568 is in row 45.

Step 3. We dynamically allocate a two-dimensional array of strings (number of entities X number of attributes). All elements are initialized to blanks.

Step 4a. A query fetches all EAV triplets (Encounter ID, Attribute ID, Value) for the given Study ID, CRF ID and Attribute ID, via a join to the Entities table. (In a slight variation to this method, which we will call Method C, one can retrieve the EAV triplets for all the Attribute IDs for the given Study ID and CRF ID. This variation decreases the load on the database and can avoid the performance degradation seen in one of the tested databases—see Section 3.)

Step 4b. Iterating through each returned row, we place the Value in the 2-D array in the row and column indicated by its corresponding Encounter and Attribute IDs, respectively: the two hash tables allow speedy row/column determination. At the end of the iterations, empty values remain blank. For a situation where the EAV data is stored across multiple data-type-specific tables (e.g., strings, integers, decimal numbers, as in TriaDB), one would repeat the second query for each necessary EAV table, as determined by metadata that indicated how many attributes of each data-type existed for the desired set.

3. Results of benchmark tests

The results for the benchmark tests using the three methods (A–C, described above) on the three DBMSs (Oracle 9i, SQL Server 2000 SP4 and SQL Server 2005 Beta 2) are illustrated in Figs. 1 and 2. To facilitate comparison across both method and DBMS, Fig. 1 illustrates the results grouped by method, while Fig. 2 illustrates the same results grouped by DBMS.

3.1. Results for Method A: using full outer joins (Fig. 1A)

Oracle 9i: Execution time increased exponentially, from 76 milliseconds (ms) for one attribute to 56,968 ms for nine attributes. (Pearson $R^2$ for log(time) versus number of attributes = 0.955.) The Java process crashed with a SQL exception on attempting a 10-attribute merge. Inserting a variety of optimizer hints in the generated SQL did not help, and further experiments were halted.

SQL Server: Execution time increased at a quadratic rate with the number of attributes ($R^2 = 0.994$ for SQL Server 2000 SP4, $R^2 = 0.999$ for SQL Server 2005 Beta 2). The query ran times for 1-42 attributes ranged from 206 to 8055 ms on SQL Server 2000 SP4 and from 87 to 18961 ms on SQL Server 2005 Beta 2.

3.2. Results for Method B: using left outer joins (Fig. 1B)

Oracle 9i: Performance scaled linearly, from 67 ms for one attribute, to 965 ms for 42 attributes, the last involving a total of 82 join operations in a single SQL statement ($R^2$ for time versus number of attributes = 0.996). SQL Server: Execution times were much higher—ranging from 190 ms for 1 attribute to above 22 s (s) and 10 s for SQL Server 2005 Beta 2 and SQL Server 2000 SP4, respectively. Execution times on SQL Server had an exponential growth for the first 11-12 attributes ($R^2 = 0.981$ for SQL Server 2000 SP4, $R^2 = 0.990$ for SQL Server 2005 Beta 2), followed by a linear increase for more than 13 attributes ($R^2 = 0.990$ for SQL Server 2000 SP4, $R^2 = 0.959$ for SQL Server 2005 Beta 2). On SQL Server, the execution times for left outer join were higher than the times for the full outer join.

3.3. Results for Method C: using hash tables and memory to perform the equivalent of multiple joins (Fig. 1C)

Oracle 9i: Execution time grew linearly ($R^2 = 0.946$) from 80 ms for 1 attribute to 547 ms for 42 attributes.

SQL Server: The behavior of SQL Server 2000 SP4 differed considerably from that of SQL Server 2005 Beta 2. SQL Server 2005 Beta 2: Execution time grew linearly ($R^2 = 0.967$) from 77 ms for 1 attribute to 474 ms for 42 attributes.

SQL Server 2000 SP4: A notable and increasing performance degradation was observed for about 9–20 attributes, beyond which the execution times became lower and grew more linearly at a smaller rate. This behavior, presumably due to SQL Server 2000's attempt to optimize the query, prompted us to try bypassing the SQL Server's optimization attempt. In this slightly modified method, called Method C', we retrieved from the database the values for all the attributes (for the given Study ID and CRF ID) and only the desired values were then picked to be stored in the pivoted array—see Fig. 1D.

As expected, the times for Method C' stayed almost flat, independent of the number of attributes desired—since most of the time was spent executing the query and then iterating through all the retrieved values to see if they need to be stored in the pivoted array or not. This modification removed the irregular increase in execution times for SQL Server 2000 SP4, but also resulted in higher execution times for Oracle 9i and SQL Server 2005 beta2, especially for a small number of attributes.

4. Discussion

4.1. Possible explanations of results

It is well known that the multi-table join problem is NP-hard with respect to performance optimization. The number of join orders to be evaluated to determine the join order that gives the fastest performance grows exponentially with the number of tables to be joined [11]. If the number of tables is large enough, the CPU time spent by the query execution engine on
Fig. 1 – Execution times for Methods A–C are compared on three different databases—Oracle 9i, SQL Server 2000 SP4 and SQL Server 2005 Beta 2. The x-axis represents the number of attributes for which the values are retrieved and loaded into an array; the y-axis represents the execution time in milliseconds (ms). (A) Execution times for Method A, using full outer joins, to retrieve the values for the desired attributes and load the data into an array. Oracle 9i times increased exponentially ($R^2 = 0.955$) with the number of attributes, and failed with a SQL error at 10 attributes. The times on SQL Server increased at a quadratic rate ($R^2 = 0.994$ for SQL Server 2000 SP4, $R^2 = 0.999$ for SQL Server 2005 Beta 2). (B) Execution times for Method B, using left outer joins, to retrieve the values for the desired attributes and load the data into an array. Execution times on SQL Server had an exponential growth for the first 11–12 attributes ($R^2 = 0.981$ for SQL Server 2000 SP4, $R^2 = 0.990$ for SQL Server 2005 Beta 2), followed by a linear increase for more than 13 attributes ($R^2 = 0.990$ for SQL Server 2000 SP4, $R^2 = 0.959$ for SQL Server 2005 Beta 2). The times on Oracle 9i increased linearly ($R^2 = 0.996$) and were much lower compared with the SQL Server times. (C) Execution times for Method C, using the in-memory hash table to pivot the values for the desired attributes and load the data into an array. In this method, only the values for the desired attributes were selected from the database. Execution times on Oracle 9i and SQL Server 2005 Beta 2 grew linearly ($R^2 = 0.946$ for Oracle 9i, $R^2 = 0.967$ for SQL Server 2005 Beta 2), with the values for the latter slightly lower in magnitude. For SQL Server 2000 SP4, a notable and increasing performance degradation was observed for about 9–20 attributes, after which the execution times became lower and grew more linearly at a smaller rate. This odd behavior, presumably due to SQL Server 2000's attempt to optimize the query, prompted us to bypass the SQL Server's optimization attempt and to investigate the alternative method where the values for all the attributes (for a desired trial) were retrieved from the database and only the desired values were then picked to be stored in the pivoted array. As expected, the times stayed almost constant—since most of the time was spent executing the query and then iterating through all the retrieved values to see if they need to be stored in the array or not.
determining the best way to perform the join can be consider-
ably more than the time that the engine might take in actually
executing the join using a naïve strategy, such as joining the
tables in the order encountered in the SQL statement.

Modern DBMSs achieve their impressive performance
through a combination of heuristics (e.g., the presence or
absence of indexes) and the use of stored database statis-
tics, such as table sizes and data distributions. This infor-
mation lets them select query execution plans that may not
be absolutely optimal, but are reasonably close to optimal,
and which can be determined in polynomial or even linear
time. The strategies are understandably vendor-specific: Since
query performance is an area where vendors compete vig-
ously, these strategies are likely to change between DBMS
versions. Further, the amount of intellectual effort that ven-
dors decide to expend in devising heuristics to accommodate
relatively uncommon situations efficiently is also likely to
vary. Finally, query optimization is a task complex enough to
require a team of programmers, with specific sub-tasks being
delegated to individual team members, some of whom may
be more skilled than others. In any event, different vendor
engines and versions will generate different plans for the same
situation.

4.2. Full outer joins versus left outer joins

Oracle degrades exponentially for full outer joins, while SQL
Server does not. Full outer joins have been introduced rela-
tively recently into DBMSs. Being needed in relatively uncom-
mon situations (the vast majority of “business” databases
do not utilize EAV design), it is possible that Oracle’s imple-
menters invested minimal effort in optimizing their perfor-
mance (to the extent of crashing the engine when the num-
ber of joins exceed 10 attributes), while SQL Server’s implementers
did not.

With SQL Server, full outer joins perform better than left
joins. The performance of left joins has been improved sig-
ificantly in SQL Server 2005 (which is desirable for these
relatively common operations) but it still falls slightly short
of full outer joins. Oracle 9i outperforms both these versions
by a wide margin for left joins. One explanation of these num-
bers is the relative effort and skill that each vendor brought to
bear on the optimization of these operations.

The older version of SQL Server performs full outer joins
more efficiently than the newer version. Most algorithms
incorporate trade-offs, and it is possible that the revised outer
join algorithm for outer joins performs much better for the
common (left join) situation, while performing worse in the
much less common (full outer join) situation. Oracle 9i’s per-
formance characteristics, which show superrelative optimiza-
tion of left joins while showing pathological behavior for full
outer joins, are possibly an extreme example of this trade-off.

4.3. In-memory joins

For all the DBMSs tested, in-memory joins give the best overall
performance, as indicated in Fig. 2, especially when the num-
ber of attributes is large. In the 42 attribute-scenario, Methods
C and C’ were 1.76 and 1.96 times faster than the next best
method (Method B) for Oracle 9i. For SQL Server 2000 SP4, for
42 attributes, Method C was four times faster than the next best performing method, Method A. For SQL Server 2005 Beta 2, Method C was 40 times faster than the next best performing method, Method A.

In the in-memory join, the SQL that is sent to the database in step 6A above, is very simple so that the DBMSs do not need to spend any CPU time trying to optimize it, and returns a large amount of data. The algorithm is essentially limited by the rate at which the Java application can deal with the rows from the resultant dataset. By shifting the work from the database to an application server, this approach scales more readily to “Web farm” parallelism [12], where multiple application servers access a shared database. This approach, employed in large-scale e-commerce scenarios, is more readily implemented than database-server parallelism. For the latter, increasing the number of CPUs does not help significantly unless the data is also partitioned across multiple independent disks, because database operations tend to be I/O bound rather than CPU bound.

This algorithm is not limited by the number of attributes, but, in the simple version described above, assumes availability of sufficient RAM. This assumption is generally reasonable on present-day commodity hardware with 2 GB-plus of RAM, but not always so. A more complex but better-scaling version of the algorithm requires a change in steps 3 and 4 above.

Modified step 3: Compute the worst-case RAM required per row, Mworst, for the 2-D array (based on the total number of attributes and their individual data types). Determine the total RAM available to the program (Mmax). Allocate the 2-D array with number of rows, Nrows × Mworst/Mmax and number of columns = number of attributes.

Modified step 4: Replace the single query of step 4 with a series of queries by traversing the ordered Entity information. Each query retrieves a horizontal “slice” of the EAV data, such that the number of distinct entities will not exceed Nrows. (To do this most directly, determine, for each query, the range of ordered patient IDs in the Entity data that does not exceed Nrows.) The filter in each query then takes the form “Study ID = x and CRF_ID = y and patient ID between ‘aaa’ and ‘bbb’ ” (where x and y are already known, and the values ‘aaa’ and ‘bbb’ are determined each time). In each iteration, write out the filled array to disk, re-initialize it, and increment the range of patients, until data for all patients is fetched.

5. Conclusions

While several algorithms can be employed for pivoting EAV data, each approach must be carefully tested on individual vendor DBMS implementations, and may need to be periodically re-evaluated as vendors upgrade their DBMS versions.

The in-memory join, while algorithmically most complex, is also the most efficient. Crucial Clinical Trials [2000] 440-461, because the SQL that it uses combines a limited number of tables, and needs only elementary optimization from the DBMS perspective.

Acknowledgments

This work was supported in part by NIH Grants U01 CA78266, K23 RR16043 and institutional funds from Yale University School of Medicine. The authors would like to thank Perry Miller for comments that improved this manuscript.

REFERENCES