IDEA GROUP PUBLISHING

701 E. Chocolate Avenue, Hershey PA 17033-1117, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com

ITP4209

# An Experimental Evaluation of Dynamic Electronic Catalog Models in Relational Database Systems

Kiryoong Kim, Dongkyu Kim, Jeuk Kim, Ighoon Lee and Sang-goo Lee
School of Computer Science and Engineering, Seoul National University, Kwanak-gu Shillim-dong, Seoul, Korea
Tel: +82-2-883-9235, Fax: +82-2-872-1858, {kosk, dkkim, ihlee, sglee}@europa.snu.ac.kr

Sang-uk Park and Sang-goo Lee
CorelogiX, Inc., Seoul National University, Seoul, Korea, {spark, sglee}@corelogix.co.kr



## ABSTRACT

*Electronic catalogs are information about products and services in the electronic commerce environment and require diverse and flexible schemas. Although relational database systems seem to be an obvious choice for their storage, traditional designs of relational schemas cannot support electronic catalogs in the most effective ways. Therefore, new models for managing diverse and flexible schemas in relational databases are required for such systems. This paper proposes several models for electronic catalogs using relational tables, and presents an experimental evaluation of their efficiency. The results of this study can be put to practical use and is, in fact, being applied in the design of a commercial software product.*

## INTRODUCTION

Electronic catalogs are electronic representations of information about the products and services of an organization [1]. A typical catalog containing 100,000 products may contain thousands of different schemas [3]. For example, a "TV" may have a 'voltage' attribute, while a "pen" may not. Consequently, one of the biggest problems in electronic catalogs is diversity of schemas for products.

For this reason, XML seems to be a suitable alternative that meets the requirements of electronic catalogs. However, it is inefficient to store large number of catalog data as XML documents. Relational database systems are still the most practical choice for managing business data. [6]. However, traditional relational databases are not geared to managing several schemas at once or managing a universal table with many nulls [3].

Consequently, careful application level design is required. For example, a frequently used model represents catalogs in the form of <id, attribute name, attribute value>. This scheme requires multiple self-joins to retrieve information about a product and, thus, is inefficient in managing a large quantity of product data. We should therefore consider other models that efficiently support the view of thousands of tables from an application perspective, yet manage the database from a finite set of verticalized tables [3].

In this paper, we suggest several models for electronic catalogs using relational databases, and verify the efficiency of each one through experiments. The goal of this paper is to find the most efficient model by experiment, and to utilize it in a practical electronic catalog system.

## TERMINOLOGY

There may be some confusion on the terminology, as the standards for electronic catalogs are not yet fully established. In this section, we introduce some terms used in this paper.

*Product group*: A group of products that are treated as the 'same type' of products [5]. We assume that products in a specific product group share the same set of attributes, and that a product belongs to only one product group.

*Common attributes:* The attributes that is common to all the product groups [5]. We represent the set of common attributes as $\boldsymbol{C}$;

$$\boldsymbol{C} = \{c_1, c_2, \ldots, c_n\} ,$$

where $c_i$ is an attribute required for every product group in the database, such as product group id., and product id.

*Dependent attribute:* The attributes those are specific to a product group. Let $\boldsymbol{D}_i$ be a set of dependent attributes for the product group $i$, then

$$\boldsymbol{D}_i = \{d_{i1}, d_{i2}, \ldots, d_{jm}\} ,$$

where $d_{ij}$ is an attribute for product group $i$ and not in $\boldsymbol{C}$, such as 'voltage' of refrigerators, 'diameter' of bearings, or 'frequency' of celluar phones.

Now, we can express a schema for product group $i$ as $\boldsymbol{P}_i$.

$$\boldsymbol{P}_i = \boldsymbol{C} *\bullet \boldsymbol{D}_i$$

*Product group id:* Attribute that identifies a product group such as the classification code of products.

*Product id:* Attribute that identifies each individual product in the database such as the SKU#[1] or EAN/UCC[2] code. Each product has a unique product id.

## PRIMITIVE MODELS

In this section, we present several data models for constructing electronic catalogs in a relational database. The name of each model describes its representative table.

### Universal Table (UT)

The UT model stores product data in a table with a schema consisting of the union of common attributes and dependent attributes of all product groups. This table, $\boldsymbol{T}_{\text{UNIVERSAL}}$ can be expressed follows:

$$\boldsymbol{T}_{\text{UNIVERSAL}} = \boldsymbol{C} *\bullet(*"\boldsymbol{D}_i)$$

Each tuple in this table represents one product. This model requires another table to keep the meta-data identifying the attributes relevant for each product group. The schema for table $\boldsymbol{T}_{\text{UNIVERSAL\_META}}$ is defined as follows:

$$\boldsymbol{T}_{\text{UNIVERSAL\_META}} = (\textit{product group id, attribute name})$$

Underlined attributes denotes key attributes. Each instance of *attribute name* should be identical with the name of each attribute in $\boldsymbol{T}_{\text{UNIVERSAL}}$. To introduce a new product group into the catalog, we should add new attributes to $\boldsymbol{T}_{\text{UNIVERSAL}}$ and insert corresponding meta-data for the product group into $\boldsymbol{T}_{\text{UNIVERSAL\_META}}$.

Now, we define the UT model as $\boldsymbol{M}_{\text{UT}}$:

$$\boldsymbol{M}_{\text{UT}} = (\boldsymbol{T}_{\text{UNIVERSAL}}, \boldsymbol{T}_{\text{UNIVERSAL\_META}})$$

The merit of this model is its simplicity. Not many tables are created. In addition, we can use the integrity constraints provided by the DBMS since the model uses the relational schema directly. On the other hand, there may be too many nulls in the instance of $\boldsymbol{T}_{\text{UNIVERSAL}}$, because each tuple uses only some of the attributes in the table. How-

This paper appears in *Issues and Trends of Information Technology Management in Contemporary Organizations,* the proceedings of the Information Resources Management Association International Conference. 

ever, because of its simplicity, UT is currently a popular model in actual electronic commerce applications [5].

### Name-Value Pair Table (PT)

The PT model stores product data in a table, $T_{PT}$, which is defined as follows:

$T_{PT}$ = (*product id, product group id, attribute name, attribute value*)

Just like $T_{UNIVERSAL\_META}$ in $M_{UT,}$ each instance of *attribute name* should be identical with the name of each attribute in $T_{UNIVERSAL.}$ In other words, it expresses name of each attribute belonging to each product group. As each attribute belongs to a specific product group, *product group id* is needed to express it. We must keep meta-data about the attributes each product group uses. Let $T_{NAME\text{-}VALUE\_PAIR\_META}$ be a table for these meta-data:

$T_{PT\_META}$ = (*product group id, attribute name,* $c_1$, $c_2$, ..., $c_n$)

$c_k$ is a constraint for each attribute; for example, its data type, whether it allows null, or whether it is a unique value.

Now, we can define $M_{PT}$ as follows:

$$M_{PT} = (T_{PT}, T_{PT\_META})$$

This is the most flexible model currently used in relational databases, as new attributes can be added without changing the schema of the table. To introduce a new product group, we simply insert tuples describing meta-data into $T_{PT\_META}$. For this reason, this model is also popular.

On the other hand, this model cannot use the integrity constraints provided by relational database systems. For example, we should translate all values into character strings or some other common data types, and check each type of data by managing the meta-data. Besides, all the data must be stored in fixed-length records.

Another drawback is that this model uses too many tuples, because it requires one tuple for each attribute, not each product.

## HYBRID MODELS

The primitive models described in Section 3 deals with all the attributes in the same way. However, it might be more efficient to handle common attributes and dependent attributes differently, because the former are fixed for all product groups, while the latter vary with the product groups. In this section, we present hybrid models, combining the primitive models.

Common attributes do not vary with product groups, so it is reasonable to manage common attributes using a single table, as done in $T_{UT}$. Thus, we define a schema $T_{COMMON}$ that is a projection of common attributes from $T_{UT.}$

$$T_{COMMON} = \text{“}_C(T_{UT})$$

### HYBRID_1

HYBRID_1 combines UT and PT. It is composed of three tables.

$$M_{HYBRID\_1} = (T_{COMMON}, T_{PT}, T_{PT\_META})$$

$T_{COMMON}$ contains common attributes of all the products in the same way as UT, whereas $T_{PT}$ and $T_{PT\_META}$ store dependent attributes in the same way as PT. In contrast with PT, instances of $T_{PT}$ and $T_{PT\_META}$ in this model do not contain tuples for common attributes because they are included in instances of $T_{COMMON}$.

### HYBRID_2

The main problem with PT and HYBRID_1 is that they require multiple tuples for a product, as they use one tuple for one attribute. To solve this problem, we introduce $T_{OPTION}$ as follows:

$T_{OPTION}$ = (*product id, product group id, optional field*$_1$, *optional field*$_2$, ..., *optional field*$_n$)

Each product group can use *optional fields* in $T_{OPTION}$ for its attributes. If the number of *optional fields* are at least the maximum of the number of elements in $P_i$, only one tuple would be sufficient for one product. To use this table, we require another table that manages its meta-data:

$T_{OPTION\_META}$ = (*product group id, optional field #, attribute name,* $c_1$, $c_2$, ..., $c_n$) ,

where $c_k$ is a constraint for the attribute.

$T_{OPTION\_META}$ is similar to $T_{PT\_META}$ except that it contains an attribute that indicates a specific optional field in $T_{OPTION}$. Combining (joining) $T_{COMMON}$ and $T_{OPTION}$, we define $T_{COMMON\_OPTION}$ as a table containing the common attributes and optional fields ($T_{COMMON}$ ·%Á% $T_{OPTION}$).

HYBRID_2 is a model using these tables.

$$M_{HYBRID\_2} = (T_{COMMON\_OPTION}, T_{OPTION\_META})$$

Values of dependent attributes are stored in *option field*s of $T_{COMMON\_OPTION}$, with $T_{OPTION\_META}$ storing their meta-data. In this case, the number of *option field*s must be large enough so that it is at least the maximum of the number of elements in $D_i$, for all i.

### HYBRID_3

HYBRID_3 is composed of three tables.

$$M_{HYBRID\_3} = (T_{COMMON}, T_{OPTION}, T_{OPTION\_META})$$

HYBRID_3 is similar to HYBRID_2 except that it separates $T_{OPTION}$ from $T_{COMMON,}$ so we can expect better performance for queries with conditions only on common attributes or only on dependent attributes, but worse performance is expected for queries with conditions on both common attributes and dependent attributes because of join operations.

## ANTICIPATED QUERIES FOR ELECTRONIC CATALOGS

When we manipulate electronic catalogs, our requirements are expressed as relational queries. There are several classes of these queries. We transformed these queries into relational queries suited to each model and used them as testing queries.

### Queries That Retrieve Desired Data From the Electronic Catalogs

These queries are most frequently used because the main purpose of an electronic catalog is to provide product information to buyers.

**Query group 1.1**: Retrieval of key with exact conditions for common attributes

**Query group 1.2**: Retrieval of key with exact conditions for dependent attributes

**Query group 1.3**: Retrieval of key with exact conditions for common and dependent attributes

**Query group 2.1**: Retrieval of key with range conditions for common attributes

**Query group 2.2**: Retrieval of key with range conditions for dependent attributes

**Query group 2.3**: Retrieval of key with range conditions for common and dependent attributes

**Query group 2.4**: Retrieval of key by pattern matching for common and dependent attributes

**Query group 3**: Retrieval of an entire set of product data

### Queries That Manipulate Data in the Electronic Catalogs

These queries are necessary to insert, delete, or update product data in the electronic catalogs.

**Query group 4.1**: Insert new product data

**Query group 4.2**: Update product data

**Query group 4.3**: Delete product data

### Queries That Manipulate Schemas of the Electronic Catalogs

Electronic catalogs change over time. For example, a new product group might be introduced, or new attributes might be required for an existing product group. These queries allow changes to schemas for product groups in electronic catalogs.

**Query group 5.1**: Add a new product group

**Query group 5.2**: Modify product group information

**Query group 5.3**: Drop a product group

# EXPERIMENT

### Environment

Experiments were null on a Solaris Ultra Sparc II machine with 332 MHz CPU and 512 MB RAM.

The tests were carried on two different DBMSs because some factors associated with a particular DBMS could affect the results. The DBMSs were Oracle 8i and MySQL version 3.22.20a

We took experimental data from electronic catalogs provided by the Public Procurement Service (PPS)[3] of the Korea, EAN Korea[4], and LOTTE.com Inc.[5] The original catalogs were in the form of universal tables with different schemas. The combined catalog contained data on 51,766 individual products in 100 product groups. The product groups shared 18 common attributes, and the number of dependent attributes ranged from 2 to 15. Product groups contained from 4 to 986 products.

All product data were transformed into the tables presented in Sections 3 and 4.

### Module for Measuring Time

We implemented a measuring module with the Java programming language and JDBC. This module reads queries in script files written in a specifically defined format we defined, and measures the time to process them.

The script file consists of groups of query sections. Each section is composed of queries required to complete one transaction, and each group corresponds to the query group described in Section 5. The module process each group of query sections and takes the average of the times to process each of them. Each script file was executed 10 times and the results were recorded.

### Index

Indexes increase the speed of retrieval, but decrease the speed of inserting or updating data. In an e-catalog system, we usually use queries to retrieve data from tables more frequently than queries to insert or update data. Moreover, the models we mentioned above manage data in one or two tables, so they would have too many tuples to process without indexes.

We experimented with indexes with each model.

*UT*: A table on UT has so many fields that it is not easy to decide which fields should be indexed. In the experiments, we used an index on *product group id*, as it was the field most frequently referenced.

*PT*: In PT, queries are usually associated with a *product group id*, and there are only a few fields in a table, so we created indexes on all the fields.

*HYBRID_1*: HYBRID_1 has features of both UT and PT, so we applied the methods of both models.

*HYBRID_2:* We created an index on *product group id* and on some fields that were frequently referenced, such as product name, price, and manufacturer.

*HYBRID_3:* We created two different groups of indexes for HYBRID_3. One was an index on *product group id* and some fields that were frequently referenced, and the other was a group of indexes on all the *option field*s and half of the common attribute fields frequently referenced.

# RESULTS

### Oracle 8i

[Figure 1] shows the results of experiments using Oracle 8i in milliseconds. A highlighted cell indicates the best result in each row. As a result of caching, values of first run results were too high for all cases and increased the variance of the results. So we excluded them.

Performance of UT was not inferior to other models. The width of a table had little detrimental effect upon performance in this case.

Model PT performed worst on average. In the experiment, instances of $\boldsymbol{T}_{PT}$ in $\boldsymbol{M}_{PT}$ included 1,354,493 tuples, and it required too many self-joins because each attribute was expressed as one tuple. While PT showed relatively good performance on **query group 3**, as self-joins were not required in those queries, it seemed to be an inefficient model overall. For the same reason, HYBRID_1 produced to be an inefficient model among the hybrid models, because it stored dependent attributes in $\boldsymbol{T}_{PT}$.

HYBRID_3 outperformed the other models on average. However, it was very inefficient for queries accessing both common attributes and dependent attributes because it manages them in separate tables. On the other hand, HYBRID_2 was uniformly efficient for most queries.

*Figure 1: results of experiments using Oracle 8i in milliseconds*

| Query | PT | UT | Hybrid-1 | Hybrid-2 | Hybrid-3 |
|---|---|---|---|---|---|
| 1.1 | 5673.04 | 379.42 | 290.93 | 696.31 | **363.47** |
| 1.2 | 2878.40 | 398.27 | 1053.71 | 368.60 | **246.69** |
| 1.3 | 3465.29 | 358.18 | 882.38 | 295.62 | **272.62** |
| 2.1 | 3156.44 | 6.85 | 6.74 | 7.26 | **6.41** |
| 2.2 | 2527.04 | 277.11 | 809.29 | 200.67 | **129.96** |
| 2.3 | 7971.07 | 290.48 | 1547.74 | **196.07** | 1323.26 |
| 2.4 | 22234.48 | **122.37** | 2824.89 | 238.11 | 338.78 |
| 3 | 2850.89 | **15.70** | 876.41 | 372.81 | 36834.59 |
| 4.1 | 57.89 | 3.67 | 14.19 | **2.37** | 4.33 |
| 4.2 | 7008.78 | **260.22** | 1021.07 | 294.63 | 411.44 |
| 4.3 | 148815.70 | **17523.26** | 64446.33 | 10521.81 | 22216.67 |
| 5.1 | 50.11 | 8.56 | 36.78 | 8.67 | **8.33** |
| 5.2 | 3911.56 | 2.67 | 1.67 | 1.78 | **1.33** |
| 5.3 | 3016.67 | 1845.89 | 2292.33 | 1320.89 | **433.33** |

### Oracle 8i with Indexes

[Figure 2] shows the results with indexes, using Oracle.

PT also showed the worst performance on average, but showed the best performance on **query group 3**, which did not require self-joins. Results for the other models were similar to those without indexes except that processing time generally decreased.

We performed another experiment on HYBRID_3 with additional indexes, to verify how much this model could be improved by indexes. These were created on all the fields of $\boldsymbol{T}_{OPTION}$ and each frequently referred field of $\boldsymbol{T}_{COMMON}$ in $\boldsymbol{M}_{HYBRID\_3}$. A column, **HYBRID_3_ext,** in [Figure 2] is the result for this experiment. As shown in the figure, indexes greatly improved the performance.

It was difficult to create indexes on the table in UT because of the large number of attributes.

### MySQL

[Figure 3] shows experimental results for UT, HYBRID_2 and HYBRID_3 on MySQL. We didn't test PT and HYBRID_1 because they showed significantly worse performance in the previous experiment on the Oracle. We also didn't test with indexes on MySQL because we didn't have enough time for the experiment, but we believe that indexes cannot change the order of results on each model, judging from the results on the Oracle.

Because MySQL is far simpler DBMS, the results on MySQL had somewhat different features in comparison with those on Oracle. UT was far worse than the others on almost all queries, and HYBRID_3 showed relatively good performance on **query group 3**. These results indicate that the efficiency of each model could vary with DBMS, because of different storage managements, query processing, transaction managements, and so on. However, hybrid models showed more regular efficiency than UT.

*Figure 2: Results of experiments with indexes, using Oracle*

| Query | PT | UT | Hybrid-1 | Hybrid-2 | Hybrid-3 | Hybrid-3-ext |
|---|---|---|---|---|---|---|
| **1.1** | 4332.02 | 301.51 | 299.18 | 621.62 | 389.20 | **8.50** |
| **1.2** | 1921.96 | 255.71 | 34.38 | 365.51 | 303.60 | **5.88** |
| **1.3** | 2939.60 | 293.60 | 17.84 | 234.20 | 14.31 | **13.18** |
| **2.1** | 2995.19 | 7.41 | 6.48 | 7.15 | 7.22 | **4.90** |
| **2.2** | 2468.89 | 244.22 | **5.15** | 244.74 | 158.89 | 10.57 |
| **2.3** | 6496.85 | 279.30 | 60.30 | 196.56 | **5.48** | 8.07 |
| **2.4** | 17116.26 | 116.81 | 7.07 | 44.78 | 6.19 | **5.93** |
| **3** | **4.89** | 16.07 | 1876.52 | 349.41 | 234.70 | 15.73 |
| **4.1** | 45.11 | **2.52** | 28.67 | 5.41 | 14.89 | 34.90 |
| **4.2** | 8708.11 | 253.26 | **151.89** | 374.44 | 253.11 | 623.43 |
| **4.3** | 149544.10 | **22214.37** | 151330.40 | 28107.93 | 30706.96 | 31239.94 |
| **5.1** | 162.56 | **9.89** | 67.00 | 16.33 | 28.11 | 171.23 |
| **5.2** | 9200.33 | 1.89 | 21.56 | **1.44** | 29.67 | 8703.41 |
| **5.3** | 7504.22 | 1410.00 | **119.44** | 1155.33 | 633.89 | 3219.81 |

*Figure 3: Shows experimental results*

| Query | UT | HYBRID-2 | HYBRID-3 |
|---|---|---|---|
| **1.1** | 2066.22 | 861.01 | **592.38** |
| **1.2** | 2068.33 | 863.69 | **551.47** |
| **1.3** | 2052.96 | **858.98** | 1001.98 |
| **2.1** | 4349.96 | 3539.67 | **3193.26** |
| **2.2** | 2136.93 | 992.19 | **691.04** |
| **2.3** | 2113.74 | **962.67** | 20251.15 |
| **2.4** | 2729.19 | 1701.74 | **1407.33** |
| **3** | 2081.00 | **843.43** | 1132.82 |
| **4.1** | **1.70** | 1.78 | 2.52 |
| **4.2** | 3874.59 | **2821.15** | 4342.56 |
| **5.1** | **5.00** | 5.89 | 5.33 |
| **5.2** | 3.00 | **1.11** | 1.13 |

### Space Analysis

[Figure 4] shows total amounts of storage used to store catalogs in each model on Oracle 8i and MySQL. Each value was measured in megabytes. We presumed that UT would have many null values and incur a good amount of wasted space. However, the result shows that UT did not overuse storage. On the contrary, PT had the worst space complexity, because each attribute name was stored in each tuple belonging to $\boldsymbol{T}_{PT}$. HYBIRD_3_ext, which we mentioned in section **7.2**, required additional storage for indexes, more than for the tables themselves.

## CONCLUSION

It is believed that PT is a proper model for managing diverse and flexible catalogs in relational databases. However, our experimental results show that this model is inefficient to be applied in practical cases, despite its flexibility. Moreover, applications would have too much work to do to keep integrity constraints, as DBMSs cannot guarantee them. Despite its popularity, we conclude that PT is not a suitable model for current relational database catalog systems.

UT, which is another popular model, also cannot be regarded as a good model because of the fact that as the number of attributes increases, it becomes harder to manage. It may also show bad performance in the DBMSs that deal poorly with nulls. Meanwhile, UT requires the least storage of all the models in the experiments. This is believed to be the result of efficient null value management in current relational database systems.

We made an attempt to combine the good features of the primitive models by using hybrid models. In the results, HYBRID_2 and HYBRID_3 were good in both performance and space complexity, so we can expect them to be applied to practical electronic commerce systems. Moreover, these models can improve their performance by using indexes since storage is currently a far cheaper resource than time.

*Figure 4: Total amounts of storage used to store each model*

| | PT | UT | HYBRID-1 | HYBRID-2 | HYBRID-3 | HYPER-3-ext |
|---|---|---|---|---|---|---|
| **Tables on Oracle** | 126.67 | 39.85 | 62.59 | **35.51** | 50.14 | 50.14 |
| **Indexes on Oracle** | 112.47 | **2.28** | 57.66 | 11.23 | 12.6 | 51.02 |
| **Tables on MySQL** | 72.29 | 24.68 | 41.20 | **23.53** | 33.28 | - |

## ACKNOWLEDGEMENTS

We would like to thank Kyungsuk Kim and Hyunyoung Song for their helpful suggestions and efforts in the implementation. We are also grateful to PPS, LOTTE.com Inc. and EAN Korea for offering their catalogs data for our experiments. The RIACT at Seoul National University provides research facilities for this study. This work was supported by Brain Korea 21 Project in 2001.

## ENDNOTES

1 Stock Keeping Unit number

2 Universal Code Council / European Article Numbering

3 An administrative agency of the Republic of Korea which is responsible for procuring commodities and related services from domestic and foreign sources, for procuring works for major government projects, for stockpile management, and for government property management.

4 A member organization of EAN International which is taking a leading role in establishing a global multi-industry system of identification and communication for products.

5 One of leading B2C companies in Korea which has much experience in managing electronic catalogs for retail goods.

## REFERENCES

1. Arie Segev, Dadong Wan and Caroline Beam, Electronic catalogs: a technology overview and survey results, *Proceedings of the 4th International conference on information and knowledge management* Baltimore, *Maryland, USA*, Nov 29-Dec 2, 1995, pp. 11-18.
2. Arie Segev, Dadong Wan, Carrie Beam, Designing Electronic Catalogs for Business Value: Results from the CommerceNet Pilot, *CITM Working Paper WP-95-1005, Haas School of Business, University of California, Berkeley*, 1995.
3. Anant Jhingran, Moving Up the Food Chain — Supporting E-Commerce Applications on Databases, *Technical Report, IBM Almaden Research Center*, 2000.
4. Michael Stonebraker, Joseph M. Hellerstein, Content Integration for E-Business, White Paper, Cohera Corp., 2000.
5. Jihye Jung, Dongkyu Kim, Sang-goo Lee, Chisu Wu and Kapsoo Kim, "EE-Cat: Extended Electronic Catalog for Dynamic and Flexible Electronic Commerce", *Proceedings of the Information Resources Management Association International Conference Anchorage, Alaska, USA*, May 21-24, 2000, pp. 303-307.
6. Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt and Jeffrey F. Naughton, Relational Databases for Querying XML Documents: Limitations and Opportunities., *Proceedings of 25th International Conference on Very Large Data Bases, Edinburgh, Scotland,*, 1999, pp. 302-314.