참조무결성을 이용한 데이타웨어하우스의 조인 실체뷰 관리

(Maintaining Join Materialized Views For Data Warehouses using Referential Integrity)

이 우 기 †

(Wookey Lee)

요 약 실체뷰는 대량의 데이타웨어하우스에서 질의처리를 효과적으로 수행하기위한 대안으로서, 그 핵심은 각 데이타 원천에서의 데이타변화에 대응한 복합적인 뷰의 효과적인 관리 문제이다. 본 연구에서는 우선 실체뷰 관리에 관한 기존의 연구들을 일별함에 있어서 즉, 갱신의 주체문제, 갱신객체, 및 갱신시간 문제의 세가지 관점에서 본 연구의 위치를 결정한 다음, 대수적 접근법으로 복합뷰 갱신문제가 복잡해지는 원인을 규명하였다. 그 해법으로서 참조무결성을 활용한 복합 조인뷰의 갱신 알고리듬을 제안하면서, 여러 가지 참조무결성 제약조건과 트랜잭션과 관련된 자체갱신적 새로운 해법을 제시했다.

Abstract View materialization has extensively been researched as one of the strongest alternatives to cope with processing huge data warehouse information. In this paper we deal with the maintenance of complex join materialized view in response effectively and efficiently to the changes of data sources. A formal approach is introduced that figures out what makes the problem difficult. Investigating a solution scheme, referential integrities are analyzed in terms of inserts and deletes both in the referencing relation and in the relevant referenced relation. Locks on the current database should be minimized, so that it can be a self-maintainable in updating the data warehouse system.

1. Introduction

Data warehouse is considered as a collection of *materialized view* over one or more operational databases, for which proper maintenance is critical [6, 10]. Maintaining *materialized view* with *join operation* is complex for the sources are decoupled, so that traditional approaches may exhibit anomalies [4, 18]. When a view can be maintained at the warehouse without accessing base relation, we say the views are *self-maintainable* and it has been identified as desirable [7, 16, 17].

Three ways can be classified to support the *materialized view maintenance* such as the *subject*, the *object*, and the *update time* respectively. The *subject* represents by which the initiative is drived: *pulled* by the view or *pushed* by the base relation [5]. The *object* means the data that should be sent to view: *base relation* per se [2, 19], *auxiliary relation* [3, 11,16, 22], and delta portion called *differential file* [12, 13, 20]. The *update time* can be classified two types: *immediate* [1, 19] and *incremental* [6, 11, 16, 20, 23], and the latter one in turn consists of *deferred* [6, 19] and *periodical* [21]. Various combinations can be generated among them.

Once a base relation is used as an object, then the deferred scheme with pushed strategy is inevitably selected. Because there is actually no room for

· This work was supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center (AITrc).

† 종신회원 : 성결대학교 컴퓨터학부 교수
wook@hana.sungkyul.edu

논문접수 : 1999년 3월 8일
심사완료 : 2000년 1월 18일

updating immediate time scale, it is liable to generate too excessive costs each time. Thus we call the method with *pull-type*, *deferred* time span, and using base relation as a *baseline* method. A popular trend is the combination as the *push* or *pull* (say, *all*) type with *incremental* method using *auxiliary relations*; we call it *all-inc-aux* method. Incremental approaches are frequently accepted in the data warehouse environment, but the main weakness of it is the lack of flexibility to cope with ad hoc queries into the data warehouse. Our approach can be said as the *push* or *pull* (say, *all*) type with all kinds of update time scale (say, *all*) using object having a *differential file* combination, thus called *all-all-df* method (*'all'* means that all the domain span can be supported).

Updating the data warehouse views, locks on the current database should be minimized which means a *self-maintainability*. If *referential integrity* constraints are present then it is not necessary to replicate the base relations in their entirety at the warehouse in order to get the *self-maintainability* of a view. Extensive researches have been devoted in investigating the referential integrity [1, 9, 14, 15]. Until now there are, if any, few relevant studies to maintain materialized views with referential integrity constraints [8, 13, 14, 19].

2. View definitions and notations

A data warehouse view formally be expressed as follows (A modification, of course, assumed to be a delete and an insert in series with the same time-stamp (*SYSDATE*));

new view(v') = *old view*(v)+*inserts*(I)−*deletes*(D), (1)

where the old view and the new view are respectively

$v = \Pi_A \sigma_C (R_1 \times R_2 \times \cdots \times R_n)$

$v' = \Pi_A \sigma_C (R_1' \times R_2' \times \cdots \times R_n')$

and the inserts(I) and the deletes(D) are

$I = \Pi_A \sigma_C (I_1 \times R_2 \times \cdots \times R_n + R_1 \times I_2 \times \cdots \times R_n + \cdots + R_1 \times R_2 \times \cdots \times I_n)$

$D = \Pi_A \sigma_C (D_1 \times R_2 \times \cdots \times R_n + R_1 \times D_2 \times \cdots \times R_n + \cdots + R_1 \times R_2 \times \cdots \times D_n)$ respectively. Specifically, for R_i is a base relation and R'_i is an after image of the relation, Π_A is a projection on an attribute(A), and σ_C is a selection on a condition(C), and Ψ is an argument set of relations $I \subseteq \Psi$ {1, 2, 3, ⋯, n}. Then the new base relation R'_i can be expressed as the old base relation R_i and its changed portion dR_i:

$R'_i = R_i + dR_i = R_i + I_i - D_i$ for $i = \Psi$ (2)

where dR_i is consisted of a set of inserts(I_i) and deletes(D_i) in the base relation(R_i).

3. Motivating Example

Consider a data warehouse for a company having product and group. Suppose this kind of data warehouse is collecting data from 3 base relations of which schema are suggested as follows (the primary key in each relation is underlined):

- *P* (*<u>pcode</u>, price, <u>go</u>*); This is a product relation having such attributes as product code, price, and group number as a foreign key that has the condition such as gno.P ⊆ gno.G. For example, the product relation P currently has five tuples as {(s100, 500, 2), (m5, 600, 3), (m6, 900, 3), (s150, 1000, 2), (v111, 3000, 1)}.
- *G* (*<u>gno</u>, gname*); this relation called a group relation, contains the group number as a primary key and group name. For example, the product relation G currently has four tuples as {(1, computer), (2, tv), (3, video), (4, audio)}.

The view is defined as follows:

$V = \Pi_{\text{p.pcode, P.price, G.gon, G.gname}} \sigma_{\text{P gno=G.gono}} (P \times G)$. For example, the view then has five tuples as {(s100, 500, 2, tv), (m5, 600, 3, video), (m6, 900, 3, video), (s150, 1000, 2, tv), (v111, 3000, 1, computer)}.

4. Differential Files

In this paper we want to use *differential file* (*DF*). That can be derived from the active log of a base relation [13, 21]. The schema of *DF* of base relation (R_i) is defined as $dR_i(A_k$, *operation*), where A_k is relevant attribute set of R_i and *operation* indicates the type of operations applied to the tuple. It has one of the two operation types: *'insert'* or *'delete'*. Then each record of changes in a base table (R_i) is appended in the *DF* (say, dR_i) with respect to non-decreasing order of time-stamp. It is assumed to be located the same site of the base relation.

Example 1. In the previous example, an item *'eq1'* is inserted in *P*. It can be represents in *dP* as {(eq1, 1500, 4, insert)}. Then transactions that the price of 'm-5' is raised from 600 to 800 and a product *'S100'* is deleted are represented as {(m-5, 600, 3, delete), (m-5, 800, 3, insert)}, {(s100, 500, 2, delete)} respectively.

4.1 Referential integrity in maintaining data warehouse views

Without loss of generality, we can assume that a referencing relation (R_i) has a relationship with referenced relation(s) (say, R_j) such that $R_i.A_{FK} \subseteq R_j.A_K$. Where the $FK \subseteq i \subseteq \Psi$ is said to be a foreign key that is relevant to a key (K) of R_j. A tuple is changed (i.e., inserted, deleted, or updated) in a relation, a referential integrity (*RI*) constraint might be fired to check the relevance of the change. Then we define an insert MI_{ji} and a delete MD_{ji} as a *modified insert in* R_i *due to the change in* R_j and a *modified delete in* R_i due to the change in R_j respectively. Then I_i^0 and D_i^0 represent the net insert and the net delete in R_i respectively. Then we can extend the changed portion of the base relation as follows:

$$\Delta R_i = I_i - D_i = (I_i^0 + MI_{ji}) - (D_i^0 + MD_{ji}) \text{ for } i,\ j \subseteq \Psi. \quad (3)$$

Example 2. When a tuple is deleted in *G*, it may effect the tuples in *P*. In this case, the referential integrity option may be assumed *'ON UPDATE NULLIFY'*. Then MD_{GP} and MI_{GP} are in series as tuples of *dP*. If a tuple in G, say *gno*=1 is deleted, then it triggers a change (modification) in P as {(v11, 3000, 1, delete), (v11, 3000, Null, insert)}. If the option is *'ON UPDATE CASCADE'*, then the MD_{GP} will be{(v11, 3000, 1, delete)}.

4.2 Algebraic representation for the view

Then the join by the two relations are expressed as follows:

$$R_i'\times R_j' = (R_i + dR_i)\times(R_j + dR_j)$$
$$= (R_i + I_i^0 - D_i^0 + MI_{ji} - MD_{ji})\times(R_j + I_j - D_j)$$
$$= \{R_i\times R_j + R_i\times I_j\} + \{I_i^0\times R_j + I_i^0\times I_j + MI_{ji}\times(R_j + I_j)\}$$
$$-\{R_i\times D_j + I_i^0\times D_j - (R_j + I_j - D_j)\times(MD_{ji} - D_i^0) + MI_{ji}\times D_j\} \quad (4).$$

When a tuple is deleted (and even though it is relevant to the view), it is useless to refer to other tables for joining the deleted tuples. Thus the last term of the equation (4) can be expressed as follows.

$$R_i\times D_j + I_i^0\times D_j - (R_j + I_j - D_j)\times(MD_{ji} - D_i^0) + MI_{ji}\times D_j$$
$$= D_i\times R_j' + D_j\times R_i \quad (5)$$

Where $R_i \cap I_j = \varnothing$, which means that the insert in the referenced relation does not affect the (existing) referencing relation. For MI_{ji} is generated by the change of the referenced relation, which means;

$I_{ji} \cap R_j = \varnothing$, $I_{ji} \cap R_j \subseteq I_{ji} \cap I_j$. (6)

Then the equation (4) can be expressed as follows.

$$R_i'\times R_j' = \{R_i\times R_j\} + \{I_i^0\times R_{ji}\} + \{(I_i^0 + MI_{ji})\times I_j\}$$
$$-(D_i\times R_j' + D_j\times R_i'). \quad (7)$$

Then the new view will be

$$v' = \Pi_A\,\sigma_C[\{R_i\times R_j\} + \{I_i^0\times R_j\}$$
$$+\{(I_i^0 + MI_{ji})\times I_j\} - (D_i\times R_j' + D_j\times R_i')]$$
$$= \Pi_A\,\sigma_C[R_i\times R_j] + \Pi_A\,\sigma_C[I_i^0\times R_j] + \Pi_A\,\sigma_C[(I_i^0 + MI_{ji})\times I_j]$$
$$-\Pi_A\,\sigma_C[D_i\times R_j' + D_j\times R_i'] \text{ for } i, j \subseteq \Psi. \quad (8)$$

In the data warehouse environment, it is sufficient for the deleted tuples just to delete the tuples in the join view. For the old image is stored, so there is no need to refer another relation for join. Thus the last term of the equation (8) is represented as follows.

$$\Pi_A\,\sigma_C[D_i\times R_j' + D_j\times R_i'] = \Pi_A\,\sigma_C[D_i + D_j]. \quad (9)$$

Therefore by the equation (3) and the equation (9), the view is represented as follows:

$$\Pi_A\,\sigma_C R_i\times R_j + \Pi_A\,\sigma_C I_i^0\times R_j + \Pi_A\,\sigma_C I_i\times I_j - \Pi_A\,\sigma_C D_i\times D_j \quad (10)$$

The equation (10) can be explained as follows. The first term of the above result is the old view (v). The second term represents that by the net insert in the referencing relation the relevant *base* table should inevitably be searched and joined. The third term means that it is sufficient to use the inserted tuples in the differential file instead of the base table. The fourth term represents that the delete operation can be made, just by sending the deleted tuples to the views. In this paper, we emphasize that due to the second term of equation (10), the data warehouse views cannot help referring the current database.

5. Maintaining Data Warehouse Views

5.1 Additional join file and Notations

In this paper in order to update materialized views self-maintainable, a new file called an *additional join file* (*AF*) is introduced. The schema of the *AF* of table R_j is defined as the same as R_j (the referenced relation). Without loss of generality, the time-stamp that the tuple of *AF* is appended is assumed the same that the insert transaction occurs. There are four kinds of transactions as well as three representative *RI* cases in dealing with the *AF*: insertions and deletions in the referencing relation, those in the referenced relations as well as restrict, cascade, and nullify. Thus there exist all 12 sub-problems. Here we consider the transaction cases with respect to the *RI's*. The *AF* of R_t is represented as aR_t.

We denote t a tuple and by t[X] the subtuple of t corresponding to the attribute set X. Let's assume that an operation is represented as a braced format appended after the relation expression such that R_i\{insert\} or $R_i.A_r$\{insert\} are said the inserted attribute of R_i. Then t[$R_i.A_r$\{insert\}], for example, represents an inserted tuple of R_i. We define $\Uparrow$ as 'indicated by RI', such that $t[R_j \Uparrow R_i\{\text{insert}\}]$ represents a tuple of R_j indicated by an insert of R_i.

5.1.1 Insertion in the referencing relation

If there is an insert in the referencing relation, the trial to commit the insertion must require a *RI* check across the referenced relation in any case of *RI* conditions. Which means that the trial will be committed if only there exist a relevant tuple in the referenced relation. The *AF* is derived from the tuples of the referenced relation that the *RI* constraint indicates. It can be represented formally as follows:

$aR_j = t[R_j \Uparrow R_i\{\text{insert}\}]$

5.1.2 Deletion in the referencing relation and Insertion in the referenced relation

These tow operations may be done without considering the *RI* condition. But there are some detailed influences to the tuples of referencing relation as follows: Restrict: MI_{ji}=Ø; Cascade: $MI_{ji} \neq$ Ø; and Nullify: MI_{ji}=Ø.

5.1.3 Deletion in the referenced relation

If there is a delete in the referenced relation, the transaction must require a *RI* check across the referencing relation in all *RI* conditions. In case of restrict condition the transaction will be failed, if there exist any relevant tuple in the referencing relation. In cases of cascade and nullify, all the relevant tuples should be deleted. It can be represented formally as follows: Restrict: Not permitted, if $R_i.\ A_K = R_t.\ A_{FK}$; Cascade: $MD_{ji} \rightarrow dR_i$\{delete\}, and Nullify: $v.\ t[R_i.\ A_{FK}=$ Null] = delete.

Example 3. In order to insert a tuple (say, *top1*) into the *P* in the previous example, the *RI* constraint is activated the relevance. By the *RI* constraint, we can get the tuple (4, *audio*) from *G* and append the tuple to the *AF* of table *G* (say, *aG*). By the transaction in *P*, the tuple with (2, tv) is appended in *aG*.

5.2 Screening process for the AF

In maintaining the *AF*, the duplicated tuples should be eliminated. In order to efficiently screen duplicated tuples various methods are suggested [6, 13], in this paper we adopt an incremental method. When a tuple in the *DF* or in the *AF* is appended, then the screening process is activated and duplicated tuples, if any, should be eliminated. It can be represented formally as: $dR_j \cap aR_j = \varnothing$ for all $j \subseteq \Psi$.

5.3 Maintaining join materialized views

Using the *DF*s and *AF*s introduced above, the data warehouse views can be maintained without interfering the current database tables. In this paper, we emphasize that maintaining the data warehouse views by the *AF*, there is no need to lock the base relations.

Example 6. With the *DF's* (*dP* and *dG*) and the *aG*, the view in the example can be refreshed as follows; {(m5, 800, 3, video), (m6, 900, 3, video), (s150, 1000, 2, tv), (eq1, 1500, 4, audio)}.

5.4 Cost Models

We analyze three kinds of cost model; (1) a *baseline* method that uses the base relations to update the views, (2) an *all-inc-aux* method as the *push* or *pull* (say, *all*) type with *incremental* method using *auxiliary relations*; (3) Our approach can be said as the *push* or *pull* (say, *all*) type with all kinds of update time scale (say, *all*) using objects *differential files*, thus called *all-all-df* method.

The cost of updating an object (O_i) in terms of transactional operation types (T_i) is obtained and let $Cost(O_i, T_i)$ denote this cost. If a method uses an additional object except the base relation, then a penalty cost is added for maintaining. It is assumed to be proportional to the update frequency (f_i). Then the total cost of the updates for propagating the updates to the view can be suggested as follows:

$$\text{TotalCost} = \sum_{i=\psi} [Cost(O_i, T_i) + \text{Penalty Cost}(O_i)^{*} f_i].$$

6. Evaluations

The size of the base table is assumed to be the same whose cardinality is varied from 10MB to 1GB. The size of auxiliary relation and the other files suggested in the previous description is assumed to be the same, and to be proportional to the base relation varied from 1% to 50% of base relation. The experiments are executed via Mathematica v3.0 in SUN spark. Three methods are analyzed such as (1) the *baseline* method, (2) *all-inc-aux* method, and (3) *all-all-df* method.

Table 1~3 represent the cost comparisons in terms of modified view sizes according to size of the base relation changes. The costs are increased along with the view size as well as the update ratios on the base relations. The costs of *all-inc-aux* method and *all-all-df* method are relatively more increased than that of the *baseline* method. Which naturally means that the more changes in the base relation, the more works to do in those two methods.

Fig. 1 represents the cost trajectories according to the update ratios with a huge size of base relation (1G). The cost of the *baseline* method is stable as update rations increasing, but the costs of the other two methods are increased rapidly according to the update ratio. If a base table is updated frequently roughly less than 30% of base relations, then the view maintenance by *all-inc-aux* method or by *all-all-df* method is significantly advantageous. It is mainly derived from the fact that the cost penalties due to the change of base relations have burdened each factor.

Table 1 Costs for updating 1% of base relations

dV	10	100	200	500	800	1000
baseline	7200	79200	151200	367200	583200	655200
all-inc-aux	3677.2	40449.2	77221.2	187537	297853	334625
all-all-df	152.4	1676.4	3200.4	7772.4	412344.4	13868.4

Table 2 Costs for updating 10% of base relations

dV	10	100	200	500	800	1000
baseline	7200	79200	151200	367200	583200	655200
all-inc-aux	4480	49280	94080	228480	362880	407680
all-all-df	1560	17160	32760	79560	126360	141960

Table 3 Costs for updating 50% of base relations

dV	10	100	200	500	800	1000
baseline	7200	79200	151200	367200	583200	655200
all-inc-aux	10400	114400	218400	530400	842400	946400
all-all-df	8600	94600	180600	438600	696600	782600

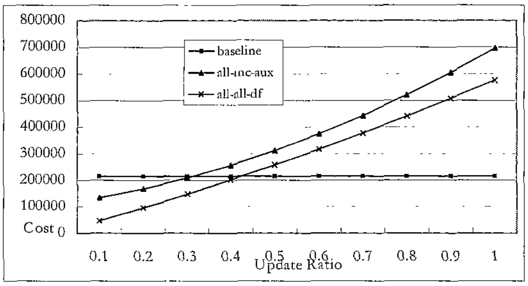

Fig. 1 Cost trajectories in terms of the update ratios with fixed size (1G) of base relation

7. Concluding Remarks

In this paper we dealt with the maintenance of complex join materialized view for data warehouses. First of all, an integrated point of view on view maintenance such as update object, update subject, and update time is addressed. We have a formal approach to figure out what makes it difficult in maintaining data warehouse views and to suggest a solution scheme with relevant algorithms for a data warehouse environment. The solution scheme investigates RI constraints in terms of changes both in the referencing relation and in the relevant referenced relation. Three methods are analyzed such as the *baseline* method, *all-inc-aux* method, and *all-all-df* method. Experimental results represent that the costs of *all-inc-aux* method and *all-all-df* method are relatively more advantageons than that of the *baseline* method. The experiment represent that the more changes in the base relation, the more works to do in those two methods. Unless a huge size (more that 1GB) of base relation is updated roughly more than 30% of base relations, the view maintenance by *all-inc-aux* method or by *all-all-df* method is advantageous. It is mainly derived from the fact that the cost penalties due to the change of base relations have burdened each factor. The solution scheme is shown to be appropriate in maintaining the data warehouse join views self-maintainable.

References

[1] Braham, T. O., "Integrating of Inheritance and Reference Links in the Building of an Object Distributed Database Management Systems," Proc. IEEE Data Engineering, pp. 535-541, 1997.

[2] Chao, D., Diehr, G. and Saharia, A., "Maintaining Join-based *Remote Snapshots Using Relevant* Logging," proc. ACM SIGMOD, Monteal, Canada, pp. 10-16, 1996.

[3] Colby, L., Griffin, T., Libkin, L., Mumick, I. and Trickey, H., "Algorithms for Deferred View Maintenance," ACM SIGMOD, pp.469-480, 1996.

[4] Date, C. J. and Darwen, H., Foundation for Future Database Systems, 2nd ed., Addison-Wesley, 2000.

[5] Goldring, R., "A Discussion of Relational Database Replication Technology," InfoDB, Spring, 1994.

[6] Gupta, A. and Blakeley, J., "Using partial information to update a materialized view," Information Systems, Vol. 20, No. 8, pp. 641-662, 1995.

[7] *Gupta, A., Jagadish, H. and Mumick, I.,* "Data Integration Using Self-Maintainable Views," proc. EDBT96, pp. 140-144, 1996.

[8] Harder, T. and Reinert, J., "Access Path Support for Referential Integrity in SQL2," The VLDB journal, Vol. 5, No. 3, pp. 196-214, 1996.

[9] Horowitz, B., "A Run-Time Execution Model for Referential Integrity Maintenance," IEEE TKDE, pp.548-556, 1992.

[10] Kotidis, Y. and Roussopoulos, N., "DynaMat: A Dynamic View Management System for Data Warehouses," ACM SIGMOD, pp. 371-382, 1999.

[11] Laurent, D., Lechtenborger, J., Spyratos, N., and *Vossen, G., "Complements for Data* Warehouses," proc. the 15th Int. conf. Data Engineering, proc. 15th ICDE, 1999.

[12] Lee, W. Park, J. and Kang, S., "An Asynchronous Differential Join in Distributed Data Replications," Journal of Database Management, Vol. 10, No. 3, Idea-Group Publishing, pp.3-12, 1999.

[13] Lee, W., "On the Independence of Data Warehouse from Databases in Maintaining Join Views," Lecture Note *in Computer* Science, Springer Verlag, Vol. 1676, 1999.

[14] Lee, W., "Data Warehouse Engin independent of Legacy Database System," IEEE Trans. on knowledge and Data Engineering accepted to appear.

[15] Markowitz, V., "Safe Referential Integrity Structures in Relational Databases," Proc. VLDB, Barcelona, Sept. pp.123-132, 1991.

[16] Mohania, M. and Kambayashi, Y., "Making Aggregate Views Self-maintainable," Data and Knowledge Engineering, Vol. 32, No. 1, pp. 87-109, 2000.

[17] Quass, D., Gupta, A., Mumick, I. and Widom, J., "Making Views Self-Maintainable for Data Warehousing," proc. *Parallel and Distributed Information* Systems, Miami, FL, 1996.

[18] Ram, P. and Do, L., "Extracting Delta for Incremental Data Warehouse Maintenance," proc. IEEE Data Engineering, pp. 220-229, 2000.

[19] Ross, K. Srivastava, D. and Sudarshan, S., "Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time," ACM SIGMOD, pp. 447-458, 1996.

[20] Roussopoulos, N., "An Incremental Access Method for View Cache: Concepts, Algorithms, and Cost Analysis," ACM TODS, Vol. 16, No. 3, Sept. 1991.

[21] Segev, A. and Park, J., "Updating Distributed Materialized Views," IEEE TKDE, Vol. 1, No. 2, June, pp.173-184, 1989.

[22] SQL2, ISO/IEC 9075:1992, "Database Language SQL," July 1992.

[23] Staudt, M. and Jarke, M., "Incremental Maintenance of Externally Materialized Views," Proc. VLDB, Bombay, India, pp.75-86, 1996.

이 우 기

1987년 서울대학교 산업공학과 학사. 1993년 서울대학교 산업공학과 석사. 1996년 서울대학교 산업공학과 박사. 2000년 ~ 카네기멜런대학교 MSE 과정 수료. 1996년 ~ 현재 성결대학교 컴퓨터학부 조교수. 2001년 ~ 현재 성결대학교 컴퓨터학부 학부장. 관심분야는 데이타웨어하우스, 데이타베이스 모델링, 멀티미디어 및 인터넷 정보검색, MIS 등.