Available online at www.sciencedirect.com
ScienceDirect
Decision Support Systems 42 (2006) 267–282
Decision Support Systems
www.elsevier.com/locate/dsw

Denormalization strategies for data retrieval from data warehouses

Seung Kyoon Shin[a,*], G. Lawrence Sanders[b,1]

[a]College of Business Administration, University of Rhode Island, 7 Lippitt Road, Kingston, RI 02881-0802, United States
[b]Department of Management Science and Systems, School of Management, State University of New York at Buffalo, Buffalo, NY 14260-4000, United States

Available online 20 January 2005

Abstract

In this study, the effects of denormalization on relational database system performance are discussed in the context of using denormalization strategies as a database design methodology for data warehouses. Four prevalent denormalization strategies have been identified and examined under various scenarios to illustrate the conditions where they are most effective. The relational algebra, query trees, and join cost function are used to examine the effect on the performance of relational systems. The guidelines and analysis provided are sufficiently general and they can be applicable to a variety of databases, in particular to data warehouse implementations, for decision support systems.

Keywords: Database design; Denormalization; Decision support systems; Data warehouse; Data mining

1. Introduction

With the increased availability of data collected from the Internet and other sources and the implementation of enterprise-wise data warehouses, the amount of data that companies possess is growing at a phenomenal rate. It has become increasingly important for the companies to better manage their data warehouses as issues related to database design for high performance are receiving more attention. Database design is still an art that relies heavily on human intuition and experience. Consequently, its practice is becoming more difficult as the applications that databases support become more sophisticated [32].Currently, most popular database implementations for business applications are based on the relational model. It is well known that the relational model is simple but somewhat limited in supporting real world constructs and application requirements [2]. It is also readily observable that there are indeed wide differences between the academic and the practitioner focus on database design. Denormalization is one example that

* Corresponding author. Tel.: +1 401 874 5543; fax: +1 401 874 4312.
E-mail addresses: shin@uri.edu (S.K. Shin), mgstand@mgt.buffalo.edu (G.L. Sanders).
[1] Tel.: +1 716 645 2373; fax: +1 716 645 6117.

0167-9236/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.dss.2004.12.004

has not received much attention in academia but has been a viable database design strategy in real world practice.

From a database design perspective, normalization has been the rule that should be abided by during database design processes. The normal forms and the process of normalization have been studied by many researchers, since Codd [10] initiated the subject. The objective of normalization is to organize data into normal forms and thereby minimize update anomalies and maximize data accessibility. While normalization provides many benefits and is indeed regarded as the rule for relational database design, there is at least one major drawback, namely "poor system performance" [15,31,33,50]. Such poor performance can be a major deterrent to the effective managerial use of corporate data.

In practice, denormalization techniques are frequently utilized for a variety of reasons. However, denormalization still lacks a solid set of principles and guidelines and, thus, remains a human intensive process [40]. There has been little research illustrating the effects of denormalization on database performance and query response time, and effective denormalization strategies. The goal of this paper is to provide comprehensive guidelines regarding when and how to effectively exercise denormalization. We further propose common denormalization strategies, analyze the costs and benefits, and illustrate relevant examples of when and how each strategy can be applied.

It should be noted that the intention of this study is not to promote the concept of denormalization. Normalization is, arguably, the mantra upon which the infrastructure of all database systems should be built and one of the foundational principals of the field [20,25]. It is fundamental to translating requirement specifications into semantically stable and robust physical schema. There are, however, instances where this principal of database design can be violated or at least compromised to increase systems performance and present the user with a more simplified view of the data structure [39]. Denormalization is not necessarily an incorrect decision, when it is implemented wisely, and it is the objective of this paper to provide a set of guidelines for applying denormalization to improve database performance.

This paper is organized as follows: Section 2 discusses the relevant research on denormalization and argues the effects of incorporating denormalization techniques into decision support system databases. Section 3 presents an overview of how denormalization fits in the database design cycle and develops the criteria to be considered in assessing database performance. Section 4 summarizes the commonly accepted denormalization strategies. In Section 5, relational algebra, query trees, and join cost function approach are applied in an effort to estimate the effect of denormalization on the database performance. We conclude in Section 6 with a brief discussion of future research.

2. Denormalization and data warehouses

2.1. Normalization vs. denormalization

Although conceptual and logical data models encourage us to generalize and consolidate entities to better understand the relationships between them, such generalization does not guarantee the best performance but may lead to more complicated database access paths [4]. Furthermore, normalized schema may create retrieval inefficiencies when a comparatively small amount of data is being retrieved from complex relationships [50]. A normalized data schema performs well with a relatively small amount of data and transactions. However, as the workload on the database engine increases, the relational engine may not be able to handle transaction processing with normalized schema in a reasonable time frame because relatively costly join operations are required to combine normalized data. As a consequence, database designers occasionally trade off the aesthetics of data normalization with the reality of system performance.

There are many cases when a normalized schema does not satisfy the system requirements. For example, existing normalization concepts are not applicable to temporal relational data models [9] because these models employ relational structures that are different from conventional relations [29]. In addition, relational models do not handle incomplete or unstructured data in a comprehensive manner, while for many real-world business applications such as data warehouses where multiple heterogeneous data sources are consolidated into a large data repository, the data are frequently incomplete and uncertain [19].

In an OLAP environment, the highly normalized form may not be appropriate because retrieval often entails a large number of joins, which greatly increases response times and this is particularly true when the data warehouse is large [48]. Database performance can be substantially improved by minimizing the number of accesses to secondary storage during transaction processing. Denormalizing relations reduces the number of physical tables that need to be accessed to retrieve the desired data by reducing the number of joins needed to derive a query [25].

2.2. Previous work on denormalization

Normalization may cause significant inefficiencies when there are few updates and many query retrievals involving a large number of join operations. On the other hand, denormalization may boost query speed but also degrade data integrity. The trade-offs inherent in normalization/denormalization should be a natural area for scholarly activity. The pioneering work by Schkolnick and Sorenson [44] introduced the notion of denormalization. They argue that, to improve database performance, the data model should be viewed by the user because the user is capable of understanding the underlying semantic constraints.

Several researchers have developed a lists of normalization and denormalization types, and have also suggested that denormalization should be carefully deployed according to how the data will be used [24,41]. The common denormalization types include pre-joined tables, report tables, mirror tables, split tables, combined tables, redundant data, repeating groups, derivable data, and speed tables. The studies [24,41] show that denormalization is useful when there are two entities with a one-to-one relationship and a many-to-many relationship with non-key attributes.

There are two approaches to denormalization [49]. In the first approach, the entity relationship diagram (ERD) is used to collapse the number of logical objects in the model. This will in effect shorten application call paths that traverse the database objects when the structure is transformed from logical to physical objects. This approach should be exercised when validating the logical model. In the second approach, denormalization is accomplished by moving or consolidating entities and creating entities or attributes to facilitate special requests, and by introducing redundancy or synthetic keys to encompass the movement or change of structure within the physical model. One deficiency of this approach is that future development and maintenance are not considered.

Denormalization procedures can be adopted as pre-physical database design processes and as intermediate steps between logical and physical modeling, and provide an additional view of logical database refinement before the physical design [7]. The refinement process requires a high level of expertise on the part of the database designer, as well as appropriate knowledge of application requirements.

There are possible drawbacks of denormalization. Denormalization decisions usually involve trade-offs between flexibility and performance, and denormalization requires an understanding of flexibility requirements, awareness of the update frequency of the data, and knowledge of how the database management system, the operating system and the hardware work together to deliver optimal performance [12].

From the previous discussion, it is apparent that there has been little research regarding a comprehensive procedure for invoking and assessing a process for denormalization.

2.3. Denormalizing with data warehouses and data marts

Data warehousing and OLAP are long established fields of study with well-established technologies. The notion of OLAP refers to the technique for performing complex analysis over the information stored in a data warehouse. A data warehouse (or smaller-scale data mart) is typically a specially created data repository supporting decision making and, in general, involves a "very large" repository of historical data pertaining to an organization [28]. The goal of the data warehouse is to put enterprise data at the disposal of organizational decision makers. The typical data warehouse is a subject-oriented corporate database that involves multiple data models implemented on multiple platforms and architectures. While there are data warehouses for decision support based on the star schema[2], which is a suitable alternative, there are also

[2] A star schema is a database design in which dimensional data are separated from fact data. A star schema consists of a fact table with a single table for each dimension.

some cases where a star schema is not the best database design [39]. It is important to recognize that star schemas do not represent a panacea for data warehouse database design.

It should be noted that denormalization can indeed speed up data retrieval and that there is a downside of denormalization in the form of potential update anomalies [17]. But updates are not typical in data warehouse applications, since date warehouses and data marts involve relatively fewer data updates, and most transactions in a data warehouse involve data retrieval [30]. In essence, data warehouses and data marts are considered good candidates for applying denormalization strategies because individual attributes are rarely updated.

Denormalization is particularly useful in dealing with the proliferation of star schemas that are found in many data warehouse implementations [1]. In this case, denormalization may provide better performance and a more intuitive data structure for data warehouse users to navigate. One of the prime goals of most analytical processes with the data warehouse is to access aggregates such as sums, averages, and trends, which frequently lead to resource-consuming calculations at runtime. While typical production systems usually contain only the basic data, data warehousing users expect to find aggregated and time-series data that is in a form ready for immediate display. In this case, having pre-computed data stored in the database reduces the cost of executing aggregate queries.

3. Database design with denormalization

The database design process, in general, includes the following phases: conceptual, logical, and physical design [25,43]. Denormalization can be separated from those steps because it involves aspects that are not purely related to either logical or purely physical design. We propose that the denormalization process should be implemented between the data model mapping and the physical design and that it is integrated with logical and physical design. In fact, it has been asserted by Inmon [26] that data should be normalized as the design is being conceptualized and then denormalized in response to the performance requirements. Fig. 1 presents a suggested database design procedure incorporating the denormalization process.

A primary goal of denormalization is to improve the effectiveness and efficiency of the logical model implementation in order to fulfill application requirements. However, as in any denormalization process, it may not be possible to accomplish a full denormalization that meets all the criteria specified. In such cases, the database designer will have to make trade-offs based on the importance of each criteria and the application processing requirement, taking into account the pros and cons of denormalization implementation.

The process of information requirements determination or logical database design produces a description of the data content and processing activities that must be supported by a database system. In addition, in order to access the efficiency of denormalization, the hardware environment such as page size, storage and access cost must also be described. This information is needed to develop the criteria for denormalization.

A generally accepted goal in both denormalization and physical design is to minimize the operational costs of using the database. Thus, a framework for information requirements for physical design can be applied to denormalization. Carlis et al. [6] developed a framework for information requirements that can be readily applied to denormalization. Based on this framework and the literature review related to denormalization, Table 1 presents a more detailed overview of the numerous processing issues that need to be considered when embarking on a path to denormalization [3,13,21,25,32,37]. The main inputs to the denormalization process are the logical structure, volume analysis (e.g. estimates of tuples, attributes, and cardinality), transaction analysis (e.g. statement of performance requirements and transaction assessment in terms of the frequency and profile), and volatility analysis.

An estimate of the average/maximum possible number of instances per entity is a useful criterion to supply the model's ability to fulfill access requirements. In developing a transaction analysis, the major transactions required against the database need to be identified. Transactions are logical units of work made up of pure retrievals, insertions, updates, or deletions, or a mixture of all four. Each transaction is analyzed

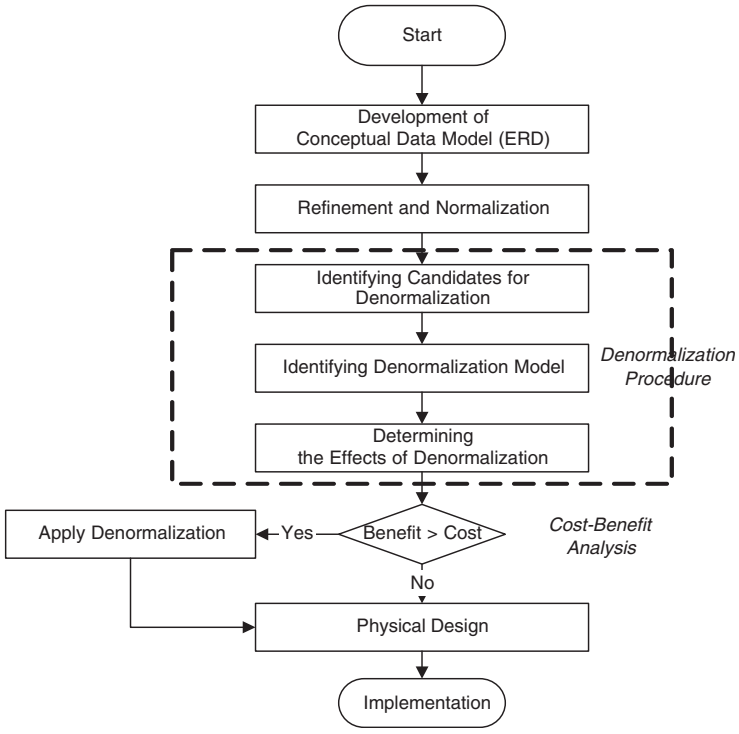

Fig. 1. Database design cycle incorporating denormalization.

to determine the access paths used and the estimated frequency of use. Volatility analysis is useful in that the volatility of a file has a clear impact on its expected update and retrieval performance [47].

4. Denormalization strategies for increasing performance

In this section, we present denormalization patterns that have been commonly adopted by experienced database designers. We also develop a mathematical model for assessing the benefits of each pattern using cost–benefit analysis. After an extensive literature review, we identified and classified four prevalent models for denormalization in Table 2 [7,15,16,23,24,38,49]. They are *collapsing relations (CR), partitioning a relation (PR), adding redundant attributes (RA), and adding derived attributes (DA)*.

Our mathematical model for the benefits of denormalization relies on performance of query processing measured by the elapsed time difference of each query with normalized and denormalized schemas. Suppose that there are i (i 1,2,...,I) data retrieval queries and j (j 1,2,...,J) update queries processing with a normalized schema. The schema can be denormalized with n (n 1,2,...,N) denormalization models, and i' (i' 1,2,..,I') data retrieval queries and j' (j' 1,2,...,J') update queries are modified to retrieve the same data from the denormalized schema as those with the normalized. Each query is executed t_i times per hour. Let x_i denote the elapsed time of query i so that enhanced query performance by the denormalized model can be measured by the time difference of two queries, $X_i = x_i - x_{i'}$. The benefit, K, of denormalization model, DN ($DN = \{CR, PR, RA, DA\}$), can be stated as

$$K_{DN} = \sum_i X_i t_i + \sum_j X_j t_j. \quad (1)$$

It is assumed that $\sum_i X_i t_i > 0$, as it is the goal of denormalization. On the contrary, $\sum_j X_j t_j \le 0$ because most denormalization models entail data duplication that may increase the volume of data to be updated as well as may require additional update transactions.

Table 1
Information requirements for denormalization

Type of information	Variables to be considered during denormalization
Data description	Cardinality of each relation
	Optionality of each relation
	Data volume of each relation
	Volatility of data
Transaction activity	Data retrieval and update patterns of the user community
	Number of entities involved by each transaction
	Transaction type (update/query, OLTP/OLAP)
	Transaction frequency
	Number of rows accessed by each transaction
	Relations between transactions and relations of entities involved
	Access paths needed by each transaction
Hardware environment	Pages/blocks size
	Access costs
	Storage costs
	Buffer size
Other consideration	Future application development and maintenance considerations

Note that any given physical design is good for some queries but bad for others [17]. While there are a queries (q_a, a 1,2,...,A) that are to be enhanced by the denormalized schema ($(\forall_{q_a})(X_a>0)$), there may be $i-a$ queries (q_b, b 1,2,...,$I-A$) that are degraded ($(\forall_{q_b})(X_a\leq 0)$). This results in the model structure below:

$$K_{DN} = \sum_a X_a t_a + \sum_b X_b t_b + \sum_j X_j t_j \quad (2)$$

where $\sum_a X_a t_a>0$ and $\sum_b X_b t_b\leq 0$. The mathematical model for selecting the optimal set of the relations needs to deal with two requirements. First, the objective of denormalization should be to enhance data retrieving performance $\left(\sum_i X_i>0\right)$, trading with the costs of update transaction caused by data redundancy $\left(\sum_j X_j<0\right)$. However, this cost would be negligible at data warehouses and data marts because the databases for those systems do not involve a great deal of update transactions. Second, as denormalization results in the costs of data duplication, if the schema involves update transactions, the cost of update transaction should not exceed the sum of the benefits. We can state these requirements in the model structure shown below.

$$\text{Max} \quad K_{DN} = \sum_a X_a t_a + \sum_b X_b t_b + \sum_j X_j t_j \quad (3)$$

$$\text{Subject to} \quad \sum_a X_a t_a + \sum_b X_b t_b>0 \quad (4)$$

$$\sum_a X_a t_a>0 \quad (5)$$

$$\sum_a X_a t_a + \sum_b X_b t_b> \sum_j X_j t_j \quad (6)$$

Based on this formula, the benefits of the four denormalization models, *CR*, *PR*, *RA*, and *DA*, will be discussed.

4.1. Collapsing relations (CR)

One of the most commonly used denormalization models involves the collapsing of one-to-one relationships [38]. Consider the normalized schema with two relations in a one-to-one relationship, CUSTOMER and CUSTOMER_ACCOUNT as shown in Fig. 2. If there are a large number of queries requiring a join operation between the two relations, collapsing both relations into CUSTOMER_DN may enhance query performance. There are several benefits of this model in the form of reduced number of foreign keys on relations, reduced number of indexes (since most indexes are based on primary/foreign keys), reduced storage space, and reduced join operations. Collapsing a one-to-one relationship does not cause data dupli-

Table 2
Denormalization models and patterns

Denormalization strategies	Denormalization patterns
Collapsing relations (CR)	Two relations with a one-to-one relationship
	Two relations with a one-to-many relationship
	Two relations with a many-to-many relationship
Partitioning a relation (PR)	Horizontal partitioning
	Vertical partitioning
Adding redundant attributes (RA)	Reference data in lookup/ code relations
Derived attributes (DA)	Summary data, total, and balance

Normalized Schema

CUSTOMER (Customer_ID, Customer_Name, Address)
CUSTOMER_ACCOUNT (Customer_ID, Account_Bal, Market_Segment)

Denormalized Schema

CUSTOMER_DN (Customer_ID, Customer_Name, Address, Account_Bal, Market_Segment)

Fig. 2. An example of collapsing one-to-one relationship.

cation and, in general, update anomalies are not a serious issue $\left(\sum_j X_j = 0\right)$.

Combining two relations in a one-to-one relationship does not change the business view but does decrease overhead access time because of fewer join operations. In general, collapsing tables in a one-to-one relationship has fewer drawbacks than other denormalization approaches.

One-to-many relationships of low cardinality can be a candidate for denormalization, although data redundancy may interfere with updates [24]. Consider the normalized relations CUSTOMER and CITY in Fig. 3, where *Customer_ID* and *Zip* are, respectively, the primary keys for each relation. These two relations are in 3rd normal form and the relationship between CITY and CUSTOMER is one-to-many. Suppose the address of CUSTOMER does not frequently change, this structure does not necessarily represent a better design from an efficiency perspective, particularly when most queries require a join operation. On the contrary, the denormalized relation is superior because it will substantially decrease the number of join operations. In addition, the costs for data anomalies caused by a transitive functional dependency (Cust-Number → Zip, Zip → {City, State}) may be minimal when the "one" side relation (CITY) does not involve update transactions.

If a reference table in a one-to-many relationship has very few attributes, it is a candidate for *collapsing relations* [25]. Reference data typically exists on the "one" side of a one-to-many relationship. This entity typically does not participate in a different type of relationship because a code table typically participates in a relationship as a code table. However, as the number of attributes in the reference relation increases, it may be reasonable to consider *RA* rather than *CR* (this will be discussed in detail in Section 4.3). In general, it is appropriate to combine the two relations in a one-to-many relationship when the "many" side relation is associated with low cardinality. Collapsing one-to-many relationships can be relatively less efficient than collapsing one-to-one relationships because of data maintenance processing for duplicated data. In the case of a reference table, the update costs are negligible $\left(\sum_j X_j t_j \approx 0\right)$ because update transactions rarely occur in reference relations $\left(t_j \approx 0\right)$.

A many-to-many relationship can also be a candidate for *CR* when the cardinality and the volatility of involved relations are low [25,41]. The typical many-to-many relationship is represented in the physical database structure by three relations: one for each of two primary entities and another for cross-referencing them. A cross-reference or intersection between the two entities in many instances also represents a business entity. These three relations can be merged into two if one of them has little data apart from its primary key. Such a relation could be merged into a cross-reference relation by duplicating the attribute data.

Normalized Schema

CUSTOMER (Customer_ID, Customer_Name, Address, ZIP)
CITY (Zip, City, State)

Denormalized Schema

CUSTOMER_DN (Customer_ID, Customer_Name, Address, ZIP, City, State)

Fig. 3. An example of collapsing one-to-many relationship.

CR eliminates the join, but the net result is that there is a loss at the abstract level because there is no conceptual separation of the data. However, Bracchi et al. [5] argues, "we see that from the physical point of view, an entity is a set of related attributes that are frequently retrieved together. The entity should be, therefore, an access concept, not a logical concept, and it corresponds to the physical record." Even though the distinction between entities and attributes is real, some designers conclude it should not be preserved, at least for conceptual modeling purposes [36]. These discussions present us with the opportunity to explore strategies for denormalization and free us from the stringent design restrictions based on the normalization rules.

4.2. Partitioning relation (PR)

Database performance can be enhanced by minimizing the number of data accesses made in the secondary storage upon processing transactions. *Partitioning a relation* (*PR*) in data warehouses and data marts is a major step in database design for fewer disk accesses [8,34,35]. When separate parts of a relation are used by different applications, the relation may be partitioned into multiple relations for each application processing group. A relation can be partitioned either vertically or horizontally. It is the process of dividing a logical relation into several physical objects. The resulting file fragments consist of the smaller number of attributes (if vertically partitioned) or tuples (if horizontally partitioned) and therefore fewer accesses are made in secondary storage to process a transaction.

4.2.1. Vertical partitioning (VP)

Vertical Partitioning (*VP*) involves splitting a relation by columns so that a group of columns be placed into the new relation and the remaining columns be placed in another [13,38]. PART in Fig. 4 consists of quite a few attributes associated with part specification and inventory information. If query transactions can be classified into two groups in terms of association with the two attribute groups, and there are few transactions involving the two groups at the same time, PART can be partitioned into PART_SPEC_INFO and PART_INVENTORY_INFO to decrease the number of pages needed for each query. In general *VP* is particularly effective when there are lengthy text fields with a large number of columns and these lengthy fields are assessed by a limited number of transactions. In actuality, a view of the joined relations may make these partitions transparent to the users. A vertically partitioned relation should contain one row per primary key as this facilitates data retrieval across relations.

VP does not cause much data duplication except for the relating primary keys among the partitioned relations. As those primary keys are rarely modified, the cost caused by the data duplication can be ignored $\left(\sum_j X_j \approx 0\right)$.

4.2.2. Horizontal partitioning (HP)

Horizontal partitioning (*HP*) [13] involves a row split and the resulting rows are classified into groups by key ranges. This is similar to partitioning a relation during physical database design, except that each relation has a respective name. *HP* can be used when a relation contains a large number of tuples as the sales database of a super store. SALE_HISTORY of super store database in Fig. 5 is a common case of the relation including a number of historical sales transactions to be considered for *HP*. It is likely that most OLAP transactions with this relation include a time-dependent variable. As such, *HP* based on the analysis period considerably decreases unnecessary data access to the secondary data storage.

HP is usually applied when the table partition corresponds to a natural separation of the rows such as in the case of different geographical sites or when there

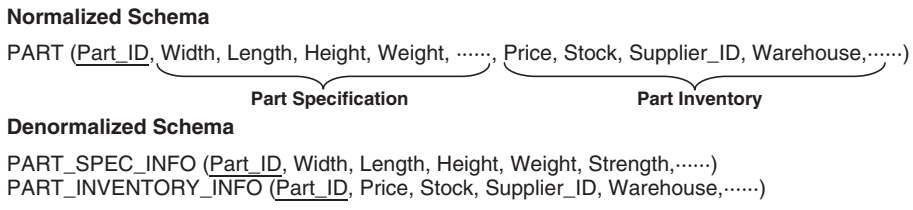

Fig. 4. An example of vertical partitioning.

Normalized Schema

SALE_HISTORY (Sale_ID, Timestamp, Store_ID, Customer_ID, Amount, ·····)

Denormalized Schema

SALE_HISTORY_Period_1 (Sale_ID, Timestamp, Store_ID, Customer_ID, Amount, ·····)
SALE_HISTORY_Period_2 (Sale_ID, Timestamp, Store_ID, Customer_ID, Amount, ·····)
SALE_HISTORY_Period_3 (Sale_ID, Timestamp, Store_ID, Customer_ID, Amount, ·····)
••••••

Fig. 5. An example of horizontal partitioning.

is a natural distinction between historical and current data. *HP* is particularly effective when historical data is rarely used and current data must be accessed promptly. A database designer must be careful to avoid duplicating rows in the new tables so that "UNION ALL" may not generate duplicated results when applied to the two new tables. In general, *HP* may add a high degree of complexity to applications because it usually requires different table names in queries, according to the values in the tables. This complexity alone usually outweighs the advantages of table partitioning in most database applications.

Partitioning a relation either vertically or horizontally does not cause data duplication $\left(\sum_j X_j = 0\right)$. Therefore, data anomalies are not an issue in this case.

4.3. Adding redundant attributes (RA)

Adding redundant attributes (*RA*) can be applied when data from one relation is accessed in conjunction with only a few columns from another table [13]. If this type of table join occurs frequently, it may make sense to add a column and carry them as redundant [24,38]. Fig. 6 presents an example of *RA*. Suppose that PART information is more useful with Supplier_Name (not Supplier_ID) and that it does not change frequently due to the company's stable business relationship with part suppliers, and that joins are frequently executed on the PART and SUPPLIER in order to retrieve part data from PART and only one attribute, Supplier_Name, from SUPPLIER. In the normalized schema, queries require costly join operations, while a join operation is not required in the denormalized schema. The major concern of denormalization, the occurrence of update anomalies, would be minimal in this case because Supplier_Name would not be frequently modified.

In general, adding a few redundant attributes does not change either the business view or the relationships with other relations. The concerns related to future development and database design flexibility, therefore, are not critical in this approach. Normally, $\sum_b X_b$ for RA is negligible $\left(\sum_b X_b \approx 0\right)$. However the costs for an update transaction may be critical when the volatility of those attributes is high.

Reference data, which relates to sets of code in one relation and a description in another, is accomplished via a join, and it presents a natural case where redundancy can pay off [3]. While *CR* is good for short reference data, adding redundant columns is suitable for a lengthy reference field. A typical strategy for a relation with long reference data is to duplicate the short descriptive attribute in the entity, which would otherwise contain only a code. The result is a redundant attribute in the target entity that is functionally independent of the primary key. The code attribute is an arbitrary convention invented to normalize the corresponding description and, of course, reduce update anomalies.

Normalized Schema

PART (Part_ID, Width, Length, Height, Weight, Price, Stock, Supplier_ID, ···)
SUPPLIER (Supplier_ID, Supplier_Name, Address, State, ZIP, Phone, Sales_Rep, ···)

Denormalized Schema

PART_DN (Part_ID, Width, Length, Height, Weight, Price, Stock, Supplier_ID, *Supplier_Name*, ···)
SUPPLIER (Supplier_ID, Supplier_Name, Address, State, ZIP, Phone, Sales_Rep, ···)

Fig. 6. An example of adding redundant columns.

Denormalizing a reference data relation does not necessarily reduce the total number of relations because, in many cases, it is necessary to keep the reference data table for use as a data input constraint. However, it does remove the need to join target entities with reference data tables as the description is duzplicated in each target entity. In this case, the goal should be to reduce redundant data and keep only the columns necessary to support the redundancy and infrequently updated columns used in query processing [24].

4.4. Derived attributes (DA)

With the growing number of large data warehouses for decision support applications, efficiently executing aggregate queries, involving aggregation such as MIN, MAX, SUM, AVG, and COUNT is becoming increasingly important. Aggregate queries are frequently found in decision support applications where relations with large history data often are joined with other relations and aggregated. Because the relations are large, the optimization of aggregation operations has the potential to provide huge gains in performance.

To reduce the cost of executing aggregate queries, frequently used aggregates are often precomputed and materialized in an appropriate relation. In statistical database applications, derived attributes and powerful statistical query formation capabilities are necessary to model the data and to respond to statistical queries [18,45]. Business managers typically concentrate first on aggregate values and then delve into more detail. Summary data and derived data, therefore, are important in decision support systems (DSS) based on a data warehouse [22]. The nature of many DSS applications dictates frequent use of data calculated from naturally occurring large volume of information. When a large volume of data is involved, even simple calculations can consume processing time as well as increase batch-processing time to unacceptable levels. Moreover, if such calculations are performed on the fly, the user's response time can be seriously affected.

An example of derived attributes is presented in Fig. 7. Suppose that a manager frequently needs to obtain information regarding the sum of purchases of each customer. In the normalized schema, the queries require join operations with the CUSTOMER and ORDER relations and an aggregate operation for the total balance of individual customers. In the denormalized schema containing the derived attribute *Sum_Of_Purchase* in CUSTOMER_DN, those queries do not require join or aggregate operations but rather a select operation to obtain the data. However, in order to keep the data concurrent in the denormalized format, additional transactions are needed to keep the derived attributes updated in an operation database, which in turn increases the costs of additional update transactions.

As *DA* does not complicate data accessibility, or change the business view, $\sum_b X_b$ is negligible $\left(\sum_b X_b \approx 0\right)$. It is worth noting that the effects of *DA* may decrease, particularly when the data volume of the relation increases.

Storing such derived data in the database can substantially improve performance by saving both CPU cycles and data read time, although it violates normalization principles [24,38]. Optimal database design with derived attributes, therefore, is sensitive to the capability of the aggregation functionality of the database server [37]. This technique can also be combined effectively with other forms of denormalization.

DA reduces processing time at the expense of higher update costs [33]. Updates costs $\left(\sum_j X_j t_j\right)$, therefore, may become the main deterrent for adopting derived attributes [42] and the access frequency, the response time requirement, and increased maintenance costs should be considered [22]. A compromise is to

Normalized Schema

CUSTOMER (Customer_ID, Customer_Name, Address, ZIP)
ORDER (Order_ID, Customer_ID, Order_Date, Standard_Price, Quantity)

Denormalized Schema

CUSTOMER_DN (Customer_ID, Customer_Name, Address, ZIP, ***Sum_Of_Purchase***)
ORDER (Order_ID, Customer_ID, Order_Date, Standard_Price, Quantity)

Fig. 7. An example of derived attribute.

determine the dimension of derived data at lower level so that the update cost can be reduced. Maintaining data integrity can be complex if the data items used to calculate a derived data item change unpredictably or frequently and a new value for derived data must be recalculated each time a component data item changes.

Data retention requirements for summary and detail records may be quite different and must also be considered as well as the level of granularity for the summarized information [25]. If summary tables are created at a certain level and detailed data is destroyed, there will be a problem if users suddenly need a slightly lower level of granularity.

5. Justifying denormalization

Determining the criteria for denormalization is a complex task. Successful denormalization of a database requires a through examination of the effects of denormalization in terms of 1) cardinality relationships, 2) transaction relationships, and 3) access paths required for each transaction/module.

In this section, we develop a framework for assessing denormalization effects using relational algebra, query tree analysis and the join cost function. The primary purpose of the relational algebra is to represent a collection of operations on relations that is performed on a relational database [11]. In relational algebra, an arbitrary relational query can be systematically transformed to a semantically equivalent algebraic expression in which the operations and volume of associated data can be evaluated [14]. The relational algebra is appropriate for assessing denormalization effects in the cost-based perspective.

A query tree is a tree data structure that corresponds to a relational algebra expression. It represents the input relations of the query as leaf nodes of the tree and the relational algebra operation as internal nodes. Relational algebra and query tree analysis provide not only an assessment of both approximate join costs but also join selectivity and access path, and they can be used to assist the database designer in obtaining a clear-cut assessment of denormalization effects.

In this study, we assume that the database processing capacity is a function of the CPU processing capacity and that it is measured in terms of the volume of data accessed per unit time. We also assume that the operational intricacies can be neglected for reasons of analytical tractability [46].

5.1. Algebraic approach to assess denormalization effects

To illustrate the approach, we will use an example *CR* of many-to-many relationship from Section 4.1. Suppose there are three relations in the 3rd normal form, R={r1, r2, r3}, S={s1, s2}, and T={t1, t2, t3} as shown in Fig. 8(A), where S is the M:N relationship between entity R and T [3], and s1 and s2 are the foreign keys of R and T, respectively. When the cross-reference relation S stores foreign keys for R and T (or likely with a few attributes), collapsing relation entity S into T (or R) can be a good strategy to increase database performance such that after denormalization we have two denormalized relations, R={r1, r2, r3} and T′={t1, s1, t2, t3} as shown in Fig. 8 (B).

Now, we consider two queries running *Cartesian products* with non-indexed tables with normalized and denormalized schemas respectively. As illustrated in Fig. 9, these are two sets of SQL statements and relational algebra operations for the nested-loop join queries that retrieve the identical data sets from each data structure. We also present two query trees (see Fig. 10) that correspond to the relational algebra of each schema.

Both queries retrieve the appropriate tuples from each schema. Suppose that there are n tuples in both R and T, and $\dot{m}n$ tuples in entity S assuming that the cardinality between the primary and cross-reference relations is 1:m. Let s denote selection cardinality[4] satisfying the operation condition ($\sigma_{r1='x'}(R)$). The number of resulting join operations of entity R and S ($Res1 \bowtie_{r1=s1} (S)$) is $\dot{m}n\dot{m}s$. Likewise, as the selection cardinality of the first join operation is $\dot{m}s$, the final project operation ($Res2 \bowtie_{s2=t1} (T)$) produces $\dot{m}n\dot{m}s$ relations which results from the Cartesian product

[3] In order to clearly illustrate and amplify the effects of denormalization, it is assumed that all R.r1 are related to S.s1 (∃S)(R.r1= S.s1) and all T.t1 are related to S.s2 (∃S)(T.t1= S.s2), respectively. Also, we use weak entity types that do not have key attributes of their own, although many cases in reality involve with strong entity type that do have key attributes of their own.

[4] The *selection cardinality* is the average number of records that will satisfy an equality selection condition on that attribute.

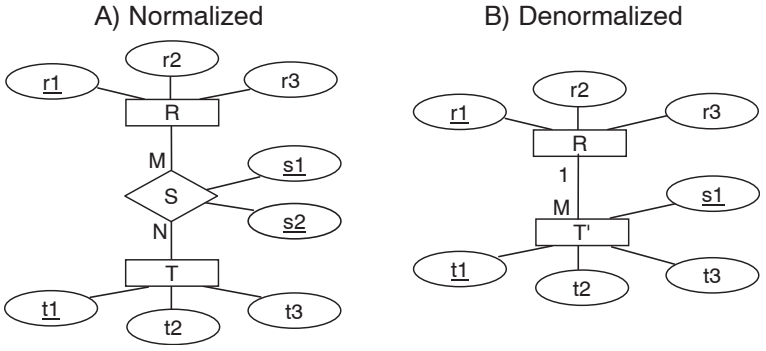

Fig. 8. ER diagrams of exemplary data structure.

operation with the three relations. Thus, the total number of relations produced from the normalized data structure is $2m \cdot n \cdot s$.

After collapsing the two relations S and T, we have $\dot{m}n$ tuples in a denormalized T' (as S and T have been merged into new entity T', the number of tuples of relation T' should be identical to the number of tuples of relation S). In order to retrieve the same dataset from the denormalized schema, the total number of join operations produced is $\dot{m}\dot{n}s$. Notice that join operation with the denormalized schema is 50% $((m \cdot n \cdot s)/(2m \cdot n \cdot s))$ of the join operation required for the normalized schema.

5.2. Cost-based approach to assess denormalization effects

The effects of denormalization can also be assessed by the join operation cost function [20]. Consider the following formula:

$$Cost = b_A + (b_A{*}b_B) + ((js{*}r_A{*}r_B)/bfr_{A-B}) \quad (7)$$

where A and B denote the outer and inner loop files, respectively; r_A and r_B indicate the number of tuples stored in b_A and b_B disk blocks, respectively; js is the join selectivity[5]; and $bfr_{A\text{-}B}$ denotes the blocking factor for the resulting join file (records/block) [20]. Notice that the 3rd term of the right-hand side of the formula is the cost of writing the resulting file to disk. Assuming that the lengths of all columns are equal, the blocking factors of relationsas bfr_R=4, bfr_S=6, $bfr_{R\text{-}S}$=3, bfr_T=4, $bfr_{RS\text{-}T}$=2, $bfr_{T'}$=3 and $bfr_{R\text{-}T'}$=2[6]with the join cardinalities as js_{RS}=1/1000, js_{RST}=1/2000, and $js_{RT'}$=1/2000[7], we estimate the costs for the join operation with normalized schema, C_N, and with denormalized schema, C_D, as follows:

$$\begin{aligned} C_N &= (b_R + (b_R{*}b_S) + ((js_{RS}{*}r_R{*}r_S)/bfr_{R-S})) \\ &\quad +(b_{RS}+ (b_{RS}{*}b_T) + ((js_{RST}{*}r_{RS}{*}r_T)/bfr_{RS-T})) \\ &= (4 + (4{*}334) + ((1/1000{*}1000{*}2000)/3)) \\ &\quad +(7 + (7{*}250)+ ((1/2000{*}20{*}1000)/2)) \\ &= 3768 \end{aligned}$$

$$\begin{aligned} C_D &= b_R + (b_R{*}b_T') + ((js_{RT}'{*}r_R{*}r_T')/bfr_{RT}') \\ &= 4 + (4{*}667) + ((1/2000{*}1000{*}2000)/2) \\ &= 3172 \end{aligned}$$

[5] *Join selectivity* (js) is an estimate for the size of the file that results after the join operation, which is obtained by the ratio of the size of the resulting join file to the size of the Cartesian product file (js=|(A ⋈ c B)|/|(A×S)|) (Elmasri and Navathe, 2000).

[6] Blocking factors (bfr=records/block) are calculated based on assumption that the size of all columns across entities is identical and that the density of block across entities is equal so that the analysis takes into account the change of data volume caused by denormalization. In this case, we assume that bfr_R=4 and bfr_S=6 because entity R has 3 columns and entity S has 2 columns. Based on the blocking factors, we also calculated the number of blocks used for each entity (b_R=4, b_S=334, b_{RS}=7, b_T=250, and $b_{T'}$=667). b_R is obtained after the first selection operation ($\sigma_{r1=\text{'}001\text{'}}$ (R)). Since the cost for this operation is common for C_N and C_D, it was not considered in this estimation.

[7] Join cardinality (js) is calculated based on the assumption that the relationship between entity R and S, and T and S are 1:2, 1000 tuples per each relation R and T, 2000 tuples in cross-reference relation, S, and the selection cardinality of entity R and T is equally 10. In this example, as the cardinality increases, the benefit from denormalization generally increases.

A) SQL and Relational Algebra with Normalization

```
Select R.r1, R.r2, R.r3, T.t1, T.t2, T.t3
From R, S, T
Where R.r1 = '001' and
      R.r1 = S.s1 and
      R.s2 = T.t1;
```

$Res1 \leftarrow \sigma_{r1 = '001'} (R)$
$Res2 \leftarrow (Res1 \bowtie_{r1 = s1} (S))$
$Res3 \leftarrow (Res2 \bowtie_{s2 = t1} (T))$
$Result \leftarrow \pi_{r1, r2, r3, t1, t2, t3} (Res3)$

B) SQL and Relational Algebra with Denormalization

```
Select R.r1, R.r2, R.r3, T.t1, T.t2, T.t3
From R, T'
Where R.r1 = '001' and
      R.r1 = T'.s1;
```

$Res1 \leftarrow \sigma_{r1 = '001'} (R)$
$Res2 \leftarrow (Res1 \bowtie_{r1 = s1} (T'))$
$Result \leftarrow \pi_{r1, r2, r3, t1, t2, t3} (Res2)$

Fig. 9. SQL and relational algebra with normalization/denormalization.

The result shows that the suggested denormalization technique leads to a decrease of 15.8% in cost over the normalized schema. While the example uses a minimal number of attributes as well as a relatively small number of tuples in each entity, most databases have more attributes and more tuples, and the benefits could be greater as far as it keeps low cardinalities. In reality, the positive effect of *CR* of many-to-many relationship is expected to increase in proportion to the number of attributes and tuples involved. By using the relational algebra and join cost function approaches, it is concluded that the join process with the denormalized data structure is less expensive than the normalized data structure. However, the actual benefit will depend on a variety of factors including the length or number of tuples, join and selection cardinalities, and blocking factors.

6. Concluding remarks

The effects of denormalization on relational database system performance have been discussed in the context of using denormalization strategies as a database design methodology. We present four prevalent denormalization strategies, evaluate the effects of each strategy, and illustrate the conditions where they are most effective. The relational algebra, query trees, and join cost function are adopted to examine the effects on the performance of relational systems to

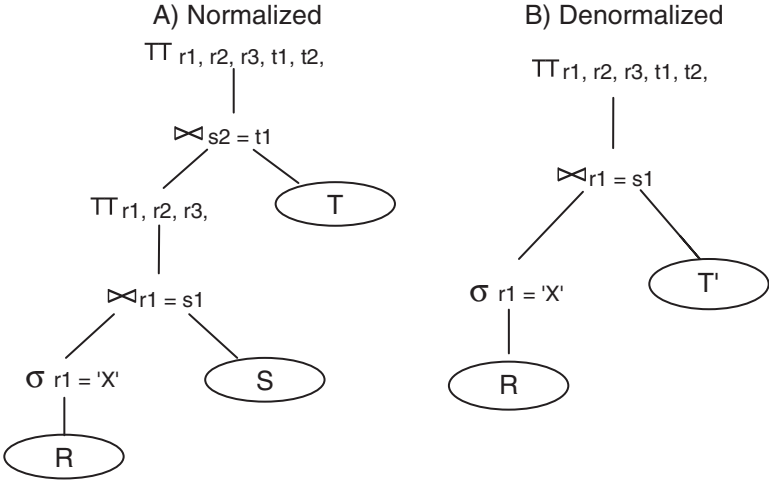

Fig. 10. Query trees analysis.

the end. Two significant conclusions can be derived from these analytical techniques: 1) the denormalization strategies may offer positive effects on database performance and 2) the analytical methods provide an efficient way to assess the effects of the denormalization strategies. The guidelines and analysis provided are sufficiently general and they can be applicable to a variety of databases, in particular to data warehouse implementations.

One of the powerful features of relational algebra is that it can be efficiently used in the assessment of denormalization effects. The approach clarifies denormalization effects in terms of mathematical methods and maps it to a specific query request. The methodological approach of the current study may, however, be somewhat restricted in that it focuses on the computation costs between normalized and denormalized data structures. In order to explicate the effects of denormalization, future studies are needed to develop a more comprehensive way of measuring the impact of denormalization on storage costs, memory usage costs, and communications cost.

It is important to note that although denormalization may speed up data retrieval, it may lead to additional update costs as well as slow the data modification processes due to the presence of duplicate data. In addition, since denormalization is frequently case-sensitive and reduces the flexibility of design inherent in normalized databases, it may hinder the future development [27]. It is, thus, critical for a database designer to understand the effects of denormalization before implementing the strategies.

While the results of this study are generally applicable to database design, the results are particularly applicable to data warehouse applications. Future research on denormalization should focus on a wide variety of operational databases as well as on examining the effects of denormalization on a variety of data models, including semantic, temporal and object-oriented implementations. Denormalizing databases is a critical issue because of the important trade-offs between system performance, ease of use, and data integrity. Thus, a database designer should not blindly denormalize data without good reason but should carefully balance the level of normalization with performance considerations in concert with the application and system needs.

Acknowledgement

The authors are grateful to the anonymous referees for their constructive suggestions and detailed comments on earlier drafts of this article.

References

[1] R. Barquin, H. Edelstein, Planning and designing the data warehousing, Prentice Hall, Upper Saddle River, NJ, 1997.
[2] D. Batra, G.M. Marakas, Conceptual data modelling in theory and practice, European Journal of Information Systems 4 (1995) 185–193.
[3] P. Beynon-Davies, Using an entity model to drive physical database design, Information and Software Technology 34 (12) (1992) 804–812.
[4] N. Bolloju, K. Toraskar, Data clustering for effective mapping of object models to relational models, Journal of Database Management 8 (4) (1997) 16–23.
[5] G. Bracchi, P. Paolini, G. Pelagatti, Data independent descriptions and the DDL specifications, in: B.C.M. Douque, G.M. Nijssen (Eds.), Data Base Description, North-Holland, Amsterdam, 1975, pp. 259–266.
[6] J.V. Carlis, S.T. March, G.W. Dickson, Physical database design: a DSS approach, Information & Management 6 (4) (1983) 211–224.
[7] N. Cerpa, Pre-physical data base design heuristics, Information & Management 28 (6) (1995) 351–359.
[8] C.H. Cheng, Algorithms for vertical partitioning in database physical design, OMEGA International Journal of Management Science 22 (3) (1993) 291–303.
[9] J. Clifford, A. Croker, A. Tuzhilin, On data representation and use in a temporal relational DBMS, Information Systems Research 7 (3) (1996) 308–327.
[10] E.F. Codd, A relational model of data for large shared data banks, Communication of the ACM 13 (6) (1970) 377–387.
[11] E.F. Codd, Relational completeness of data base sublanguages, in: R. Rustin (Ed.), Data Base Systems, Prentice-Hall, Englewood Cliffs, NJ, 1972, pp. 65–98.
[12] G. Coleman, Normalizing not only way, Computerworld 1989 (12) (1989) 63–64.
[13] C.E. Dabrowski, D.K. Jefferson, J.V. Carlis, S.T. March, Integrating a knowledge-based component into a physical database design system, Information & Management 17 (2) (1989) 71–86.
[14] M. Dadashzadeh, Improving usability of the relational algebra interface, Journal of Systems Management 40 (9) (1989) 9–12.
[15] C.J. Date, The normal is so. . .interesting, Database Programming & Design (1997 (November)) 23–25.
[16] C.J. Date, The birth of the relational model, Intelligent Enterprise Magazine (1998 (November)) 56–60.
[17] C.J. Date, An Introduction to Database Systems, Pearson Addison Wesley, Reading, MA, 2002.

[18] D.E. Denning, Secure statistical databases with random sample queries, ACM Transactions on Database Systems 5 (3) (1980) 291–315.
[19] D. Dey, S. Sarkar, Modifications of uncertain data: a Bayesian framework for belief revision, Information Systems Research 11 (1) (2000) 1–16.
[20] R. Elmasri, S.B. Navathe, Fundamental of Database Systems, 3rd ed., Addison-Wesley, New York, 2000.
[21] S. Finkelstein, M. Schkolnick, P. Tiberio, Physical database design for relational databases, ACM Transactions on Database Systems 13 (1) (1988) 91–128.
[22] S.F. Flood, Managing data in a distributed processing environment, in: B. von Halle, D. Kull (Eds.), Handbook of Data Management, Auerbach, Boston, MA, 1993, pp. 601–617.
[23] J. Hahnke, Data model design for business analysis, Unix Review 14 (10) (1996) 35–44.
[24] M. Hanus, To normalize or denormalize, that is the question, Proceedings of 19th International Conference for the Management and Performance Evaluation of Enterprise Computing Systems, San Diego, CA, 1994, pp. 416–423.
[25] J.A. Hoffer, M.B. Prescott, F.R. McFadden, Modern database management, 6th ed., Prentice Hall, Upper Saddle Review, NJ, 2002.
[26] W.H. Inmon, What price normalization? Computerworld 22 (10) (1988) 27–31.
[27] W.H. Inmon, Information Engineering for the Practitioner: Putting Theory into Practice, Yourdon Press, Englewood Cliffs, NJ, 1989.
[28] W.H. Inmon, Building the Data Warehouse, 2nd ed., Wiley & Sons Inc., New York, 1996.
[29] C.S. Jensen, R.T. Snodgrass, Temporal data management, IEEE Transactions on Knowledge and Data Engineering 11 (1) (1999) 36–44.
[30] M.Y. Kiang, A. Kumar, An evaluation of self-organizing map networks as a robust alternative to factor analysis in data mining applications, Information Systems Research 12 (2) (2001) 177–194.
[31] H. Lee, Justifying database normalization: a cost/benefit model, Information Processing & Management 31 (1) (1995) 59–67.
[32] S.T. March, Techniques for structuring database records, ACM Computing Surveys 15 (1) (1983) 45–79.
[33] D. Menninger, Breaking all the rules: an insider's guide to practical normalization, Data Based Advisor 13 (1) (1995) 116–120.
[34] S.B. Navathe, S. Ceri, G. Wiederhold, J. Dou, Vertical partitioning algorithms for database design, ACM Transactions on Database Systems 9 (4) (1984) 680–710.
[35] S.B. Navathe, M. Ra, Vertical partitioning for database design: a graphical algorithm, Proceedings of International Conference on Management of Data and Symposium on Principles of Database Systems, Portland, OR, 1989, pp. 440–450.
[36] G.M. Nijssen, T.A. Halpin, Conceptual Schema and Relational Database Design, Prentice-Hall, Sydney, 1989.
[37] P. Palvia, Sensitivity of the physical database design to changes in underlying factors, Information & Management 15 (3) (1988) 151–161.
[38] K. Paulsell, Sybase Adaptive Server Enterprise Performance and Tuning Guide, S.P. Group, Sybase, Inc., Emeryville, CA, 1999.
[39] V. Poe, L.L. Reeves, Building a Data Warehouse for Decision Support, 2nd ed., Prentice Hall, Upper Saddle River, NJ, 1998.
[40] M.J. Prietula, S.T. March, Form and substance in physical database design: an empirical study, Information Systems Research 2 (4) (1991) 287–314.
[41] U. Rodgers, Denormalization: why, what, and how? Database Programming & Design 1989 (12) (1989) 46–53.
[42] D. Rotem, Spatial join indices, Proceedings of Seventh International Conference on Data Engineering, 1991, pp. 500–509.
[43] G.L. Sanders, Data modeling, International Thomson Publishing, Danvers, MA, 1995.
[44] M. Schkolnick, P. Sorenson, Denormalization: a performance oriented database design technique, Proceedings of the AICA, Bologna, Italy, 1980, pp. 363–367.
[45] J. Schloerer, Disclosure from statistical databases: quantitative aspects of trackers, ACM Transactions on Database Systems 5 (4) (1980) 467–492.
[46] J.A. Stankovic, Misconceptions about real-time computing: a serious problem for next-generation systems, IEEE Computer 21 (10) (1988) 10–19.
[47] F. Sweet, Process-driven data design (Part 3): objects and events, Datamation (1985 (September)) 152.
[48] H. Thomas, A. Datta, A conceptual model and algebra for online analytical processing in decision support databases, Information Systems Research 12 (1) (2001) 83–102.
[49] C. Tupper, The physics of logical modeling, Database Programming & Design (1998 (September)) 56–59.
[50] J.C. Westland, Economic incentives for database normalization, Information Processing & Management 28 (5) (1992) 647–662.

Seung Kyoon Shin is an assistant professor of Information Systems in the College of Business Administration at the University of Rhode Island. Prior to joining academia, he worked in industry as a software specialist and system integration consultant. His research has appeared in *Communications of the ACM*, *Decision Support Systems*, and *International Journal of Operation & Production Management*. His current research interests include strategic database design for data warehousing and data mining, web-based information systems success, and economic and cultural issues related to intellectual property rights.

G. Lawrence Sanders, Ph.D., is professor and chair of the Department of Management Science and Systems in the School of Management at the State University of New York at Buffalo. He has taught MBA courses in the Peoples Republic of China and Singapore. His research interests are in the ethics and economics of digital piracy, systems success measurement, cross-cultural implementation research, and systems development. He has published papers in outlets such as *The Journal of Management Information Systems, The Journal of Business, MIS Quarterly, Information Systems Research*, the *Journal of Strategic Information Systems*, the *Journal of Management Systems, Decision Support Systems, Database Programming and Design*, and *Decision Sciences*. He has also published a book on database design and co-edited two other books.