Denormalization Effects on Performance of RDBMS

G. Lawrence Sanders
Management Science & Systems
State Univeristy of New York at Buffalo
mgtsand@mgt.buffalo.edu

Seungkyoon Shin
Management Science & Systems
State Univeristy of New York at Buffalo
ss27@acsu.buffalo.edu

Abstract

In this paper, we present a practical view of denormalization, and provide fundamental guidelines for incorporating denormalization. We have suggested, using denormalization as an intermediate step between logical and physical modeling to be used as an analytic procedure for the design of the applications requirements criteria. Relational algebra and query trees are used to examine the effect on the performance of relational systems. The guidelines and methodology presented are sufficiently general, and they can be applicable to most databases. It is concluded that denormalization can enhance query performance when it is deployed with a complete understanding of application requirements.

1. Normalization vs. Denormalization

Normalization is the process of grouping attributes into refined structures. The normal forms and the process of normalization have been studied by many researchers, since Codd [5] initiated the subject. First Normal Form (1NF), Second Normal Form (2NF), and Third Normal Form (3NF) were the only forms originally proposed by Codd, and they are the normal forms supported by commercial case tools. The higher form of normalization such as Boyce/Codd Normal Form, the Fourth Normal Form (4NF), and the Fifth Normal Form (5NF) are academically important but are not widely implemented.

The objective of normalization is to organize data into stable structures, and thereby minimize update anomalies and maximize data accessibility. Although normalization is generally regarded as the rule for the statue of relational database design, there are still times when database designers may turn to denormalizing a database in order to enhance performance and ease of use. Even though normalization results in many benefits, there is at least one major drawback – poor system performance [22],[13], [15], [24], [18], [7].

Normalization can also be used as a supplemental tool to provide an additional check on the stability and integrity of an Entity Relationship Diagram (ERD) and produce naturally normalized relational schemas. However, converting the conceptual entity-relationship model into database tables does not guarantee the best performance, nor the ideal geometrical distribution of the data [20]. Thus, even though conceptual data models encourage us to generalize and consolidate entities to better understand the relationships between them, such generalization can lead to more complicated database access path [2]. Furthermore, normalization can create retrieval inefficiencies where a comparatively small amount of information is being sought and retrieved from the database [24]. As a consequence, database designers occasionally trade off the aesthetics of data normalization with the reality of system performance.

Denormalization can be described as a process for reducing the degree of normalization with the aim of improving query processing performance. One of the main purposes of denormalization is to reduce the number of physical tables that must be accessed to retrieve the desired data by reducing the number of joins needed to derive a query answer [17].

Denormalization has been utilized in many strategic database implementations to boost database performance and reduce query response times. One of the most useful areas for applying denormalization techniques is in data warehousing implementations for data mining transactions. The typical data warehouse is a subject-oriented corporate database that involves multiple data models implemented on multiple platforms and architectures. The goal of the data warehouse is to put enterprise data at the disposal of organizational decision makers. The retrieval, conversion and migration of data from the source to the target computing environments is one of many tasks charged to the data warehouse administrator. The transformation of that data in a manner that ensures that the target database holds only accurate, timely, integrated, valid and credible data represents the most complex task on the technical agenda. This is also an area where the tools and techniques are not abundant. It is not enough to simply capture the data needed for the

data warehouse. It is also necessary to optimize the potential of the data warehouse.

Denormalization is particularly useful in dealing with the proliferation of star schemas that are found in many data warehouse implementations. In this case, denormalization provides better performance and a more intuitive data structure for data warehouse users to navigate [1]. The goal of most analytical processes against the data warehouse is to access aggregates such as sums, averages, and trends. 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.

Important and common components in a data warehouse that are good candidates for denormalization include: multidimensional analysis in a complex hierarchy, aggregation, and complicated calculations. Time is a good example of a multidimensional hierarchy (e.g. year, quarter, month, and date). Basic design decisions such as how many dimensions to define and what facts to aggregate can affect database size and query performance.

Although denormalization techniques have been utilized for various types of database design, denormalization is still one issue that lacks solid principles and guidelines. There has been little research related to illustrating how denormalization enhances database performance and reduces query response time. This paper aims at providing a comprehensive guideline regarding when and how to effectively exercise denormalization. The main contribution is to provide unambiguous principles for conducting denormalization. In regards to the organization of the paper, we start by giving an overview of prior research on denormalization in Section 2. A generally applicable denormalization process model and commonly accepted denormalization techniques are to be presented, in Sections 3 and 4. In Section 5 respectively, we use relational algebra and query trees to develop a mechanism to access the effect of denormalization on the performance of Relational Database Management Systems (RDBMS). Finally, Section 6 presents a summary of our findings.

2. Previous work on denormalization

It is commonly believed by database professionals and researchers that normalization degrades response time. A full normalization results in a number of logically separate entities that, in turn, result in even more physically separate stored files. The net effect is that join processing against normalized tables requires an additional amount of system resources.

Normalization may also 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. However, this has not been the case. The pioneering work by Schkolnick and Sorenson [22] introduced the notion of denormalization. In that work, They argues that, to improve performance quality of a database design, the data model viewed by the user has to be capable of notifying the user about semantic constraints. The goals of this paper are to illustrate how semantic constraints can be exhibited and to illustrate an algorithm to carry out the denormalization process.

Hanus [12] developed a list of normalization and denormalization types, and suggested that denormalization should be carefully used according to how the data will be used. A limitation of this work is that the approach to be used in denormalization was not described in sufficient detail.

Tupper [23] proposed two separate dimensions for denormalization. In the first approach, the 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 in this approach is that future development and maintenance are not considered.

Date [8] argued that denormalization should be done at the physical storage level, not at the logical or base relation level. He argued further that denormalization usually has the side effect of corrupting a clear logical design with undesirable consequences.

Hahnke [11] illustrated the positive effects of denormalization on analytical business applications. In designing analytical systems, a more denormalized data model provides better performance and a more intuitive data structure for users to navigate. The two primary components of denormalized data models, facts and dimensions, are also provided to solve the complexity of hierarchies that are an important component of multidimensional analysis when they support drill-down functions in navigation functions. (Facts are the data elements of interest contained in the result set returned by

a query and dimensions define the constraints used to select the facts.)

Cerpa [4] proposes a pre-physical database design process as an intermediate step between logical and physical modeling and provides a practical view of logical database refinement before the physical design. He maintained that the refinement process requires a high level of expertise on the part of the database designer, as well as appropriate knowledge of application requirements.

Rodgers [20] discusses the general trade-offs of denormalization and some of the more common situations in which a database designer should consider denormalization. For example, denormalization is useful when there are two entities with a One-to-One relationship and a Many-to-Many relationship with non-key attributes.

Bolloju and Toraskar [2] present an approach to avoid or minimize the need for denormalization. They introduce the concept of data clustering as an alternative to denormalization. This approach is important in view of the current popularity of object-oriented techniques for information system analysis and design. One problem with the use of data clustering is that it is limited to a particular type of physical data organization.

Another possible drawback of denormalization relates to flexibility. Coleman [6] argues that denormalization decisions usually involve the 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.

It is apparent from the previous discussions that there has been little research regarding a comprehensive taxonomy and procedure model for the denormalization process. The next question of this paper sets the stage for the development of an analytical approach for denormalization.

3. A denormalization process model

As noted above, the primary goals of denormalization are to improve query performance and to present the end-user with a less complex and more user-oriented view of data. This is in part accomplished by reducing the number of physical tables and reducing the number of actual joins necessary to derive the answer to a query.

As a rule, denormalization should be considered only when performance is an issue and then only after there has been a thorough analysis of the various impacted systems. For example, if additional system development is under way, denormalization may not be appropriate and could lead to additional data anomalies and reduce flexibility, integrity, and accessibility. Consequently, denormalization should be deployed only when performance issues indicate that it is needed, and then only after there has been a thorough analysis of the problem domain and the application requirements.

It has been asserted by Inmon [14] that data should be firstly normalized as the design is being conceptualized and then denormalized in response to the performance requirements. Prior to the denormalization procedure, the database designer should develop a logical entity relationship model that indicates; 1) cardinality for each relationship, 2) volume estimation for each entity, and defines 3) process decompositions for the application. Point 3 could be accomplished via data flow diagrams. Figure 1 illustrates the up-front activities that that should be attended to before proceeding to denormalization.

- Development of a conceptual data model (ER Diagram).
- Refinement and Normalization.
- Identifying candidates for denormalization.
- Determining the effect of denormalizing entities on data integrity.
- Identifying what form the denormalized entity may take.
- Map conceptual scheme to physical scheme.

Figure 1. DB Design Cycle with Denormalization.

A typical implementation of the database design process includes the following phases: conceptual database design, logical database design, and physical database design [21]. Conceptual data models encourage the generalization and consolidation of entities. In practice, however, the generalization process leads to more complex database access paths. Physical database design means merely converting the conceptual entity-relationship model into database tables. However, this approach does not account for the trade-offs necessary for performance or for geographic distribution of the database. Therefore, denormalization can be separated from both steps because it involves aspects that are neither purely logical nor purely physical. We propose that the denormalization process should be implemented between the data model mapping and the physical database design so that the procedure can be based on logical and physical database design.

- General application performance requirements indicated by business needs.
- On-line response time requirements for application queries, updates ad processes.
- Minimum number of data access paths.
- Minimum amount of storage.

Figure 2. Criteria for Denormalization

The criteria for denormalization should address both logical and physical issues. After an exhaustive review of journal articles and experts' recommendations in professional journals, we have identified four criteria (Figure 2) that have been used as the rationale for turning to denormalization. These criteria are focused on reducing database access costs, and are affected by database activity, computer system characteristics and physical factors [19]. They also take into account that the dynamics of applications requires that a database design should be reviewed according to maintenance and development plans for the application system.

A primary goal of denormalization relates to how it improves the effectiveness and efficiency of the logical model implementation and how it fulfills application requirements. This requires an analysis of the advantages and disadvantages of possible model implementation. 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 should exploit knowledge about the application requirements in order to evaluate the degree of importance of each criterion in conflict.

Finally, in Figure 3, in order to contribute to database designers' seamless knowledge in addition to criteria previously specified, we identified a list of variables from literature [10] [3] that should be taken into account when considering an absolute denormalization.

Denormalization has many drawbacks, such as data duplication, more complex data-integrity rules, update anomalies, and increased difficulty in expressing the type of access [16], [20]. The denormalization process can easily lead to data duplication and that in turn leads to update anomaly issues requiring increased database storage requirements.

An update anomaly problem can generally be resolved with database management techniques such as triggers, application logic, and batch reconciliation [9]. A trigger can update duplicated or derived data anytime the base data changes. Triggers provide the best solution from an integrity point of view, although they can be costly in terms of performance. In addition, application logic can be included into transactions in each application in order to update denormalized data to ensure that changes are atomic. Finally, a batch reconciliation process can be run at appropriate intervals to bring the denormalized data back into agreement. Using application logic to manage denormalized data is risky, because the same logic must be used and maintained in all applications that modify the data.

Denormalization usually speeds up retrieval, but it can slow the data modification processes. It is noteworthy that both on-line and batch system performance is adversely affected by a high degree of normalization [15]. The golden rule is: When in doubt, don't denormalize. The next section of this paper will provide a strategy for increasing performance, but at the same time, minimizing the deleterious effects related to denormalization.

- Application performance criteria.
- Future application development and maintenance considerations.
- Volatility of application requirements.
- Relations between transactions and relations of entities involved.
- Transaction type (update/query, OLTP/OLAP).
- Transaction frequency.
- Access paths needed by each transaction.
- Number of rows accessed by each transaction.
- Number of pages/blocks accessed by each transaction.
- Cardinality of each relation.

Figure 3. Other Considerations of Denormalization

4. Denormalizing for performance

In this Section, we discuss denormalization patterns that have been commonly adopted by experienced database designers. After an exhaustive literature review, we identified and classified four prevalent strategies for denormalization in Figure 4. They are collapsing tables, splitting a table, adding redundant columns and adding derived columns.

- Collapsing Tables.
  - Two entities with a One-to-One relationship.
  - Two entities with a Many-to-Many relationship.
- Splitting Tables (Horizontal/Vertical Splitting).
- Adding Redundant Columns (Reference Data).
- Derived Attributes (Summary, Total, and Balance).

Figure 4. Denormalization Strategies.

4.1. Collapsing Tables

One of the most common and secure denormalization techniques is the collapsing of One-to-One relationships. This situation occurs when for each row of entity A, there is only one related row in entity B. While the key attributes for the entities may or may not be the same, their equal participation in a relationship indicates that they can be treated as a single unit. For example, if users frequently need to see COL1, COL2, and COL3 at the same time and the data from the two tables is in a One-to-One relationship, the solution is to collapse the two tables into one (See Figure 5).

There are several nice advantages of this technique in the form of reduced number of foreign keys on tables, reduced number of indexes (since most indexes are created based on primary/foreign keys), reduced storage space, and reduced amount of time for data modification. Moreover, combining the attributes does not change the business view but does decrease access time by having fewer physical objects and reducing overhead. In general, collapsing tables in One-to-One relationship has fewer drawbacks than others.

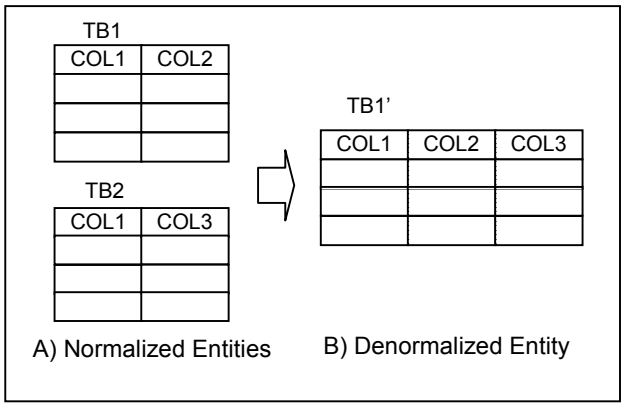

Figure 5. Collapsing Tables.

A Many-to-Many relationship can also be a candidate for the table collapsing. The typical Many-to-Many relationship is represented in the physical database structure by three tables: one table for each of two primary entities and another table for cross-referencing them. A cross-reference or intersection between the two entities in many instances also represents a business entity. These three tables can be merged into two if one of the entities has little data apart form its primary key (i.e. there are not many functional dependencies with the primary key). Such an entity could be merged into the cross-reference table by duplicating the attribute data. There is of course a drawback to this approach. Because data redundancy may interfere with updates, update anomalies may occur when the merged entity has instances that do not have any entries in the cross-reference table. Collapsing the tables in both One-to-One and One-to-Many eliminates the join, but the net result is that there is a significant loss at the abstract level because there is no conceptual separation of the data. In general, collapsing tables in Many-to-Many relationship has a significant number of problems compared to other denormalization approaches.

4.2. Splitting Tables

When separate parts of a table are used by different applications, the table may be split into a denormalized table for each application processing group. In this case, the table can be split either vertically or horizontally. However, it should be noted that there are cases that the whole table should be able to be used for a single transaction.

A vertical split involves splitting a table by columns so that a group of columns is placed into the new table and the remaining columns are placed in another new table (Figure 6). Vertical splitting can be used when some columns are rarely accessed rather than other columns or when the table has wide rows. The net result of splitting a table is that it may reduce the number of pages/blocks that need to be read because of the shorter row length in the main table. With more rows per page, I/O is decreased when large numbers of rows are accessed in physical sequence. A vertically split table should contain one row per primary key in the split tables as this facilitates data retrieval across tables. In actuality, a view of the joined tables may make this split transparent to the users. This technique has been particularly effective when there are lengthy text fields in a long row. If the text fields are rarely accessed, they may be placed in a separate table.

A horizontal split involves a row split, resulting rows classified into groups by key ranges (Figure 6). This is similar to partitioning a table, except that each table has a different name. A horizontal split can be used when a table is large, and reducing its size reduces the number of index pages read in a query. B-tree indexes are generally very flat, and large numbers of rows can be added to a table with small index keys before the B-tree requires more levels. However, an excessive number of index levels may be an issue with tables that have very large keys. Thus, as long as the index keys are short, and the indexes are used for queries on the table rather than on whole table scan, a horizontal split prevents doubling or tripling the number of rows in the table. The result is that the number of disk reads required for a query by only one index level is reduced.

A horizontal split is usually applied when the table split corresponds to a natural separation of the rows such

as different geographical sites or historical vs. current data. It can be used when there is a table that stores a large amount of rarely used historical data, and at the same time when there are applications that require a quick result from the data. 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, horizontal splitting adds a high degree of complexity to applications. Consider that it usually requires different table names in queries, according to the values in the tables. This complexity alone usually outweighs the advantages of table splitting in most database applications.

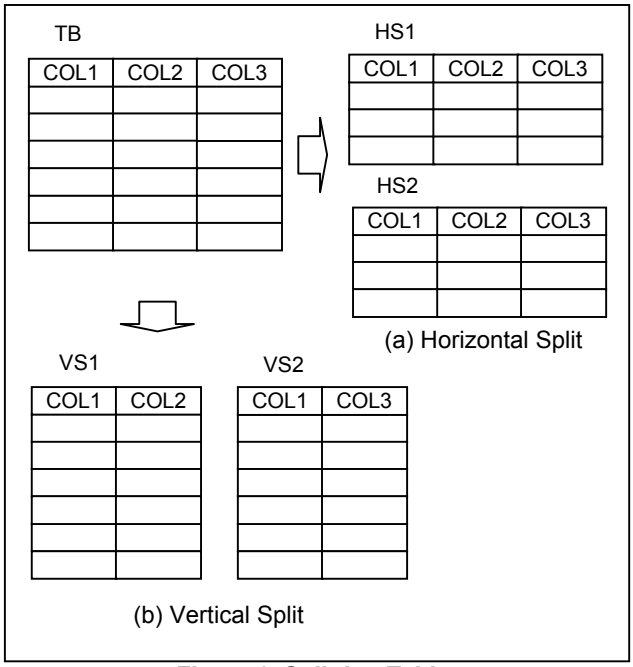

Figure 6. Splitting Tables.

4.3. Adding Redundant Columns

Adding redundant columns can be used when a column from one table is being accessed in conjunction with a column from another table. If this occurs frequently, it may be necessary to combine the data and carry them as redundant. For example, if frequent joins are performed on the TB1 and TB2 in order to retrieve COL1, COL2, and COL3, it is an appropriate strategy to add COL3 to TB1 (Figure 7).

Reference data, which relates to sets of code in one table and a description in another, obtaining a description of a code is accomplished via a join, presents a natural case where redundancy can pay off. A typical strategy with reference data tables is to duplicate their descriptive attribute in the entity, which would otherwise contain only the 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.

Denormalizing reference data tables does not necessarily reduce the total number of tables because, in most case, 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 duplicated in each target entity. In this case, the goal should be to reduce redundant data or keep only the columns necessary to support the redundancy and infrequently updated columns used by many queries [12].

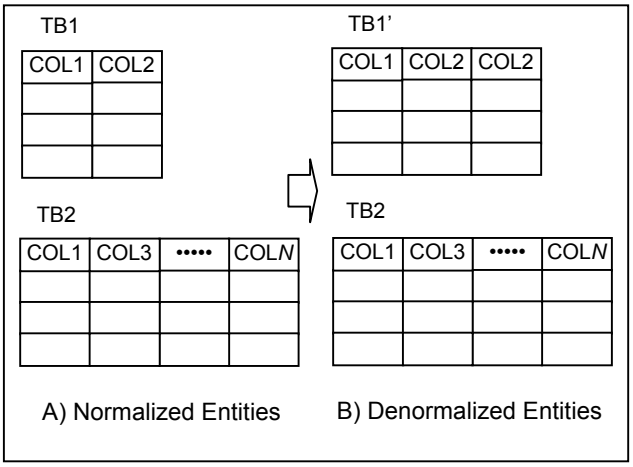

Figure 7. Adding Redundant Columns.

4.4. Derived Attributes

Summary data and derived data are extremely important in the Decision Support Systems (DSS) environment. The nature of many applications dictates frequent use of data calculated from naturally occurring information. On large volumes of data, 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 is seriously affected.

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. Storing derived data can help eliminate joins and reduce time-consuming calculations at runtime. This technique can be combined effectively with other forms of denormalization. Maintaining data integrity can be complex if the data items used to calculate a derived data item change unpredictably or

frequently, as the new value for derived data must be recalculated each time a component data item changes. Thus, the frequency of access, required response time, and increased maintenance costs must be considered.

Data retention requirements for summary and detail records may be quite different and must be considered, and the level of granularity for the summarized information must be carefully designed. 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.

4.5 Using Views as an Alternative to Denormalization

Using a view is not a pattern of materialized denormalization technique, but it can help to reduce the drawbacks of denormalization as well as it is the most natural way to start the denormalization of a database. Views are virtual tables that are defined as a database object containing references to other items in the database. Since no data is stored in the view, the integrity of the data isn't an issue. Also, because data is stored only once, the amount of disk spaced required is minimal.

However, since most DBMS products process view definitions at run time, a view does not solve performance issues, but can solve some of ease-of-use issues. A query processing through a view would be naturally slower than a query that references base tables in direct manner, as the query optimizer of a DBMS server cannot be as intelligent with views as it can be without them. Despite recent advances in DBMS technologies that permit a database programmer to choose the appropriate access path, it turns out to be quite complicated and even inefficient when applied it to views that is associated with many large tables. In order for a user to use such technology, s/he must be able to identify a better data access path than the query optimizer. The advantage of these technologies also can be realized only when the database programmer can identify an index that provides an access path to a certain query. In such cases that the application designer can force the query optimizer to use a particular access path, the resulting query can be more efficient. Finally, since a view is a combination of pieces of multiple tables, data updates through a view can be restricted if a view references multiple tables that are not related vis-a-vis One-to-One relationship.

5. Justifying denormalization

There is considerable complexity of the decision-making process arising from the need to find out the criteria for denormalization. Some guidelines to follow during denormalization processes, which are not theory-based, but are derived from practical experience in designing relational database, were presented in the previous section. In order to obtain a thorough knowledge of applications and query processes, the most fundamental procedures that a database designer should go through are to 1) identify relations and their cardinalities, 2) identify the transaction/module relations being accessed by them, and 3) identify the access paths required by each transaction/module.

In this Section, we present a theory-based method for assessing the effects of denormalization. Using relational algebra operations and query trees, we provide a simple, but precise assessment of denormalization effects based on retrieval requests.

The relational algebra operations enable the user to specify basic retrieval requests with a basic set of relational model operations [9]. The algebra operations produce new relations, which can be further manipulated using operations of the same algebra. A sequence of relational algebra operations forms a relational algebra expression, whose result will also be a relation.

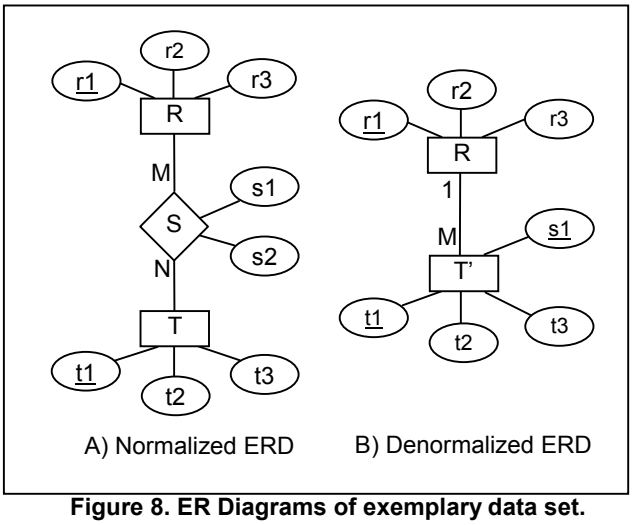

Figure 8. ER Diagrams of exemplary data set.

We apply the relational algebra operation to an example of denormalization for a Many-to-Many relationship. Suppose we're given three 3NF normalized entities, R = {r1, r2, r3}, S = {s2, s2}, and T = {t1, t2, t3}, where S is a relationship between R and T, and r1, s1, s2, and t1 are super-keys of each entity R, S, and T respectively. Notice that s1, s2 are foreign keys of entity R, T, respectively. The ERD of normalized data structure is provided in Figure 8 (A), and the ERD of denormalized data structure is provided in Figure 8 (B).

Two sets of SQL and relational algebra operations for the queries that retrieve the same information from each data structures are shown in Figure 9. The queries retrieve all tuples whose r1 is equal to '001' from each data structure. Suppose the cross-reference S stores only foreign keys for R and T (or with a few attributes), we may want to merge S into T as suggested in Section 4 such that we have two entities, R = {r1, r2, r3} and T' = {t1, t2, t3, s1}, after denormalization. The ERD of the denormalized data structure is provided as (B) in Figure 8.

(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;
```

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

Figure 9. SQL and Relational Algebra

In Figure 10, we present two query trees that respectively correspond to the relational algebra expressions in Figure 9. A tree query characterizes 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 operations as internal nodes.

Suppose there are 1,000 tuples in R, 2,000 tuples in S, and 500 tuples in T. Also, if there are 10 tuples retrieved from R, which satisfying operation condition ($\bullet_{r1 = '001'}$ (R)), the number of combination of both entities caused by the first join operation (Res1 $\bullet_{r1 = s1}$ (S)) comes to 20,000 (= 10 * 2,000). Likewise, if there are 20 tuples retrieved from the first join operation, the final join operation (Res2 $\bullet_{s2 = t1}$ (T)) results in 10,000 (= 20 * 500) combinations. Thus, the total number of combinations produced from the normalized data structure is 30,000.

Now, suppose we have 2,000 tuples in a denormalized T' (since S has been merged into T, the number of tuples in T' should be identical to the number of tuples in S). For the identical information retrieval, the total number of combinations produced in the denormalized data structure is 20,000 (10 * 2,000). Thus, with this example, we conclude that the exemplary data retrieval process is more expensive with the normalized data structure.

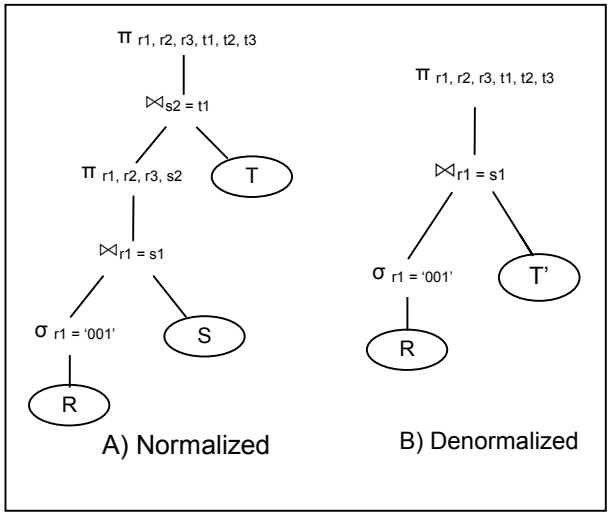

Figure 10. Comparison Using Query Trees.

6. Concluding remarks

In this paper, we have presented a practical view of denormalization with fundamental guidelines to be used in database design. We have also presented a methodology for the assessment of denormalization effects, using relational algebra operations and query trees. We have proposed denormalization as an intermediate step between logical and physical modeling, which is concerned with analyzing the functionality of the design with regards to the applications requirements criteria.

The guidelines and methodology are sufficiently general so that they can be applicable to most databases. It is concluded that denormalization can enhance query performance when it is deployed with a complete understanding of application's requirements.

One of the powerful features of relational algebra is that it can be 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. There are some limitations to the approach.

The methodology is limited because it compares computation costs between normalized and denormalized data structures. In order to explicate denormalization effects in greater detail, we still need to invest effort in creating a method for measuring other effects such as storage costs, memory usage costs, and communication cost.

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 balance the level of normalization with performance considerations in concert with the application and system needs.

7. References

[1] R. Barquin, Edelstein, H., *Planning and Designing the Data Warehousing*, Upper Saddle River, New Jersey: Prentice Hall, 1997.

[2] N. Bolloju, Kranti Toraskar, "Data clustering for effective mapping of object models to relational models," *Journal of Database Management*, vol. 8, no. 4, 1997, pp. 16-23.

[3] J.V. Carlis, C.E. Dabrowski, D.K. Jefferson, and S.T. March, "Integrating a Knowledge-Based Component into a Physical Database Design System," *Information Management*, vol. 17, no. 2, 1989, pp. 71-86.

[4] N. Cerpa, "Pre-physical data base design heuristics," *Information Management*, vol. 28, no. 6, 1995, pp. 351-359.

[5] E. Codd, "Relational Completeness of Data Base Sublanguages," in R. Rustin, ed., *Data Base Systems*, Englewood Cliffs, NJ: Prentice-Hall, 1972.

[6] G. Coleman, "Normalizing not only way," *Computerworld*, December, 1989, pp. 63 - 64.

[7] C.J. Date, "The Normal is so ... Interesting," *Database Programming & Design*, November, 1997, pp. 23 - 25.

[8] C.J. Date, The Birth of the Relational Model, *Intelligent Enterprise Magazine*, November, 1998,

[9] R. Elmasri, Bavathe, S. B, *Fundamental of Database Systems*, 3 ed., New York: Addison-Wesley, 2000.

[10] S. Finkelstein, M. Schkolnick, and P. Tiberio, "Physical Database Design For Relational Databases," *ACM Transactions on Database Systems*, vol. 13, no. 1, 1988, pp. 91-128.

[11] J. Hahnke, "Data model design for business analysis," *Unix Review*, Vol. 14, No. 10, September 1996,

[12] M. Hanus, "To normalize or denormalize, that is the question," *CMG Proceedings*, no. 1, 1994, ubl by CMG, Chicago, IL, USA.

[13] W.H. Inmon, "Denormalize for Efficiency," *Computerworld*, vol. 21, no. 11, 1987, pp. 19-21.

[14] W.H. Inmon, What price normalization?, *Computerworld*, October 1988, pp. 27 - 31.

[15] W.H. Inmon, *Information Engineering for the Practitioner: Putting Theory into Practice*, Englewood Cliffs, New Jersey: Yourdon Press, 1989.

[16] H. Lee, "Justifying database normalization: A cost/benefit model," *Information Processing & Management*, vol. 31, no. 1, 1995, pp. 59-67.

[17] F.R. McFadden, Jeffrey A. Hoffer, *Modern Database Management*, 4th ed., Redwood City, California: The Benjamin/Cummings Publishing Company, 1994.

[18] D. Menninger, "Breaking all the rules: an insider's guide to practical normalization.," *Data Based Advisor*, vol. 13, no. 1, 1995, pp. 116 - 120.

[19] P. Palvia, "Sensitivity of the Physical Database Design to Changes in Underlying Factors," *Information Management*, vol. 15, no. 3, 1988, pp. 151-161.

[20] U. Rodgers, "Denormalization: Why, What, and How?," *Database Programming & Design*, Vol. December, 1989, pp. 46 - 53.

[21] G.L. Sanders, *Data Modeling*, Danvers, MA: International Thomson Publishing, 1995.

[22] M. Schkolnick, Sorenson, P., "Denormalization: A Performance Oriented Database Design Technique.," *In Proceedings of the AICA 1980 Congress (Bologna, Italy)*, AICA, Brussels, 1980, pp. 363 - 367.

[23] C. Tupper, "The Physics of Logical Modeling," *Database Programming & Design*, September, 1998,

[24] J.C. Westland, "Economic Incentives for Database Normalization," *Information Processing & Management*, vol. 28, no. 5, 1992, pp. 647-662.