# Physical Databases Design
# Tuning Techniques
# &
# Procedures

**Anirudhan Velur**
**Data Architecture Center of Excellence**
**Cognizant Technology Solutions**
**25 Feb 2008**

## DOCUMENT HISTORY

| Version | Date | Author | Comments |
|---|---|---|---|
| 0.01 | 04 Feb 2008 | Anirudhan Velur | Initial Draft |
| 0.02 | 25 Feb 2008 | Anirudhan Velur | Updates as per review comments |
| | | | |

# 1 INTRODUCTION

Physical Database Design Performance tuning/ De-normalization is the process of attempting to optimize the performance of a database by adding redundant data. It is sometimes necessary because current DBMSs implement the relational model poorly. A true relational DBMS would allow for a fully normalized database at the logical level, while providing physical storage of data that is tuned for high performance.
A normalized design will often store different but related pieces of information in separate logical tables (called relations). If these relations are stored physically as separate disk files, completing a database query that draws information from several relations (a join operation) can be slow. If many relations are joined, it may be prohibitively slow. There are two strategies for dealing with this. The preferred method is to keep the logical design normalized, but allow the DBMS to store additional redundant information on disk to optimize query response.

This paper will present such a systematic approach for improving performance by adding some controlled redundancy. Some design tuning techniques like denormalization techniques are examined to deal with different data situations and some controlled data redundancy strategies are discussed to enable top take appropriate decisions

# 2 PURPOSE AND SCOPE

The purpose and scope of this document is to highlight the areas to be focused while finalizing a physical database design. This document will give you some insight on the methodologies that can be used for tuning the physical data model for performance

# 3 TARGET AUDIENCE

This document is written for Data Modelers, Database Administrators, Project Leaders and Project Managers to help them review, validate, and decide on their physical database design

# 4 DISCLAIMER

The points/methodologies suggested in this document are not the only way for physical database design tuning. This document explains some common methodology used for physical database design tuning . These are not standards suggested by any standard body.

# 5 REASONS FOR PHYSICAL DESIGN TUNING

During physical design the database design needs to be tuned for better performance. For fine tuning a design Denormalization can be used. The denormalization is a technique to

move from higher to lower normal forms of database modeling in order to speed up database access. The valid reason exists for de-normalizing/ adding redundancy a relational design are

- For Ad-hoc reporting
- For Better Query Performance

## 5.1 Ad-Hoc Reporting

Another reason to denormalize a database is to simplify ad-hoc reporting. Ad-hoc reporting is the unstructured reporting and querying performed by end users. End users are often confused when they have to join a significant number of tables. To avoid the confusion, DBAs can create a special set of tables designed for ad-hoc reporting. If the data is used for reporting and not online processing, you can avoid some of the problems associated with a denormalized design.

## 5.2 For better Query Performance

Once you have created your database in normalized form, you can perform benchmarks and back away from normalization to improve performance for specific queries or applications.

There are several indicators which will help to identify systems and tables which are potential denormalization candidates. These are:

- Many critical queries and reports exist which rely upon data from more than one table. Often times these requests need to be processed in an on-line environment.
- Repeating groups exist which need to be processed in a group instead of individually.
- Many calculations need to be applied to one or many columns before queries can be successfully answered.
- Tables need to be accessed in different ways by different users during the same timeframe.
- Many large primary keys exist which are clumsy to query and consume a large amount of DASD when carried as foreign key columns in related tables.
- Certain columns are queried a large percentage of the time. Consider 60% or greater to be a cautionary number flagging denormalization as an option.

# 6 PHYSICAL DATABASE DESIGN TUNING TECHNIQUES

The following methodologies can be used for fine tuning a Database design for better performance

- Denormalization
- Duplicating Tables
- Splitting of tables

## 6.1 Denormalization

A normalized set of relational schemes is the optimal environment and should be implemented for whenever possible. Yet, in some situations, denormalization may be necessary. Retrieval performance needs dictate very quick retrieval capability for data stored in relational databases, especially more and more access to database for reporting, where users concern more prompt responses than an optimum design of databases. With respective to performance of retrieval, denormalization is not necessarily a bad decision if implemented following a systematic approach to large scale database, where there are dozens of relational tables.

### 6.1.1 Rationale behind De-normalization/adding Redundancy

The following issues should be considered before Denormalizing/ adding redundancy.

- Can the system achieve acceptable performance *without* Denormalizing/ adding redundancy?
- Will the performance of the system *after* Denormalizing/ adding redundancy still be unacceptable?
- Will the system be less reliable due to denormalization/ adding redundancy?

If the answer to any of these questions is "yes," then you should avoid denormalization / adding redundancy because any benefit that is accrued will not exceed the cost. If, after considering these issues, you decide to denormalize be sure to adhere to the general guidelines that follow. The most prevalent denormalization techniques are:

- Adding redundant columns
- Adding derived columns
- Collapsing tables
- Roll up – Generalization Hierarchy

In addition, you can duplicate or split tables to improve performance. While these are not denormalization techniques, they achieve the same purposes and require the same safeguards.

### 6.1.2 Adding Redundant Columns

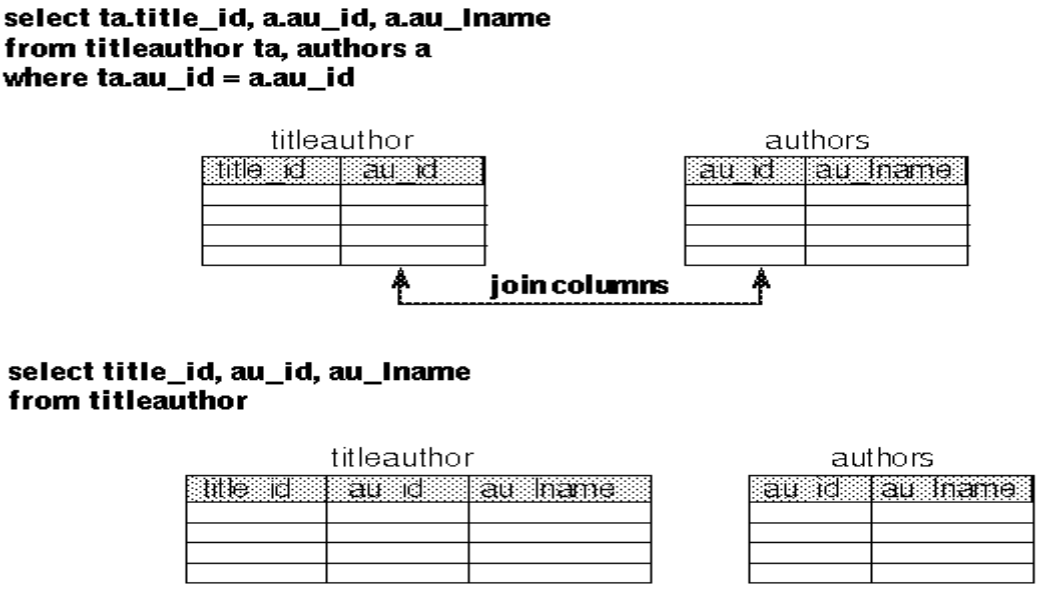
You can add redundant columns to eliminate frequent joins. Sometimes one or more columns from one table are accessed whenever data from another table is accessed. If these columns are accessed frequently with tables other than those in which they were initially defined, consider carrying them in those other tables as redundant data. By carrying these additional columns, joins can be eliminated and the speed of data retrieval will be enhanced .For example, if frequent joins are performed on the *titleauthor* and *authors* tables in order to retrieve the author's last name, you can add the *au_lname* column to *titleauthor*.

.The ideal candidates for redundant duplication are table columns that meet the following criteria:

1. The introduction of redundancy will eliminate the need to repeatedly join two tables together.
2. only a few columns are necessary to support the redundancy
3. the columns should be stable, being updated only infrequently

4. The columns should be used by either a large number of users or a few very important users

Figure 2-10: Denormalizing by adding redundant columns




Adding redundant columns eliminates joins for many queries. The problems with this solution are that it:

- Requires maintenance of new columns. All changes must be made to two tables, and possibly to many rows in one of the tables.
- Requires more disk space, since *au_lname* is duplicated.

### 6.1.3 Adding Derived Columns

Adding derived columns can eliminate some joins and reduce the time needed to produce aggregate values. If the cost of deriving data using complicated formulae is prohibitive then consider storing the derived data in a column instead of calculating it. However, when the underlying values that comprise the calculated value change, it is imperative that the stored derived data also be changed otherwise inconsistent information could be reported. This will adversely impact the effectiveness and reliability of the database.

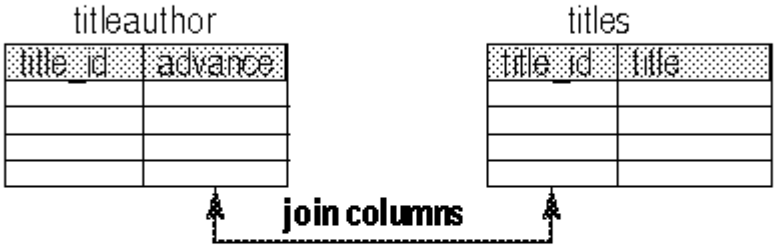
Sometimes it is not possible to immediately update derived data elements when the columns upon which they rely change. This can occur when the tables containing the derived elements are off-line or being operated upon by a utility. In this situation, time the update of the derived data such that it occurs immediately when the table is made available for update. Under no circumstances should outdated derived data be made available for reporting and inquiry purposes.

The *total_sales* column in the *titles* table of the *pubs2* database provides one example of a derived column used to reduce aggregate value processing time.

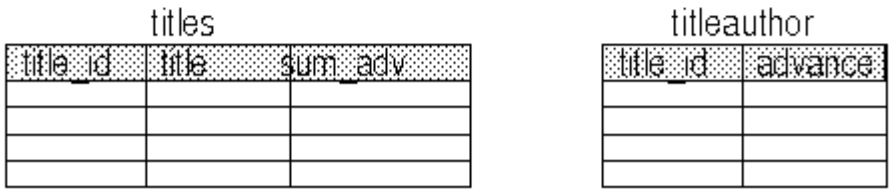
The example in Figure 3.1.2 shows both benefits. Frequent joins are needed between the *titleauthor* and *titles* tables to provide the total advance for a particular book title.

**Figure 3.1.2: Denormalizing by adding derived columns**

```
select title, sum(advance)
from titleauthor ta, titles t
where ta.title_id = t.title_id
group by title_id
```




```
select title, sum_adv
from titles
```



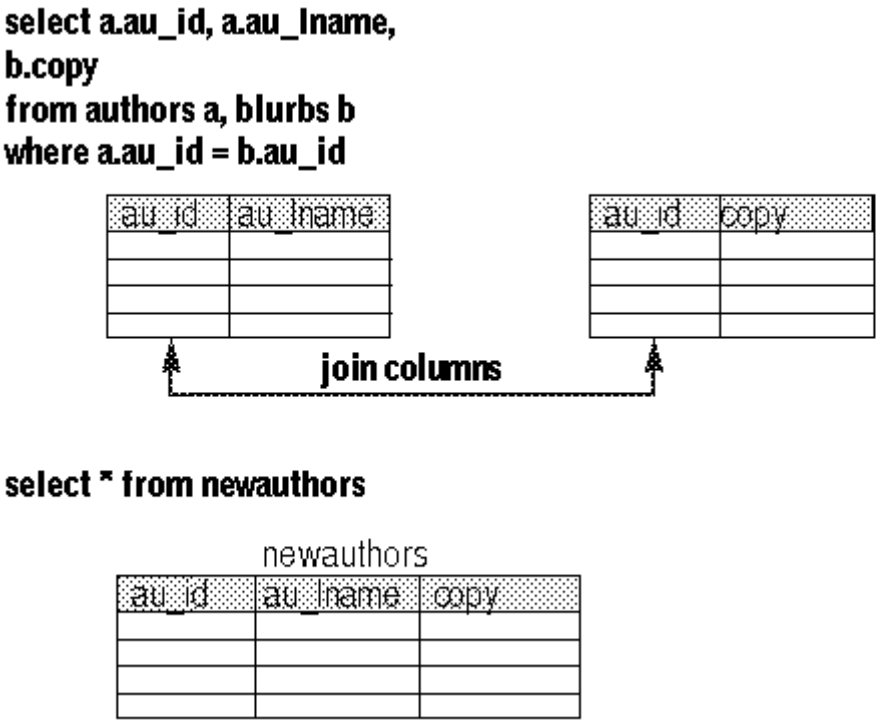

You can create and maintain a derived data column in the *titles* table, eliminating both the join and the aggregate at run time. This increases storage needs, and requires maintenance of the derived column whenever changes are made to the *titles* table.

### 6.1.4 Collapsing Tables

If most users need to see the full set of joined data from two tables, collapsing the two tables into one can improve performance by eliminating the join.

For example, users frequently need to see the author name, author ID, and the *blurbs* copy data at the same time. The solution is to collapse the two tables into one. The data from the two tables must be in a one-to-one relationship to collapse tables.

**Figure 3.1.3: Denormalizing by collapsing tables**

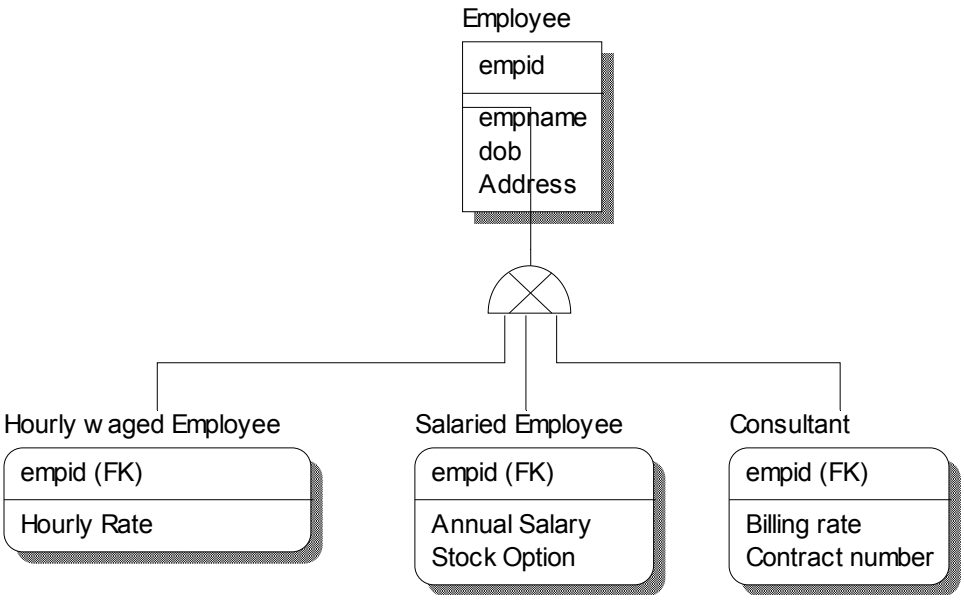



Collapsing the tables eliminates the join, but loses the conceptual separation of the data. If some users still need access to just the pairs of data from the two tables, this access can be restored by using queries that select only the needed columns or by using views.

### 6.1.5 Roll up – Generalization Hierarchy

In a generalization hierarchy, all common attributes are assigned to the super type. The super type is also assigned an attribute, called a discriminator, whose values identify the categories of the subtypes. Attributes unique to a category, are assigned to the appropriate subtype. Each subtype also inherits the primary key of the super type. Subtypes that have only a primary key should be eliminated. Subtypes are related to the super types through a one-to-one relationship.

**Example- Generalization Hierarchy**

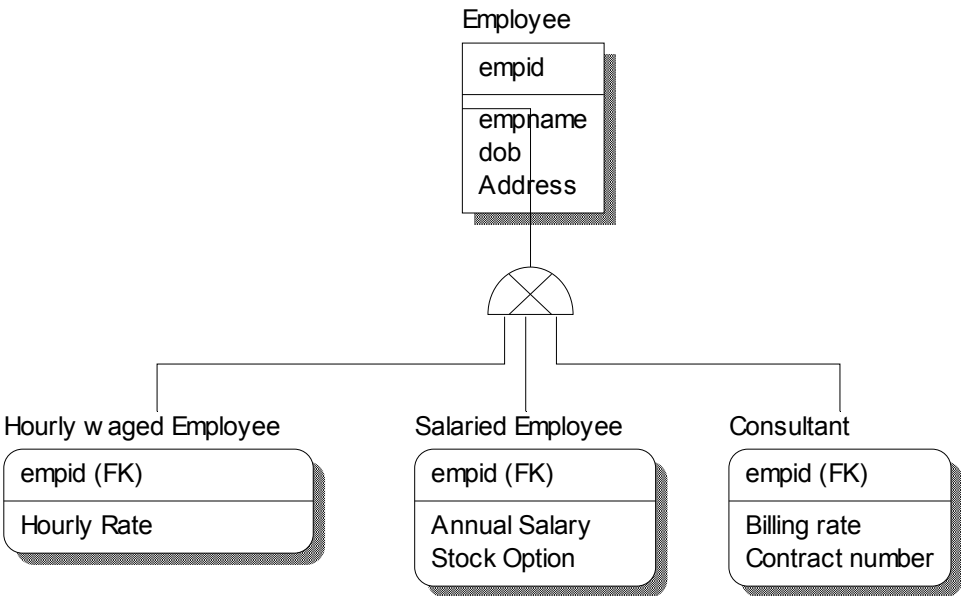



The Generalization hierarchy can be roll up /row down depend on the conditions

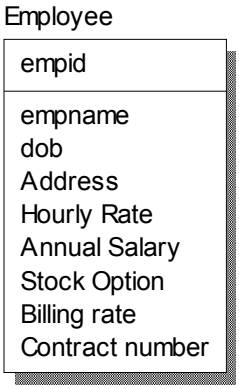
- Rolling up: Remove the subtypes and copy all of the data elements and relationships from each subtype to the super type. Also add a type code to distinguish the subtypes.
- Rolling down: Remove the super type entity and copy all of the data elements and relationships from the super type to each of the subtypes.

By Rolling up ( De-normalizing/ adding redundancy) the designer forced to make all of my sub type columns are moved to Super type to make it de-normalized. Means that some columns are nullable depend up on the type of sub type.

**Before Roll up**




**After Roll up**




The employee table is now denormalized. Here the hourly rate is filled up only If the employee is daily waged and Annual salary is filled up if the employee is salaried employee

### 6.1.6 Advantages of Denormalization

Denormalization can improve performance by:

- Minimizing the need for joins
- Reducing the number of foreign keys on tables
- Reducing the number of indexes, saving storage space, and reducing data modification time

- Pre-computing aggregate values, that is, computing them at data modification time rather than at select time
- Reducing the number of tables (in some cases)

### 6.1.7 Disadvantages of Denormalization

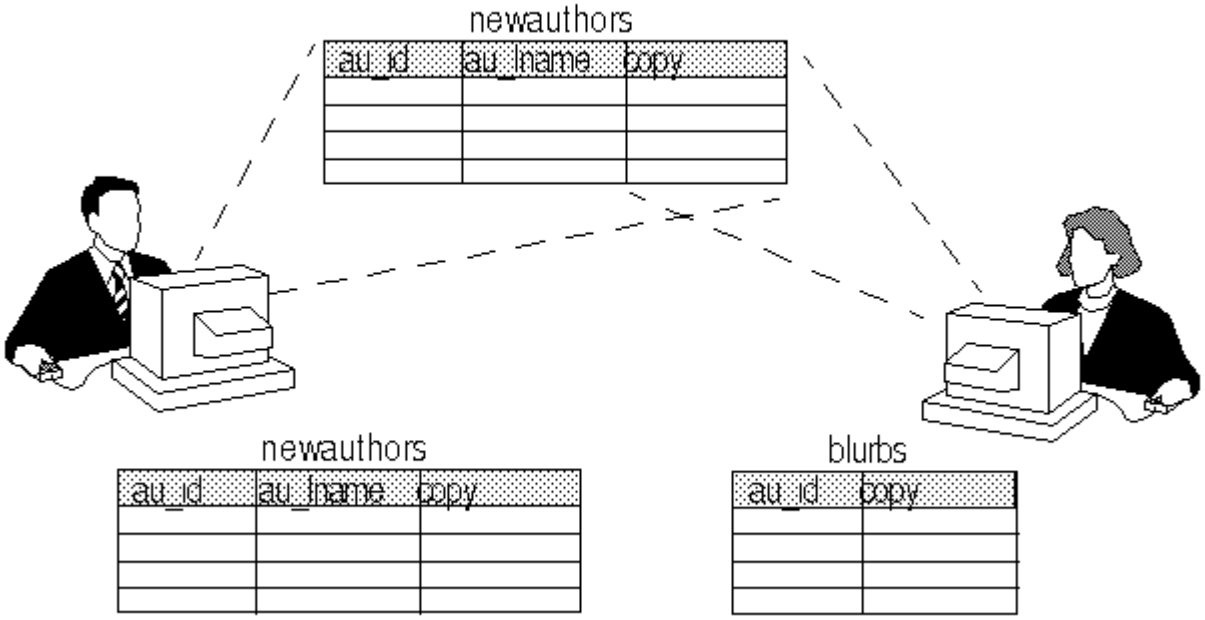
Denormalization has these disadvantages:

- It usually speeds retrieval but can slow data modification.
- It is always application-specific and needs to be reevaluated if the application changes.
- It can increase the size of tables.
- In some instances, it simplifies coding; in others, it makes coding more complex.

## 6.2 Duplicating/Mirroring Tables

If a group of users regularly needs only a subset of data, you can duplicate the critical table subset for that group. This requires the creation of duplicate, or mirror tables. For example Consider an application system that has very heavy on-line traffic during the morning and early afternoon hours. This traffic consists of both querying and updating of data. Decision support processing is also performed on the same application tables during the afternoon. The production work in the afternoon always seems to disrupt the decision support processing causing frequent time outs and dead locks. This situation could be corrected by creating mirror tables. A foreground set of tables would exist for the production traffic and a background set of tables would exist for the decision support reporting. A mechanism to periodically migrate the foreground data to background tables must be established to keep the application data synchronized. One such mechanism could be a batch job executing UNLOAD and LOAD utilities. This should be done as often as necessary to sustain the effectiveness of the decision support processing.

**Figure 3-13: Increase Performance by duplicating tables**



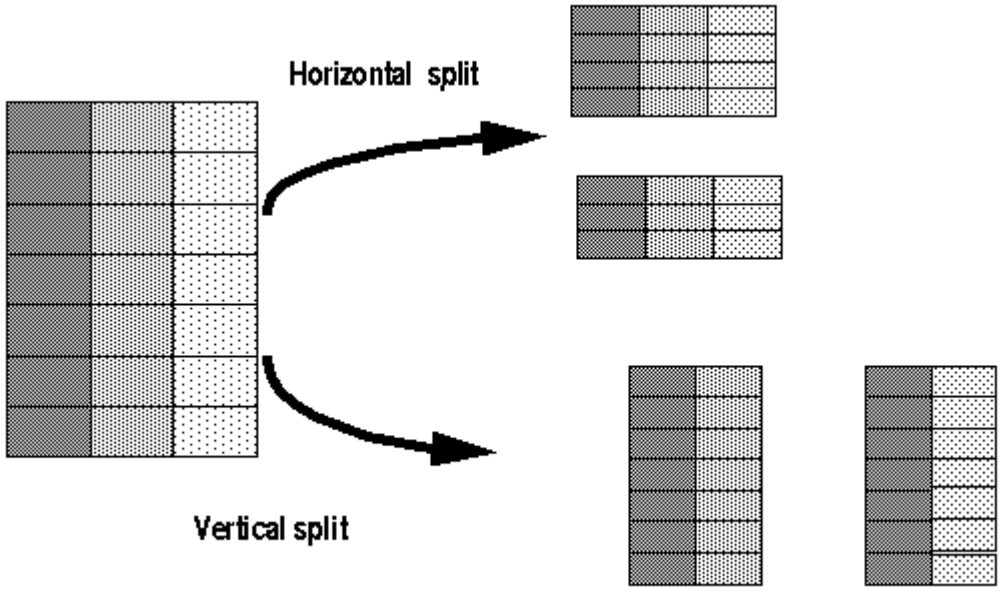

The kind of split shown in Figure 2-13 minimizes contention, but requires that you manage redundancy. There may be issues of latency for the group of users who see only the copied data.

## 6.3 Splitting Tables

Sometimes splitting normalized tables can improve performance. You can split tables in two ways:

- Horizontally, by placing rows in two separate tables, depending on data values in one or more columns
- Vertically, by placing the primary key and some columns in one table, and placing other columns and the primary key in another table

**Figure 2-14: Horizontal and vertical partitioning of tables**




Splitting tables¾either horizontally or vertically¾adds complexity to your applications. There usually needs to be a very good performance reason.

### 6.3.1 Horizontal Splitting

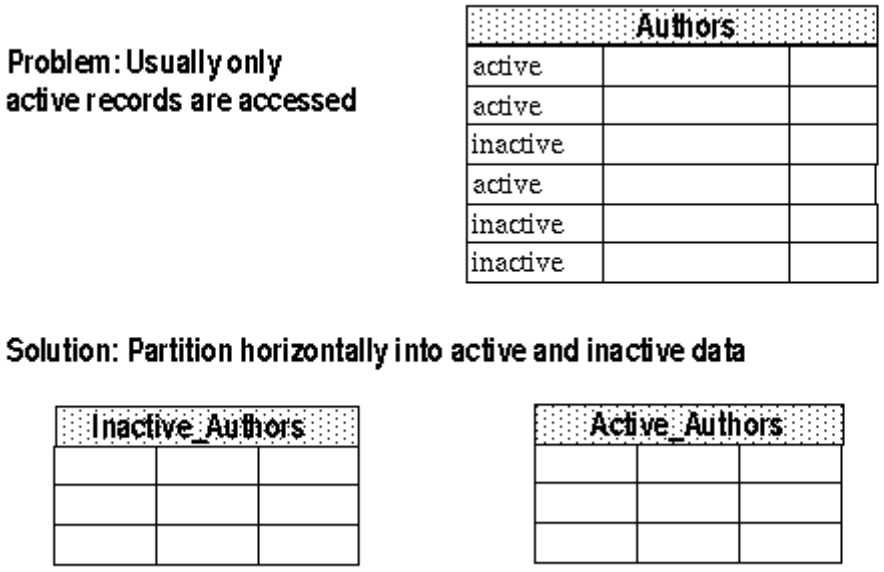
Use horizontal splitting in the following circumstances:

- A table is large, and reducing its size reduces the number of index pages read in a query. B-tree indexes, however, are generally very flat, and you can add large numbers of rows to a table with small index keys before the B-tree requires more levels. An excessive number of index levels may be an issue with tables that have very large keys.
- The table split corresponds to a natural separation of the rows, such as different geographical sites or historical vs. current data. You might choose horizontal splitting if you have a table that stores huge amounts of rarely used historical data, and your applications have high performance needs for current data in the same table.
- Table splitting distributes data over the physical media (there are other ways to accomplish this goal, too).

Generally, horizontal splitting requires different table names in queries, depending on values in the tables. This complexity usually far outweighs the advantages of table splitting in most database applications. As long as the index keys are short and indexes are used for queries on the table, doubling or tripling the number of rows in the table may increase the number of disk reads required for a query by only one index level. If many queries perform table scans, horizontal splitting may improve performance enough to be worth the extra maintenance effort.

Figure 2-15 shows how the *authors* table might be split to separate active and inactive authors:

**Figure 2-15: Horizontal partitioning of active and inactive data**




### 6.3.2 Vertical Splitting

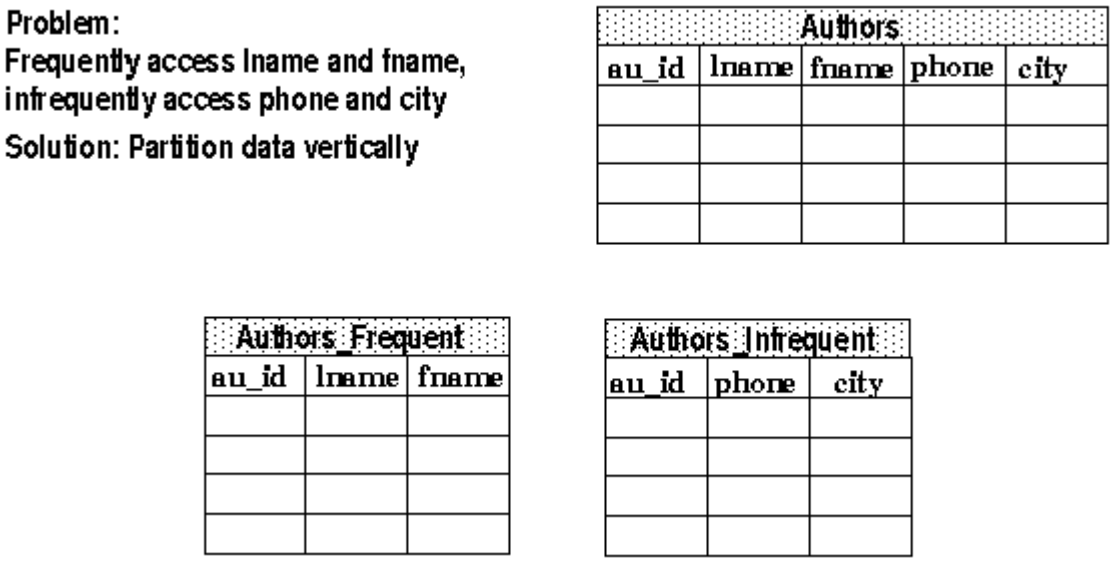
Use vertical splitting in the following circumstances:

- Some columns are accessed more frequently than other columns.
- The table has wide rows, and splitting the table reduces the number of pages that need to be read.

Vertical table splitting makes even more sense when both of the above conditions are true. When a table contains very long columns that are accessed infrequently, placing them in a separate table can greatly speed the retrieval of the more frequently used columns. With shorter rows, more data rows fit on a data page, so, for many queries, fewer pages can be accessed.

Figure 2-16 shows how the *authors* table can be partitioned.

**Figure 2-16: Vertically partitioning a table**




# 7 MANAGING DATA REDUNDANCY

Whatever denormalization / Split /Mirror techniques you use, you need to ensure data integrity. Techniques you can use include:

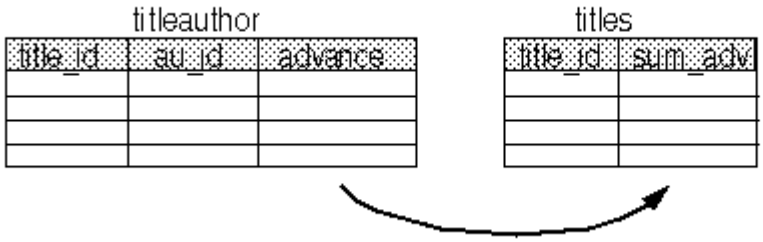
- Triggers, which can update derived or duplicated data anytime the base data changes
- Application logic, using transactions in each application that update denormalized data, to ensure that changes are atomic
- Batch reconciliation, run at appropriate intervals, to bring the denormalized data back into agreement

From an integrity point of view, triggers provide the best solution, although they can be costly in terms of performance.

## 7.1 Using Triggers to Manage Data Redundancy

In Figure 2-17, the *sum_adv* column in the *titles* table stores denormalized data. A trigger updates the *sum_adv* column whenever the *advance* column in *titleauthor* changes.

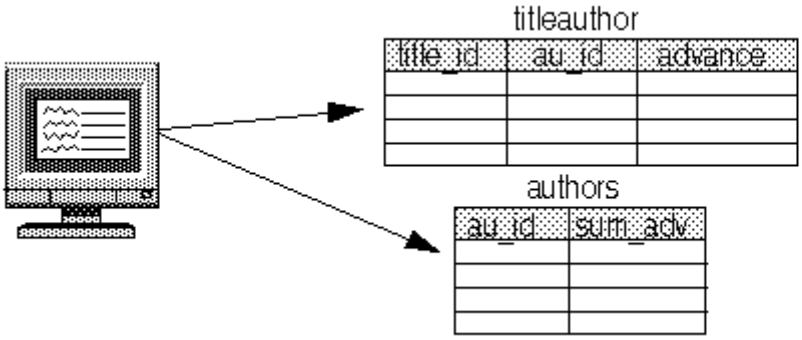
**Figure 2-17: Using triggers to maintain normalized data**




## 7.2 Using Application Logic to Manage Data Redundancy

If your application has to ensure data integrity, it must ensure that the inserts, deletes, or updates to both tables occur in a single transaction (see Figure 2-18).

**Figure 2-18: Maintaining denormalized data via application logic**




If you use application logic, be very sure that the data integrity requirements are well documented and well known to all application developers and to those who must maintain applications.

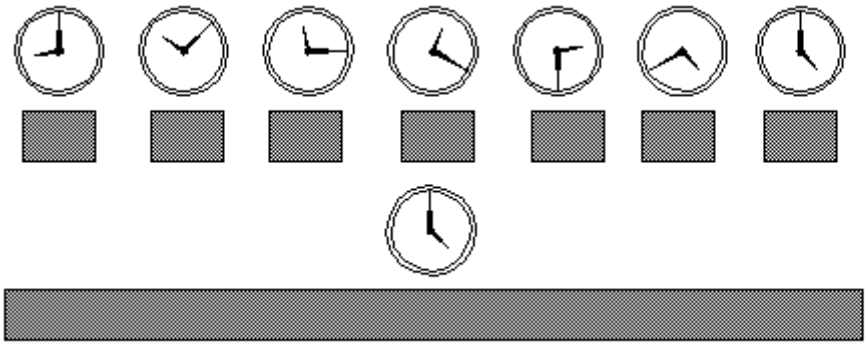
**Note:** , Using application logic to manage denormalized data is risky. The same logic must be used and maintained in all applications that modify the data.

## 7.3 Batch Reconciliation to Manage Data Redundancy

If 100 percent consistency is not required at all times, you can run a batch job or stored procedure during off-hours to reconcile duplicate or derived data.

You can run short, frequent batches or long, infrequent batches (see Figure 2-19).

**Figure 2-19: Using batch reconciliation to maintain data**



## 8 SUMMARY

The decision to denormalize/ adding redundancy should never be made lightly because it involves a lot of administrative dedication. This dedication takes the form of documenting the denormalization/redundancy decisions, ensuring valid data, scheduling of data migration, and keeping end users informed about the state of the tables. In addition, there is one more category of administrative overhead: periodic analysis.

Whenever denormalized data exists for an application the data and environment should be periodically reviewed on an on-going basis. Hardware, software, and application requirements will evolve and change. This may alter the need for denormalization /redundancy. To verify whether or not denormalization is still a valid decision ask the following questions:

- Have the processing requirements changed for the application such that the join criteria, timing of reports, and/or transaction throughput no longer require denormalized data?
- Did a new DBMS release change performance considerations? For example, did the introduction of a new join method undo the need for pre-joined tables?
- Did a new hardware release change performance considerations? For example, does the upgrade to a new high-end processor reduce the amount of CPU such that denormalization is no longer necessary? Or did the addition of memory enable faster data access enabling data to be physically normalized?

In general, periodically test whether the extra cost related to processing with normalized tables justifies the benefit of denormalization. You should measure the following criteria:

- I/O saved
- CPU saved
- complexity of update programming
- cost of returning to a normalized design

It is important to remember that denormalization/ adding data redundancy was initially implemented for performance reasons. If the environment changes it is only reasonable to re-evaluate the decision. Also, it is possible that, given a changing hardware and software

environment, the data redundancy may be causing performance degradation instead of performance gains.

Simply stated, always monitor and periodically re-evaluate all applications which has controlled data redundancy for better performance.

# 9 APPENDIX –A (REFERENCES & COURTESY)

http://www.oracle.com/
http://www.devx.com/
http://sybase.com
http://techrepublic.com.com/