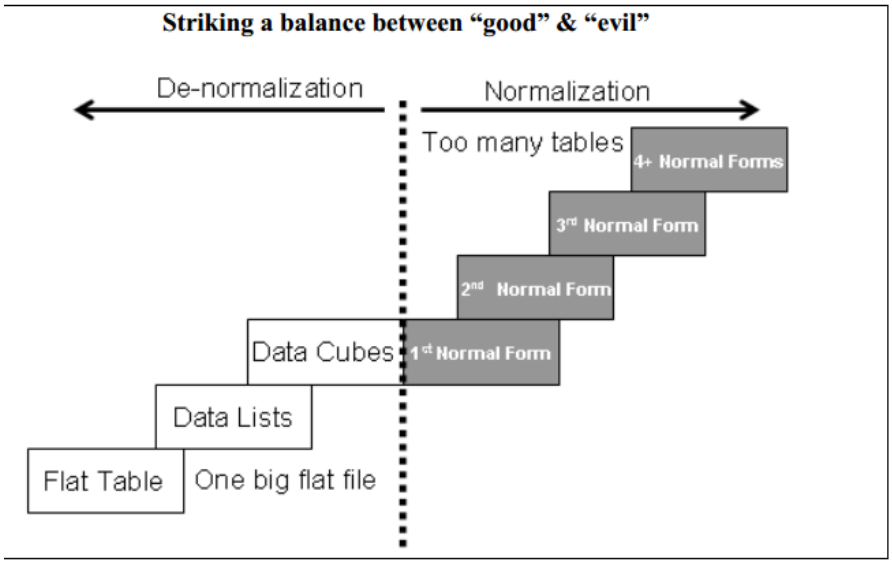
LECTURE 5

De-Normalization

Leaning Goals

1. What is De-Normalization?
2. Why De-Normalization used in DSS?
3. How De-Normalization improves performance?
4. Guidelines for De-normalization



Striking a balance between normalization and de-normalization

There should be a balance between normalized and de-normalized forms. In a fully normalized form, too many joins are required and in a totally de-normalized form, we have a big, wide single table. Database should be aligned someplace in between so as to strike a balance, especially in the context of the queries and the application domain.

A "pure" normalized design is a good starting point for a data model and is a great thing to do and achieve in academia. However, as briefly mentioned in the previous lecture, in the reality of the "real world", the enhancement in performance delivered by some selective de-normalization technique can be a very valuable tool. The key to success is to undertake de-normalization as a design technique very cautiously and consciously. Do not let proliferation of the technique take over your data warehouse or you will end up with a single big flat file!

What is De-Normalization?

It is not chaos, more like a "controlled crash" with the aim of performance enhancement without loss of information.

1. Normalization is a rule of thumb in DBMS, but in DSS ease of use is achieved by way of denormalization
2. De-normalization comes in many flavors, such as combining tables, splitting tables, adding data etc., but all done very carefully.

'Denormalization' does not mean that anything and everything goes. Denormalization does not mean chaos or disorder or indiscipline. The development of properly de- normalized data structures follows software engineering principles, which insure that information will not be lost. De-normalization is the process of selectively transforming normalized relations into un-normalized physical record specifications, with the aim of reducing query processing time. Another fundamental purpose 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 required to answer a query. Some people tend to confuse dimensional modeling with de-normalization. This will become clear when we will cover dimensional modeling, where indeed tables are collapsed together.

Why De-Normalization in DSS?

1. Bringing "close" dispersed but related data items.
2. Query performance in DSS significantly dependent on physical data model.
3. Very early studies showed performance difference in orders of magnitude for different number de-normalized tables and rows per table.
4. The level of de-normalization should be carefully considered.

The efficient processing of data depends how close together the related data items are. Often all the attributes that appear within a relation are not used together, and data from different relations is needed together to answer a query or produce a report. Although normalized relations solve data maintenance anomalies (discussed in last lecture), however, normalized relations if implemented one for one as physical records, may not yield efficient data processing times. DSS

query performance is a function of the performance of every component in the data delivery architecture, but is strongly associated with the physical data model. Intelligent data modeling through the use of techniques such as de-normalization, aggregation, and partitioning, can provide orders of magnitude performance gains compared to the use of normalized data structures.

The processing performance between totally normalized and partially normalized DSSs can be dramatic. Inmon (grand father of data warehousing) reported way back in 1988 through a study, by quantifying the performance of fully and partially normalized DSSs. In his study, a fully normalized DSS contained eight tables with about 50,000 rows each, another partially normalized DSS had four tables with roughly 25,000 rows each, and yet another partially normalized DSS had two tables. The results of the study showed that the less than fully normalized DSSs could muster a performance as much as an order of magnitude better than the fully normalized DSS. Although such results depend greatly on the DSS and the type of processing, yet these results suggest that one should carefully consider whether the physical records should exactly match the normalized relations for a DSS or not?

How De-Normalization improves performance?

De-normalization specifically improves performance by either:

1. Reducing the number of tables and hence the reliance on joins, which Consequently speeds up performance.
2. Reducing the number of joins required during query execution, or
3. Reducing the number of rows to be retrieved from the Primary Data Table

The higher the level of normalization, the greater will be the number of tables in the DSS as the depth of snowflake schema would increase. The greater the number of tables in the DSS, the more joins are necessary for data manipulation. Joins slow performance, especially for very large tables for large data extractions, which is a norm in DSS not an exception. De-normalization reduces the number of tables and hence the reliance on joins, which consequently speeds up performance.De-normalization, can help minimize joins and foreign keys and help resolve aggregates. By storing values that would otherwise need to be retrieved (repeatedly), one may be

able to reduce the number of indexes and even tables required to process queries.

Four Guidelines for De-normalization

1. Carefully do a cost-benefit analysis (frequency of use, additional storage, join time).
2. Do a data requirement and storage analysis.
3. Weigh against the maintenance issue of the redundant data (triggers used).
4. When in doubt, don't de-normalize

Guidelines for Denormalization

Following are some of the basic guidelines to help determine whether it's time to de- normalize the DSS design or not:

1 . Balance the frequency of use of the data items in question, the cost of additional storage to duplicate the data, and the acquisition time of the join.

2 . Understand how much data is involved in the typical query; the amount of data affects the amount of redundancy and additional storage requirements.

3 . Remember that redundant data is a performance benefit at query time, but is a performance liability at update time because each copy of the data needs to be kept up to date. Typically triggers are written to maintain the integrity of the duplicated data.

4 . De-normalization usually speeds up data retrieval, but it can slow the data modification processes. It may be noted that both on-line and batch system performance is adversely affected by a high degree of de-normalization. Hence the golden rule is: When in doubt, don't de-normalize.

Areas for Applying De-Normalization Techniques

1. Dealing with the abundance of star schemas.
2. Fast access of time series data for analysis.
    1. Fast aggregate (sum, average etc.) results and complicated calculations.
    2. Multidimensional analysis (e.g. geography) in a complex hierarchy.
    3. Dealing with few updates but many join queries.

De-normalization will ultimately affect the database size and query performance.

De-normalization is especially useful while dealing with the abundance of star schemas that are found in many data warehouse installations. For such cases, de-normalization provides better performance and a more natural data structure for supporting decision making. The goal of most analytical processes in a typical data warehouse environment is to access aggregates such as averages, sums, complicated formula calculations, top 10 customers etc. Typical OLTP systems contain only the raw transaction data, while decision makers expect to find aggregated and time-series data in their data warehouse to get the big picture through immediate query and display. Important and common parts of a data warehouse that are a good candidate for de-normalization include (but are not limited to):

- Aggregation and complicated calculations.
- Multidimensional analysis in a complex hierarchy.
- Few updates, but many join queries.

Geography is a good example of a multidimensional hierarchy (e.g. Province, Division, District, city and zone). Basic design decisions for example, the selection of dimensions, the number of dimensions to be used and what facts to aggregate will ultimately affect the database size and query performance.

Five principal De-Normalization techniques

Now we will discuss de-normalization techniques that have been commonly adopted by experienced database designers. These techniques can be classified into four prevalent strategies for demoralization which are:

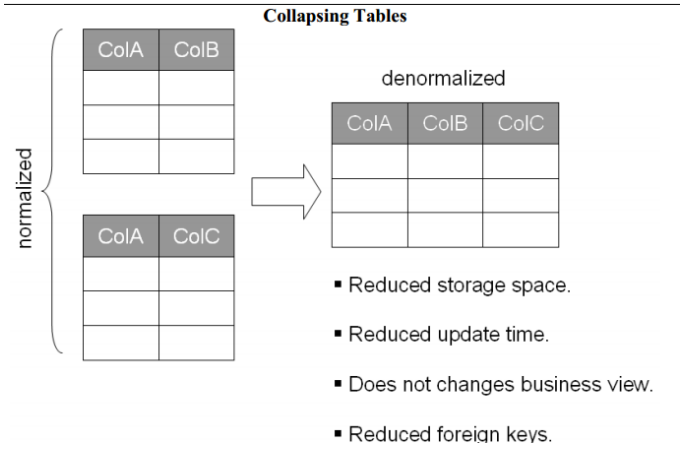
Collapsing Tables.
- Two entities with a One-to-One relationship.
- Two entities with a Many-to-Many relationship.

Pre-joining.

Splitting Tables (Horizontal/Vertical Splitting).

Adding Redundant Columns (Reference Data).

Derived Attributes (Summary, Total, Balance etc).



Collapsing Tables

Collapsing Tables

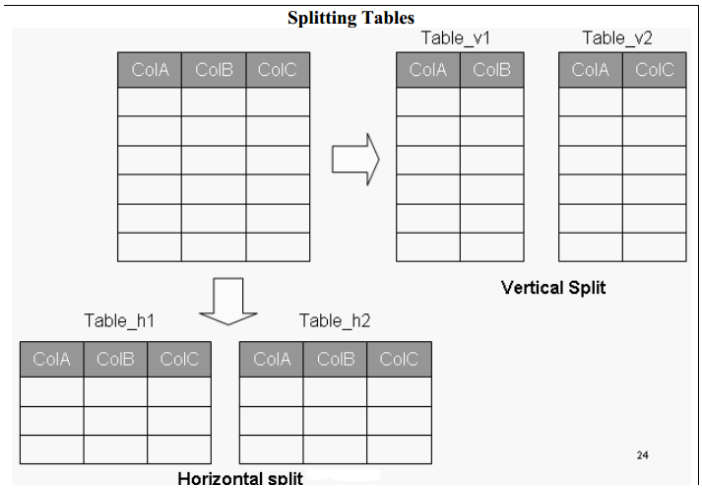
One of the most common and safe denormalization techniques is combining of one-to- One relationship. This situation occurs when for each row of entity A, there is only one related row in entity B. While the key attributes for the entities may or may not be the same, their equal participation in a relationship indicates that they can be treated as a single unit. For example, if users frequently need to see COLA, COLB, and COLC together and the data from the two tables are in a One-to-One relationship, the solution is to collapse the two tables into one. For example, SID and gender in one table, and SID and degree in the other table.

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

De-Normalization Techniques

Learning Goals

- Splitting Tables
- Pre-Joining
- Adding Redundant Columns



Splitting Tables

The denormalization techniques discussed earlier all dealt with combining tables to avoid doing run-time joins by decreasing the number of tables. In contrast, denormalization can be used to create more tables by splitting a relation into multiple tables. Both horizontal and vertical splitting and their combination are possible. This form of denormalization - record splitting- is especially common for a distributed DSS environment.

Splitting Tables: Horizontal splitting

Breaks a table into multiple tables based upon common column values. Example: Campus specific queries.

GOAL

1. Spreading rows for exploiting parallelism.
2. Grouping data to avoid unnecessary query load in WHERE clause.

Splitting Tables

Horizontal splitting breaks a relation into multiple record set specifications by placing different rows into different tables based upon common column values. For the multi- campus example being considered; students from Islamabad campus in the Islamabad table, Peshawar students in corresponding table etc. Each file or table created from the splitting has the same record lay out or header.

Goals of horizontal splitting:

There are typically two specific goals for horizontal partitioning: (1) spread rows in a large table across many HW components (disks, controllers, CPUs, etc.) in the environment to facilitate parallel processing, and (2) segregate data into separate partitions so that queries do not need to examine all data in a table when WHERE clause filters specify only a subset of the partitions. Of course, what we would like in an ideal deployment is to get both of these benefits from table partitioning.

Advantages of Splitting tables:

Horizontal splitting makes sense when different categories of rows of a table are processed separately: e.g. for the student table if a high percentage of queries are focused towards a certain campus at a time then the table is split accordingly. Horizontal splitting can also be more secure since file level of security can be used to prohibit users from seeing certain rows of data. Also each split table can be organized differently, appropriate for how it is individually used. In terms of page access, horizontally portioned files are likely to be retrieved faster as compared to un-split files, because the latter will involve more blocks to be accessed.

Splitting Tables: Horizontal splitting

ADVANTAGE

Enhance security of data.

1. Organizing tables differently for different queries.

2. Reduced I/O overhead.
3. Graceful degradation of database in case of table damage.
4. Fewer rows result in flatter B-trees and fast data retrieval.

Other than performance, there are some other very useful results of horizontal splitting the tables. As we discussed in the OLAP lecture, security is one of the key features required from an OLAP system. Actually DSS is a multi-user environment, and robust security needs to ensure. By splitting the tables and restricting the users to a particular split actually improves the security of the system. Consider time-based queries, if the queries have to cover last years worth of data, then splitting the tables on the basis of year will defiantly improve the performance as the amount of data to be accessed is reduced. Similarly, if for a multi-campus university, most of the queries are campus specific, then splitting the tables based on the campus would result in improved performance. In both of the cases of splitting discussed, i.e. time and space, as the number of records to be Retrieved is reduced, resulting in more records per block that translates into fewer page faults And high performance. If the table is not partitioned, and for some reason the table is damaged, then in the worst case all data might be lost. However, when the table gets partitioned, and even if a partition is damaged, ALL of the data is not lost. Assuming a worst case scenario that tables crash i.e. all of them, the system will not go down suddenly, but would go down gradually i.e. gracefully.

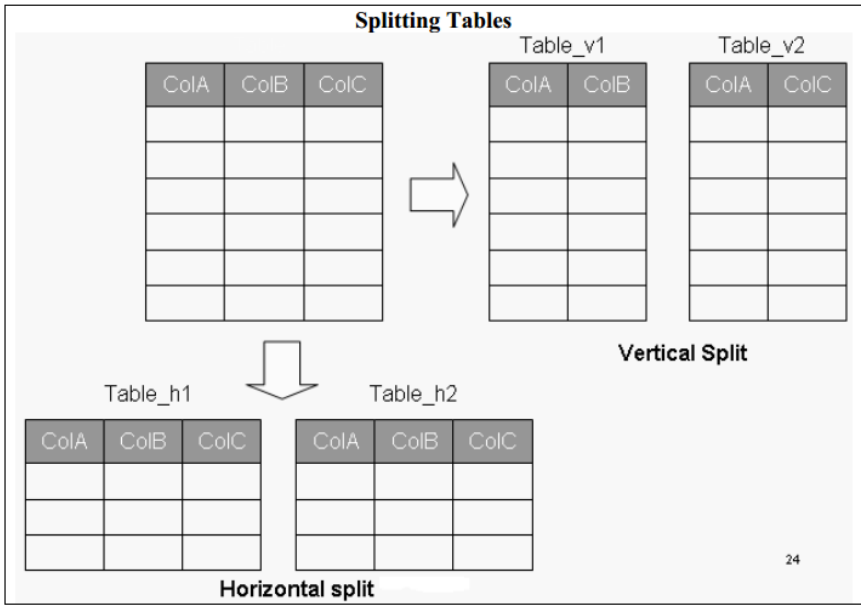
Splitting Tables: Vertical Splitting

1. Splitting and distributing into separate files with repeating primary key.
2. Infrequently accessed columns become extra “baggage” thus degrading performance.
3. Very useful for rarely accessed large text columns with large headers.
4. Header size is reduced, allowing more rows per block, thus reducing I/O.
5. For an end user, the split appears as a single table through a view.

Splitting Tables: Vertical Splitting

Vertical splitting 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. Thus columns are distributed into separate files, such that the primary key is repeated in each of the files. An example of vertical splitting would be breaking apart the student registration table by creating a

personal info table by placing SID along with corresponding data into one record specification, the SID along with demographic-related student data into another record specification, and so on. Vertical splitting can be used when some columns are rarely accessed rather than other columns or when the table has wide rows or header or both. Thus the infrequently accessed columns become extra "baggage" degrading performance. 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 header length allowing more rows to be packed in a block, thus reducing I/O. A vertically split table should contain one row per primary key in the split tables as this facilitates data retrieval across tables and also helps in dissolving the split i.e. making it reversible. In reality, the users are unaware of the split, as view of a joined table is presented to the users



Pre-Joining

Pre-joining Tables

The objective behind pre-joining is to identify frequent joins and append the corresponding tables together in the physical data model. This technique is generally used when there is a one-to-many relationship between two (or more) tables, such as the master-detail case when there are header and detail tables in the logical data model. Typically, referential integrity is assumed from the foreign key in one table (detail) to the primary key in the other table (header). Additional space will be required, because information on the master table is stored once for each detail record i.e. multiple times instead of just once as would be the case in a normalized design.

Pre-Joining

- Typical of Market basket query
- Join ALWAYS required
- Tables could be millions of rows
- Squeeze Master into Detail
- Repetition of facts. How much?

Shows a typical case of market basket querying, with a master table and a detail table. The sale_ID column is the primary key for the master table and uniquely Identifies a market basket. There will be one "detail" record for each item listed on the "Receipt" for the market basket. The tx_ID column is the primary key for the detail table.

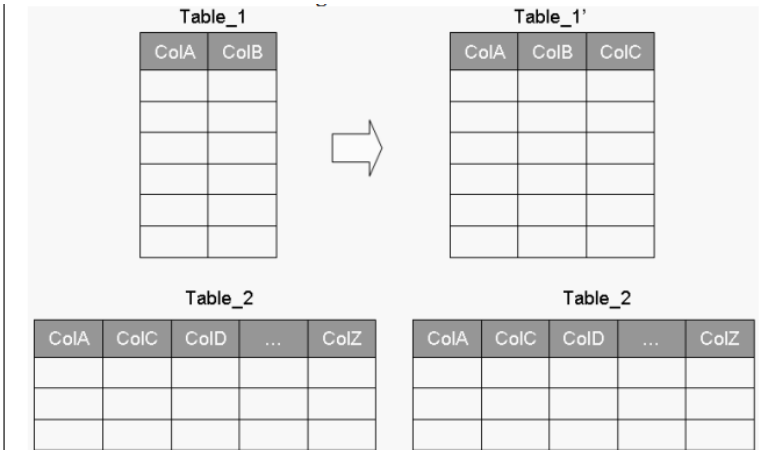
Observe that in a normalized design the store and sale date for the market basket is on one table (master) and the item (along with quantity, sales Rs, etc.) are on a separate table (detail). Almost all analysis will require product, sales date, and (sometimes) sale person (in the context of HR). Moreover, both tables can easily be millions of rows for a large retail outlet with significant historical data. This means that a join will be forced between two very large tables for almost every query asked of the data warehouse. This could easily choke the system and degrade performance. Note that this same header/detail structure in the data applies across many industries as we have discussed in the very early lectures, such as healthcare, transportation, logistics, billing etc. To avoid the run-time join, we use the pre-join technique and "squeeze" the sales master information into the detail table. The obvious drawback is repetition of facts from the master table into the detail table. This avoids the join operation at run-time, but stores the header information redundantly as part of the sales detail. This redundant storage is a violation of normalization, but will be acceptable if the cost of storage is less than the performance achieved by virtue of eliminating the join.

Adding Redundant Columns

This technique can be used when a column from one table is frequently accessed in a large scale join in conjunction with a column from another table. To avoid this join, the column is added

(redundant) or moved into the detail table(s) to avoid the join. For example, if frequent joins are performed using Table_1 and Table_2 using columns ColA, ColB and ColC, then it is suitable to add ColC to Table_1.

Adding Redundant Columns



Adding redundant columns

Note that the columns can also be moved, instead of making them redundant. If closely observed, this technique is no different from a pre-joining. In pre-joining all columns are moved from the master table into the detail table, but in the current case, a sub-set of columns from the master table is made redundant or moved into the detail table. The performance, and storage trade-offs are also very similar to pre-joining.

Adding Redundant Columns

Columns can also be moved, instead of making them redundant. Very similar to pre-joining as discussed earlier.

EXAMPLE

Frequent referencing of code in one table and corresponding description in another table.

1. A join required is required.
2. To eliminate the join, a redundant attribute added in the target entity which is functionally independent of the primary key.

A typical scenario for column redundancy/movement is frequent referencing of code in one table

and the corresponding description in another table. The description of a code is retrieved via a join. In such a case, redundancy will naturally pay off. This is implemented by duplicating the descriptive attribute in the entity, which would otherwise contain only the code. The result is a redundant attribute in the target entity which is functionally independent of the primary key. Note that this foreign key relationship was created in the first place to

Normalize the corresponding description reduce update anomalies.

Redundant Columns: Surprise

Note that:

1. Actually increases in storage space, and increase in update overhead.
2. Keeping the actual table intact and unchanged helps enforce RI constraint.
3. Age old debate of RI ON or OFF.

Redundant Columns Surprise

Creating redundant columns does not necessarily reduce the storage space requirements, as neither the reference table is removed, nor the columns duplicated from the reference table.

The reason being to ensure data input RI constraint, although this reasoning falls right in the middle of the age old debate that Referential Integrity (RI) constraint should be turned ON or OFF in a DWH environment. However, it is obvious that column redundancy does eliminate the join and increase the performance.

Derived Attributes

A. Objectives

- Ease of use for decision support applications
- Fast response to predefined user queries
- Customized data for particular target audiences
- Ad-hoc query support

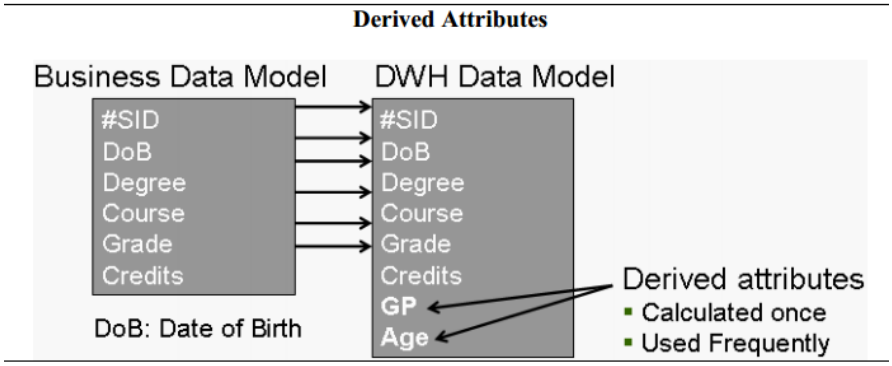
B. Feasible when…

- Calculated once, used most
- Remains fairly “constant”

- Looking for absoluteness of correctness.
- Pitfall of additional space and query degradation.

Derived Attributes

It is usually feasible to add derived attribute(s) in the data warehouse data model, if the derived data is frequently accessed and calculated once and is fairly stable. The justification of adding derived data is simple; it reduces the amount of query processing time at run-time while accessing the data in the warehouse. Furthermore, once the data is properly calculated, there is little or no apprehension about the authenticity of the calculation. Put in other words, once the derived data is properly calculated it kind of becomes absolute i.e. there is hardly any chance that someone might use a wrong formula to calculate it incorrectly. This actually enhances the credibility of the data in the data warehouse



Business Data Model vs. DWH Data Model

GP (Grade Point) column in the data warehouse data model is included as a derived value. The formula for calculating this field is Grade*Credits.

Age is also a derived attribute, calculated as Current Date – DOB (calculated periodically).

In most cases, it will only make sense to use derived data if the ratio of detail rows to derived rows is at least 10:1. In such cases, the 10% storage cost for keeping the derived data is less than the temporary and sort space storage costs for many concurrent queries aggregating at runtime.