CT3532 Database Systems - Further Design

Colm O'Riordan

We have seen that through design by synthesis we can obtain a *good* design.

This guarantees that the final design schema exhibits certain desirable features

However, occasionally we wish to violate the above guidelines to improve performance.
We must pay attention to transaction requirements.
Try to:

- Identify future required transactions
- their relative frequency
- the required response time

Indexing strategies

The information gleaned from the above analysis can inform the design of the indexing strategy.

Usually place index on fields that are to be frequently queried upon.

Choice of index (primary, secondary, B-tree, B+-tree, clustering) will depend on:

- type of query expected
- types permitted by DBMS
- type of field involved

Also try to identify which tables will be joined frequently and on which attributes. Common to build indexes on these attributes.

Key choice

Performance requirements can lead to a change in the logical design.
Consider a table containing 1000 employees. The SSN number was chosen in the conceptual design as a choice of key.

Now consider the performance requirements involve joining this table to another table which has SSN as a foreign key.

This query is expected to be very frequent with a short short response time required.

SSN numbers are 8 characters long. Any index built on this will contain index values of 8 characters wide.

We only have 1000 employees - could introduce surrogate key (emp id) and use this for indexes to handle joins.

Denormalisation

Denormalisation is the process of making compromises to the normalised tables by introducing intentional redundancy for performance reasons.

Decisions regarding denormalisation are made during the transaction requirements analysis.

Denormalisation involves a tradeoff between:

- introducing redundancy and potential for anomalies
- increased efficiency of certain transactions

Downward Denormalisation

Consider the following relations:

CUSTOMER: ID, address, name, telephone
ORDER: orderno, date, date invoice, cus ID

Assume that the queries of the following type are extremely frequent and require a fast response time:

```
SELECT ID, name
FROM CUSTOMER, ORDER
WHERE CUSTOMER.ID = ORDER.cus_ID
AND orderno = 46;
```

Main cost in evaluating this query is the join operation.
We can avoid this costly join be adding the `name` field to **ORDER** table.

This gives us:

ORDER: <u>orderno</u>, date, date invoice, cus ID, name

We have now introduced redundancy (violates 3NF), which leads to potential update, delete and insert anomalies.
However, queries of the type above can be dealt with more efficiently.

Other types of denormalisation exist:

- Upward denormalisation: store aggregate of values from one table in another
- Intra-table denormalisation: explicitly store information in a table that can be derived from other attributes