Lessons Learnt

With the 1970 paper of Edgar Codd outlining a relational database model, a leap in thinking concerning data was made. One of the most important concepts is data independence, namely the distinction between the logical data model and physical storage layer of any database. The summary below gives an overview of the content. I will also briefly summarize key concepts from the lecture, but this is not exhaustive, nor does it include every detail from the exercises.

Definitions:

- Table: Collection
- Attribute: Column, Field, Property
- Primary Key: Row ID, Name
- Row: Business Object, Item, Entity, Document, Record
- CRUD: Create, Read, Update Delete
- ACID: Atomicity, Consistency, Isolation, Durability (in database time)
- CAP: Consistency, Availability, Partition tolerance (in the new times, also note eventual consistency != consistency).
- OLTP: OnLine Transaction Processing (writing)
- OLAP: OnLine Analytical Processing (reading)

Relational queries overview (think SQL):

Set queries
- Union
- Intersection
- Substraction

Set queries
- Selection
- Projection
- Filter queries

Renaming queries
- Relation renaming
- Attribute renaming

Binary queries
- Cartesian product
- Natural join
- Theta join

Normal Forms & denormalization

- 1st Normal Form: The key (atomic)
- 2nd Normal Form: The whole key (no subset of the key dependency)
- 3rd Normal Form: Nothing but the key (no non-key dependency)
**SQL**

Declarative and functional language, which separates logical model from physical, and has nested expressions. Based heavily on concepts of data independence, and implements the relational model as proposed by Codd. Below a diagram showing the sequence of query processing (Joins, sorting and more is not visible.

![Diagram showing query processing](https://via.placeholder.com/150)

**1970 Paper by Edgar Codd, “A Relational Model of Data for Large Shared Data Banks”:**

Let me preface this with a note of caution: This summary is written to condense the concepts of the paper, not repeat the relational algebra introduced by Codd (otherwise I couldn’t condense it much). If you’re still interested, give the paper a read, but from the course, SQL and some set theory you should have a decent overview.

The paper introduces an “application of relation theory to systems which provide shared access to large banks of formatted data”. **Data independence** is defined as independence between the usage of programs and activities from additions or changes in data representation. The relational model is favorable for multiple reasons:

- Describes the data without adding any structure for machine representation. This aids data independence
- Can be used to determine derivability, redundancy and consistency
- Highlights the scope and limitations of the model in a clear fashion

In no particular order (pun intended), three limitations of existing models were: ordering dependence, indexing dependence and access path dependence, hindering data independence. **Ordering independence** would mean a system can operate fully, with the data stored in any order, and independent of the hardware addresses of the data. **Index independence** is desirable, as they represent a performance improvement, and should not affect the function of the database. The index should also be independent of the primary key, and the database continue to function with a modified or removed index. Lastly, **access path independence** is critical, so that a database does not require files to be in a specific hierarchical location and fail when the access path is modified.

The relational view of Codd on data is as follows: we have not necessarily distinct sets $S_1, S_2, ..., S_n$, and then $R$ is a relation on these $n$ sets, if it is a set of $n$-tuples, where the first element is $S_1$, the second $S_2$ and so on. The relation can be said to have degree $n$, where we have unary, binary, ternary and $n$-nary relations. We can now observe that a relational table has the following properties: every row is an $n$-tuple of the relation, the order is irrelevant, the rows are distinct, the
order of the columns is significant, the name of each column matters. This allows for multiple columns with the same name (hence of the same domain), but their order distinguishes them from one another. To achieve data independence, we have to abstract away from this relational view, and for example, make column ordering independent, by having a unique column name (even if they are of the same domain).

**Introduced definitions:**

- **Primary key:** domain (or combination of) uniquely identifying each element (row). It is nonredundant if it is atomic or combination such that no member of the key is superfluous. Multiple candidate primary keys may exist, then one is chosen to be the primary key.

- **Foreign key:** domain (or combination of) that is not the primary key of the current relation, but the primary key of another relation (relation is somewhat analogous to table here).

- **Nonsimple domain:** domains which are themselves relations (non-atomic)

- **Normalization:** eliminating nonsimple domains

Normalization can be defined by the following procedure: begin at the top-level ‘node’ of the relation tree and insert its primary key into every direct subdomain. The primary key of every direct subdomain is then the concatenation of the inserted primary key, and its pre-existing primary key. Then remove from the top-level relation all nonsimple domains. Repeat for every subtree recursively. Further normalization is possible. An additional advantage of normalization is the improved suitability of communication with other systems, that have a different representation of the data as 1) we no longer have pointers 2) we can avoid dependencies 3) we are order independent. Furthermore, the naming can now be simplified: usually \( R.d \) is adequate (\( R \) is the relation name, \( d \) the domain name).

The paper goes on to discuss the importance of the data sublanguage \( R \) and host languages \( H \), and their importance in analyzing the expressivity of the data language. \( R \) should be as descriptive as possible, not computationally powerful. We want to have query, insert, delete and update (CRUD) operations, some of which might be triggered by internal dependencies in \( R \). Another concept is that when a user knows a relation is stored, he expects to be able to exploit the relation knowing some arguments and not knowing others (think columns). This is symmetric exploitation of relations.

The named set of a data base is what has been defined (columns), the expressible set is the total collections of relations that can be expressed using the data language (think joins, projections, arithmetic). The named set is a subset of the expressible set.

The “responsibility for providing efficient response and throughput shifts from the individual user to the data system”

**List of transformations:**

- **Permutation:** re-ordering of the columns, identical for queries, but relevant for storage.
- **Projection:** removing columns, using selection operator \( \pi \).
- **Join:** combining relations, natural join always exists (think cartesian product). Other joins exist. When a join grows the number of elements, there exists a point of ambiguity, which is the element for which not only a single join exists. Ambiguity can sometimes be resolved with another relation, introducing cyclic joins (different from linear 3-way-joins we’re probably all familiar with). The tie of a relation is then producing a \((n - 1)\) degree relation from \( \pi \), by tying the ends together. The cyclic join is then the tie of the cartesian product.
• **Composability**: only possible if relations are joinable, is, in essence, a projection on the join (on the middle term)

• **Restriction**: a way of filtering rows by using another relation. The diagram from the paper shows it well: \( R' \) is a restriction of R given S, by restricting R’s 2nd and 3rd column based on S’ 1st and 2nd columns. Hence only \( a=A \) and \( b=B \) elements were ‘permitted’

<table>
<thead>
<tr>
<th></th>
<th>R (s p j)</th>
<th>S (p j)</th>
<th>R’ (s p j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a A</td>
<td>a A</td>
<td>1 a A</td>
</tr>
<tr>
<td>2</td>
<td>a A</td>
<td>c B</td>
<td>2 a A</td>
</tr>
<tr>
<td>2</td>
<td>a B</td>
<td>b B</td>
<td>2 b B</td>
</tr>
<tr>
<td>2</td>
<td>b A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>b B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Redundancy**: Let us consider redundancy in transformation (projection, natural join, tie and restriction). A set of relations is **strongly redundant**, if it contains a projection which is derivable from other projections. This can be advantageous for storage optimization purposes. **Weak redundancy** may exist if a set of relations has a projection of some join of other projections of relations.

**Consistency**: A system may never be fully aware of all redundancies (strong and weak), and consistency ensures that values that are expected to have a certain relationship (could be equality, could be a ‘chain’ of redundancy) are consistent. The state describes the instantaneous state of the data base, irrelevant of the cause for or against it. Checking for it at every insertion, deletion or update is possible, but costly.