Design Patterns for Relational Databases

Eugenia Stathopoulou, Panos Vassiliadis
University of Ioannina,
Dept. of Computer Science,
Ioannina, Hellas
{jenny_stathopoulou@hotmail.com, pvassil@cs.uoi.gr}

1 Introduction

A design artifact at the logical level comprises abstract mathematical symbol structures to hide implementation details from the designer [Kolp01, Mylo98]. Logical models are the bridge between the requirements-oriented, subjective, highly intuitive conceptual models and the concrete, physical-level models that represent the way things are actually implemented in the system. This property provides a reasonable compromise between formality, intuition and implementation and makes the logical models the fundamental blueprints of the software architecture of an information system. In the world of databases, the fundamental design artifacts at the logical level are the database schemata. A database schema is the platform over which (a) applications are developed and (b) tuning of the physical structure of the database is performed. In other words, logical schemata are the most important design artifact for the full lifecycle of a database-centric information system.

Why patterns? Patterns constitute a principled way of teaching, designing and documenting software systems [GHJV95]. Moreover, patterns allow us to evaluate the quality of a design by measuring the compliance of a logical schema to a set of underlying patterns. Given a well-founded theory of database patterns, the less deviations a schema has from the theory, the less is the risk of *maintenance traps*, since the improvisations that a designer makes are minimized.

In this paper, we provide a discussion of a template structure for database-related patterns. We make the following assumptions:

(i) we are primarily interested in patterns concerning *relational databases* (on top of which, object-relational or other structures can be applied), and,

(ii) we view the problems of database design from the perspective of maintenance and evolution (as opposed to other viewpoints, like, for example, performance).

In the next section, we provide a template pattern structure. Then, we discuss three design problems along with their respective patterns, specifically, pivoting, materialization and generalization.

2 Template Pattern Structure

Why do we organize database design in patterns? What fundamental contribution is there in the proposal of trying to provide a wide, structured list of common situations? Like in all engineering principles, the goal is to equip the designer with commonly accepted alternative design solutions for recurring problems. There are more than one solution for every problem, be it ad hoc or recurring, but some of them have better characteristics than others – even if none is a clear winner in every aspect of the problem. Providing the designer with a toolbox of *best practices* does not attempt to rigidly enforce a fixed set of solutions to standard problems; the goal is to plainly explain –in a measurable way, if possible- the motivations, assumptions, benefits and risks of each solution and, then, let the designer build, customize, reuse and adjust these template solutions *in knowledge* of what the properties of the produced solution are.

Ontological foundations. Patterns should address the fundamental concerns around the design of a database schema; therefore, the comprehensive treatment of all these concerns by a design pattern is unavoidable. To this day, there is a common agreement around the concerns that a designer faces:

- *Data integrity*. The first concern for a database schema, introduced at the seminal paper by E.F. Codd that introduced the relational model already dealt with the issue of data integrity [Codd70]. Early enough, E.F. Codd realized that unnecessary replication in a database can lead to data entry errors and, subsequently, to inconsistencies in the information presented to the user. Normal Forms were born together with the relational model and constitute the only textbook-level pattern-related design method that is deeply incorporated in the corpus of the database literature, in terms of theoretical foundations, and part of the curriculum of a database course.

- *Query efficiency*. Bruce Lindsay [Wins05] is quoted as having said that the three most important aspects of a DBMS are "performance, performance, and performance". A database is built with the primary goal of answering user queries and efficiency in this task is of uttermost importance. So, once the data integrity and completeness aspects are resolved at the logical level, a designer is obliged to fine-tune the design of a database (both at the logical and, mostly, at the physical level) in order to achieve acceptable response time and throughput for the user workload.
- *Evolution*. Typically, maintenance, or evolution (as we choose to call it in the '00s) involves around 50% of the resources of a software project. Database centric systems are no exception to this rule. The difference of database-centric systems from the software developed by the procedural or object-oriented paradigm is the strict layering of the developed software.

A database with a physical configuration (indexes, ISAM files, disk placement, clustering, etc) is placed at the bottom of this layered architecture. The data independence principle envisioned by E.F. Codd places a logical level abstraction on top of the physical layer, providing a mathematical abstraction for the construction of applications in terms of the relational model. Plainly speaking, this paradigm requires the designer to come up with a database schema, i.e., a set of relations, a.k.a. tables, over which applications or ad-hoc queries are to be posed (without any regard to their physical implementation). This logical-level schema constitutes a primitive API over which the applications of the database-centric system are built as the third layer of this architecture. Still, since database schemata have become large and complicated, the coupling of applications with the underlying schemata becomes more and more intense. One of the ideas behind this paper has to do with the introduction of an auxiliary API (mainly supported by views) that abstracts the complexity of the logical schema from the application developer and reduces the coupling of the database and constructed applications on top of it.

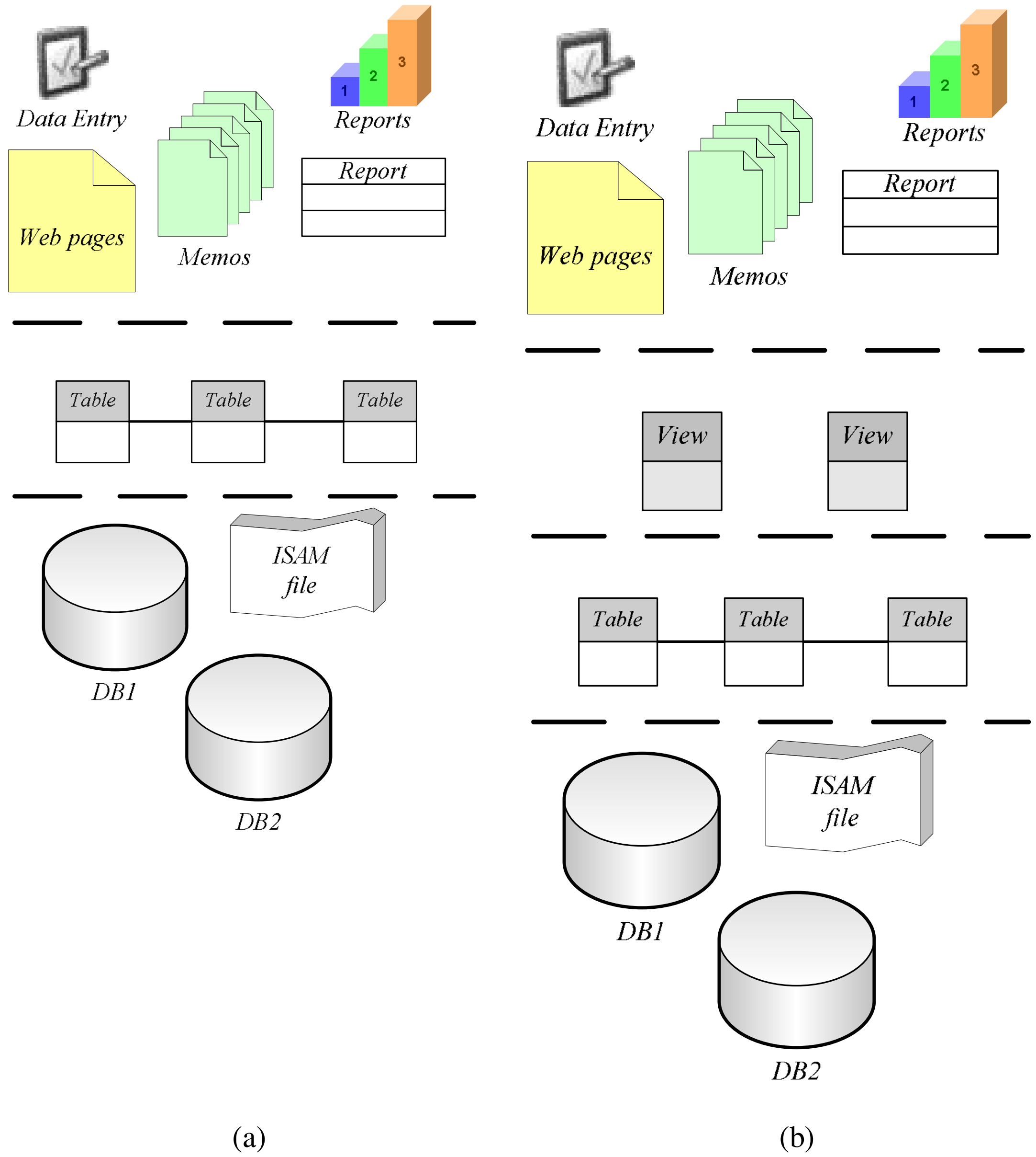

Figure 2.1. Applications built (a) directly on top of the logical database schema, (b) over an API-like layer of views

Pattern structure. How should we structure the presentation of patterns to correspond to abovementioned ontological foundations? In this paper we adopt the following structure for pattern presentation. Before proceeding, we would like to clarify the terminology, in order to avoid confusion:

- A design *problem* is a frequently encountered situation where the designer needs to map user requirements, or conceptual-level constructs (ER, UML diagrams) to logical or physical constructs in the database. In this paper, we are *not* interested in providing alternative ways to construct queries over a given schema; on the contrary, we are interested in designing database schemata on the basis of higher-level requirements.

- A design *pattern*, or design *solution*, or simply *design*, is a response to a problem.

The structure of a pattern is based (a) on the traditional pattern structure as delivered by Gamma et al [GHJV95] and (b) on the fundamentals of everyday operations around a database system.

- Motivation. The motivation discusses the situation that produces a puzzle for the designer. The problem is contextualized and its parameters analyzed.
- Alternative Solutions. The answers to the problem, in other words, the design patterns are presented. The description of each solution should normally incorporate a definition of the database schema, and an illustrative example both at the schema and the instance level.
- Interface to developers. Assuming a developer would like to have a certain level of guarantees over the schema that his applications see, how can the database provide an API-like layer on top of the relations at the logical level? Every pattern must describe a mechanism that buffers schema evolution effects (as much as possible) so that the developer can judge how the application must interface with the database in order to minimize their coupling.
- Behavior at the instance level. The first of the dynamic properties of a solution (i.e., properties characterizing how the system will behave over time) has to do with the management of insertion, deletion and updates of tuples in the database.
- Behavior at the schema level. The second kind of dynamic properties has to do with how the system is going to respond to future schema changes. These changes are expected to stem from changes in the reality that the database schema model.
- Overall discussion and comparison of alternatives. Finally, the presentation of a set of patterns should include a comparative critical assessment of them.

Again, we would like to stress that our focus is on maintenance and not performance. In the following, we explore three cases of problems and patterns, specifically, (a) pivoting, (b) materialization and (c) generalization.

3 Pivoting

3.1 Motivation

The main motivation for the case of *pivoting* is the management of *attribute-value* pairs. The case of attribute value pairs appears whenever attributes of similar functionality and type appear within an entity. Take for example a database of the public sector containing information about pensioners. Apart from the personal information, a pensioner has a group of similar attributes concerning the kinds of bonuses he is awarded and a group of similar attributes concerning the amounts of money he is granted every month. Specifically, the first group might comprise attributes like `HandicapBonusPct`, `HeavyDutyProfessionType`, `WarVeteranMonths`, each denoting whether the pensioner deserves an extra bonus due to (a) some injury or physical handicap (expressed as a 0-100 value on the pensioner's ability to operate normally), (b) the type of profession he exercised before retiring (constrained to heavy duty professions), or, (c) his military service (in terms of months in combat). The second group comprises attributes like `Pension`, `Tax`, `HandicapBonus`, `HeavyDutyProfessionBonus`, `WarVeteranBonus`, with the obvious semantics, in terms of monthly revenue or tax.

Assume that every bonus type and every type of amount that the pensioner receives is modeled as a separate attribute. Then, constraints are easy to check and queries are easily constructed and efficiently executed. Still, the database designer faces the following problem: if an extra type of bonus is introduced, *all* the applications that operate over the Pensioner relation have to be appropriately maintained (in fact, all the queries of these applications have to be maintained as well as their mapping to the graphical user interface that presents the results to the user).

3.2 Design solutions

To deal with this problem, we introduce two alternative modeling solutions for the representation of this information. We organize attributes in two classes: (a) *stable* attributes, for which no major or frequent modifications are anticipated at the schema level and (b) *evolvable* attributes that comprise the part of the schema where alterations are foreseeable. The two proposed designs are as follows

- *Flat design*: all the properties of the entity are modeled as different attributes. For example, in our case, we have the following relational structure:

```
EMP (E_ID,Name, HandicapBonusPct, HeavyDutyProfessionType, WarVeteranMonths,
     Pension, Tax, HandicapBonus, HeavyDutyProfessionBonus, WarVeteranBonus)
```

- *Attribute-value pairs*: we construct three relations, (i) the *stable* relation with the stable attributes, (ii) the *master* relation where each *category* of properties is modeled as an attribute, and, (iii) a (set of) *lookup relation*(s) where the description of the properties is maintained. For example, in our case, we have the following relational structure:

`EMP_Stable (E_ID, Name)` (*stable relation*)

`EMP_AMTS (E_ID, Amt_ID, Amt_Value)` (*master relation*)

`AMOUNT_TYPES(Amt_ID,Amt_Description)` (*lookup relation*)

The name of the problem is *pivoting* referring to the well-known spreadsheet operation where the attribute-value representation is transformed to the flat representation. Figures 3.1 and 3.2 depict the schema-level structure of the flat and the attribute-value-pair design, respectively.

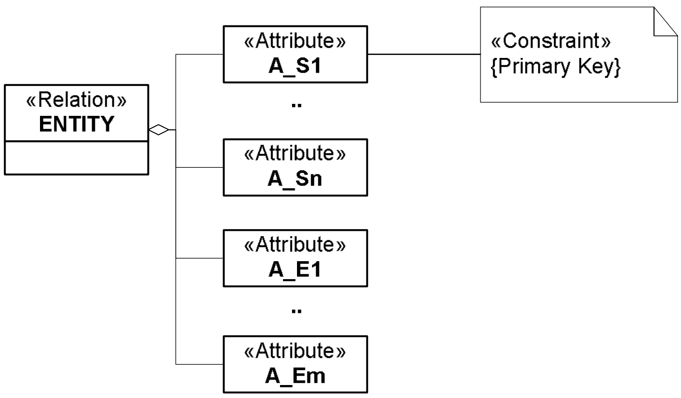

Fig. 3.1 Flat Design pattern for pivoting data.

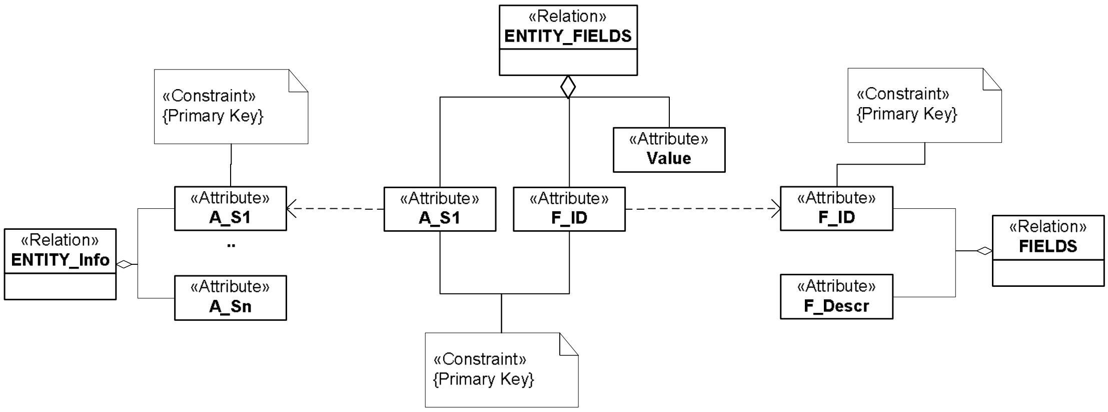

Fig. 3.2 Attribute-value pattern for pivoting data.

Figures 3.3 and 3.4 depict the instance-level structure of the flat and the attribute-value-pair design, respectively.

EMP

Emp_ID	**Salary**	**Bonus**	**Tax**	**Net**
01	1500	300	500	1300
02	2000	500	700	1800
03	1500		500	1000
04	1000			1000

Figure 3.3 Exemplary instance for the flat design.

EMP

Emp_ID	**Amt_ID**	**Amt_Value**
01	1	1500
01	2	300
01	3	500
01	4	1300
02	1	2000
02	2	500
02	3	700
02	4	1800
03	1	1500
03	3	500
03	4	1000
04	1	1000
04	4	1000

AMOUNT_TYPES

Amt_ID	**Amt_Description**
1	Salary
2	Bonus
3	Tax
4	Net

Figure 3.4 Exemplary instance for the attribute-value pairs design.

3.3 Interface to Developers

Assuming a developer is building an application on top of the database, the main decision he has to take is whether he needs to retrieve tuples in a flat or an attribute-value based manner. This is mainly imposed by performance reasons: once the structure of the database is set, then converting the instances from one pattern to the other at runtime is too slow (especially for large amounts of data). In terms of flexibility to evolution, clearly the attribute-value pattern is more flexible. A simple view can also relate the fields to their textual description; if the view is an outer join from the part of the fields, then, each entity can be related to a fixed set of fields, too.

ENTITY_FIELDS_FULL = ENTITY_FIELDS $\bowtie^{+}$ FIELDS

The conversion from one pattern to another can be done via an appropriate stored procedure; using a composition of SQL queries for this purpose incurs too much coding and maintenance effort as well as runtime overhead.

3.4 Behavior at the instance level

In this subsection, we discuss how the design decision for the schema of the master relation affects applications that query or modify its contents.

Querying. The collection of the necessary information for a particular reference entity (in our example, a pensioner) is straightforward in the case of a flat model. The case of attribute-value pair requires a join of the master table with all the lookup tables in order to reconstruct the textual description of the code id's of the parameterized properties. These differences concern also the case that a query requests a full table scan for all the contents of the involved relation.

In terms of **internal representation**, clearly indexing improves performance for both cases. In the attribute-value design with a single lookup, a clustered index might be very efficient, too. A non-obvious problem of the flat model is that it suffers from the presence of NULL values for attributes that are not pertinent to a certain record. This also requires extra care at the authoring of counting queries. In terms of the necessary disk space, the solution with the higher space overhead is determined by the average number of master records per entity and the number of NULL values in a flat representation.

Modification: Tuple Insertions/Deletions/Updates. The flat model requires the modification of a single record of the relation. On the other hand, the attribute-value model requires the modification of as many master records as necessary for a single entity that is inserted or deleted. A hidden problem with the updates is the two step-process for the performance of the correct update: first one needs to detect which code the update concerns (via the lookup relation) and then, the modification to the master table can be performed. Moreover, triggers `ON DELETE/UPDATE CASCADE` must be defined for the appropriate propagation of updates.

3.5 Behavior at the schema level

Clearly, schema modifications in the case of the flat design are the main reason for the introduction of the attribute-value design. Returning to our example, assume a new kind of bonus needs to be introduced for pensioners `NumberOfDependentFamilyMembers`, along with the respective amount `FamilyMemberBonus`. Clearly, the flat model requires all the applications accessing the relation (data entry forms, stored procedures, application logic external programs, and simple presentation reports) to be (a) located (which by itself is a task much harder than it originally appears) and (b) appropriately maintained. On the other hand, the attribute-value design simply requires the insertion of a single record for the bonus type and the bonus amount in the lookup relations. Both designs also require the population of the master relation with the appropriate values (if this results from the business requirements).

Modification: Attribute Insertions/Deletions. The insertion or deletion of attributes is straightforward in both designs. The modifications at the attribute-value design are simpler, since they only involve tuples. Most importantly though, the applications accessing the attribute-value schema are practically immune to these changes, if appropriately authored (i.e., by taking the parameterized representation of the entity's properties in the database schema into consideration). Deletions are the most painful for the case of flat design, since the applications simply crash!

3.6 Critical assessment of the alternative designs

Clearly, the flat design is more efficient in terms of instance management and querying. Most querying operations in the attribute-value design require joins of a large master relation with the smaller lookup relations. Hash joins facilitate this kind of queries quite efficiently, still, the performance degradation compared to the flat design is evident. Modifications for the attribute-value design are also painful, since they require a two–step process for relating the property description with the appropriate record in the master relation.

Both solutions have space overheads, either due to multiple records per entity or due to the presence of NULL values. No clear winner can be a-priori assumed for the space overhead problem. NULL values pose an extra concern for counting queries, too.

In terms of schema evolution, the attribute-value design is a clear winner if the applications are appropriately constructed. All schema changes are reflected to tuple insertions and deletions; the applications are also immune with respect to the danger of crashing for the case of deletions.

Applicability: one could possible accept the flat design if (i) performance requirements impose it, (ii) schema modifications are rare, and (iii) the application code is appropriately stored and documented in such a way that maintenance is guided from an organized repository. In terms of deployment, client-server applications will probably suffer from the extra cost of re-deployment in the case of flat designs; on the other hand, web-based applications with their centralized deployment of software components are much easier to handle for the problem of re-deployment.

4 Materialization

4.1 Motivation

Materialization is a relationship between an abstract class and a set of concrete implementations of it. In the context of the object-oriented world, materialization is mainly a typing issue: the abstract class provides a customizable framework for the definition of a set of classes with similar structure and similar methods; the difference of the materializations of the abstract class has to do with the types of the variables and the method parameters.

We use the term materialization in order to deal with the separation of commonly repeated information as opposed to information which is different between instances. For example, a flight schedule between two cities has the same flight number, and the same standard hours of departure and arrival; still, every day that the flight is executed there is a different airplane that executes the flight, different crew members, etc.

Assume the case of a train organization of a country. The organization is responsible for providing connections between different cities of the country. Each connection between two cities has a set of standard, scheduled itineraries. Every itinerary has departure and arrival stations as well as scheduled departure and arrival times. These 'template' itineraries are realized by specific routes that take place. Each route realization has a date and actual departure and arrival times that are possibly different from the scheduled ones. Also the database of the organization records which train was actually used for the realization of the itinerary. Trains are organized in types and the organization is in possession of 3 types of trains, specifically, trains of small, medium and large capacity. Each train type has a name, a number of train wagons and a specific engine power. Trains belong to a train type and are named after their nicknames. Due to size limitations and the particularities of the tracks, there is an upper limit to the type that each connection can support.

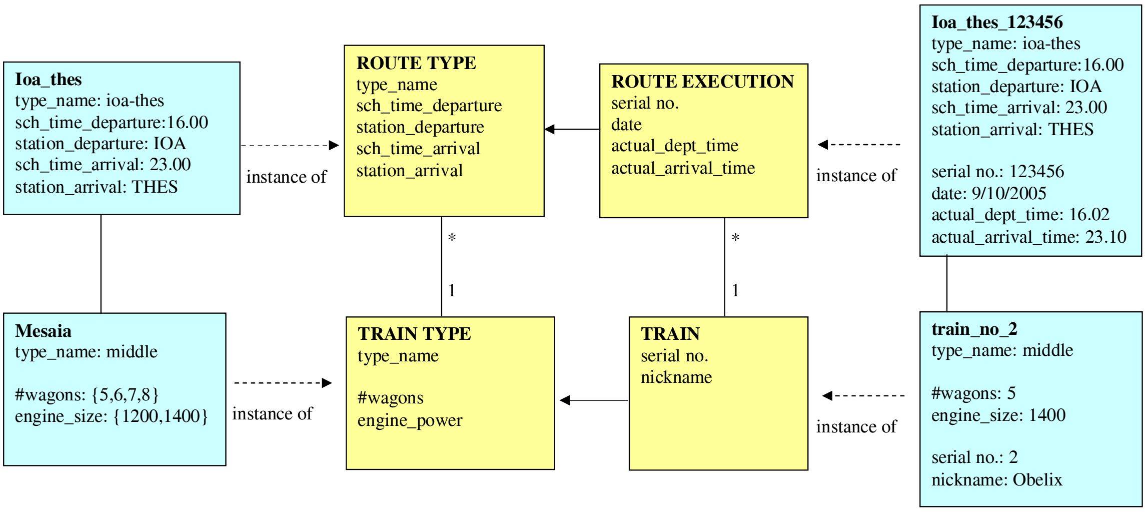

Figure 4.0. A high level informal description of the entities involved in the reference example

4.2 Design solutions

The main idea behind the solution is to separate the recurring and non-recurring parts of the data in different relations. We will refer to the former relation as the *abstract* or *template* relation and to the latter as the *concrete* or *template materialization* relation. The ABSTRACT relation contains the tuples that record categories (e.g., all the connections provided by the train company of the previous example) and the CONCRETE relation contains the specific characteristics of each individual implementation (e.g., the train used for a specific route on a specific date). The concrete relation is linked to the abstract relation via a foreign key; this way, a simple join of the two relations can give the full information for a specific flight execution. We will refer to the result of this join operation as the *full materialization* of the template.

For reasons of simple normalization, there is a need to differentiate the relation of the template from the relation of its materializations.

In Figure 4.1, we introduce a relation ABSTRACT with all the attributes capturing recurring information and a relation CONCRETE capturing the information that is differentiated in every realization of the abstract template. A foreign key connects the materialization to the template relation.

4.3 Interface to developers

There are two different aspects that need to be covered by the implementation of the database for a materialization scheme: (a) efficient management of updates and (b) efficient reconstruction of all the information for a specific instance, via the full materialization relation.

We observe that the structure of the pattern directly facilitates the update of the information. On the other hand the full materialization is obtained by a view `CONCRETE_FULL` that joins the two involved relations `ABSTRACT` and `CONCRETE` over the foreign key.

```
TRAIN_FULL = TRAIN_TYPE ⋈ TRAIN
CONCRETE_FULL = ABSTRACT ⋈ CONCRETE
```

4.4 Behavior at the instance level

Querying. The retrieval of a specific instance and the retrieval of all the instances of a certain materialization are facilitated via the view `CONCRETE_FULL`.

Modifications. The insertion, deletion and update of data is straightforward. The two relations must be linked with `ON DELETE / UPDATE CASCADE` assertions.

4.5 Behavior at the schema level

Due to its inherent normalized structure, the overall design handles schema modifications straightforwardly.

4.6 Discussion

There is nothing particularly fancy about the template pattern except that (a) it relates roughly to the idea of object-oriented factories and (b) it is an excellent tool to teach normalization in a class. The structure of a template provides an excellent testbed for the production of erroneous solutions by the students and the identification of the dangers of denormalization (specifically, inconsistent values due to data entry errors). An extra benefit is that the students visualize the template structure in their minds and have a concrete example (with the simple visual representation of Fig. 4.0, 4.3) as a reference tool that helps them understand the intuition and motivation behind the formalities of the normalization theory.

Alternative structures. The view CONCRETE_FULL can be materialized too. Clearly, this increases the query time with the extra overhead of replica maintenance. Still, since the size of the abstract class is expected to be significantly smaller that the one of the materialization, and in any case, quite small, we do not anticipate that the join of the two relations imposes a significant overhead (both hash joins and index-based joins can perform quite efficiently for this kind of queries). Therefore, materialization of the view can be envisioned only in cases with too strict QoS constraints on the response time of the queries.

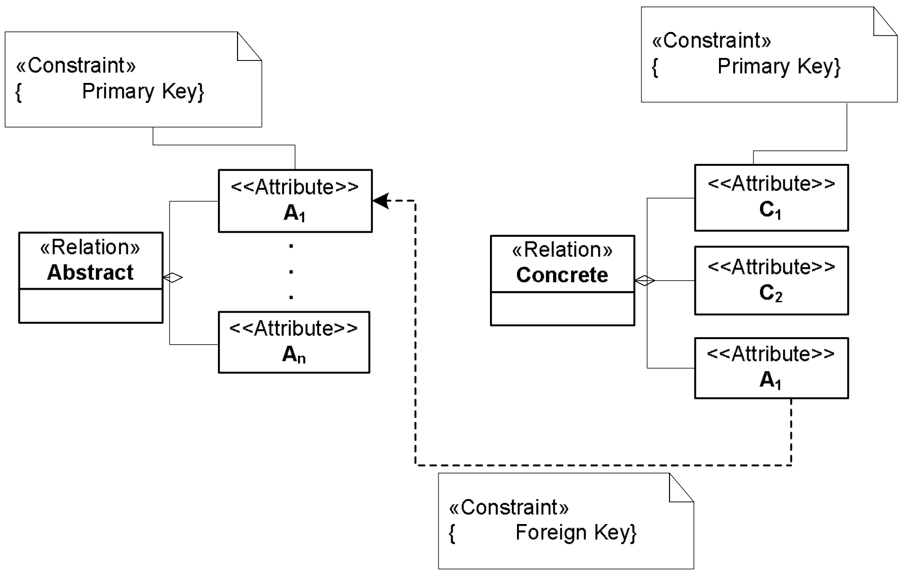

Fig. 4.1 Design Solution for the materialization of templates

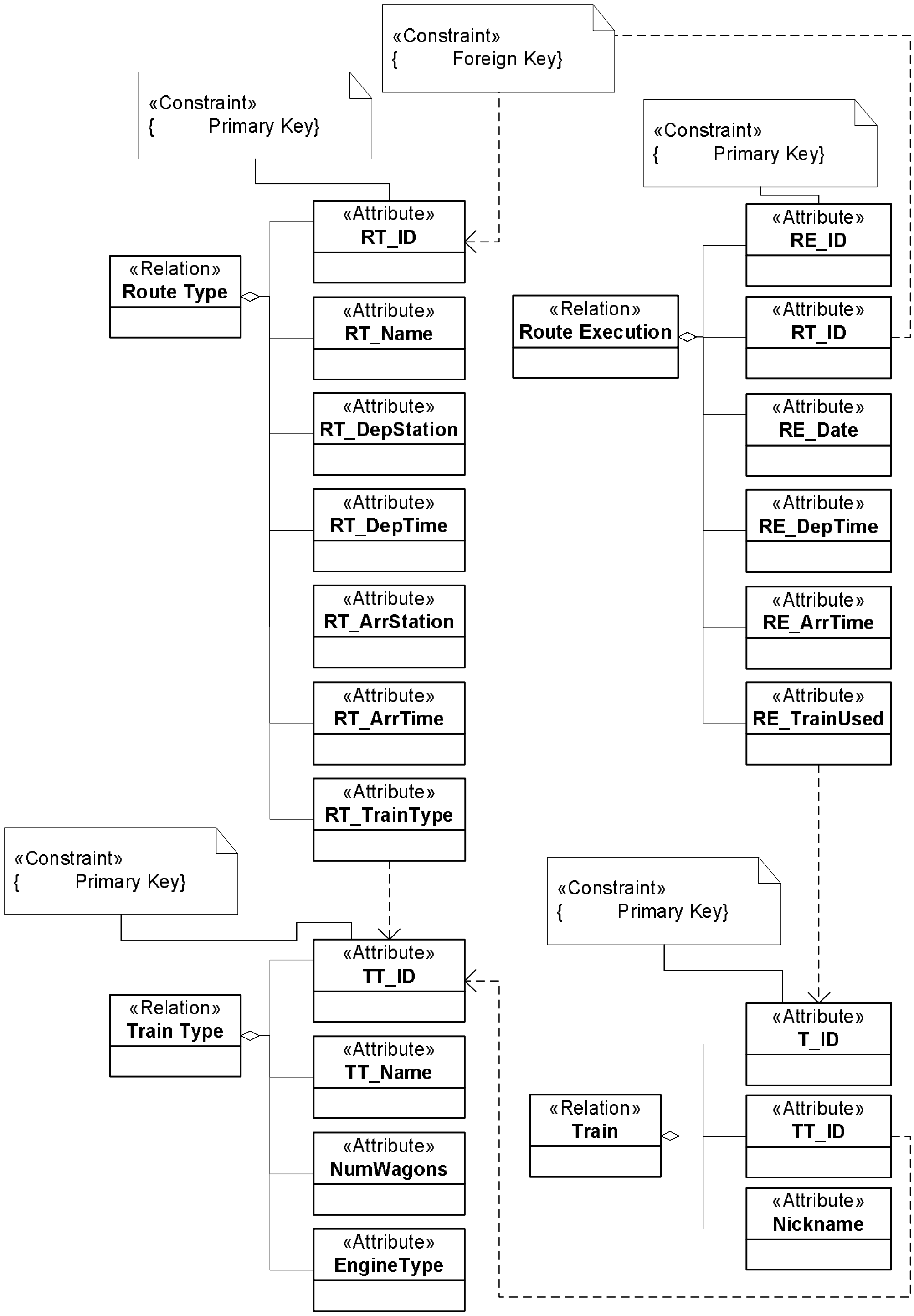

Fig. 4.2 A double, symmetric application of the materialization template for route types / routes and train types / trains

ROUTE-TYPE

RT_ID	**RT_Name**	**RT_DepStation**	**RT_DepTime**	**RT_ArrStation**	**RT_ArrTime**	**RT_TrainType**
1	M-O	Moscow	11.00	Omsk	16.00	120
2	O-T	Omsk	16.30	Tomsk	19.00	110
3	T-I	Tomsk	19.10	Irkutsk	22.00	110

ROUTE-EXECUTION

RE_ID	RT_ID	RE_Date	RE_DepTime	RE_ArrTime	RE_TrainUsed
1001	1	15/7	11.00	16.20	20
1002	2	15/7	16.31	19.05	10
1005	1	16/7	11.00	16.00	11

TRAIN-TYPE

TT_ID	TT_Name	NumWagons	EngineType
100	Small	15	1500
110	Middle	20	2000
120	Large	30	3000

TRAIN

T_ID	TT_ID	Nickname
10	110	Serko
20	120	Nikolai
11	110	Nadia

Fig. 4.3 Exemplary instance for the materialization pattern

5 Generalization and Specialization for Relational Databases

5.1 Motivation

In the context of conceptual modeling, generalization is the process via which a set of classes are abstracted via a higher-level class (also known as *parent*, or, *super class*) whose extent encompasses the instances of all these classes. Specialization is the inverse process, where a set of instances of a high-level are also assigned to a specialized new class (also known as *child*, *descendant*, or, *subclass*) with an extra, refined semantics. Typically, the relationship between a high-level class and one of its subclasses is referred to as an *IsA* relationship (shortcut for *is-a-subclass-of*). In both cases, the semantics of the IsA relationship is that the extent (i.e., the set of instances) of the subclass is a subset of the extent of the parent class. Frequently, for reasons of convenience, these subset semantics at the extent level are also accompanied with structural inheritance: the subclass inherits the structure of the super-class and extends it with extra properties, functionality or both.

Assume the following simple scheme. A mail company distributes surface mail. Each letter that the company delivers has a sender and a recipient. Letters are classified as (a) simple letters, with no extra information for them, (b) express letters, also carrying information for a guaranteed delivery data and (c) packages, whose weight is also recorded. Some of the packages are also fragile; for the latter, the kind of wrapping is also recorded.

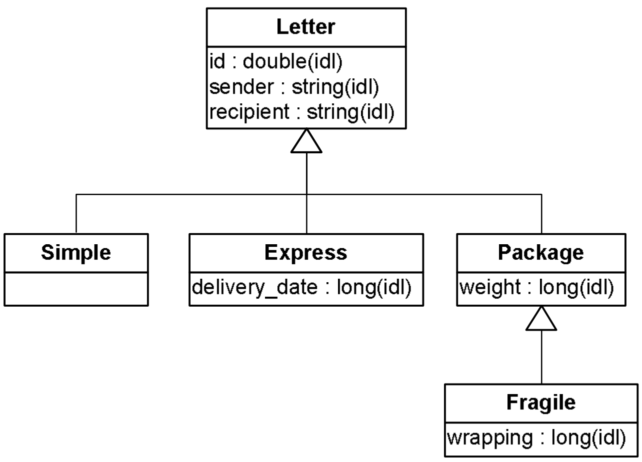

Fig 5.1. UML representation of the mail company classes.

The *requirements* for the proposed solution aim to support the following three fundamental properties of the object-oriented paradigm:

(i) Subset relationship between the extents of the super-class and its subclasses.

(ii) Polymorphic usage of the descendants by other constructs or applications.

(iii) Structural inheritance of the common super-class' attributes to the descendants and specialization of the descendants with extra attributes.

Any design pattern that provides a solution to the problem of inheritance should support the explicitly deal with the following common issues which are the direct representation of the aforementioned requirements in the relational world.

1. The pattern must allow the application developer to easily retrieve *all* the instances of a class – with the instances of its descendants included. We will consistently apply the following convention: for each class, we require (a) a view that returns all these instances and (b) a view that returns only the instances of its very own extent (i.e., without the instances of its subclasses). Assuming a class named `C` we will name these views `C_ALL`, `C_ONLY`, respectively. Any pattern, despite its internal structure must be in a position to support the definition of these two views.
2. The above solution also facilitates the requirement that a pattern must allow the polymorphic usage of the contents of relations: in other words, the application can be written with respect to the view `C_ALL` with the application developer free from the need to take care for collecting all the instances of the different subclasses. Still, there are two issues that are not resolved by the abovementioned solution: (a) how do we enforce that the population of the relations is performed correctly, and, (b) how do we allow foreign keys to parent or child relations? To deal with issues we require the patterns to explicitly deal with the issue of foreign keys to the ancestor class.
3. The final issue has to do with the location of the common and non-common attributes of the ancestor and descendent relations. In other words, the actual structure of the database schema has to be determined in order to support the aforementioned set of views that is likely to act as a programming interface for the developers who will access the database.

5.2 Modeling Solutions

In this section, we will present four design solutions that map an IsA relationship to a set of relational tables. We will discuss both the generic representation and the instantiation of the patterns to our reference example. In the rest of our deliberations, we will assume the existence of a super-class $A(\underline{A_1}, A_2, \ldots, A_n)$ and two of its subclasses, $B(B_1, B_2, \ldots, B_m)$ and $C(C_1, \ldots, C_l)$. Attribute A_1 is the primary key for the super-class relation and, due to the inheritance property it is also a primary key for the subclasses, too.

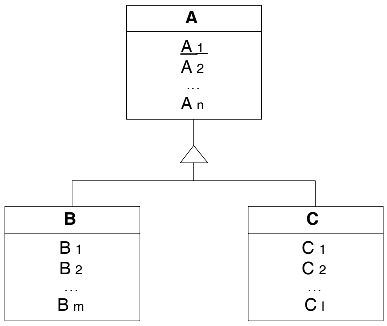

Fig 5.2 UML representation of the template IsA hierarchy we will use in the sequel

The first decision one has to make concerns whether (a) a single relation will be employed for the whole hierarchy, or, (b) a design that is coarsely directed towards one table per class will be chosen. Choosing a single relation for the whole hierarchy gives the simplest design pattern for the problem. On the other hand, choosing a strategy of one relation per class leads to a variety of design decisions that we present in detail in the following paragraphs.

Pattern: Single Table Hierarchy. The first design pattern, which we call *Single Table Hierarchy* is based on the idea of keeping a single relation with (a) all the tuples of all classes as its extent and (b) all the attributes of all the classes as its schema. An extra attribute, `Class_Type` is also part of the schema, in order to assign each tuple to the appropriate class. The class descriptions are captured in the relation `CLASSES` and

Class_Type is a foreign key to this relation. Observe that the relation CLASSES is also the place where the structure of the hierarchy is kept, via the attribute Parent.

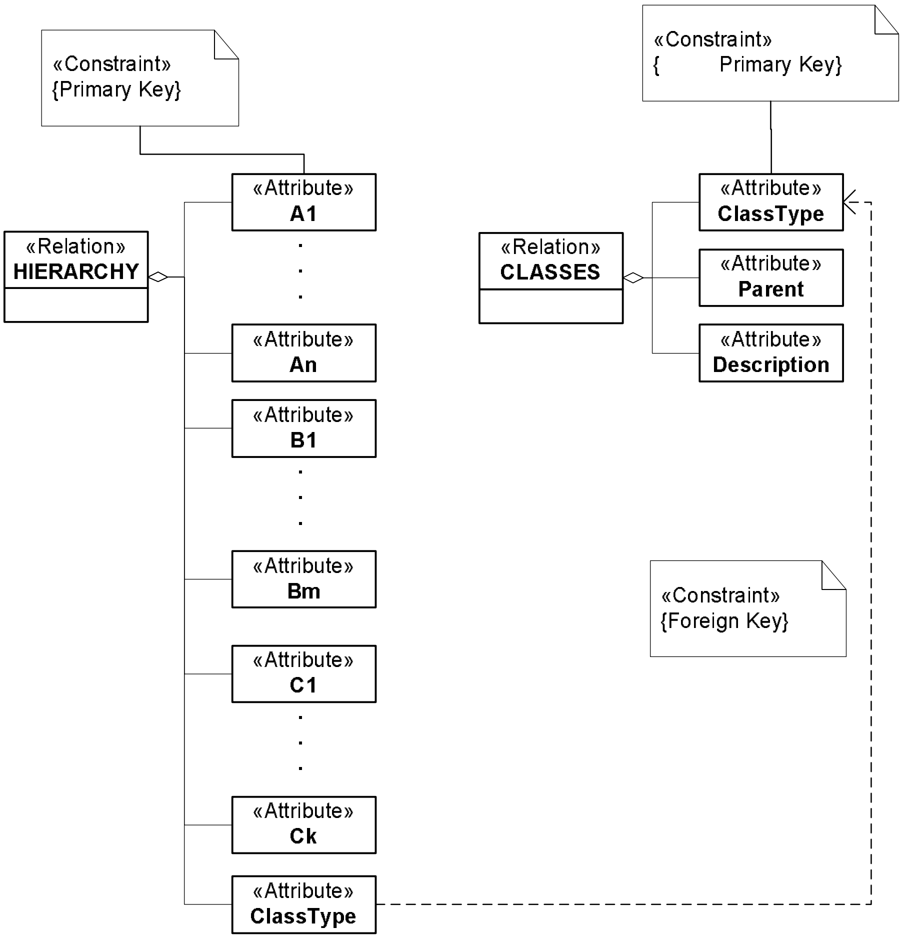

Fig 5.3 Single Table Hierarchy pattern

LETTERS

L_ID	Sender	Recipient	Dlv_date	Weight	Wrapping	Class_Type
1	Plato	Archytas				110
2	Paul	Titus				110
3	Aristotle	Theophrastus	15/07			120
4	Archimedes	Eratosthenes		200		130
5	Paul	Timothy		100	Hard	135

CLASSES

Class_Type	Parent	Class_Descr
100		Letter
110	100	Simple
120	100	Express
130	100	Package
135	130	Fragile

Fig 5.4 Exemplary Instance of the Single Table Hierarchy pattern

One relation per class. Apart from the previous strategy of storing all the hierarchy in a single table, another option is to try using one table per class, while keeping the hierarchy in auxiliary structures, too. Several decisions have to be taken in this case; these decisions are summarized in Figure 5.5.

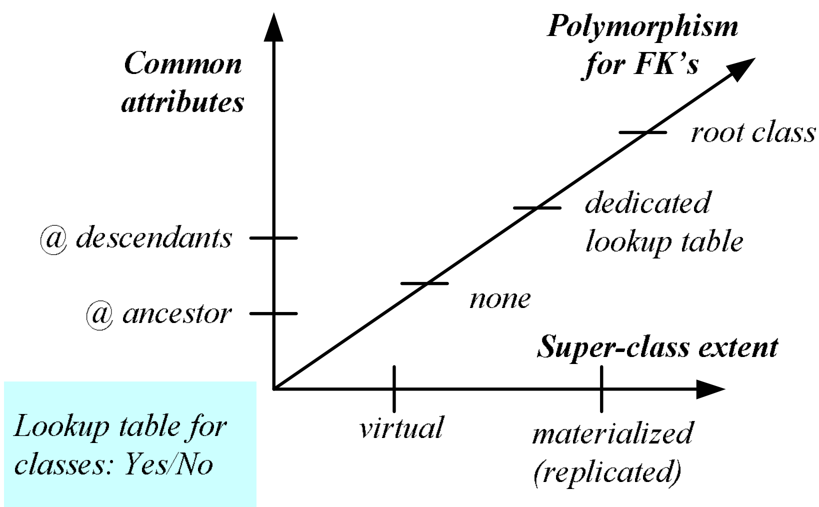

Fig 5.5 Space of alternatives for various subproblems

A first design choice has to do with the way the database schema allows the definition of foreign keys towards the tuples of the hierarchy, along with the necessary polymorphism this might entail. A first solution is to provide a reference-agnostic solution where the other relations can have foreign keys only to individual relations but not to the whole hierarchy. A second solution involves the usage of a very simple lookup relation `LOOKUP(OID,CLASS)` which keeps track of all tuples via an 'object id(`OID`)' as well as the actual table where the tuple is found. A third solution involves keeping *all* the tuples (or part of them) in the relation of the root class and allowing other relations to define foreign keys to the primary key of the root class. Moreover, there are two fundamental design choices concerning the location of the common attributes (in the super-class *only*, or in every relation) and the storage of the extent of the super-class, which includes all the tuples of its sub-classes (either to be virtually computed or replicated in the root class, too). We organize the presentation of the presented patterns around the two last design choices; Fig. 5.6 depicts the patterns that we present for these combinations.

COMMON FIELDS \ SUPERCLASS EXTENT	***Virtual***	***Materialized***
Only at ancestor	NOT APPLICABLE	*Vertical split*
At descendants	*Virtual super-class extent*	*Materialized super-class extent*

Fig. 5.6 names of presented patterns with their design choices

Pattern: Vertical split. The second design pattern that we present, *vertical split*, is based on the idea that common attributes between an ancestor and its descendants reside at the relation of the ancestor (Fig. 5.7 and 5.8). This allows the efficient querying of the super-class' full extent for the common attributes (a kind of query which is typical in polymorphic querying). At the same time, there is no need for a separate lookup relation for "object identifiers", since the root of the hierarchy encompasses all these identifiers at its primary key. Of course, the full reconstruction of a tuple of a descendant class requires joining the appropriate tuples at the ancestor and descendant relations.

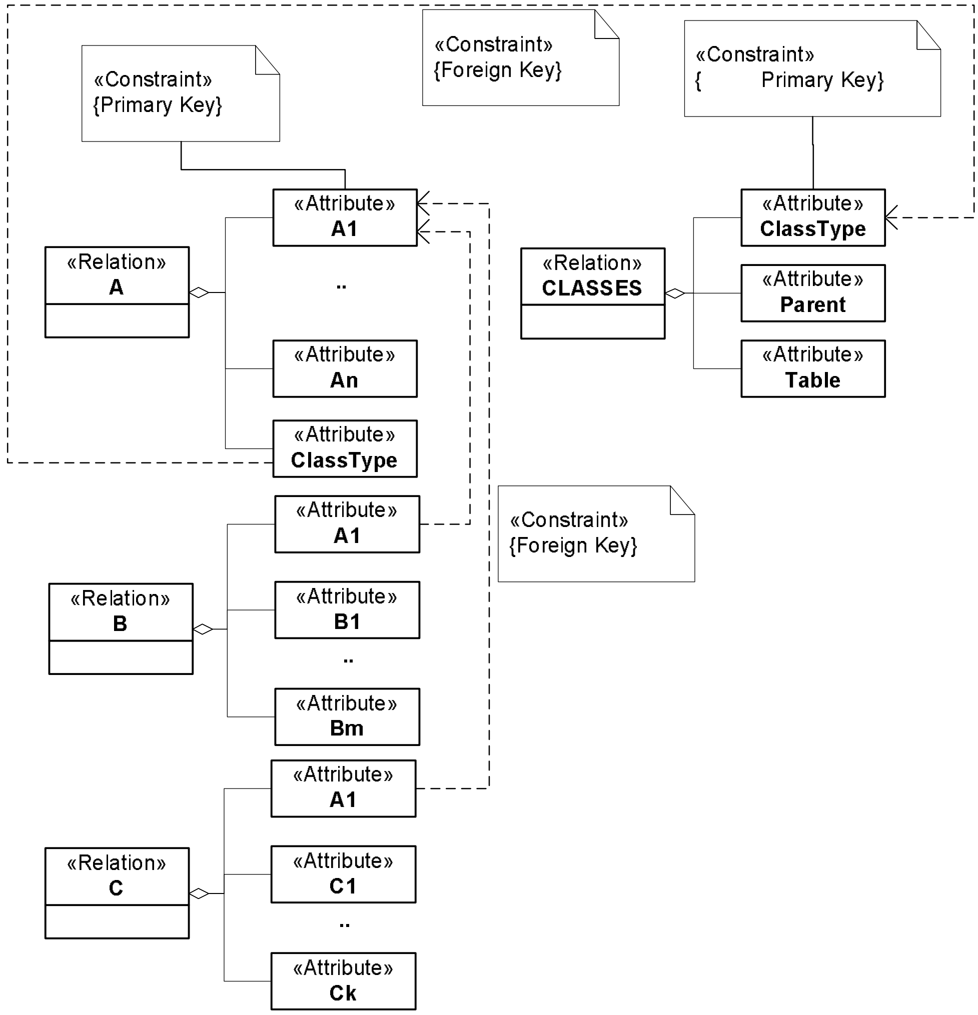

Fig 5.7 Vertical Split pattern.

Pattern: Virtual super-class extent. Once the idea of keeping the common attributes at the root class is abandoned, we result in relations whose schema has all the attributes of their corresponding class, independently on whether they are inherited or descendant-specific. The pattern *virtual super-class extent* is based on the idea that the extent of a super-class will be collected at runtime. Thus, each class has exactly the tuples that belong strictly to its very own extent; its full extent is collected via a view that performs the union of the respective sub-class relations. A consequence of this design is that there is no relation containing all the object identifiers; therefore, we introduce a lookup relation, ID_LOOKUP_TABLE, for this purpose. All polymorphic foreign keys are directed to this lookup relation. Fig. 5.9 depicts this pattern graphically and Fig. 5.10 presents an instance of this pattern.

LETTERS

L_ID	**Sender**	**Recipient**	**Class_Type**
1	Plato	Archytas	110
2	Paul	Titus	110
3	Aristotle	Theophrastus	120
4	Archimedes	Eratosthenes	130
5	Paul	Timothy	135

Express

L_ID	**Dlv_date**
3	15/07

Packages

L_ID	**Weight**
4	200
5	100

Fragile

L_ID	**Wrapping**
5	Hard

CLASSES

Class_Type	**Parent**	**Class_Descr**
100		Letter
110	100	Simple
120	100	Express
130	100	Package
135	130	Fragile

Fig 5.8 Exemplary Instance of the Vertical Split pattern

Pattern: Materialized super-class extent. This pattern aims to speed up the querying of the full class extent of a super-class, by replicating the instances of its subclasses in its extent. On the other hand, this feature incurs the danger of inconsistencies if the modifications of subclass extents are not automatically reflected to the super-class extent. A second difference with the virtual super-class extent pattern is that the replication alleviates the need for an extra lookup table; polymorphic foreign keys can now access the super-class relation which contains the common part for all the tuples for all the classes of the hierarchies.

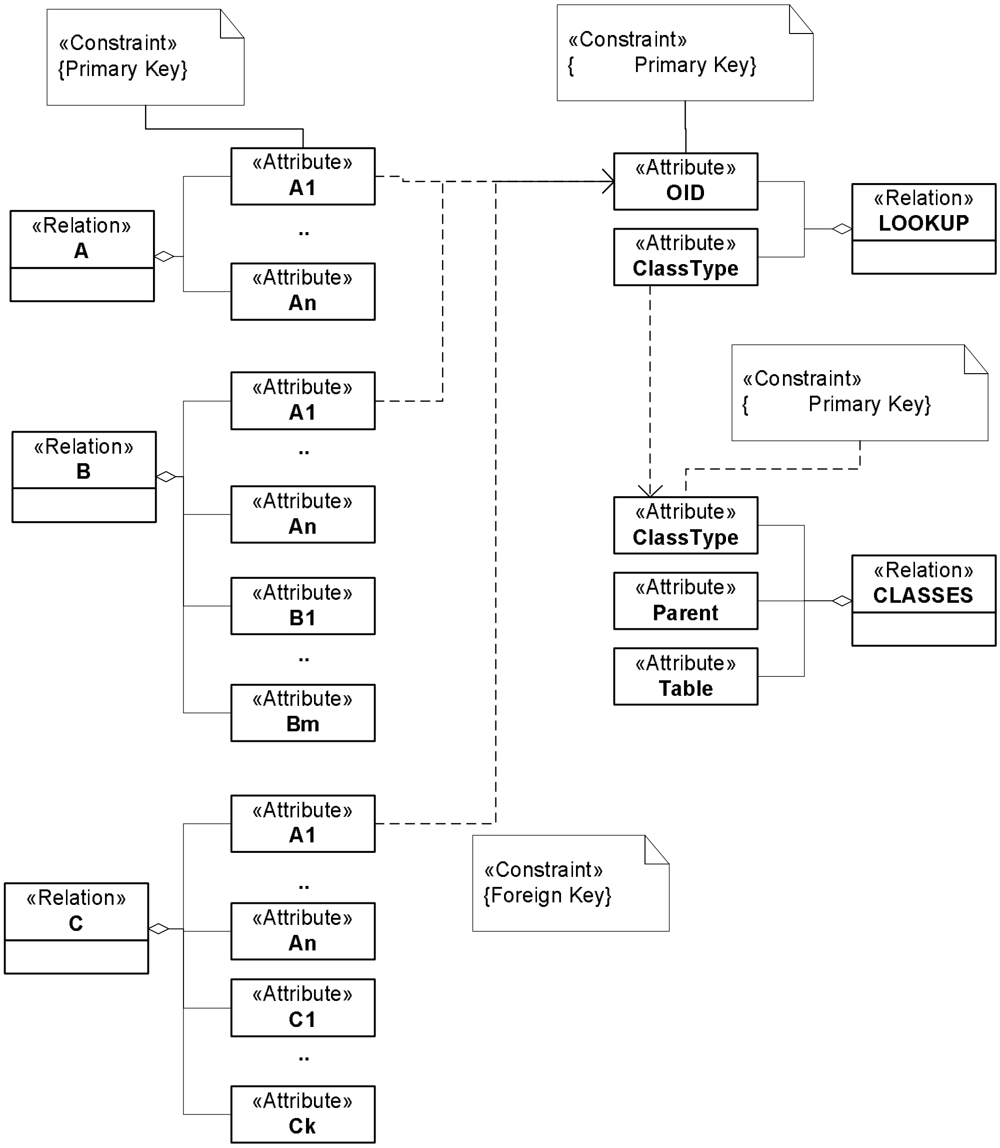

Fig 5.9 Virtual Super-class Extent pattern

SIMPLE

L_ID	Sender	Recipient
1	Plato	Archytas
2	Paul	Titus

EXPRESS

L_ID	Sender	Recipient	Dlv_date
3	Aristotle	Theophrastus	15/07

PACKAGES

L_ID	Sender	Recipient	Weight
4	Archimedes	Eratosthenes	200

FRAGILE

L_ID	Sender	Recipient	Weight	Wrapping
5	Paul	Timothy	100	Hard

LOOKUP

OID	Class_Type
1	110
2	110
3	120
4	130
5	135

CLASSES

Class_Type	Parent	Class_Descr
100		Letter
110	100	Simple
120	100	Express
130	100	Package
135	130	Fragile

Fig 5.10 Exemplary instance of the Virtual Super-class Extent pattern

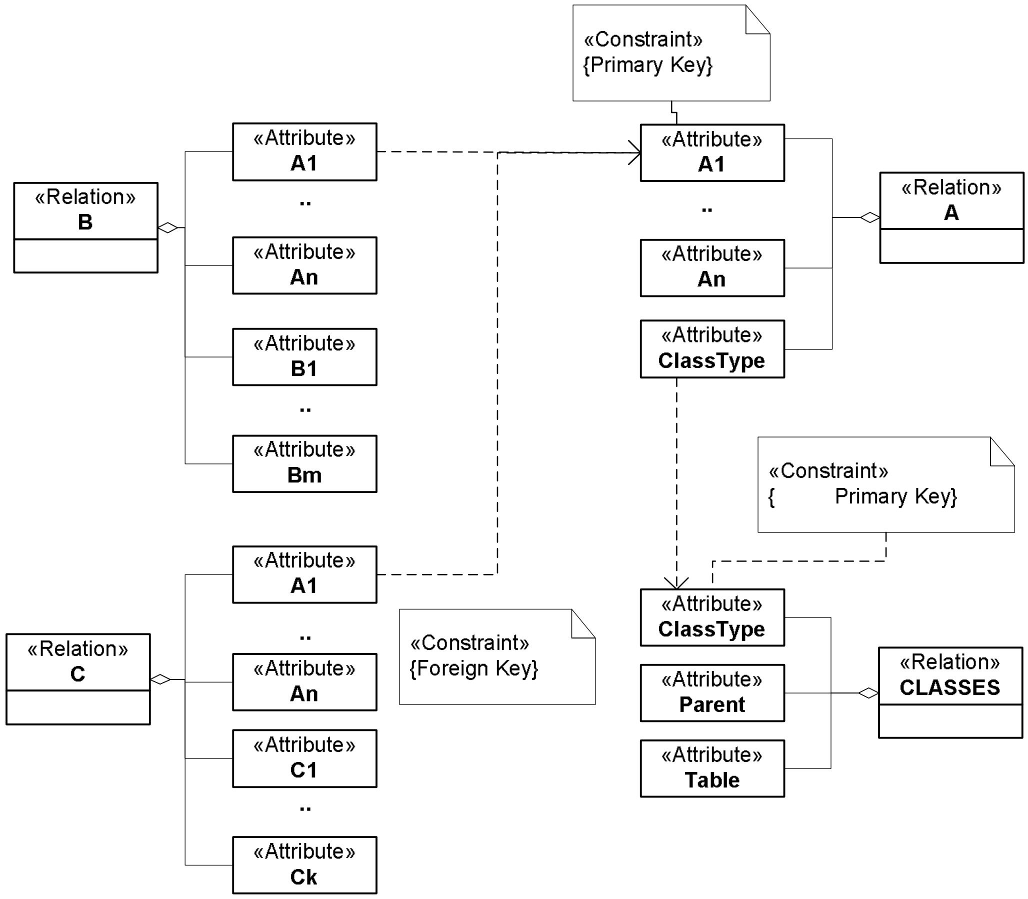

Fig. 5.11 Materialized Super-class Extent pattern.

CLASSES

Class_Type	Parent	Class_Descr
100		Letter
110	100	Simple
120	100	Express
130	100	Package
135	130	Fragile

LETTERS

L_ID	Sender	Recipient	Class_Type
1	Plato	Archytas	110
2	Paul	Titus	110
3	Aristotle	Theophrastus	120
4	Archimedes	Eratosthenes	130
5	Paul	Timothy	135

EXPRESS

L_ID	Sender	Recipient	Dlv_date
3	Aristotle	Theophrastus	15/07

PACKAGES

L_ID	Sender	Recipient	Weight
4	Archimedes	Eratosthenes	200
5	Paul	Timothy	100

FRAGILE

L_ID	Sender	Recipient	Weight	Wrapping
5	Paul	Timothy	100	Hard

Fig. 5.12 Exemplary instance of the Materialized Super-class Extent pattern.

5.3 Support of the logical-level programming interface

The querying of data in a schema that supports generalization is fundamentally based on a set of views that present a programming interface for applications and ad-hoc querying. Specifically, assuming a class named `C` we will employ a view named `C_ALL`, containing the full extent of the class. An auxiliary view `C_ONLY` can also be of help if applications require the extent of a class without the extents of its subclasses. Fig. 5.13 discusses the way to compute these two views for the different design alternatives.

	C_ONLY	C_ALL
SINGLE TABLE HIERARCHY	**Schema**: projection to the class' schema. **Extent**: selection of the instances that belong to the particular class.	**Schema**: projection to the class' schema. **Extent**: selection of the instances that belong to the classes of the hierarchy rooted at the class. Can be computed also as the union of C_ONLY and C_i_ALL of all the subclasses C_i.
VERTICAL SPLIT	**Schema**: join of the relations for all the classes in the path from the root class to the class. **Extent**: similarly, with an extra selection to remove instances that belong to subclasses.	**Schema & Extent**: join of the relations for all the classes in the path from the root class to the class.
VIRTUAL SUPER-CLASS EXTENT	**Schema & Extent**: simple query to the class' relation	**Schema & Extent**: union of all the relations of the subclasses (projected over the class' schema) with the class' relation.
MATERIALIZED SUPER-CLASS EXTENT	**Schema & Extent**: simple query to the class' relation with an extra selection to remove instances that belong to subclasses.	**Schema & Extent**: simple query to the class' relation

Fig. 5.13 View computation for different patterns

Should the two views be virtual or materialized? The reader is reminded that a virtual view acts like a macro: each time a query over a view is posed, the query is automatically rewritten to replace the view with its definition. A materialized view on the other hand has its extent fully computed; this provides the extra benefit that the tuples to be processed are already available at query time. Still apart from the space overhead, the materialized view incurs the extra maintenance cost of refreshment whenever the contents of its underlying source relations are modified. Fortunately, modern DBMS's take care of performing this refreshment automatically.

5.4 Behavior at the instance level

We will discuss the following operations at the instance level: (a) retrieval of all the information around a certain record, (b) retrieval of all the instances of a class and (c) insertion, deletion and updates of a certain tuple.

Tuple retrieval. The retrieval of the full extent of a class is straightforward, via a `SELECT * FROM C_ALL` query. The retrieval of individual tuples, nevertheless, poses additional challenges. Assuming that the user has retrieved the primary key of a tuple (e.g., via another query on any of the rest of the attributes), the task of tuple reconstruction requires (a) the identification of the class to which the tuple belongs and (b) the retrieval of the tuple from any of the two views that act as an API. Fig. 5.14 presents the way to perform this action for the alternative solutions.

	TUPLE'S CLASS KNOWN	TUPLE'S CLASS UNKNOWN
SINGLE TABLE HIERARCHY	Simple query to the relation itself; no need for views.	
VERTICAL SPLIT	Simple query to the appropriate C_ONLY or C_ALL view (depending on the faster of the two)	Derive the tuple's class via a simple query to the root class; then, a second query to the appropriate view is due.
VIRTUAL SUPER-CLASS EXTENT	Simple query to the C_ONLY view	Simple query to the lookup relation class; then, a second query to the appropriate view is due.
MATERIALIZED SUPER-CLASS EXTENT	Simple query to the appropriate C_ONLY or C_ALL view (depending on the faster of the two)	Derive the tuple's class via a simple query to the root class; then, a second query to the appropriate view is due.

Fig. 5.14 Tuple reconstruction for generalization patterns

In terms of efficiency, for the case when the appropriate view to query is not obvious, simple cost considerations clarify the appropriate choice as follows. If an index is present, then there is no real difference for all practical purposes. In the case of the absence of an index, if the computation of the irrelevant tuples from the underlying relation is expensive for C_ONLY, then C_ALL should be preferred; otherwise, C_ONLY is the appropriate choice.

Tuple modifications. Tuple modifications involve the insertion, deletion, and update of records.

- *Single table hierarchy*: All operations are straightforward. Still, insertions and updates have the extra overhead to populate the correct attributes depending on the class being updated.
- *Vertical split*: The modification program must take care of updating the appropriate relations, depending on the class of the modified tuple. Automating the consistency of deletions via ON DELETE CASDACE assertions is also useful among the subclass and super-class relations.
- *Virtual super-class extent*: The lookup relation must always be updated in insertions; every other operation is straightforward. In the case of deletions

and updates, if the class of the tuple is not known, a lookup must be performed first to the lookup relation.

– *Materialized super-class extent*: All operations are straightforward. If the class of the modified tuple is not known, then a lookup at the root relation must be performed. Assertions `ON DELETE/UPDATE CASDACE` for deletions and updates are necessary for the automation of these processes.

5.5 Behavior at the schema level

We are concerned with two types of schema modification: (a) change in the set of attributes of a class and (b) change in the set of classes of a hierarchy.

Attribute-level modifications. Attribute level modification involves the addition of a new attribute, the deletion of an existing one and the update (rename, type alteration) of an existing attribute. We assume that primary keys are not modified under any circumstance. Again, all operations are straightforward for the single table hierarchy and vertical split design solutions, as the class under modification determines and the relation to be updated too. For the cases of virtual and materialized super-class extents, the modifications must be repeated to all the descendants of the modified class.

Class-level modifications. Class-level modifications involve the addition of new classes and the deletion of existing ones. We assume deletions of leaves in the class hierarchy (all other deletions can be reduced to sequences of leaf deletions). Modifications of classes have been dealt with in the attribute-level modifications.

Single table hierarchy involves simply adding or deleting the appropriate attributes for the hierarchy's relation. All the multi-relation patterns require the addition of a new relation (with a foreign key to the appropriate root or lookup relation), or the deletion of an existing relation (respectively). All operations require the update of relation `CLASSES` and the readjustment of the views `C_ALL` and `C_ONLY`.

5.6 Critical assessment of alternative designs

In this subsection, we summarize the benefits and vulnerabilities of the alternative designs that we have proposed.

	SINGLE TABLE	VERTICAL SPLIT	VIRTUAL EXTENT	MATERIALIZED EXTENT
STORAGE				
NULL values	☹☹	☺	☺	☺
Redundancy	☺	☺	☺	☹☹
QUERYING				
Complexity of `C_ONLY`	☺☺	☹☹	☺☺	☺
Complexity of `C_ALL`	☺☺	☹	☺	☺☺
UPDATES				
INS tuple	☹	😐	😐	☹
DEL tuple	☺	☹	😐	☹
UPD tuple	☹	☺	😐	☹
SCHEMA MODIFICATIONS				
ADD field	😐	😐	😐	😐
DEL field	😐	😐	😐	😐
UPD field	😐	😐	😐	😐
ADD class	😐	😐	😐	😐
DEL class	😐	😐	😐	😐

Figure 5.15 Comparative description of alternative designs for the generalization problem

Structure. Obviously, the single table hierarchy design is practically denormalized; as such it suffers both from data entry problems and from a multitude of NULL values. Apart from the space management overheads, this has the extra overhead of having to take care of counting queries.

Virtual classes. Virtual classes are characterized by the absence of instances that belong only to their own extent and not in any of their subclasses; in other words, each of their instances belongs to the extent of one of their subclasses. Solutions with materialized super-class extents remain unaffected from the virtual character of the super-class since the subclass instances are stored in the super-class relation (in terms of the common attributes). Solutions with virtual super-class extents are also unaffected due to the usage of the lookup relation; in this case it is possible to omit the super-class relation from the schema since it has no instances anyway.

6 Conclusions

We believe that design patterns are a clear need for the database world as they can serve as guiding aids and reference language for designers, especially in their early steps. In the University of Ioannina we have used the abovementioned problems and patterns in the context of an advanced undergraduate elective database course. The results have been encouraging, since:

- the students were eager to participate and quite often they embarked in the task of devising alternatives for the solutions that we discussed,
- too many issues concerning fundamental notions of the database world were revisited with a clear viewpoint once patterns were introduced (for example, materialization is a very good starting point to discuss normalization; generalization demonstrates nicely the benefits of foreign keys, etc),
- the activity of teaching best practices via examples is always very helpful for the instructor, too, since the weaknesses of the students are very clearly demonstrated.

Clearly, too many issues are open; the main issue is a clarification of how we do view the fundamental structure of design patterns for databases. More patterns have to be devised, a balanced organization must be extracted (not too detailed and not too simplistic) and the deep foundations of why a solution is good must be further investigated (possibly via concrete metrics rather than rumor or inconclusive experiments).

References

[Codd70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. Commun. ACM 13(6): 377-387 (1970)

[Dahc98] M. Dahchour. "*Formalizing Materialization Using a Metaclass Approach*". CAISE 1998: pp. 401-421, 1998.

[DKPZ05] M. Dahchour, M. Kolp, A. Pirrote and E. Zimanyi. "*Generic Relationships in Information Modelling*", Journal of Data Semantics, Volume 4, 2005.

[Dong04] J. Dong. "*Adding pattern related information in structural and behavioural diagrams*". Information & Software Technology 46(5): pp.93-300, 2004.

[GHJV95] E.Gamma, R. Helm, R. Johnson and J. Vlissides. "*Design Patterns Elements of Reusable Object-Oriented Software*". Professional Comptuting Series. Addison Wedley, Reading, 1995.

[Kolp01] M. Kolp. "*Semantics Relationships*", Lecture Semantics Relationships, University of Toronto, 2001. In collaboration with A. Pirotte and M. Danhchour, University of Louvain.

[Mylo98] J. Mylopoulos. "*Information Modelling in the Time of the Revolution*", Information Systems 23(3-4), pp. 127-156, June 1998.

[TrBu07] A. Tropashko and D. K. Burleson. "*SQL Design Patterns: Expert Guide to SQL Programming*". Rampant Techpress. IT In-Focus, April 2007.

[Wins05] M. Winslett. Bruce Lindsay speaks out. SIGMOD Record, Vol. 34, No. 2, June 2005.