A quantum logic-based query processing approach for extending relational query languages

Sebastian Lehrack
Institut für Informatik
Brandenburgische Techn. Universität Cottbus
slehrack@informatik.tu-cottbus.de

May 1, 2009

Abstract

Evaluating a traditional database query against a data tuple returns a `true` on match and a `false` on mismatch. Unfortunately, there are many application scenarios where such an evaluation is not possible or does not adequately meet users needs. A further problematic application area is text retrieval where in general finding a complete match is impossible. Thus, there is a need for incorporating impreciseness and proximity into a logic-based query language: Objects fulfill such a *similarity condition* to a certain degree which is expressed by a result value out of the interval [0, 1]. In this work we will sketch a quantum logic-based approach, which provides the combination of classical Boolean predicates and similarity conditions into one integrating formalism.

1 Introduction

To motivate our query processing approach we want to consider an example, which is dealing with the assessment of TV sets. So, in an online shop a user may look for a very comfortable but inexpensive TV set by means of a query expressing these requirements. In a database following attributes are saved for a certain TV set: `name, price, status, handling` and `image_quality`. The two last attributes contain a rating of the respective properties decoded as marks from 1 to 6. Thereby, the mark 1 stands for an *excellent* test result and the mark 6 confirms an *inadequate* quality for the tested feature. The Table 1 gives an extracted part of the entire data spreadsheet. The user defines his query more precisely as: I want to find a device, which can be handled as easy as possible and its price does not exceed 1.000 Euro. If otherwise the cost of a certain TV set is more than 1.000 Euro, it should provide the best possible quality of image at least. The vagueness of the subconditions *handling as easy as possible* and *best possible quality of image* cannot be mapped to Boolean truth values. In Table 1 the truth values for a Boolean evaluation are given in parentheses, when the threshold value for an acceptable mark is assumed to 2.

TV sets				
`name`	`price`	`handling`	`image_quality`	...
TV1	500 (1)	3 (0)	4 (0)	...
TV2	800 (1)	2 (1)	2 (1)	...
TV3	900 (1)	1 (1)	4 (0)	...
TV4	2000 (0)	2 (1)	1 (1)	...
...	...	...	...	...

Table 1: Spreadsheet of tested TV sets

Obviously, important informations are getting lost by the usage of classic Boolean logic. Thus, the provided result items are not distinguishable at all. For instance, the TV sets `TV2, TV3` and `TV4` return all the *same* positive

query result `true`, in spite of the fact that TV set `TV2` has to be acknowledged as the best choice, when all three subconditions are taken into account.

Next section gives a short overview to the theoretical model behind the quantum-logic query processing and introduces similarity calculus CQQL: *Commuting Quantum Query Language*. Section 3 will sketch the structure of the similarity language family based on CQQL.

2 The Quantum Query Language CQQL

In the following section we present an introduction to the theoretical model behind CQQL. Especially, we will sketch the structure and the evaluation of a CQQL query. A more detailed description of theory can be found in [1].

In general, CQQL enables the logic-based construction of queries from traditional Boolean and similarity conditions. The underlying idea is to apply the theory of vector spaces, also known from quantum mechanics and quantum logic, for query processing. Table 2 gives the correspondences between query processing concepts and the adapted vector space model of CQQL.

query processing		**vector space model of CQQL**	
query system	-	vector space	**H**
tuple to be queried	t_i	vector	$v[t_i]$
query	q	vector subspace	$vs[q]$
evaluation	$eval(t_i, q)$	squared cosine of the angle between $v[t_i]$ and $vs[q]$	$cos^2(\alpha(v[t_i], vs[q]))$

Table 2: Correspondences between query processing and the model of CQQL

Before we go in more detail, we want to summarise the basic idea of evaluating a given tuple t_i against a given CQQL query q. We start by considering a vector space **H** containing all encoded elements of the query processing (see Table 2). All attribute values of a tuple t_i are embodied by the direction of a normalized tuple vector $v[t_i]$. The query q itself corresponds to a vector subspace $vs[q]$ located in **H**. The position and the expansion of the subspace $vs[q]$ bijectively correlates to the semantic of q. To distinguish the subspace $vs[q]$ from the containing vector space **H** we will denote $vs[q]$ as *query space*.

Furthermore, the evaluation result of a tuple t_i against the query q is determined by the minimal angle, denoted as $\alpha(v[t_i], vs[q])$, between the tuple vector $v[t_i]$ and the query space $vs[q]$ in **H**. The squared cosine of this angle is a value out of the interval $[0, 1]$ and can therefore be interpreted as a similarity measure as well as a score-value. If the tuple vector belongs to the query space, i.e. $\alpha(v[t_i], vs[q]) = 0°$, then we interpret the query outcome as a complete match: $cos^2(0°) = 1$. Contrarily, a right angle of 90° between $v[t_i]$ and $vs[q]$ leads to a complete mismatch: $cos^2(90°) = 0$.

Based on this main idea we want to further discuss the two central issues: (1) the construction of the tuple vector $v[t_i]$ and the query space $vs[q]$ and (2) the fast computation of the evaluation result as $cos^2(\alpha(v[t_i], vs[q]))$. To study these both topics we assume the following example query, which is associated with the already known TV set scenario:

$$q = \{(name, status, handling, \ldots) \mid TV(name, status, handling, \ldots) \wedge (status = a \vee status = o) \wedge handling \approx 1\}$$

The domain of the attribute *status* contains the three values *available*, *sold* and *ordered*, whereby the underlined abbreviations are used for the sake of convenience, i.e. $Dom(status) = \{a, s, o\}$. Thus, the query determines all TV sets, which are available or ordered and own a handling mark as good as possible. This query will be validated against the two tuple $t_1 = (TV1, o, 3, \ldots)$ and $t_2 = (TV2, s, 2, \ldots)$.

2.1 Construction of the tuple vector $v[t_i]$ and the query space $vs[q]$

To construct the elements $v[t_i]$ and $vs[q]$ we employ a typical bottom-up strategy based on the logical composition of q. Thus, atomic conditions, also called *predicates*, can be considered as smallest evaluable entities, which are getting combined by the logical connectors $\wedge, \vee$ and $\neg$ to form the final query q. We will exploit this construction principle (going from single predicates over combined subconditions to the final query) to deploy the intended elements.

According to their semantic atomic conditions can be divided in two different main types: Boolean and similarity predicates. For instance, the introduced example query includes three atomic conditions '$handling \approx 1$', '$status = a$' and '$status = s$', whereby the first one can be classified as a similarity predicate and the last two conditions are typical Boolean predicates. The left relation predicate $TV(name, status, handling, \ldots)$ is also a Boolean predicate, but for the discussion of this special case we refer to [1]. At the first construction step we set up a *separate* vector space $\mathbf{H_{p_j}}$ for each Boolean and similarity predicate p_j.

These single vector spaces possess the character of basic modules for the construction of the final vector space $\mathbf{H}$. This behaviour is similar to the structural meaning which is owned by predicates according to the query q. In the following we simple name this kind of a separate vector space as a *basic module*.

There are two information entities, which have to be encoded in such a basic module. On one hand, we have the value of the queried tuple attribute. For example, taking the predicate '$status = a$' we must encode the value o for the attribute $status$ of t_1, which has to be tested against the predicate condition: Is o equal to a? So, the value o forms the tuple vector $v[t_1^{status}]$. On the other hand, the comparison constant of the predicate condition, e.g. the constant 1 for $handling \approx 1$, must be integrated into $\mathbf{H_{p_j}}$ as the query space $vs[handling \approx 1]$.

The applied method for the encoding directly depends on the predicate type, whereby several implications must be considered. Primarily, the encoding must guarantee a valid outcome for the minimal angle between $v[t_i^{attr}]$ and $vs[p_j]$. So, the angle $\alpha(v[t_i^{attr}], vs[p_j])$ has to be either $0°$ *or* $90°$ for a Boolean predicate. In contrast, for a similarity predicate the angle $\alpha(v[t_i^{attr}], vs[p_j])$ can be evaluated as a value *between* $0°$ *and* $90°$. These constraints determine the structure of a basic module $\mathbf{H_{p_j}}$ as well the integration of $v[t_i^{attr}]$ and $vs(p_j)$ into $\mathbf{H_{p_j}}$.

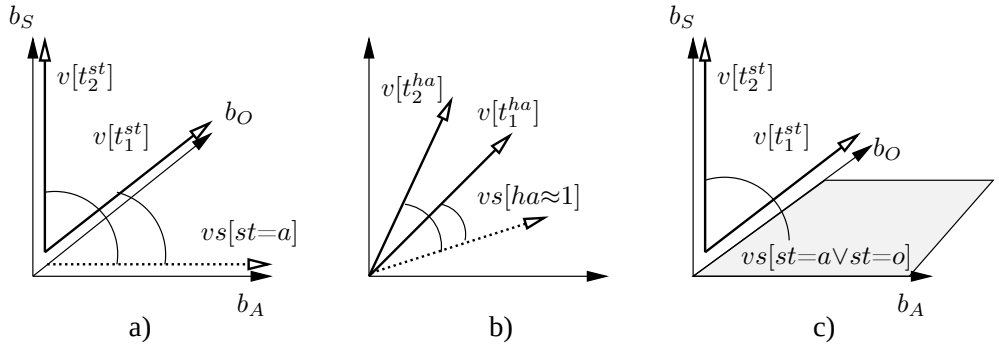

Figure 1: Basic modules

Basic modules for Boolean predicates For Boolean predicates each domain value of the queried attribute constitutes a orthonormal basis vector for $\mathbf{H_{p_j}}$. So, for instance a 3-dimensional basis module is built for the Boolean predicate '$status = a$', whereby the basic vectors b_A, b_S and b_O represent the domain values available, sold and ordered.

To encode the attribute value of t_i^{attr} and the condition constant of p_j we map the corresponding elements $v[t_i^{attr}]$ and $vs[p_j]$ to basis vectors embodying the same domain value. Figure 1a) depicts the basic module for the Boolean predicate '$status = a$' including the query space $vs[st = a]$, and the tuple vectors $v[t_1^{st}]$ and $v[t_2^{st}]$. Please notice that in a basic module the query space $vs[p_i]$ only spans one dimension and is therefore equivalent to a single vector. By studying

Figure 1a) we can confirm that all angles between tuple vectors and query spaces must be either $0°$ or $90°$ depending on the considered attribute values and condition constants. In this example both attribute values o (for t_1) and s (for t_2) leads to a mismatch: $\alpha(v[t_i^{st}], vs[st = a]) = 90°$ for $i = 1, 2$.

Basic modules for similarity predicates A basic module for an arbitrary similarity predicate has always two dimensions. So, the attribute value and the condition constant must be represented by non-orthogonal vectors embedded in the 2-dimensional vector space $\mathbf{H_{p_j}}$. In Figure 1b) the basic module for the similarity predicate '$handling \approx 1$' is given. We conclude that the angles between the attribute vectors $v[t_1^{ha}]$, $v[_2^{ha}]$ and the query space $vs[ha \approx 1]$ express the similarity between these elements in an interval from 0 ($cos^2(90°)$) to 1 ($cos^2(0°)$).

Combining basic modules For the combining of basic modules to the final vector space $\mathbf{H}$ we use the tensor product of two vector spaces [1]. By the usage of this algebraic operation two vector spaces $\mathbf{H_{p_j}}$ and $\mathbf{H_{p_k}}$ are getting entangled in a new generated vector space: $\mathbf{H_{p_j}} \otimes \mathbf{H_{p_k}} = \mathbf{H_{c_l}}$. The dimensionality of the product space $\mathbf{H_{c_l}}$ is in general higher than the dimensionalities of both input spaces.

The attribute tuples and the query spaces within $\mathbf{H_{p_j}}$ and $\mathbf{H_{p_k}}$ are also getting transfered into the product vector space, whereby the two attribute vectors merge to a single multi-attribute vector and the query spaces express a combined condition.

The specific combining operation for query spaces corresponds to the logical operators $\wedge, \vee$ and $\neg$ of the underlying formula q. For example, two query spaces $vs[c_1]$ and $vs[c_2]$ are getting intersected, when the respective subconditions c_1 and c_2 are connected conjunctively. Following operations for the merging of query spaces are defined:

$$\begin{aligned} vs(c_1 \wedge c_2) &\stackrel{def}{=} vs(c_1) \cap vs(c_2), \\ vs(c_1 \vee c_2) &\stackrel{def}{=} \text{closure}(vs(c_1) \cup vs(c_2)), \\ vs(\neg c_1) &\stackrel{def}{=} \mathbf{H_{c_l}} \setminus vs(c_1). \end{aligned}$$

The closure operation generates the set of all possible vector linear combinations. By applying these rules the final vector space $\mathbf{H}$, the multi-attribute tuple vector $v[t_i]$ and the combined query space $vs[q]$ can be constructed recursively.

The Figure 1c) gives an idea for the combination of query spaces. The condition '$status = a \vee status = o$' generates a combined query space as a plane spanned by the vectors b_A and b_O. Since the considered condition only queries the attribute $status$, the query space $vs[st = a \vee st = o]$ is still placed in a single basic module instead of a product space. The measurement of the angle between $v[t_1^{st}]$ and $vs[st = a \vee st = o]$ returns $0°$. It correlates to the fact that $t_1^{st} = o$ fulfill the subcondition '$st = a \vee st = o$'. On the other side, there is an angle of $90°$ between $vs[t_2^{st}]$ and the query space $vs[st = a \vee st = o]$, which is affirming a mismatch between $t_2^{st} = s$ and '$st = a \vee st = o$'.

2.2 Fast computation of $cos^2(\alpha(v[t_i^{attr}], vs[p]))$

Generally, the squared cosine of the minimum angle between $v[t_i]$ and $vs[q]$ can be computed by using a set of basis vectors for the constructed query space $vs[q]$. From a computational view point such a method is quite inefficient, because in general the cardinality of a basis vector set increases tremendously.

An alternative method is developed in [1]. It allows to evaluate a tuple t_i against a complex CQQL query q by using of simple arithmetic operations recursively:

$$
\begin{aligned}
eval(t_i, p) &:= \varphi_{attr}(t_i^{attr}, p), \\
eval(t_i, c_1 \wedge c_2) &:= eval(t_i, c_1) * eval(t_i, c_2), \\
eval(t_i, c_1 \vee c_2) &:= eval(t_i, c_1) + eval(t_i, c_2) - eval(t_i, c_1 \wedge c_2), \\
eval(t_i, \neg c) &:= 1 - eval(t_i, c),
\end{aligned}
$$

whereby p is an atomic condition (predicate) and c, c_1 and c_2 are arbitrary subconditions. The function $\varphi_{attr}(t_i^{attr}, p)$ returns $cos^2(\alpha(v[t_i^{attr}], vs[p]))$ for the predicate p and the respective attribute value of t_i.

For the correct application of the defined operations a specific *syntactical form* of the CQQL query q is needed. Since our language obeys the same transformation rules as held by the Boolean logic we can convert every CQQL query into the required syntactical form. An algorithm performing this transformation is presented in [1].

3 Language family

Since Codd has introduced the relational data model in the 70s a lot of relational query languages have been proposed. Especially, the relational calculuses, the relational algebra and the database query language SQL have grown a strong theoretical and practical significance. Under certain circumstances all three languages can be considered as equivalent expressive. However, they distinguish from each other by their areas of application and their user groups.

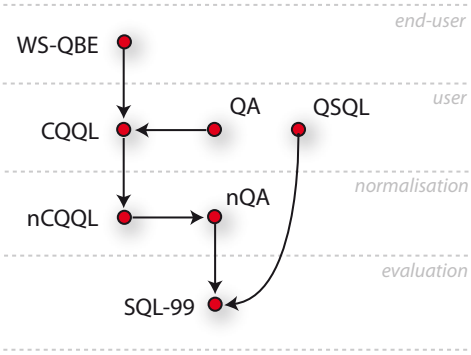

Figure 2: Similarity query languages

In the previous section we presented the basic ideas of our query processing approach by an informal introduction to the calculus language CQQL. The underlying concepts and basic constraints, which have been derived from the vector space model, are encapsulated in the CQQL semantic.

The challenge is now to incorporate these principles into further relational query languages. For this purpose a whole quantum logic-based language family has been developed. To preserve the compatibility to existing object-relational data base systems we also evolved several mappings between the languages (Fig. 3), which are all ending up to the SQL-99 standard. Following languages belongs to our group of similarity languages:

- WS-QBE: QBE-based extension for the input of multimedia and similarity conditions
- CQQL: extension of the relational domain calculus
- QA: extension of the relational algebra
- QSQL: special SQL dialect based on SQL-92
- nCQQL: normalised CQQL-version
- nQA: normalised QA-version

References

[1] Schmitt, Ingo: *QQL: A DB&IR Query Language*. The VLDB Journal, 17(1):39–56, 2008.