N

Naive Tables

Gösta Grahne
Concordia University, Montreal, QC, Canada

Synonyms

Relations with marked nulls; Extended relations

Definition

The simplest way to incorporate unknown values into the relational model, is to allow *variables*, in addition to *constants*, as entries in the columns of relations. Such constructs are called *tables*, instead of *relations*. A table is an incomplete database, and *represents a set of complete databases*, each obtained by substituting all variables with constants. Different occurrences of the same variable (marked null) are substituted with the same constant. The substitution is thus a function from the variables and constants, to the constants, such that the function is identity on the constants. A table T then represents the set of relations, denoted $rep(T)$, defined as $\{v(T) : v \text{ is a valuation}\}$. Then the *certain answer* to a query q on a table T, denoted $sure(q, T)$ is the set of tuples that occur in every answer obtained by applying the query to every database in $rep(T)$. In other words, the certain answer to q on T is $sure(q, T) = \cap q(rep(T))$.

Key Points

To illustrate the above concepts, let tables T_1 and T_2 be as below, and let q be the relational expression $\sigma_{A=a \vee A=c}(\pi_{AC}(R_1 \bowtie R_2))$. (The schema of T_i is that of R_i, $i = 1,2$). Then applying q to T_1,T_2, which is denoted $\sigma_{A=a \vee A=c}(\pi_{AC}(T_1 \bowtie T_2))$, yields table $q(T_1, T_2)$ below.

T_1

A	B
a	X
Y	b
c	b

T_2

B	C
X	d
b	Z

$q(T_1, T_2)$

A	C
a	d
c	Z

The variables/null-values are written in uppercase, to clearly distinguish them from the (lowercase) constants. Note, however, that $q(T_1, T_2)$ is not (necessarily) yet the certain answer. How was $q(T_1, T_2)$ derived from q and T_1, T_2, and how is the certain answer $sure(q,(T_1, T_2))$ obtained from $q(T_1, T_2)$? The answer to the second question is very simple: just drop all tuples containing variables from $q(T_1, T_2)$. The remaining tuples form the certain answer. In the example, the certain answer consists of tuple (a, c) only. The answer to the first question is not much more complicated: evaluate q on the tables, treating variables as "constants," pairwise distinct, and distinct from all "real" constants. This is also known as the Naive evaluation of q on T [2,3]. In the example above, tuple (a, X) is joined with tuple (X, c) since they have the same value, represented by X, in the join column. This is done even though the "actual" value of X is not known, since in any valuation v the two occurrences of X are mapped to the same value. On the other hand, when performing the selection $\sigma_{A=a \vee A=c}$ the tuple (Y, Z) is not picked, since there is at least one valuation v, for which both $v(Y) \neq a$ and $v(Y) \neq c$. A characterization of the correctness of the Naive evaluation is given below.

Before going to the characterization, note that it is not always ideal to only return the certain answer. Namely, if the answer to q is to be materialized as a view for further querying, essential information is lost if the tuples with variables are dropped. For a simple example, a evaluating π_A on $sure(q, (T_1, T_2))$ gives tuple (a) as a sure answer, whereas evaluating $\pi_A(q(T_1, T_2))$, puts tuples (a) and (c) in the sure answer. As a consequence, query evaluation would not be compositional, unless $q(T_1, T_2)$ is stored as an "intermediate" answer. This "intermediate" answer is called the *exact answer* in [1], where the theory of query rewriting in information integration systems is extended to use the exact answer, instead of the certain one.

The correctness and completeness criteria for tables and query-evaluation is formalized using the notion of the representation system [2]. Here an alternative, equivalent formulation given in [3] is used: Consider a class of tables T, and a query language $\mathcal{Q}$. A triple $(\mathrm{T}, rep, \mathcal{Q})$ is said to be a *representation system* if for every table $T \in \mathrm{T}$, and for every applicable $q \in \mathcal{Q}$, there exists a function (here also named) q, such that

$$\cap\, rep(q(T)) = \cap q(rep(T)), \text{ and} \quad (1)$$

$$q' \circ q(T) = q'(q(T)), \quad (2)$$

for all applicable $q' \in \mathcal{Q}$.

Condition (1) says that the system can correctly compute the certain answer, and condition (2) states that the computation has to be uniformly recursive, following the structure of q. The important result is that, the class of Naive tables, and the class of all negation-free relational algebra expressions, together with Naive evaluation, form such a representation system. And this result comes without any computational penalty.

Cross-references

- ▶ Certain (and Possible) Answers
- ▶ Incomplete Information
- ▶ Maybe Answer
- ▶ Naive Tables

Recommended Reading

1. Grahne G. and Kiricenko V. Towards an algebraic theory of information integration. Inf. Comput., 194(2): 79–100, 2004.
2. Imielinski T. and Lipski Jr. Incomplete information in relational databases. J. ACM, 31(4):761–791, 1984.
3. Lipski W. Jr. On relational algebra with marked nulls. In Proc. 4th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, 1985, pp. 201–203.

Name Matching

- ▶ Record Matching

Namelessness

- ▶ Anonymity

Narrowed Extended XPath I

Andrew Trotman
University of Otago, Dunedin, New Zealand

Synonyms

NEXI

Definition

NEXI is an information retrieval (IR) query language for searching structured and semi-structured document collections. The language was first introduced for searching XML documents at the annual INEX [3] evaluation forum in 2004, and it has been used ever since.

Designed as the simplest query language that could possibly work, the language is a tiny subset of XPath [1] with an added *about*() function for identifying elements about some given topic. The language has extensions for question answering, multimedia searching, and searching heterogeneous document collections. NEXI is a language with a strict syntax defined in YACC but it has no semantics; the interpretation of the query is the task of the search engine.

Historical Background

A common information retrieval query language for searching XML documents was needed for specifying information retrieval queries at the first INEX in 2002. There, XML markup was chosen as the method of identifying keywords and the elements in which they should appear. It was also chosen as the method of identifying the preferred XML element to return to the user (the target element). The INEX 2002 query from topic 05 is given in Fig. 1. In this example, QBIC should be in a bibl element, image retrieval may appear anywhere in the document, and the user is interested in a list of tig elements as the result of the query.

Two problems with this format were identified: first it allowed the specification of queries that could be resolved by a simple mechanical process; second the

```
<title>
        <te>tig</te>
        <cw>QBIC</cw><ce>bibl</ce>
        <cw>image retrieval</cw>
</title>
```

Narrowed Extended XPath I. Figure 1. INEX topic 05 in the 2002 XML format.

language was not sufficiently expressive for information retrieval queries.

A modified XPath [1] was used at INEX 2003. In this variant the *contains*() function that required an element to contain the given content was replaced by an *about*() function that required an element to be about the content. Changing XPath in this way allowed fuzzy IR queries to be specified using a highly expressive language. However, an analysis of the XPath queries showed high syntactic and semantic error rates [6].

O'Keefe and Trotman [6] proposed using the simplest query language that could possibly work and a novel syntax. The INEX Queries Working Group [7] rejected the syntax but embraced the philosophy. It identified the minimum requirements of an IR query language for information retrieval queries containing structural constraints. This language, although at the time without syntax or semantics, was to be used at INEX for evaluation purposes.

Trotman and Sigurbjörnsson [9] proposed the Narrowed Extended XPath I (NEXI) language based on the working group report. It was narrowed in so far as only the descendant axis was supported, and extended in so far as the *about*() function was added, all other functions and axis were dropped. A formal grammar and parser were published, and an online syntax checker was hosted by the authors.

The decision to reduce XPath resulted in fewer errors because it reduced the chance of making mistakes. NEXI has a precise mathematical formulation which matches intuitive user profiles [4]. For both naïve users with knowledge of just the tag names, and for more advanced users with additional knowledge of the inter-relationships of those tags, the language is safe and complete. That is, the user cannot make semantic mistakes, and can express every information need they have.

Foundations

Web queries typically contain between 2 and 3 terms per query [8]. Formal query languages for semi-structured data tend to be comprehensive. This mismatch became apparent at INEX 2003 where XPath was chosen as the preferred language for information retrieval experts to specify relatively simple queries, but where they were unable to write syntactically and semantically correct queries. Just as SQL is not an end-user query language, neither, it turned out, was XPath.

Requirements

After two years of experimentation with XML query languages at INEX, the needs of such a language became apparent. The INEX, Queries Working Group [7] specified that the language should:

- Be compatible with existing syntax for specifying content only (keyword) queries.
- Be based on XPath as that language was already well understood, but:
- Remove all unnecessary XPath axis used for describing paths. Limit to just the descendant axis was suggested. The child operator was considered particularly problematic as it was open to misinterpretation.
- Drop exact match of strings, and inequality of numbers. XPath path filtering remained, however all strings were expressed as aboutness.
- Support multiple data types including numeric and string.
- Be open for extensions for new data types (including names, locations, dates, etc.).
- Not include tag instancing (for example author[1], the first author).
- Have vague semantics open to interpretation by the search engine.
- Loosen the meaning of the Boolean operators AND and OR.
- Disallow the multiple target elements. Although not explicit in the requirement, the implication is that the target element must be about the final clause in the query. It is a simple mechanical process to add non-target elements that are not about the query to the result – such as the author, title, source details to sections about something.
- Allow queries in which the target element was not specified and in which the search engine identified the ideal element.

Content Only (CO) Queries

NEXI addresses two kinds of queries on semi-structured and structured data: Content Only (CO) and Content And Structure (CAS) queries.

Content Only (CO) queries are the traditional IR query containing only keywords and phrases. No XML restrictions are seen and no mention is given of a preferred result (target) element. For these the NEXI syntax is derived from popular search engines: search terms can be keywords, numbers, or phrases

(delineated with quotes). Term restrictions can be specified using plus and minus.

Information Retrieval queries are by their very nature fuzzy. A user has an information need and from that need they express a query. There are many different queries they might specify from the same need, some of which might be more precise than the others. If a document in the document collection satisfies the user's information need, that document is relevant regardless of the query. That is, no query term might appear in a relevant document, or all the query terms might appear, either way the document is relevant. When specifying an IR query language it is important to avoid specifying semantics that violate this principle of relevance. The semantics of the terms with and without restriction in NEXI is, for example, specified this way:

- *"The '+' signifies the user expects the word will appear in a relevant element. The user will be surprised if a '-' word is found, but this will not prevent the document from being relevant. Words without a sign are specified because the user anticipates such terms will help the search engine to find relevant elements. As restrictions are only hints, it is entirely possible for the most relevant element to contain none of the query terms, or for that matter only the '-' terms."*

Or, in other words, it is the task of the search engine to identify relevant documents even if this involves ignoring the query.

In INEX topic 210 the author states:

- *"I'm developing a new lecture for the Master course 'Content Design' and want to discuss the topic "Multimedia document models and authoring". Therefore I want to do a quick background search to collect relevant articles in a reader. I expect to find information in abstracts or sections of articles. Multimedia content is an essential component of my lecture, thus for fragments to be relevant they should address document models of content authoring approaches for multimedia content. I'm not interested in single media approaches or issues that discuss storing multimedia objects."*

The query they give is

- +multimedia "document models" "content authoring"

in which "`document models`" is a phrase and `+multimedia` is a term-restricted search term (is positively selected for by the user).

Content and Structure (CAS) Queries

The second kind of query addressed by NEXI is the Content and Structure (CAS) query. These queries contain not only keywords but also structural constraints know as *structural hints*. Just as the keywords are hints passed to the search engine in an effort to help with the identification of relevant documents, so too are structural hints. CAS queries contain two kinds of structural hints, where to look (support elements), and what to return to the user (target elements).

Formally, queries many take one of the forms in Table 1:

A and C are paths and B and D are filters. Other forms could easily be added, but since NEXI was originally designed to address the INEX query problem, they are not formally included.

Paths (A and C in Table 1) are specified as a list of descendants separated by the descendant axis //. Formally, a path is an ordered sequence of nodes $//E_1...//E_n$ starting with E_1 and finishing at E_n, and for all $e \in n$, E_e is a ancestor of E_{e+1}. An attribute node is indicated by the prefix @. Alternative paths are specified $(E_{na}|E_{nb})$. The wildcard * is used as a place holder.

For example, the path:

```
//article//*//(sec|section)//@author
```

describes an `author` attribute beneath either a `sec` or `section` element beneath something beneath an `article` element. The interpretation by the search engine is, of course, loose.

Filters (B and D in Table 1) can be either arithmetic or string. Arithmetic filters are specified as arithmetic comparisons (>, <, =, >=, <=) of numbers to relative-paths, for example:`.//year >= 2000`. String filters take the form `about(relative-path,`

Narrowed Extended XPath I. Table 1. Valid forms of NEXI CAS queries

Form	Target Element	Meaning
`//A[B]`	`A`	Return A tags about B
`//A[B]//C`	`A//C`	Return C descendants of A where A is about B
`//A[B]//C[D]`	`A//C`	Return C descendants of A where A is about B and a C descendant of A are about D

COquery). Filters can be combined using the Boolean operators and, and or. Paths and filters are all considered hints and there is no requirement for the search engine to distinguish between the Boolean operators.

The target elements for the forms given in Table 1 are specified in column 2. Target elements, like support elements, are also hints. If, for example, the user specified paragraphs a subsection element might fulfill the user's information need.

An example of a valid NEXI CAS query (again from INEX topic 230) is:

```
//article[about(.//bdy, "artificial intelligence") and.//yr < = 2000]//bdy[about(., chess) and about(., algorithm)]
```

in which the target element is //article//bdy. The user has specified an arithmetic filter.//yr <= 2000. Several string filters are used including about(.//bdy, ''artificial intelligence"). A Boolean operator is also used to separate two filters about(., chess) and about(., algorithm).

The NEXI CAS query from INEX topic 210 is an alternative expression of the information need given in the previous section. That query is:

```
//article//(abs|sec)[about(.,+multimedia ''document models'' ''content authoring'')]
```

in which the target element is either //article//abs or //article//sec. The same documents and elements are relevant to both queries, as relevance is with respect to the information need and not the specific query.

Key Applications

Information retrieval from structured and semi-structured document collections.

Future Directions

Although proposed as an XML query language for use in an evaluation forum, there is evidence it may also be an effective end-user language. Van Zwol et al. [13] compared NEXI to a graphical query language called Bricks. They found that a graphical query language reduced the time needed to find information, but that users were more satisfied with NEXI. Inherent in text query languages is the problem that users are required to know the structure (the DTD) of the documents. In a heterogeneous environment this may not be possible, especially if new and different forms of data are constantly being added. Graphical query languages that translate into an intermediary text-based query language are one solution. This solution is seen with graphical user interfaces to relational databases.

Woodley et al. [12] further the model of NEXI as an intermediate language and compare NLPX (a natural language to NEXI translator) to that of Bricks (a graphic to NEXI translator). They show that users prefer a natural language interface, and that the performance of the two is comparable.

Ogilvie [10] examined the use of NEXI for question answering and proposed extensions to the language for this purpose. Dignum and van Zwol [2] proposed extensions for heterogeneous searching. Trotman and Sigurbjörnsson [10] unified these proposals and formally extended the language to include both – however, these extensions are not considered core to the language (language extensions philosophically deviate from the principle of simplest that could possibly work). Multimedia extensions to the language have also been used at INEX [11], again the extensions are not considered core to the language.

Experimental Results

The analysis of XPath queries used at INEX 2003 showed 63% of queries containing either syntactic or semantic errors [3]. An analysis of the errors in NEXI queries used at INEX 2004 showed that only 12% contained errors [12]. NEXI has been in use at INEX ever since.

Data Sets

NEXI queries from INEX 2004 onwards can be downloaded from the INEX web site: http://inex.is.informatik.uni-duisburg.de/

INEX queries for 2003 and 2002 were translated into NEXI (where possible) and can be downloaded from the NEXI web page hosted by the University of Otago: http://metis.otago.ac.nz/abin/nexi.cgi

URL to Code

An online syntax checker, lex and yacc scripts, and a command line syntax checker can be downloaded from the NEXI web page hosted by the University of Otago: http://metis.otago.ac.nz/abin/nexi.cgi

Cross-references

- ▶ Content-and-Structure Query
- ▶ Content-Only Query
- ▶ INitiative for the Evaluation of XML Retrieval
- ▶ Processing Structural Constraints
- ▶ Query by Humming
- ▶ Query Languages for the Life Sciences
- ▶ Semi-Structured Query Languages
- ▶ Temporal Query Languages
- ▶ XML
- ▶ XPath/XQuery
- ▶ XQuery Full-Text
- ▶ XSL/XSLT

Recommended Reading

1. Clark J. and DeRose S. XML path language (XPath) 1.0, W3C recommendation. The World Wide Web Consortium. Available at: http://www.w3.org/TR/xpath 1999.
2. Dignum V. and van Zwol R. Guidelines for topic development in heterogeneous collections. Available at: http://inex.is.informatik.uni-duisburg.de:2004/internal/hettrack/downloads/hettopics.pdf 2004.
3. Fuhr N., Gövert N., Kazai G., and Lalmas M. INEX: initiative for the evaluation of XML retrieval. In Proc. ACM SIGIR 2002 Workshop on XML and Information Retrieval, 2002.
4. Kamps J., Marx M., Rijke Md., and Sigurbjörnsson B. Articulating information needs in XML query languages. Trans. Inf. Syst., 24(4):407–436, 2006.
5. Ogilvie P. Retrieval using structure for question answering. In Proc. 1st Twente Data Management Workshop - XML Databases and Information Retrieval, 2004, pp. 15–23.
6. O'Keefe R.A. and Trotman A. The simplest query language that could possibly work. In Proc. 2nd Workshop of the Initiative for the Evaluation of XML Retrieval, 2003.
7. Sigurbjörnsson B. and Trotman A. Queries: INEX 2003 working group report. In Proc. 2nd Workshop of the Initiative for the Evaluation of XML Retrieval, 2003.
8. Spink A., Wolfram D., Jansen B.J., and Saracevic T. Searching the web: the public and their queries. J. Am. Soc. Inf. Sci. Tech., 53(2):226–234, 2001.
9. Trotman A. and Sigurbjörnsson B. Narrowed extended XPath I (NEXI). In Proc. 3rd Workshop of the Initiative for the Evaluation of XML Retrieval, 2004, pp. 16–40.
10. Trotman A. and Sigurbjörnsson B. NEXI, now and next. In Proc. 3rd Workshop of the Initiative for the Evaluation of XML Retrieval, 2004, pp. 41–53.
11. Westerveld T. and van Zwol R. Multimedia retrieval at INEX 2006. SIGIR Forum, 41(1):58–63, 2007.
12. Woodley A., Geva S., and Edwards S.L. Comparing XML-IR query formation interfaces. Aust. J. Intell. Inf. Process. Syst., 9(2):64–71, 2007.
13. van Zwol R., Baas J., van Oostendorp H., and Wiering F. Bricks: the building blocks to tackle query formulation in structured document retrieval. In Proc. 28th European Conf. on IR Research, 2006, pp. 314–325.

NAS

▶ Network Attached Storage

NAS Servers

▶ Storage Devices

NASD

▶ Network Attached Secure Device

Natural Human-Computer Interaction (NHCI)

▶ Natural Interaction

Natural Interaction

Stefano Baraldi, Alberto Del Bimbo, Lea Landucci, Nicola Torpei
University of Florence, Florence, Italy

Synonyms

Natural human-computer interaction; NHCI

Definition

The aim of Natural Human-Computer Interaction (NHCI) research is to create new interactive frameworks that integrate human language and behaviour into tech applications, focusing on the way people live, work, play and interact with each other. Such frameworks have to be easy to use, intuitive, entertaining and non-intrusive.

The design of *natural interaction* systems is focused on recognizing innate and instinctive *human expressions* in relation to some object, and return to the user a corresponding feedback that has the characteristics of being both *expected* and *inspiring*. All of the technology and the intelligence is built inside the digital artifacts and the user is not asked to use external devices, wear anything, or learn any commands or procedures. An interesting challenge for NHCI is

therefore to make systems self-explanatory by working on their "affordance" [11] and introducing simple and intuitive interaction languages.

The human expressions that can be utilized are those considered innate, meaning that they don't have to be learned. This includes vocal expressions and all the gestures used by humans to explore the nearby space or the immediate surroundings with their bodies, like: touching, pointing, stepping into zones, grabbing and manipulating objects (see Fig. 1). These direct actions express a clear sign of interest and necessitate a sudden reaction from the system.

The application fields range from browsing multimedia contents to exploration of knowledge structures; the scenarios involved can be either task-specific (in office or research contexts) or experiences created for casual interaction between the visitors and the media contents (like in museum exhibits and creative installations).

Natural interaction interfaces are very interesting to access, and explore large data sets like the ones contained in multimedia or geo-referenced databases. Data mining applications, which usually ask the user to enter complex search criteria and tweak the parameters to obtain the wanted data, can be designed with a visual analytic interface following these design guidelines. Queries are expressed visually, selecting interactive objects that embody selection criteria, creating clusters of them. Result sets provided by the database back-end can be iteratively shaped by directly manipulating the visual elements mapping the data, including other sets or reducing them.

Historical Background

At the beginning of the nineties, Human-Computer Interaction [17] was completely integrated into the Computer Science field, its quick growth led to the development of technologies able to go over the standard limiting user-computer communication paradigm, approaching a kind of natural interaction.

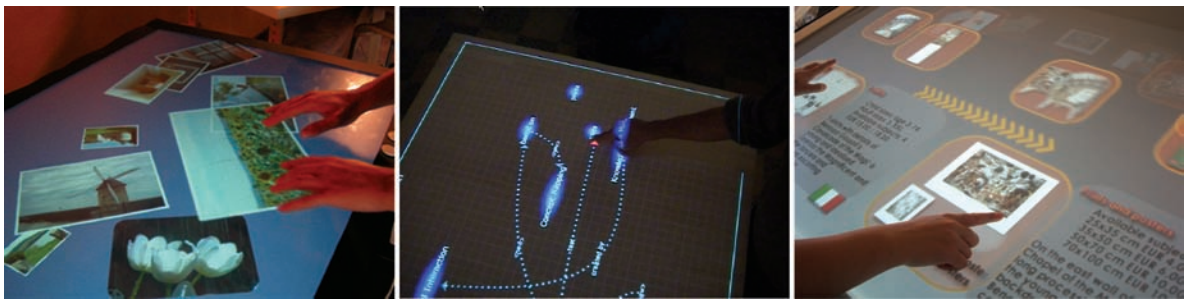
Alex Pentland in his "The Dance of Bits and Atoms" (1996, [13]) said that "There is a deep divide between the world of bits, and the world of atoms. Current machines are blind and deaf; they are unaware of us or our desires unless we explicitly instruct them. Consequently, only experts use most machines, and even they must spend most of their time battling arcane languages and strange, clunky interface devices."

The key point is that technologies must be designed and developed adapting them to the users, not the contrary: this is the foundation of the "human centered design" [11,12].

The goal, finally, becomes a technology at one user's service, suitable to the task requested and characterized by the complexity naturally embedded in the task itself [11,12].

Foundations

Interactive artifacts, augmented spaces, ambient technology and ubiquitous computing are all fields of study in HCI, which concentrate on both the technological aspects and the modalities in which users can perform activities. The techniques proposed are often referred to as *multi-modal interaction*, focusing on how machines can understand commands coming from different channels of human communication [10]. Among those: speech recognition, natural language understanding and gesture recognition, have been applied in different mixtures with roles ranging from active (the system observes the user and is pro-active in interacting with him) to passive (the system expects some kind of command, often through a device that is worn by the user). In the last years, studies have been following the concept of natural interaction as a conjunction of



Natural Interaction. Figure 1. Examples of natural interfaces for accessing digital contents and structures.

different technologies and design principles, with a more radical view about the user freedom in using interactive artifacts.

Natural interaction systems can be modelled as the sum of different modules: the *sensing* subsystem, which gathers sensor data about user expressions and behaviour, and the *presentation* module, which realizes the dialogue with the user orchestrating the output of different kind of actuators (graphics display, audio, haptics).

Sensing

The initial expression of interest towards the digital artefact is distracted from its main role: transferring information and stimuli. For this reason, the artefact should hide all the needs for an external controlling device, and be able to *sense* what the user is trying to do.

From a technological point of view, sensing involves the use of different sensors that provide data about some physical dimension in the surroundings of the artefact. There are a great number of electronic sensors that have been used in many industrial fields for robotics, automation and automatic inspection. Ranging from video cameras and image processing algorithms, to capacitive sensors for touch and pressure sensing or accelerometers for gesture recognition and body articulation.

Every sensor can have very different ways of providing this kind of data in terms of resolution, range, tolerances and errors. Some of them are able to provide discrete data with a high certainty, while others (like video cameras) just provide a great amount of data that has to be processed and interpreted by algorithms in order to extract useful information.

Considering the sensing architecture as a whole, a processing logic must be applied on top of it to abstract all the singularities of the sensors and create a homogeneous model of events for the later stages of interaction. Discrete events arriving from the sensors can be considered as belonging to three different categories of data:

- *Presence*: this data usually comes with a high signal-to-noise ratio, meaning that it cannot be directly interpreted as a human expression. Nonetheless, it reports a general activity in a spatial area.
- *Behaviour*: as the certainty level of some event increases (thanks to the combination of different sensors data), data can be studied for a certain period of time and provide information about the behaviour of single users or groups.
- *Activity*: this kind of data are considered "certain" and interpreted as a clear user intention to be used for a direct control of the interface. Instead of providing just the bi-dimensional screen coordinates as, it happens with traditional touch-screens, advanced sensing can interpret speed or pressure as well as distance of the hands as they approach a surface.

Sensors can be displaced in the environment, embedded in the artefact and also worn by the user but for natural interaction systems the last option is not suggested. For indoor use, the most meaningful displacement of sensors is inside the artefact itself. In this way, space can remain flexible and structures can be moved around. Such a thing could not be done if the whole environment was disseminated with.

Intelligence and Presentation

The intelligence substratum is what orchestrates the signals coming from sensors and generates output for presentation.

The devices embedded in the artefacts that manage the output phase are called *actuators* and can range from visual displays, to audio systems, surfaces with tactile feedback, holograms and many others. In scenarios like museums or art installations, even simple forms of stimuli can be used and mixed to convey a sensation or to catch the attention of the user. In an information space like an office or a data-intensive environment, the main channel of parallel communication remains the visual one, and the actuator is the graphical display.

On traditional GUIs (Graphical User Interfaces), what happens "inside the screen" can be subject to inference and prediction. Modern interfaces gather data about their use and are predictive in searching and suggesting data to the user in a light and "polite" way. Natural interfaces can extend this reasoning to a lot more data coming from the sensors, because the interactive artefact itself tries to *sense* beyond the screen. Also, unlike traditional interfaces whose informational layout is totally independent from the physical place in which the device is installed and used, digital artefacts can consider their position in the environment and what is happening around them.

In natural graphical interfaces, the basic assumption is that digital elements are not just a representation of data, but a part of the environment, and they

need to follow aesthetics rules too. As a first principle: digital elements should behave like objects in physical systems. Following this *analogy* the graphical display becomes a real *space*, and therefore has to provide the same affordance of a normal surface. If something changes on the visual interface it should happen in a smooth way, without sudden jumps, similar to what happens with hyperlinks on web pages. The models of movement and forces used in transitions have to mimic those of one real world, like: gravity, accelerating forces, momentum and friction. The user has the chance to understand what is going on without feeling disoriented, and thanks to these visual cues he can also expect what is going to happen.

The anticipation is more important because in shared natural interfaces the model of control of traditional interfaces, modelled around a question to confirm every action, cannot be used. A popup message box would block the interface occupying space and this is not feasible in a shared user environment. Instead, every action is directly executed and, in the time in which this happens, the visual features of objects influenced by the action gradually change. Another guideline for the visual interface design is *lightness*. On every graphical display, there is a conflict between the level of contents belonging to the domain of the application (diversified media), and the level of widget elements (labels, buttons, etc) that explain the interface to the user and give hints about the options. In natural interfaces this is even truer. Basically, the space is for contents, so any other element is stealing some attention from the user. For this reason, the widget level is kept to a minimal amount of symbols and there is little or no use of "global" interface elements.

Handling Complexity with TUIs

A challenge in natural interaction systems is tackling the complexity with simplicity. While the category of exploring applications can be realized using only innate gestures, in the case of complex applications (featuring multiple options and actions) simple and spontaneous hand gestures turn out to be not enough, and methods like speech understanding do not adapt well to noisy environments or to scenarios in which multiple people are interacting simultaneously.

The solutions could be:

1. Enriching the interaction language by adding new complex gestures that map to actions. This could distort the naturalness of interaction forcing users to learn unnatural gestures.
2. Introducing an intermediate visual level using interface elements (such as menus, icons etc.). This would reduce the interaction directness causing a conflict between digital contents and interface elements, both sharing the same visualization area.

The result is that such solutions could increase the user cognitive load. Tangible user interfaces (TUIs [8]) can be an alternative solution to those mentioned. They introduce physical, tangible objects that the system interprets as embodiment [6] of the elements of the interaction language. Users, manipulating those objects, inspired by their physical affordance, can have a more direct access to functions mapped to different objects [2].

There is a broad literature about TUIs which dates back to the first experiments of Hiroshi et al at the MIT, where a set of normal objects where "sensed" by a system who could recognize and track some of their features, like the position in space. Many other systems have been developed using this paradigm, and today there is a growing familiarity with taxonomies that try to define the different styles and scenarios in which it can be used [7]. Table 1 illustrates the main different branches.

The physical objects, called *tangibles,* could address some of the issues related to complexity. They can become the *embodiment* of some aspects of the interaction between the user and the domain of multi-media contents handled by the application. In particular three possible roles can be distinguished.

- *The tangible as a simulacrum*. The physical object can be used as a representative of a single digital object or a collection. This means either a uniquely identified static element

Natural Interaction. Table 1. Tangible interaction themes

Theme	Features
Tangible manipulation	Physical manipulation of tactile qualities
Spatial interaction	Movement in space
Embodied facilitation	Configuration of objects affecting group behavior
Expressive representation	Focus on digital and physical expressiveness

with a one-to-one mapping between the physical and the digital world, or a re-assignable element that can be associated with different data (like a container).

- *The tangible as a manipulator.* In this case the tangible represents a *function* that can be applied to a digital object. The closest metaphor is that of a tool in order to change one of its aspects, reveal other connected contents etc. This modality can be also extended to global functions whose target is the entire viewport or collection of digital objects.
- *The tangible as an avatar of the user.* In this case, every user would have a personal object that represents himself, to move across the contexts. An avatar object can be used in conjunction with the others when the application needs an authentication or when the activity proposed requires the user to express a preference.

While the manipulator role is specific to the nature and local interaction with the artefact (e.g., a digital tabletop), the simulacrum and avatar roles exactly provide the abstraction that is needed in order to expand the affordance of the environment, providing a natural way to transport the productions across the artifacts. In this fashion, activities can be initiated on an artefact and continued somewhere else, like in a laboratory, different places provide different contexts and options for the same data.

Key Applications

Multimedia Browsing

Usually they are easy-to use systems where people can interact simultaneously with multimedia contents through their own bare-hand gesture.

This kind of application offers an intuitive approach to various multimedia objects, they don't require any kind of training or instructions.

Knowledge Exploration and Building

Interactive workspace featuring vision-based gesture recognition that allows multiple users to collaborate [3] in order to realize face-to-face contexts, designing a common workspace where users can build knowledge (activities like brainstorming or problem solving sessions), exploiting the useful scenario-specific characteristics.

Interactive Museum and Cultural Exhibits

Museums and exhibitions are often just a collection of objects, standing deaf in front of visitors. In many cases, objects are accompanied by textual descriptions, usually too short or long to be useful for the visitor. In the last decade, progress in multimedia has allowed for new, experimental forms of communication (using computer technologies) in public spaces [1].

Interactive Music Systems

Usually built over a tabletop tangible user interface, Interactive Music Systems allow several simultaneous performers to share complete control over the instrument by moving physical artefacts on the table surface while constructing different audio topologies in a kind of tangible modular synthesizer. The reacTable [9], a clear example, is a novel multi-user electro-acoustic musical instrument with a tabletop tangible user interface.

Cross-references

- ▶ Human-Computer Interaction
- ▶ Multimedia Databases
- ▶ Multimodal Interfaces
- ▶ Object Recognition
- ▶ Visual Interfaces
- ▶ Visual Perception
- ▶ Visual Representation

Recommended Reading

1. Alisi T.M., Del Bimbo A., and Valli A. Natural interfaces to enhance visitors' experiences. IEEE Multimed., 12(3):80–85, 2005.
2. Baraldi S., Del Bimbo A., Landucci L., Torpei N., Cafini O., Farella, E., Pieracci A., and Benini L. Introducing TANGerINE: a tangible interactive natural environment. In Proc. 5th ACM Int. Conf. on Multimedia, 2007, pp. 831–834.
3. Baraldi S., Del Bimbo A., Landucci L., and Valli A. wikiTable: finger driven interaction for collaborative knowledge-building workspaces. In Proc. 2006 IEEE Int. Conf. on Computer Vision and Pattern Recognition Workshop, 2006, p. 144.
4. Colombo C., Del Bimbo A., and Valli A. Visual capture and understanding of hand pointing actions in a 3-D environment. IEEE Trans. Syst. Man. Cybern. B Cybern., 33(4):677–686, 2003.
5. Dietz P. and Leigh D. DiamondTouch: a multi-user touch technology. In Proc. 14th Annual ACM Symp. on User Interface Software and Technology, 2001, pp. 219–226.
6. Fishkin K.P. A taxonomy for and analysis of tangible interfaces. Pers. Ubiquitous Comput., 8(5):347–358, 2004.
7. Hornecker E. and Buur J. Getting a grip on tangible interaction: a framework on physical space and social interaction. In Proc. SIGCHI Conf. on Human Factors in Computing Systems, 2006, pp. 437–446.

8. Ishii H. and Ullmer B. Tangible bits: towards seamless interfaces between people, bits and atoms. In Proc. SIGCHI Conf. on Human Factors in Computing Systems, 1997, pp. 234–241.
9. Kaltenbrunner M., Jordà S., Geiger G., and Alonso M. The reacTable: a collaborative musical instrument. In Proc. Workshop on Tangible Interaction in Collaborative Environments, 2006.
10. Marsic I., Medl A., and Flanagan J. Natural communication with information systems. In Proc. IEEE, 88(8):1354–1366, 2000.
11. Norman D.A. The Design of Everyday Things. MIT PRESS, Cambridge, MA, 1998.
12. Norman D.A. The Invisible Computer. MIT PRESS, Cambridge, MA, 1999.
13. Pentland A., Smart rooms. Sci. Am., 274(4):54–62, 1996.
14. Prante T., Streitz N.A., and Tandler P. Roomware: computers disappear and interaction evolves. IEEE Comput., 37 (12):47–54, 2004.
15. Ulmer B. and Ishii H. Emerging frameworks for tangible user interfaces. IBM Syst. J., 39(3–4):915–931, 2000.
16. Valli A. Notes on Natural Interaction. http://naturalinteraction.org, 2005.
17. Wania C.E., Atwood M.E., and McCain K.W. How do design and evaluation interrelate in HCI research? In Proc. 6th Conf. on Designing Interactive Systems, 2006, pp. 90–98.

Natural Language Generation (NLG)

► Text Generation

Navigation System Interfaces

► Mobile Interfaces

Near-Duplicate Video Retrieval

► Video Querying

Nearest Neighbor Classification

Thomas Seidl
RWTH Aachen University, Aachen, Germany

Synonyms

NN classification; *k*-nearest neighbor classification; k-NN classification

Definition

Nearest neighbor classification is a machine learning method that aims at labeling previously unseen query objects while distinguishing two or more destination classes. As any classifier, in general, it requires some training data with given labels and, thus, is an instance of supervised learning. In the simplest variant, the query object inherits the label from the closest sample object in the training set. Common variants extend the decision set from the single nearest neighbor within the training data to the set of k nearest neighbors for any $k > 1$. The decision rule combines the labels from these k decision objects, either by simple majority voting or by any distance-based or frequency-based weighting scheme, to decide the predicted label for the query object. Mean-based nearest neighbor classifiers group the training data and work on the means of classes rather than on the individual training objects. As nearest neighbor classifiers only require distance computation of objects as a basic operation, their applicability exceeds the domain of vector spaces and attribute-based tuple data, and includes metric data spaces even if there are no attributes or dimensions available. Nearest neighbor classification counts for being highly accurate but inefficient; the latter assessment is to be invalidated by using appropriate indexing structures that support fast k-nearest neighbor retrieval. The abbreviation "NN classification" tends to cause confusion with the totally different concept of neural network classification.

Historical Background

Nearest neighbor classification is one of the earliest classification schemata. Nevertheless, it is one of the most highly accurate ones, whilst obeying a very broad range of applicability. The following aspects distinguish it from other classification methods.

Multiple class labels. Nearest neighbor classification is not restricted to two-class problems where classification decisions have to distinguish two classes only. Moreover, a high number of different classes does not deteriorate the efficiency in contrast to many other classifiers which create individual models for all the classes in the training set.

Applicability to general metric data spaces. Nearest neighbor classification is applicable to data without any structured attribute representation, i.e., to non-vectorial data. As distance computations are the only required basic operation, the applicability includes

complex multimedia data, time series and measurement series data, sequences and structural data to name just a few supported domains. Among the competing approaches in the field, only support vector machines equipped with respective kernel functions share this advantage. Other classifiers including decision trees, neural networks and Bayes density models rely on the attribute structure of the data.

Instance-based learning. Nearest neighbor classification derives decisions close to the individual training instances as any other method. No model is created for the training data and, thus, it is called lazy evaluation. Most of the competing classification approaches follow the eager evaluation paradigm which spends significant training effort in the determination of models for the training data. They produce decision tree structures, synaptic weights in neural networks, support vector weights, or density probability functions, respectively. The lazy model of nearest neighbor classification nevertheless allows for fast class decisions when spending (training) effort in the creation of an index on the training data.

Training data that changes dynamically. If the classifier has to follow training data that changes over time, without or even with respect to its statistical characteristics, a nearest neighbor classifier is superior to almost all of the competing classifiers. No model parameters such as decision tree structures or density weights have to be recomputed where the training data has changed. The lazy evaluation strategy ensures that the results always rely on the current training data status. The efficiency depends on how the underlying index structure supports the dynamic data changes.

Intuitive explanation component. Nearest neighbor classifiers provide illustrative explanations of their decisions by revealing the closest neighbors of the query object from the training data. Moreover, the corresponding similarity scores, i.e., the distances of the training items to the query object, provide some insight into the decision process, and the users may derive their confidence in the final decision result. In comparison, support vector machines analogously give illustrative weights to individual training objects, Bayes classifiers have a similar explanation power by providing probability values for each target class, and decision trees are slightly more intuitive by presenting the rules that produced the decision result. On the other hand, neural network classifiers lack illustrative explanations of their decisions.

Foundations

Classification Model

Nearest neighbor classifiers are a common classification model for which several variants exist. Along with the simple nearest neighbor model, *k*-nearest neighbor classification uses a set of k neighbors and the mean-based nearest neighbor model where individual training objects are generalized uses group representatives.

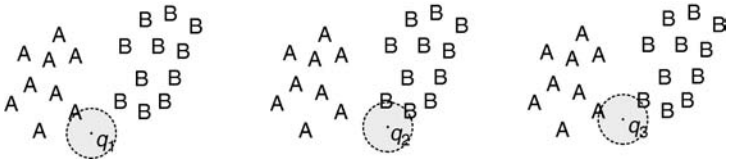
Simple nearest neighbor classification. The simplest variant of nearest neighbor classification may be formalized as follows: For any metric object space O, let $TS \subseteq O$ denote the set of labeled training data and $d : O \times O \rightarrow \Re_0^+$ is the chosen and thus distance function that reflects the dissimilarity of any two objects from O. Then, for any query object $q \in O$, the classifier evaluates the function $C_{\text{preliminary}}(q) = \{label(o) |\ o \in TS, \forall p \in TS - \{o\}: d(o,q) < d(p,q)\}$.

Unfortunately, this formalization is not valid for cases where several objects $o \in TS$ share the same minimal distance $d(o,q)$ to the query object, i.e., when the query hits perpendicular bisectors of these objects. Though in practice, these ties occur very rarely, a formalization that is equivalent to the previous one in other respects but in which ambiguities may be broken simply in a nondeterministic way is preferred (Fig. 1):

$$C_{nn}(q) = \{label(o) |\ o \in TS, \forall p \in TS : d(o,q) \leq d(p,q)\} \quad (1)$$

The decision model of nearest neighbor classification allows for a particularly broad range of applications. Its applicability covers vector data which are characterized by numerical attributes and includes general relational data where the tuples are represented by both numerical and categorical data as long as some metrics are defined on the categories. Moreover, if training and test data are neither taken from a vector space nor carry any categorical attributes but are only compared in terms of a distance function that reflects the dissimilarity of objects, nearest neighbor classifiers are still applicable. Examples of such applications include sequence data or structural data which may be compared by the edit distance, time series and measurement series compared by dynamic time warping, text and document data compared by the cosine distance, or multimedia data compared by complex similarity models.

Mean-based nearest neighbor classification. Nearest neighbor classifiers yield high quality decisions in



Nearest Neighbor Classification. Figure 1. Ambiguity of nearest neighbors for a sample training data set with two classes, A and B. Query object q_1 has a unique nearest neighbor and will be assigned to class A. For query object q_2, two neighbors share the smallest distance but there is no ambiguity with respect to the class decision since both neighbors are in class B. Query q_3 yields a conflict as it has two nearest neighbors from different classes, namely A and B, respectively. In practice, these ties occur very rarely at least for numerical reasons but may be solved nondeterministically.

terms of classification accuracy, and they are quite robust even for small training sets. Nevertheless, they tend to suffer from overfitting since the decisions are made closer at the training data than it is the case for any other method. As a consequence, erroneous training data, noise and outliers may badly affect the decisions. In order to increase the generalization power and to approach the overfitting problem, mean-based nearest neighbor classifiers and *k*-nearest neighbor classifiers have been developed as common variants of the simple model.

For mean-based nearest neighbor classification, the objects in the training data set are grouped into one or more clusters per class. These clusters are represented by their means, and nearest neighbors are then selected among the means rather than among the original individual training objects. The means inherit the class label of their cluster members, and the decision rule (1) from simple nearest neighbor classification is easily adopted to the mean model just by replacing the training data set by the set of means:

$$C_{m-nn}(q) = \{label(m) |\ m \in means(TS), \forall n \in means(TS) : d(m,q) \leq d(n,q)\} \quad (2)$$

Note that the computation of mean values requires the object space O to be a vector space, since the objects of a cluster need to be summed up followed by a scalar multiplication by the reciprocal value of the group's cardinality. Thus, the mean-based model does not apply to non-vectorial metric spaces which, in contrast, are supported by *k*-nearest neighbor classifiers.

k-nearest neighbor classification. Beside mean-based nearest neighbor classifiers, *k*-nearest neighbor classification is another quite common approach to increase the generalization power and to decrease overfitting effects. Instead of looking at a single object to the query among the training data, a set of k nearest neighbors, $k > 1$, is taken into account when making the decision. One starts by defining the decision set to be a subset of the training data TS that contains the k objects closest to the query object q. Again, ties may be broken by a nondeterministic choice among equidistant neighbors:

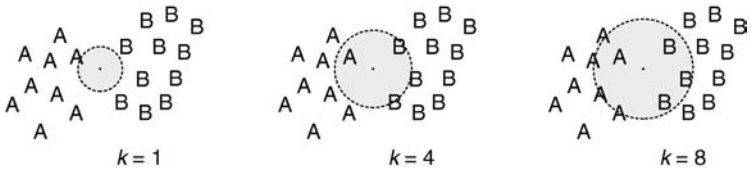
$$NN(q,k) = \{o |\ o \in TS, \forall p \in TS - NN(q,k) : d(o,q) \leq d(p,q)\},\ |NN(q,\ k)| = k \quad (3)$$

The decision rule based on majority vote now looks as follows (Fig. 2):

$$C_{majority}(q) = \arg\max_{l \in label(NN(q,k))} \{card\{o \in NN(q,k) |\ label(o) = l\}\}. \quad (4)$$

Weighted k-nearest neighbor classification. For *k*-nearest neighbor classifiers, different weighting schemata have been developed which introduce distances or frequencies into the decision rule. Conceptually, majority voting represents a weighting schema with unit weights. A quite common variant is to use the squared distances of the decision objects to the query object as reciprocal weights. This way, the desired effect is obtained that the closer an object is to the query, the higher is its influence to the classification decision. The corresponding decision rule is as follows:

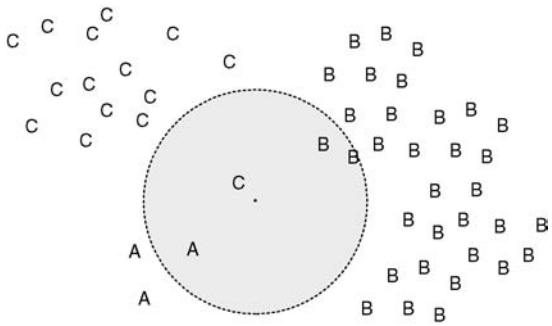
$$C_{dist}(q) = \text{argmax}_{l \in label(NN(q,k))} \left\{ \sum_{\substack{o \in NN(q,k) \\ label(o)=l}} \frac{1}{d(o,q)^2} \right\}. \quad (5)$$



Nearest Neighbor Classification. Figure 2. Dependency of the classification decision on the number of nearest neighbors. In the examples from left to right, the same query object is considered. A choice of k = 1 yields the class label A. For k = 4, class B holds the majority in the nearest neighbor set, and for k = 8, again label A is assigned to the query object.

An alternative weighting schema takes the relative frequency of the classes into account. Naturally, rare classes are represented only by a small number of instances in the training data. If at decision time, a nearest neighbor set contains a few of these rare instances, this occurrence is honored by the frequency weighting schema. This way, rare classes may outvote highly frequent classes even if these dominate the decision set by their number. Formally speaking, decisions are based on the fraction of training instances a class contributes to the decision set rather than on the absolute number of instances among the nearest neighbors (Fig. 3).

$$C_{freq}(q) = \text{argmax}_{l \in label(NN(q,k))} \left\{ \frac{card\{o|\ o \in NN(q,k), label(o) = l\}}{card\{o|\ o \in TD, label(o) = l\}} \right\}. \quad (6)$$



Nearest Neighbor Classification. Figure 3. Dependency of class decision on the weighting schema. In the example, a 4-nearest neighbor set is selected for the query object. Pure majority voting decides for class B as the decision set includes two instances of class B. Distance weighting decides for class C as the close neighbor from class C dominates the farer neighbor from class A and even the two neighbors from class B. Frequency-based weighting yields class A since A contributes the highest fraction of its overall occurrences in the training data to the nearest neighbor set.

Lazy Evaluation Model

Nearest neighbor classification follows the model of lazy evaluation. This means that there is no computation or estimation of model parameters at training time, but all calculations to make decisions are deferred to query time. Lazy evaluation is in contrast to the eager evaluation paradigm which requires some effort at training time to build an appropriate model. Examples of eager classifiers include decision trees, Bayes classifiers, neural networks, or support vector machines for which attribute-oriented decision flow diagrams, probability (mixture) density models, synaptic weights, or support vector weights are computed at training time, respectively. Lazy evaluation does not build models for the training data but derives its decisions directly from the training objects and, therefore, is called instance-based learning. Nearest neighbor classification is a prominent example of lazy evaluation, and the simple 1-nn and extended *k*-nn variants purely reflect the instance-based paradigm. The mean-based variant, however, deviates a little from this idea since means of classes are computed at training time, and decisions are based on the means rather than on the individual instances. For lazy evaluation in general, questions about decision efficiency, dynamic changes of training data, and explanations of decisions arise; these issues are discussed in the following subsections. Note that mean-based nearest neighbor classification is not a pure lazy variant since means of classes are computed at training time.

Decision efficiency. By following the lazy evaluation model, nearest neighbor classifiers defer all data analysis to query time. So query processing tends to be computationally more expensive than with eager

classifiers. Obviously, the most complex task is to determine the decision set, that is the set of (k) nearest neighbors for a query object; the subsequent weighting and voting is negligibly done in $O(k)$. Retrieving the k nearest neighbors in a set of n training objects depends on the data organization. Whereas a sequential scan requires $O(n)$ operations, the use of multidimensional or metric indexing structures may significantly speed-up the retrieval of the decision set depending on the dimensionality and on the statistical characteristics of the data. For high-dimensional data, techniques for reduction of dimensionality and multi-step query processing help to keep k-nearest neighbor retrieval efficient. As strict lazy evaluation does not waste any training time to create a model, it is nevertheless highly recommended to spend some effort for the creation of an appropriate index on the training data. In short, to train an efficient nearest neighbor classifier means to instantiate model creation by index creation.

Dynamically changing training data. As for lazy evaluation, no model is created at training time, a lazy classifier in general allows for dynamic changes of training data particularly well. For a nearest neighbor classifier, efficiency depends on how the underlying index structure maintains the dynamic data set. This represents a common requirement for indexing and is supported well by almost all multidimensional or metric access methods. As training data changes, new examples are inserted into the index and outdated training instances are removed. This way, incremental training for low change rates and also online training with potentially high rates of incoming training data are enabled. Nearest neighbor classification, thus, always produces a result that relies on the current training data status.

Decision explanation. Lazy classifiers do not create models for the training data and, as an inherent limitation, no explicit knowledge is extracted which reflects the structure of the data. Though nearest neighbor classifiers do not provide intuitive models of the data characteristics, they nevertheless explain their individual decisions illustratively by revealing the closest neighbors of the query object within the labeled training data to the user. Additional information is provided by the similarity scores, i.e., the distances of the training items to the query object, thus yielding some insight into the decision process from which the users may derive their confidence in the final decision result.

Key Applications

Key applications of nearest neighbor classification include all areas where classification problems occur. Particularly well suited are nearest neighbor classifiers in the following cases:

1. Many classes need to be distinguished
2. Objects are represented by general metric data without an attribute structure
3. Classifier has to follow training data that change dynamically
4. Users require intuitive explanations of the system's decisions

Real applications are found in all areas of multimedia data exploration and cognitive systems including computer vision, speech recognition, medical imaging, robot motion planning, or sensor-based object recognition in general to name just a few.

Cross-references

- ▶ Applications
- ▶ Indexing Metric Spaces
- ▶ Indexing: R-Trees
- ▶ Multidimensional Indexing
- ▶ Multimedia Information Retrieval
- ▶ Nearest Neighbor Query
- ▶ Spatial Indexing Techniques
- ▶ Spatial
- ▶ Spatiotemporal
- ▶ Text Indexing Techniques

Recommended Reading

1. Ankerst M., Kastenmuller G., Kriegel H.-P., and Seidl T. Nearest neighbor classification in 3D protein databases. In Proc. 7th Int. Conf. on Intelligent Systems for Molecular Biology, 1999, pp. 34–43.
2. Athistos V. Nearest neighbor retrieval and classification. Available at: http://cs-people.bu.edu/athitsos/nearest-neighbors/, 2007.
3. Djouadi A. and Bouktache E. A fast algorithm for the nearest-neighbor classifier. IEEE Trans. Pattern Anal. Mach. Intell., 19(3): 277–282, 1997.
4. Duda R.O., Hart P.E., and Stork D.G. Pattern Classification (2nd ed.). Wiley, New York, 2001.
5. Efros A.A., Berg A.C., Mori G., and Malik J. Recognizing action at a distance. In Proc. 9th IEEE Conf. Computer Vision, 2003, pp. 726–733.
6. Ghosh A.K., Chaudhuri P., and Murthy C.A. On visualization and aggregation of nearest neighbor classifiers. IEEE Trans. Pattern Anal. Mach. Intell. 27(10):1592–1602, 2005.

7. Han J. and Kamber M. Data Mining, Concepts and Techniques (2nd ed.). Elsevier, Amsterdam, 2006.
8. Hastie T., Tibshirami R., and Friedman J. The Elements of Statistical Learning: Data Mining, Inference and Prediction. Springer Series in Statistics. Springer, New York, 2001.
9. Kriegel H.-P., Pryakhin A., and Schubert M. Multi-represented kNN-classification for large class sets. In Proc. 10th Int. Conf. on Database Systems for Advanced Applications, 2005, pp. 511–522.
10. Li Y., Yang J., and Han J. 1Continuous K-nearest neighbor search for moving objects. In Proc. 16th Int. Conf. on Scientific and Statistical Database Management, 2004, pp. 123–126.
11. Shibata T., Kato T., and Wada T. K-D. Decision tree: an accelerated and memory efficient nearest neighbor classifier. In Proc. 2003 IEEE Int. Conf. on Data Mining, 2003, pp. 641–644.
12. Veenman C.J. and Reinders M.J.T. The nearest subclass classifier: a compromise between the nearest mean and nearest neighbor classifier. IEEE Trans. Pattern Anal. Mach. Intell., 27(9): 1417–1429, 2005.

Nearest Neighbor Query

Dimitris Papadias
Hong Kong University of Science and Technology, Hong Kong, China

Synonyms

NN query; *k*NN query

Definition

Given a dataset P and a query point q, a *k nearest neighbor* (kNN) query returns the k closest data points $p_1,p_2,...,p_k$ to q. For any point $p \in P-\{p_1,p_2,...,p_k\}$: $dist(p_i, q) \leq dist(p, q)\ \forall\ 1 \leq i \leq k$, where *dist* depends on the underlying distance metric. The usual metric is the L_x norm, or formally, given two points q, p whose coordinates on the i-th dimension ($1 \leq i \leq m$) are q_i and p_i respectively, their L_x distance is:

$$L_x(p, q) = (\Sigma_{i=1\sim m} |p_i - q_i|^x)^{1/x}, \text{ for } x \neq \infty,$$

$$L_x(p, q) = \max_{i=1\sim m}(|p_i - q_i|), \text{for } x = \infty.$$

Most work on spatial and spatio-temporal databases focuses on Euclidean distance (i.e., L_2), but alternative definitions have been used in various domains (e.g., road networks, time series).

Key Points

Nearest neighbor search constitutes an important component for several tasks such as clustering, outlier detection, time series analysis, and image and document retrieval. For instance, finding the most similar series, image, or document to a given input is a NN query in a corresponding data space defined by the features of interest. Unlike spatial and spatio-temporal databases that usually involve 2–3 dimensions, these applications may lead to high-dimensional spaces. In order to avoid the high cost of distance computations in such cases, several methods follow a multi-step framework for processing NN queries [3]: (i) a dimensionality reduction technique is used to decrease the number of dimensions, (ii) the low dimensional data are indexed, (iii) the index is used to efficiently retrieve a candidate NN (in low dimensional space), (iv) the actual distance of the candidate (in the original space) is computed, and (v) steps (iii) and (iv) are repeated until no other candidate can lead to a better solution than the one already discovered.

In addition to conventional NN search, several alternative types of nearest neighbor queries have been proposed in the database literature. Given a multi-dimensional dataset P and a point q, a *reverse nearest neighbor* query retrieves all the points $p \in P$ that have q as their NN (The nearest neighbor relationship is not symmetric; i.e., the fact that q is the NN of p does not necessarily imply that p is the NN of q.) [1]. Given a set P of data points and a set Q of query points, an *aggregate nearest neighbor* query returns the data point p with the minimum *aggregate distance* [2]. The *aggregate distance* between a data point p and $Q = \{q_1,...,q_n\}$ is defined as $f(dist(p,q_1),...,dist(p, q_n))$. If, for instance, $f = sum$, the corresponding query reports the data point that minimizes the total distance from all query points.

Cross-references

► Nearest Neighbor Query in Spatio-temporal Databases
► Reverse Nearest Neighbor Query

Recommended Reading

1. Korn F. and Muthukrishnan S. Influence sets based on reverse nearest neighbor queries. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2000, pp. 201–212.
2. Papadias D., Tao Y., Mouratidis K., and Hui K. Aggregate nearest neighbor queries in spatial databases. ACM Trans. Database Sys., 30(2):529–576, 2005.
3. Seidl T. and Kriegel H-P. Optimal multi-step k-nearest neighbor search. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1998, pp 154–165.

Nearest Neighbor Query in Spatio-temporal Databases

Dimitris Papadias
Hong Kong University of Science and Technology, Hong Kong, China

Synonyms

NN query; NN search

Definition

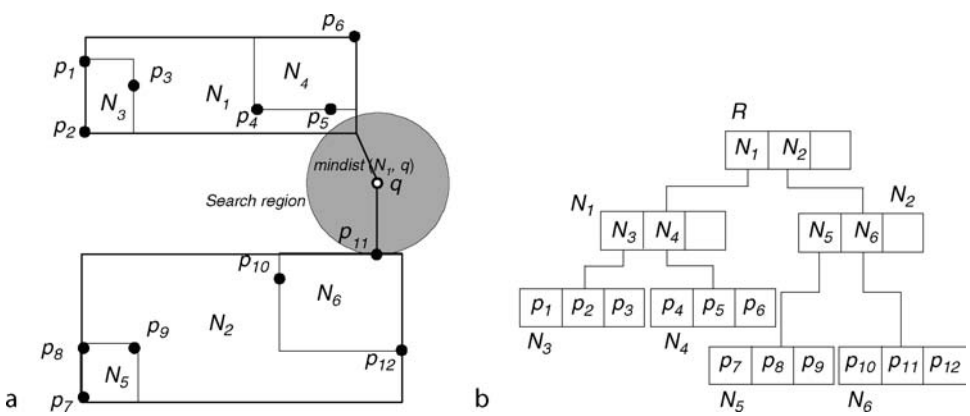
Given a set of points P in a multi-dimensional space, the nearest neighbor (NN) of a query point q is the point in P that is closest to q. Similarly, the k nearest neighbor (kNN) set of q consists of the k points in P with the smallest distances from q. In spatial and spatio-temporal databases, the distance is usually defined according to the Euclidean metric, and the dataset P is disk-resident. Query algorithms aim at minimizing the processing cost. Other optimization criteria in the case of moving objects (or queries) include the network latency, or the number of queries required for keeping the results up-to-date.

Historical Background

Nearest neighbor (NN) search is one of the oldest problems in computer science. Several algorithms and theoretical performance bounds have been devised for exact and approximate processing in main memory [1]. In spatial databases, existing algorithms assume that P is indexed by a spatial access method (usually an *R-tree* [2]) and utilize some pruning bounds to restrict the search space. Figure 1 shows an *R-tree* for point set $P = \{p_1,p_2,...,p_{12}\}$ with a capacity of three entries per node (typically, the capacity is in the order of hundreds). Points that are close in space (e.g., p_1, p_2, p_3) are clustered in the same leaf node (N_3). Nodes are then recursively grouped together with the same principle until the top level, which consists of a single root. Given a node N and a query point q, the *mindist* (N,q) corresponds to the closest possible distance between q and any point in the subtree of node N.

The first NN algorithm for *R-trees* [8] searches the tree in a *depth-first* (DF) manner, recursively visiting the node with the minimum *mindist* from q, e.g., in Fig. 1, DF accesses the root, followed by N_1 and N_4, where the first potential nearest neighbor is found (p_5). Note that p_5 is not the actual NN (it is p_{11}), and the search continues. During backtracking to the upper level (node N_1), the algorithm prunes entries whose *mindist* is equal to or larger than the distance (*best_dist*) of the nearest neighbor already retrieved. In the example of Fig. 1, after discovering p_5, (i) *best_dist* is set to *dist*(p_5,q), (ii) DF backtracks to the root level (without visiting N_3), and (iii) it follows the path N_2, N_6 where the actual NN p_{11} is found. DF can be easily extended for the retrieval of $k > 1$ nearest neighbors: the k points discovered so far with the minimum overall distances are maintained in an ordered list of k pairs $<p$, *dist*(p,q)$>$ (sorted on ascending *dist*(p,q) order) and *best_dist* equals the distance of the k-th NN. Whenever a better neighbor is found, it is inserted in the list, the last element is removed, and the value of *best_dist* is updated.

The DF algorithm is sub-optimal, i.e., it accesses more nodes than necessary. Specifically, an optimal algorithm should visit only nodes intersecting the

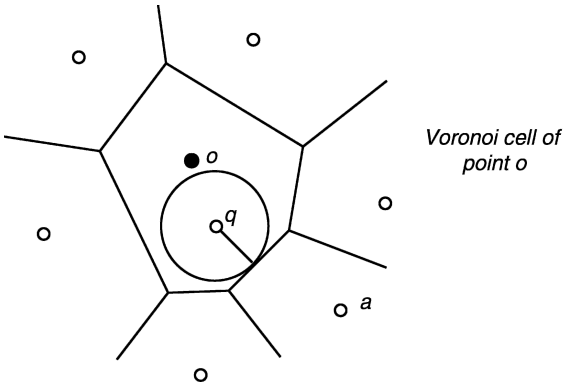


Nearest Neighbor Query in Spatio-temporal Databases. Figure 1. Example of an R-tree and a point NN query.

search region, i.e., a circle centered at the query point q with radius equal to the distance between q and its nearest neighbor [4]. In Fig. 1, for instance, an optimal algorithm should visit only the root, N_1, N_2, and N_6, whereas DF also visits N_4. The *best-first* (BF) algorithm of [5] achieves the optimal I/O performance by maintaining a heap H with the entries visited so far, sorted by their *mindist*. As with DF, BF starts from the root, and inserts all the entries into H (together with their *mindist*), e.g., in Fig. 1, $H = \{<N_1, mindist(N_1,q)>, <N_2, mindist(N_2,q)>\}$. Then, at each step, BF visits the node in H with the smallest *mindist*. Continuing the example, the algorithm retrieves the content of N_1 and inserts all its entries in H, after which $H = \{<N_2, mindist(N_2,q)>, <N_4, mindist(N_4,q)>, <N_3, mindist(N_3,q)>\}$. The next two nodes accessed are N_2 and N_6 (inserted in H after visiting N_2), in which p_{11} is discovered as the current NN. At this time, the algorithm terminates (with p_{11} as the final result) since the next entry (N_4) in H is farther (from q) than p_{11}. Similarly to DF, BF can be easily extended to kNN queries ($k > 1$). In addition, BF is *incremental*, i.e., it can output the nearest neighbors in ascending order of their distance to the query without a pre-defined termination condition.

Foundations

In dynamic environments, the result of a conventional NN query may be immediately invalidated due to the query or data movement. Several techniques augment the result with some additional information regarding its validity. For instance, Zheng and Lee [14] assume an architecture where moving clients (e.g., mobile devices) send their queries to a server. The server precomputes and stores in an *R-tree* the *Voronoi diagram* of the (static) dataset. When a NN query q arrives at the server, the *Voronoi diagram* is used to efficiently compute its nearest neighbor; i.e., the point (o in Fig. 2), whose Voronoi cell covers q. In addition to the result, the server sends back to the client its *validity time* T, which is a conservative approximation assuming that the query speed is below a maximum value. In particular, T is the time that q will cross the closest boundary of the Voronoi cell of object o (in which case point a will become the nearest neighbor). Zhang et al. [13] propose the concept of *location-based* queries that return the *validity region* around the query point, where the result remains the same. For instance, in Fig. 2, a location-based query q will return object o

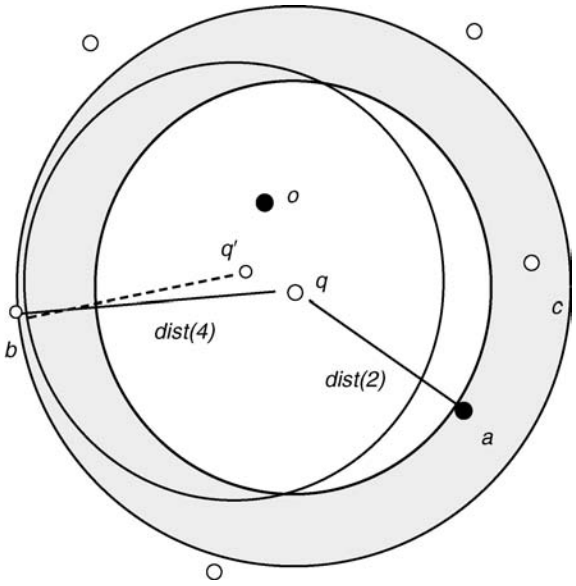


Nearest Neighbor Query in Spatio-temporal Databases.
Figure 2. Example of [6].

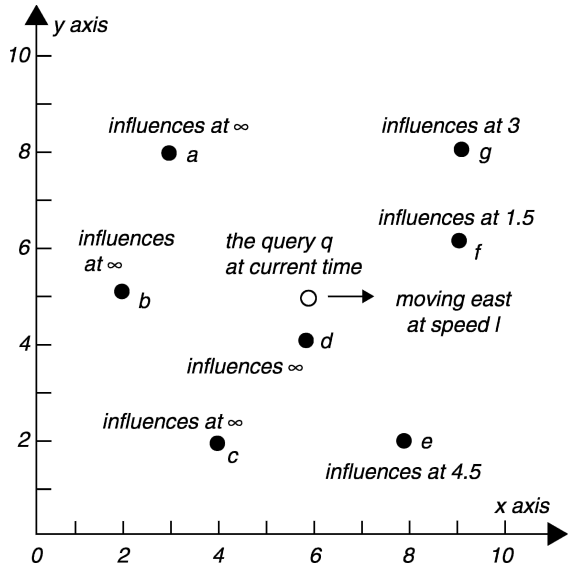
and its Voronoi cell, which is computed on-the-fly using an *R-tree* on the data points.

For the same settings (moving query - static data objects), Song and Roussopoulos [10] reduce the number of queries required to keep the result up-to-date by introducing redundancy. In particular, when a kNN query arrives, the server returns to the client a number $m > k$ of neighbors. Let $dist(k)$ and $dist(m)$ be the distances of the kth and mth nearest neighbor from the query point q. If the client re-issues the query at a new location q', it can be proven that the new k nearest neighbors will be among the m objects of the first query, provided that $2 \cdot dist(q',q) \leq dist(m) - dist(k)$. Figure 3 shows an example for a 2NN query at location q, where the server returns four results o, a, b and c (the two nearest neighbors are o and a). When the client moves to a nearby location q', the 2 NN are o and b. If $2 \cdot dist(q,q') \leq dist(4) - dist(2)$, the client can determine this by computing the new distances (with respect to q') of the four objects, without having to issue a new query to the server.

Tao and Papadias [11] propose *time-parameterized* (TP) queries, assuming that the clients move with linear and known velocities. In addition to the current result R, the output of a TP query contains its *validity period* T and the *next change* C of the result (that will occur at the end of the validity period). Given the additional information (C and T), the client only needs to issue another TP query after the expiry of the current result. Figure 4 shows a TP NN, where the query point q is moving east with speed 1. Point d is the current nearest neighbor of q. The *influence time* $T_{INF}(p, q)$ of an object p is the time that p starts to get closer to q than the current nearest neighbor. For



Nearest Neighbor Query in Spatio-temporal Databases.
Figure 3. Example of [8].



Nearest Neighbor Query in Spatio-temporal Databases.
Figure 4. Example of TP NN query.

example, $T_{INF}(g,q) = 3$, because at this time g will come closer to q than d. The expiry time of the current result is the minimum $T_{INF}(o,q)$ of all objects, i.e., in Fig. 4, $T = 1.5$, at which point f will replace d as the NN of q. Based on the above observations, a TP NN query is processed in two steps: (i) a conventional algorithm (e.g., [5,8]) retrieves the NN at the current time, and (ii) a second pass computes the time-parameterized component (i.e., C and T) by applying again NN search and treating $T_{INF}(o,q)$ as the distance metric: the goal is to find the objects (C) with the minimum T_{INF}.

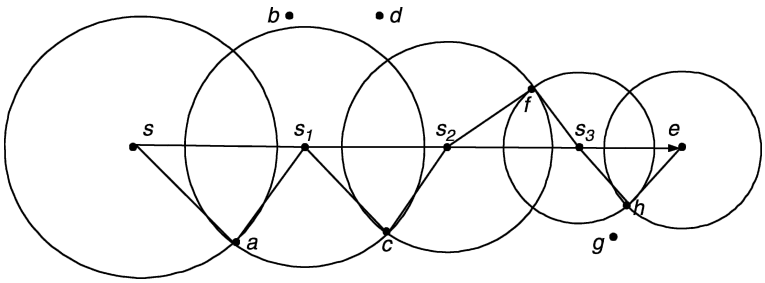
A *continuous* nearest neighbor (CNN) query [3,11] retrieves the nearest neighbor (NN) of every point in a line segment $q = [s, e]$. In particular, the result contains a set of $<R,T>$ tuples, where R (for result) is a point of P, and T is the interval during which R is the NN of q. As an example consider Fig. 5, where $P = \{a,b,c,d,f,g,h\}$. The output of the query is $\{<a, [s,s_1]>, <c, [s_1,s_2]>, <f, [s_2,s_3]>, <h, [s_3,e]>\}$, meaning that point a is the NN for interval $[s,s_1]$; then at s_1, point c becomes the NN etc. The points of the query segment (i.e., s_1, s_2, s_3) where there is a change of neighborhood are called *split points*. CNN algorithms use DF or BF traversal on *R-trees* to visit nodes and data points according to their proximity to the query segment. Visited data points introduce new split points, which are used to prune the search space. For instance, in Fig. 5, if a, c, f and h have already been discovered, every node and data point (b, d, g) outside a circle defined by a split point (center) and its NN (radius) can be eliminated since they cannot affect the result.

A *predictive* NN query retrieves the expected nearest neighbor of a (static or moving) query point (e.g., 30 seconds from now) based on the current motion patterns. Assuming that data points move linearly, they can be indexed by a TPR-tree [9]. The TPR-tree is similar to the *R-tree*, but takes into account both location and velocity in order to group data objects into nodes. Furthermore, each node is assigned a velocity vector so that its extent continuously encloses all the objects inside (i.e., a moving node that may grow with time). A predictive query is answered in the same way as in the *R-tree* (e.g., by adaptations of [5,8]), except that the node extends at the (future) query time are dynamically computed (using the current location and velocity vector of the node). The concepts of TP and continuous queries also apply in this case. Tao et al. [12] propose an architecture and index for processing queries on objects moving with arbitrary motion patterns, unknown in advance.

Key Applications

Geographic Information Systems

Nearest neighbor search is one of the most common query types. Efficient algorithms are important for



Nearest Neighbor Query in Spatio-temporal Databases. Figure 5. Example CNN query.

dealing with the large and ever increasing amount of spatial data in several GIS applications.

Location-based Services

Advanced NN algorithms will provide the means for enhanced location-based services based on the proximity of mobile clients to potential facilities of interest.

Multi-criteria Decision Making

A number of decision support tasks can be modeled as nearest neighbor search. For instance, news WWW-sites usually recommend to users the articles that are most similar to their previous choices. Nearest neighbor algorithms have also been used to process skyline queries.

Future Directions

All the above techniques take as input a single query, and report its nearest neighbor set at the current time, possibly with some validity information (e.g., expiration time, Voronoi cell), or generate future results based on predictive features (e.g., velocity vectors of queries or data objects). On the other hand, *continuous monitoring of spatial queries* [6] (i) involves multiple, long-running, queries (from geographically distributed clients), (ii) is concerned with both computing *and* keeping the results up to date, (iii) usually assumes main-memory processing to provide fast answers in an on-line fashion, and (iv) attempts to minimize factors such as the CPU or communication cost (as opposed to I/O overhead).

Another interesting problem concerns NN search in non-Euclidean spaces. For instance, in road networks the distance between two points can be defined as the length of the shortest path connecting them, or by the minimum time it takes to travel between them. In either case, the problem requires different algorithmic solutions [7].

Experimental Results

In general, for every presented method, there is an accompanying experimental evaluation in the corresponding reference. [5] compares BF and DF traversal for conventional NN search. [9] proposes cost models for TP NN and CNN queries and evaluates their accuracy, as well as the relative performance of the two query types for static (indexed by *R-trees*) and dynamic (indexed by *TPR-trees*) objects.

Data Sets

A large collection of real datasets, commonly used for experiments, can be found at:

http://www.rtreeportal.org/

URL to Code

R-tree portal (see above) contains the code for most common spatial and spatio-temporal indexes, as well as data generators and several useful links for researchers and practitioners in spatio-temporal databases.

Cross-references

- ▶ Continuous Monitoring of Spatial Queries
- ▶ Nearest Neighbor Query
- ▶ Rtree
- ▶ Voronoi Diagrams

Recommended Reading

1. Arya S., Mount D., Netanyahu N., Silverman R., and Wu A. An optimal algorithm for approximate nearest neighbor searching. J. ACM, 45(6):891–923, 1998.
2. Beckmann N., Kriegel H., Schneider R., and Seeger B. The R*-tree: an efficient and robust access method for points and rectangles. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1990, pp. 322–331.
3. Benetis R., Jensen C., Karciauskas G., and Saltenis S. Nearest neighbor and reverse nearest neighbor queries for moving objects. VLDB J., 15(3):229–249, 2006.

4. Bohm C. A cost model for query processing in high dimensional data spaces. ACM Trans. Database Sys., 25 (2):129–178, 2000.
5. Hjaltason G. and Samet H. Distance browsing in spatial databases. ACM Trans. Database Sys., 24(2):265–318, 1999.
6. Mouratidis K., Hadjieleftheriou M., and Papadias D. Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2005, pp. 634–645.
7. Papadias D., Zhang J., Mamoulis N., and Tao Y. Query processing in spatial network databases. In Proc. of the 29th Conf. on Very Large Databases, 2003, pp. 790–801.
8. Roussopoulos N., Kelly S., and Vincent F. Nearest Neighbor Queries. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1995, pp. 71–79.
9. Saltenis S., Jensen C., Leutenegger S., and Lopez M. Indexing the positions of continuously moving objects. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2000, pp. 331–342.
10. Song Z., and Roussopoulos N. K-nearest neighbor search for moving query point. In Proc. 7th Int. Symp. Advances in Spatial and Temporal Databases, 2001, pp. 79–96.
11. Tao Y. and Papadias D. Spatial queries in dynamic environments. ACM Trans. Database Sys., 28(2):101–139, 2003.
12. Tao Y., Faloutsos C., Papadias D., and Liu B. Prediction and indexing of moving objects with unknown motion patterns. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2004, pp. 611–622.
13. Zhang J., Zhu M., Papadias D., Tao Y., and Lee D. Location-based spatial queries. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2003, pp. 443–454.
14. Zheng B., and Lee D. Semantic caching in location-dependent query processing. In Proc. 7th Int. Symp. Advances in Spatial and Temporal Databases, 2001, pp. 97–116.

Negative Dictionary

► Stoplists

Nested Loop Join

Jingren Zhou
Microsoft Research, Redmond, WA, USA

Synonyms

Nested loop join; Loop join

Definition

The nested loop join is a common join algorithm in database systems using two nested loops. The algorithm starts with reading the *outer* relation R, and for each tuple $\mathcal{R} \in R$, the *inner* relation S is checked and matching tuples are added to the result.

```
Algorithm 1: Nested Loop Join: R⋈pred(r, s)S
 foreach R ∈ R do
    foreach S ∈ S do
       if pred (R.r, S.s) then
          add {R, S} to result
       end
    end
 end
```

Key Points

One advantage of the nested loop join is that it can handle any kind of join predicates, unlike the sort-merge join and the hash join which mainly deal with an equality join predicate. An improvement over the simple nested loop join is the block nested loop join which effectively utilizes buffer pages and reduces disk I/Os.

Block Nested Loop Join

Suppose that the memory can hold B buffer pages. If there is enough memory to hold the smaller relation, say R, with at least two extra buffer pages left, the optimal approach is to read in the smaller relation R and to use one extra page as an input buffer to read in the larger relation S and the other extra buffer page as an output buffer.

If there is not enough memory to hold the smaller relation, the best approach is to break the outer relation R into *blocks* of $B - 2$ pages each and scan the whole inner relation S for each block of R. As described before, one extra page is used as an input buffer and the other as an output buffer. In this case, the outer relation R is scanned only once while the inner relation S is scanned multiple times.

Cross-references

► Parallel Join Algorithms
► Evaluation of Relational Operators

Recommended Reading

1. Mishra P. and Eich M.H. Join processing in relational databases. ACM Comput. Surv., 24(1):63–113, 1992.

Nested Transaction Models

George Karabatis
University of Maryland, Baltimore Country (UMBC), Baltimore, MD, USA

Definition

A *nested transaction model* as proposed by Moss is a generalization of the flat transaction model that allows nesting. A nested transaction forms a tree of transactions with the root being called a *top-level transaction* and all other nodes called *nested transactions (subtransactions)*. Transactions having no subtransactions are called *leaf* transactions. Transactions with subtransactions are called *parents (ancestors)* and their subtransactions are called *children (descendants)*.

A subtransaction can commit or rollback by itself. However, the effects of the commit cannot take place unless the parent transaction also commits. Therefore, in order for any subtransaction to commit, the top-level transaction must commit. If a subtransaction aborts, all its children subtransactions (forming a subtree) are forced to abort even if they committed locally.

Historical Background

Nested transactions were introduced by Moss in 1981 [8] to overcome some of the limitations of the flat transaction model, as they allow a finer level of control in a transaction. For example, a failed operation in a flat transaction causes the entire transaction to be rolled back. On the contrary, a failed operation in a part of a nested transaction may be acceptable and the entire transaction may be allowed to commit, even if some of its operations failed. As an example, a trip consisting of a flight reservation, hotel accommodation and car rental can be implemented as a nested transaction with three subtransactions. If some of its subtransactions commit (e.g., flight and hotel), and some fail (e.g., car rental) the outcome may still be acceptable and allow the nested transaction to commit with partial results.

Nested transactions are based on the underlying concept of *spheres of control* developed by Bjork and Davies in 1973 [1,3]. Davies describes the spheres of control in more detail in [2]; however, he portrays overall semantics of the spheres which convey a more abstract concept than nested transactions. Moss provides implementation details of nested transactions for a distributed environment. Prior to that, Reed in his dissertation [9] also describes an implementation of nested transactions in a distributed environment and one can see similarities and differences between the two approaches. For example, Moss uses locking whereas Reed uses timestamps. For a detailed comparison of the two approaches see [8].

Foundations

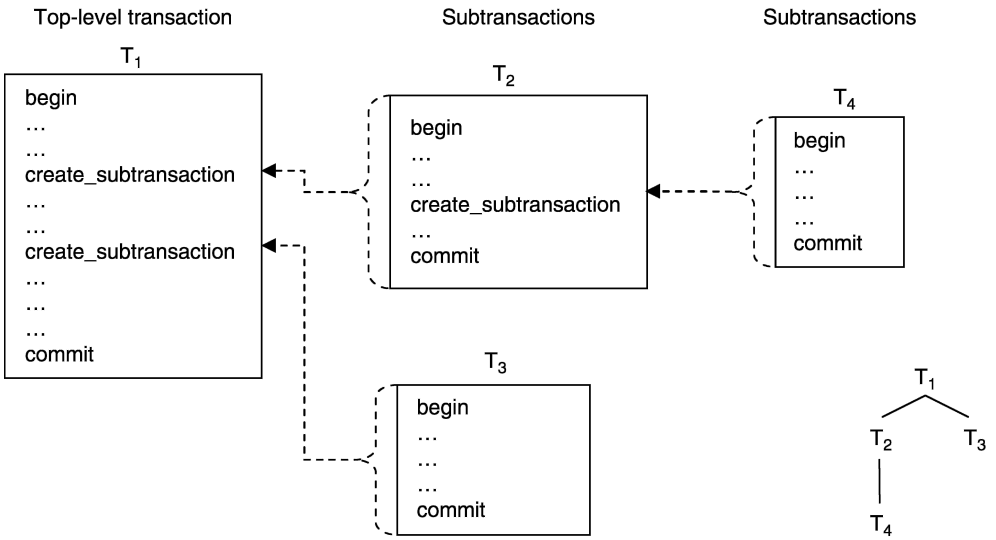
Structure of a Nested Transaction

The structure of a nested transaction is depicted by a transaction tree: its root represents the *top-level* transaction and its children represent *subtransactions* each one corresponding to a transactional unit. A subtransaction may be either a simple (*leaf)* transaction or a nested transaction, recursively expanding the structure to a hierarchy with multiple levels of transactions.

Leaf transactions are very similar to traditional ACID transactions, but they do not preserve the durability property of ACID transactions as will be explained in the Commit/Abort rules. Leaf transactions are the ones that actually manipulate the data in the database and perform the work. Intermediate level subtransactions and the top-level transaction operate on a higher level of abstraction: their purpose is to control when to create a new subtransaction. This hierarchy corresponds to the notion of nested spheres of control [3]. Figure 1 illustrates an example of a nested transaction and its corresponding transaction tree. It consists of a top-level transaction T_1 with two subtransactions T_2 and T_3; subtransaction T_2 is itself a nested transaction invoking subtransaction T_4. In this example, subtransactions T_3 and T_4 are leaf transactions performing the actual work. When multiple such nested transactions coexist their trees form a forest.

Synchronization of Nested Transactions

Subtransactions of nested transactions can execute concurrently and for proper synchronization Moss devised the following locking protocol: A subtransaction can either *hold* a lock or *retain* a lock. When a lock is held the transaction has exclusive rights to the object. After a subtransaction commits, its held or retained locks are *inherited* by the parent subtransaction. These inherited locks are not available to other subtransactions outside the subtree (sphere of control) of the holder; however, subtransactions within the subtree (descendants) can acquire a retained lock.



Nested Transaction Models. Figure 1. Structure of an example nested transaction; to the lower right is a depiction of the corresponding nested transaction tree.

The Locking Rules of nested transactions slightly paraphrased from Moss [8] are the following:

1. A transaction may hold a lock in write mode if no other transaction holds the lock (in any mode) and all retainers of the lock are ancestors of the requester.
2. A transaction may hold a lock in read mode if no other transaction holds the lock in write mode and all retainers of write locks are ancestors of the requester.
3. When a transaction commits, its parent inherits all locks the descendants held or retained in the same mode as the child held or retained them.
4. When a transaction aborts, all locks it holds or retains are released. Ancestors of the aborted transaction who were retaining the lock they continue to do so in the same mode as before the abort.

Commit/Abort of Nested Transactions

Moss describes a set of rules (state restoration algorithm in [8]) that dictate the behavior of nested transactions at commit/abort time. A subtransaction can unilaterally commit or abort independently of the others preserving atomicity, consistency and isolation from the ACID properties. The rules reworded from [8] are:

1. All children of a subtransaction must be resolved (committed or aborted) before it commits itself.
2. A subtransaction that unilaterally commits, first reaches its local commit point. However, it cannot finally commit until its parent transaction commits. Consequently, a subtransaction will commit only if it commits locally and all its ancestors commit including the top-level transaction.
3. If one of the ancestors of a subtransaction aborts, then the subtransaction itself will have to abort. In other words, the abort of a subtransaction causes the abort of all its children. Even if the subtransaction commits, its actions must be undone due to the abort of an ancestor. This is the reason that nested subtransactions violate the durability of ACID properties.

A nested transaction at the top level (i.e., the entire nested transaction as a whole) does observe durability. Therefore, nested transactions as a whole obey the ACID properties, despite the fact that individual nested subtransactions may violate the durability property.

Advantages of Nested Transactions

1. Nested transactions allow for a simple composition of subtransactions improving modularity of the overall structure.
2. The concurrent execution of subtransactions that follow the prescribed rules allows for enhanced concurrency while preserving consistency. This enhanced concurrency occurs because transactions can create subtransactions which execute concurrently.
3. The success or failure of a subtransaction is independent of the success of its siblings. When a subtransaction fails, its parent may try to execute it again, or ignore the failure, depending on the

situation. Therefore, failures that are contained within the scope of a subtransaction may be tolerated and do not necessarily cause the failure of the entire nested transaction.

Gray and Reuter present a mechanism to emulate nested transactions using database savepoints [6]. This approach emulates recovery as accurately as prescribed in the nested transaction model; however, it emulates locking not as flexibly as described in nested transactions.

Key Applications

Several systems were influenced by the nested transactions model, and as such they either incorporated or adapted its concept. The implementation of nested transactions by Moss, and especially the locking mechanism was used in the Argus system in which Moss worked with Liskov [7]. Camelot is another system that used nested transactions. It is a transaction processing system developed at Carnegie Mellon University using the Avalon programming language [5]. Camelot was a precursor to the Encina distributed transaction processing monitor which uses the Transactional-C language. Encina was commercialized under the name Transarc Corporation, which was acquired by IBM in 1994.

Nested transactions have also been implemented in Berkeley DB – a database library with bindings for several programming languages, which originated at the University of California – Berkeley, distributed by Sleepycat Software, which was acquired by Oracle Corporation in 2006.

Several researchers have also proposed systems with transaction processing components that are designed for application domains such as manufacturing, CSCW, software engineering, etc. where the ACID properties may be considered too strict; therefore, relaxed or advanced transaction models have been proposed to address the specific requirements that exist in these domains. Several of these transaction models appear in a book by Elmagarmid [4].

A notable example of an extension to the concept of nested transactions is the *multi-level transaction model* and its closely related *open nested transaction model* by Weikum and Schek [10,12]. The multi-level transaction model is represented by a transaction tree like in nested transactions; however, unlike nested transactions the transaction tree is balanced. Multi-level concurrency control takes advantage of the semantics of conflicting operations in the same level of the tree, such as commutativity, in order to enhance concurrency. In open nested transactions, the subtransactions are not restricted to have the same depth and are allowed to commit their results without waiting for their ancestors to commit [11].

Cross-references

- ▶ ACID Properties
- ▶ Concurrency Control – Traditional Approaches
- ▶ Data Conflicts
- ▶ Database Management System
- ▶ Distributed DBMS
- ▶ Distributed Transaction Management
- ▶ Extended Transaction Models
- ▶ Multilevel Transactions and Object-Model Transactions
- ▶ Open Nested Transaction Models
- ▶ Serializability
- ▶ TP Monitor
- ▶ Transaction
- ▶ Transaction Management
- ▶ Transaction Manager

Recommended Reading

1. Bjork L.A. Recovery scenario for a DB/DC system. In Proc. ACM Annual Conference, 1973, pp. 142–146.
2. Davies C.T. Data processing spheres of control. IBM Syst J., 17:179–198, 1978.
3. Davies C.T. Recovery semantics for a DB/DC system, In Proc. ACM Annual Conference, 1973, pp. 136–141.
4. Elmagarmid A.K. Database Transaction models for advanced applications. Morgan Kaufmann Publishers Inc., CA, 1992.
5. Eppinger J.L., Mummert L.B., and Spector A.Z. Camelot and avalon: A Distributed Transaction Facility. Morgan Kaufmann Publishers Inc., CA, 1991.
6. Gray J. and Reuter A. Transaction Processing: Concepts and Techniques. 1st edn. Morgan Kaufmann Publishers Inc., CA, 1992.
7. Liskov B. Distributed programming in Argus. Commun ACM., 31:300–312, 1988.
8. Moss E.B. Nested Transactions: An Approach to Reliable Distributed Computing. Technical Report. PhD Thesis. UMI Order Number: TR-260: Massachusetts Institute of Technology, 1981, pp. 178.
9. Reed D.P. Naming and Synchronization in a Distributed Computer System. Technical Report. PhD Thesis, UMI Order Number: TR-205: Massachusetts Institute of Technology, 1978, pp. 181.
10. Weikum G. Principles and Realization Strategies of Multilevel Transaction Management. ACM Trans Database Sys., 16:132–180, 1991.
11. Weikum G. and Schek H.-J. Concepts and Applications of Multilevel Transactions and Open Nested Transactions. In Database Transaction Models for Advanced Applications. Morgan Kaufmann Publishers Inc., 1992, pp. 515–553.

12. Weikum G. and Schek H.-J. Multi-level Transactions and open nested transactions. Q. Bull. IEEE TC on Data Engineering, 14:60–66, 1991.

.NET Remoting

Aniruddha Gokhale
Vanderbilt University, Nashville, TN, USA

Synonyms

Subsumed by windows communication framework

Definition

.NET Remoting [1,2] is a Microsoft technology comprising a set of application programming interfaces (APIs) for interprocess communication available within the .NET Framework. .NET Remoting serves as the distribution middleware layer, which enables communication between distributed applications that can choose their own transport protocol, serialization format, an application model and different schemes to manage object lifetimes.

Key Points

Central to the .NET Remoting architecture is the notion of a *remotable object* and a brokering capability that enables communication between *application domains*. An application domain in the .NET framework provides a type-safe and secure unit of execution and isolation. .NET Remoting makes it feasible for multiple application domains to communicate with each other without concern for whether they are hosted within the same operating system process, or across multiple different processes within the same node or across a network.

A remotable object is an object which is declared to be serializable remotable objects can cross application domain boundaries. Remotable Objects are published via an *Activation URL*. Client applications invoking services of a remotable object do so via proxies.

An important feature of .NET Remoting is its ability to allow applications to define their own serialization format, the choice of transport protocol used, and application model used by the communicating entities. By virtue of using the Common Language Runtime (CLR), .NET Remoting can provide interoperability across multiple managed languages. But its portability is restricted to Microsoft platforms only.

Cross-references

- ▶ Client-server architecture
- ▶ CORBA
- ▶ DCE
- ▶ DCOM
- ▶ J2EE Middleware
- ▶ Java RMI
- ▶ Request Broker
- ▶ SOAP

Recommended Reading

1. .NET Framework Remoting Architecture, http://msdn2.microsoft.com/en-us/library/2e7z38xb.aspx.
2. Microsoft Technologies, .NET Remoting: Core Protocol Specification, [MS-NRTP] v20080207, February 2008.

Network Attached Secure Device

Kazuo Goda
The University of Tokyo, Tokyo, Japan

Synonyms

NASD; Object-based Storage Device; OSD

Definition

A Network Attached Secure Device (often abbreviated to NASD) is a networked storage device which offers object-level storage accesses. NASD was first explored in a research project of the same name, which started at Carnegie Mellon University in 1995. NASD is now more often called Object based Storage Device (shortened to OSD).

Key Points

NASD is an approach of decoupling only low-level functions of file systems from NAS devices. A NASD manages only stored objects, each of which is identified with a unique file descriptor, and their additional information such as metadata and attributes, whereas a central policy server manages higher-level information which is used for name space management and authorization. A bunch of NASDs, with the assistance of the central policy server, can together work as a file store. This may look similar to NAS at a glance, but the difference is that most commands and data are directly transferred between clients and NASDs. Conventional storage subsystems have a number of storage devices under a storage controller, which is thus likely to

become a performance bottleneck. NASD has benefits of scaling aggregate bandwidth by spreading partial functions over a number of disk processors.

Object-level storage abstraction, introduced by NASD, is recognized to be the third alternative in addition to block-level storage abstraction typically seen in SAN storage environments and file-level storage abstraction in NAS storage environments. Content-Addressable Storage is another example which offers object-level storage accesses and is now widely deployed in enterprise systems.

The idea of NASD has been standardized as ANSI T10 SCSI OSD, which specifies an extension of the SCSI protocol for clients and devices to exchange objects and their related information. Thus, OSD is sometimes seen as a promising infrastructure of intelligent storage devices in which more intelligence will be incorporated.

Cross-references

► Active Disks
► Intelligent Storage Systems

Recommended Reading

1. ANSI. Information Technology - SCSI Object-Based Storage Device Commands (OSD). Standard ANSI/INCITS 400–2004. 2004.
2. Gibson G.A. and Van Meter R. Network attached storage architecture. Commun. ACM, 43(11):37–45, 2000.
3. Gibson G.A., Nagle D.F., Amiri K., Chang F.W., Feinberg E.M., Gobioff H., Chen Lee, Ozceri B., Riedel E., Rochberg D., and Zelenka J. File server scaling with network-attached secure disks. In Proc. 1997 ACM SIGMETRICS Int. Conf. on Measurement and Modeling of Comp. Syst., 1997, pp. 272–284.

Network Attached Storage

Kazuo Goda
The University of Tokyo, Tokyo, Japan

Synonyms

NAS

Definition

Network attached storage is a storage device which is connected to a network and provides file access services. Network attached storage is often abbreviated to NAS. Although the term NAS refers originally to such a storage device, when the term is used in the context of explaining and comparing storage network architectures it sometimes refers to a storage network architecture in which NAS devices are mainly implemented.

Key Points

A NAS device is basically comprised of disk drives which store files and controllers which export access services to the files. A file sever which runs network file system (NFS) and/or common internet file system (CIFS) to export file sharing services is a type of NAS implementation, but recent NAS products are sometimes implemented using dedicated hardware and software to increase reliability and performance. A diskless NAS device which contains only controllers is sometimes referred to as a NAS gateway or a NAS head. A NAS gateway/head has two types of network ports: one is connected to disk storage devices over a SAN and the other is to NAS clients over IP networks. The clients are thus provided with access services towards files stored in the storage devices. That is, a NAS gateway/head can be seen as a service bridge between SAN and NAS systems.

Cross-references

► Direct Attached Storage
► Storage Area Network
► Storage Network Architectures

Recommended Reading

1. Storage Network Industry Association. The Dictionary of Storage Networking Terminology. Available at: http://www.snia.org/.
2. Troppens U., Erkens R., and Müller W. Storage Networks Explained. Wiley, New York, 2004.

Network Data Model

Jean-Luc Hainaut
University of Namur, Namur, Belgium

Synonyms

CODASYL data model; DBTG data model

Definition

A database management system complies with the *network data model* when the data it manages are organized as data records connected through binary relationships. Data processing is based on navigational primitives according to which records are accessed and

updated one at a time, as opposed to the set orientation of the relational query languages. Its most popular variant is the CODASYL DBTG data model that was first defined in the 1971 report from the CODASYL group, and that has been implemented into several major DBMSs. They were widely used in the seventies and eighties, but most of them are still active at the present time.

Historical Background

In 1962, C. Bachman of General Electric, New-York, started the development of a data management system according to which data records were interconnected via a network of relationships that could be navigated through [2]. Called Integrated Data Store (IDS), this disk-based system quickly became popular to support the storage, the management and the exploitation of corporate data.

IDS was the main basis of the work of the CODASYL Data Base Task Group (DBTG) that published its first major report in 1971 [3,6], followed by a revision in 1973 [3,9]. This report described a general architecture for DBMSs, where the respective roles of the operating system, the DBMS and application programs were clearly identified. It also provided a precise specification of languages for data structure definition (Data Description Language or Schema DDL), for data extraction and update (Data Manipulation Language or DML) and for defining interfaces for application programs through language-dependent views of data (Sub-schema DDL).

The 1978 report [7,9] clarified the model. In particular physical specifications such as indexing structures and storage were removed from the DDL and collected into the Data Storage Description Language (DSDL), devoted to the physical schema description. In 1985, the X3H2 ANSI Database Standard committee issued standards for network database management systems, called NDL and based on the 1978 CODASYL report. However, due to the increasing dominance of the relational model these proposals have never been implemented nor updated afterwards.

Some of the most important implementations were Bull IDS/II (an upgrade of IDS), NCR DBS, Siemens UDS-1 and UDS-2, Digital DBMS-11, DBMS-10 and DBMS-20 (now distributed by Oracle Corp.), Data General DG/DBMS, Philips Phollas, Prime DBMS, Univac DMS 90 and DMS 1100 and Culliname IDMS, a machine-independent rewriting of IDS (now distributed by Computer Associates). Other DBMSs have been developed, that follow more or less strictly the CODADYL specifications. Examples include Norsk-Data SYBAS, Burroughs DMS-2, CDC IMF, NCR IDM-9000, Cincom TOTAL and its clone HP IMAGE (which were said to define the *shallow* data model), and MDBS and Raima DbVista that both first appeared on MS-DOS PCs.

In the seventies, IBM IMS was the main competitor of CODASYL systems [11]. From the early eighties, they both had to face the increasing influence of relational DBMSs such as Oracle (from 1979) and IBM SQL/DS (from 1982 [8]). Nowadays, most CODASYL DBMSs provide an SQL interface, sometimes through an ODBC API. Though the use of CODASYL DBMSs is slowly decreasing, many large corporate databases are still managed by network DBMSs. This state of affairs will most probably last for the next decade. Network databases, as well as hierarchical databases, are most often qualified *legacy*, inasmuch as they are often expected to be replaced, sooner or later, by modern database engines.

Foundations

The presentation of the network model is based on the specifications published in the 1971 and 1973 reports, with which most CODASYL DBMSs comply.

The Languages

The data structures and the contents of a database can be created, updated and processed by means of four languages, namely the *Schema DDL* and *Sub-schema DDL*, through which the global schema and the subschemas of the database are declared, the *Data Storage Description Language* or *DSDL* (often named *DMCL*) that allows physical structures to be defined and tuned, and the *DML* through which application programs access and update the contents of the database.

Gross Architecture of a CODASYL DBMS

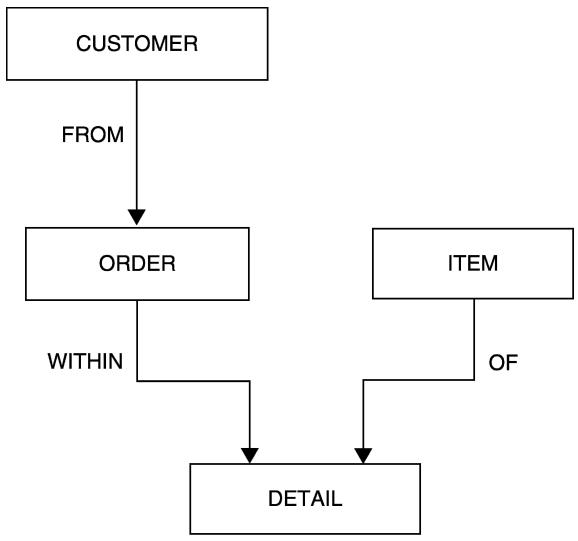
The CODASYL reports define the interactions between client application programs and the database. The resulting architecture actually laid down the principles of modern DBMSs. The DBMS includes (at least) three components, namely the DDL compiler, the DML compiler and the database control system (DBCS, or simply system). The DDL compiler translates the data description code into internal tables that are stored in the database, so that they can be exploited by the DML compiler at program compile time and by the DBCS at program run time. Either the DML compiler

is integrated into the host language compiler (typically COBOL) or it acts as a precompiler. It parses applications programs and replaces DML statements with calls to DBMS procedures. The DBCS receives orders from the application programs and executes them. Each program includes a *user working area* (UWA) in which data to and from the database are stored. The UWA also comprises registers that inform the program on the status of the last operations, and in particular references to the last records accessed/updated in each record type, each area, each set type and globally for the current process. These references, called *currency indicators*, represent static predefined cursors that form the basis for the navigational facilities across the data.

The Data Structures

The pictorial representation of a schema, or *Data Structure Diagram* [1], looks like a graph, where the nodes are the database *record types* and the edges are *set types*, that is, binary 1:N relationship types between record types, conventionally directed from the *one* side to the *many* side. Both record types and set types are named (Fig. 1). In this popular representation, record fields as well as various characteristics of the data structures are ignored.

Records and Record Types A *record* is the data unit exchanged between the database and the application program. A program reads one record at a time from the database and stores one record at a time in the database. Records are classified into *record types* that define their common structure and default behavior. The intended goal of a record type is to represent a real world entity type. A database key, which is a database-wide system-controlled identifier, is associated with each record, acting as an object-id.



Network Data Model. Figure 1. Diagram representation of the schema of a sample database.

Each database includes the SYSTEM record type, with one occurrence only, that can be used to define access paths across user record types through SYSTEM-owned singular set types.

The schema specifies, via the record type *location mode*, how a record is stored and how it is retrieved when storing a dependent record. This feature is both very powerful and, when used at its full power, fairly complex. Two main variants are proposed:

- location mode *calc using field-list*: the record is stored according to a hashing technique (or later through B-tree techniques) applied to the record key, composed of one or several fields of the record type (*field-list*); at run time, the default way to access a record will be through this record key;
- location mode *via set type S*: the record is physically stored as close as possible to the current record of set type *S*; later on, the default way to access a record will be through an occurrence of S identified by its *set selection* mode.

Record Fields A record type is composed of *fields*, the occurrences of which are *values*. Not surprisingly (CODASYL was also responsible for the COBOL specifications), their structure closely follow the record declaration of COBOL. The DDL offers the following field structures:

- *data item*: elementary piece of data of a certain type (arithmetic, string, implementor defined);
- *vector*: array of values of the same type; its size can be fixed or variable;
- *repeating group*: a somewhat misleading name for a possibly repeating aggregate of fields of any kind.

The fields of a record type can be atomic or compound, single-valued or multi-valued, mandatory or optional (through the null value); these three dimensions allow complex, multi-level field structures to be defined.

Sets and Set Types Basically, a CODASYL *set* is a list of records made up of a head record (the *owner* of the set) followed by zero or more other records (the *members* of the set). A *set type* S is a schema construct defined by

its name and comprising one owner record type and one or more member record type(s). Considering set type S with owner type A and member type B, any A record is the owner of one and only one occurrence of S and no B record can be a member of more than one occurrence of S. In other words, a set type materializes a 1:N relationship type. The owner and the members of a set type are distinct. This limitation has been dropped in the 1978 specifications, but has been kept in most implementations (exceptions: SYBAS and MDBS). Cyclic structures are allowed provided they include at least two record types. It must be noted that a set type can include more than one member record type.

The member records of S can be ordered (first, last, sorted, application-defined). This characteristic is static and cannot be changed at run-time as in SQL. The insertion of a member record in an occurrence of S can be performed at creation time (automatic insertion mode) or later by the application program (manual insertion mode). Once a record is a member of an occurrence of S, its status is governed by the retention mode; it can be removed at will (optional), it cannot be changed (fixed) or it can be moved from an occurrence to another but cannot be removed (mandatory).

The *set [occurrence] selection* of S defines the default way an occurrence of S is determined in certain DML operations such as storing records with automatic insertion mode.

Areas An area is a named logical repository of records of one or several types. The records of a definite type can be distributed in more than one area. The intended goal is to offer a way to partition the set of the database records according to real world dimensions, such as geographic, organizational or temporal. However, since areas are mapped to physical devices, they are sometimes used to partition the data physically, e.g., across disk drives.

Schema and Sub-schemas The data structures of each database are described by a schema expressed in the DDL. Though DDL is host language independent, its syntax is reminiscent of COBOL. Views are defined by sub-schemas. Basically, a subschema is a host language dependent description of a subset of the data structures of a schema. Some slight variations are allowed, but they are less powerful than relational database capabilities. Figure 2 shows a fragment of the schema declaring the data structures of Fig. 1.

Data Manipulation

The DML allows application programs to ask the DBCS data retrieval and update services. The program accesses the data through a sub-schema that identifies the schema objects the instances of which can be retrieved and updated as well as their properties, such as the data type of each field. Exchange between the host language and the DBCS is performed via the UWA, a


```
schema name is ORDER-MANAGEMENT.

area name is DOMESTIC.
area name is FOREIGN.

record name is CUSTOMER within DOMESTIC, FOREIGN;
location mode is calc using CUST-NO duplicates not allowed.
2 CUST-NO type is character 10.
2 CUST-NAME type is character 45.
2 CUST-ADDRESS.
  4 STREET character 32.
  4 CITY character 20.
  4 PHONE type is character 15 occurs 3 times.

record name is ORDER within DOMESTIC;
location mode is via FROM set;
duplicates not allowed for ORD-NO.
  2 ORD-NO type is decimal 12.
  2 ORD-DATE type is date.

set name is FROM;
owner is CUSTOMER;
order is sorted duplicates not allowed;
member is ORDER mandatory automatic key is ascending ORD-NO;
set selection is through current of FROM.
```

Network Data Model. Figure 2. Fragment of the DDL code defining the schema of Fig. 1.

shared set of variables included in each running program. This set includes the currency indicators, the process status (e.g., the error indicators) and record variables in which the data to and from the database are temporarily stored. Many DML statements use the currency indicators as implicit arguments. Such is the case for set traversal and for record storing. Based on the currency indicators, on the *location mode* of record types and on the *set selection* option of set types, sophisticated positioning policies can be defined, leading to tight application code.

Data Retrieval The primary aim of the *find* statement is to retrieve a definite record on the basis of its position in a specified collection and to make it the current of all the *communities* which it belongs to, that is, its database, its area, its record type and each of its set types. For instance, if an *ORDER* record is successfully retrieved, it becomes the current of the database for the running program (the *current of run unit*), the current of the *DOMESTIC* area, the current of the *ORDER* record type and the current of the *FROM* and *WITHIN* set types. The variants of the *find* statement allow the program to scan the records of an area, of a record type and of the members and the owner of a set. They also provide selective access among the members of a set.

The *get* statement transfers field values from a current record in the UWA, from which they can then be processed by the program.

Data Update

A record *r* is inserted in the database as follows: first, field values of *r* are stored in the UWA, then the current of each set in which *r* will be inserted is retrieved and finally a *store* instruction is issued. The *delete* instruction applies to the current record. For this operation, the DBCS enforces a *cascade* policy: if the record to be deleted is the owner of sets whose members have a mandatory or fixed retention mode, those members are deleted as well. The *modify* statement transfers in the current of a record type the new values that have been stored in the UWA. Insertion and removal of the current of a record type is performed by *insert* and *remove* instructions. Transferring a mandatory member from a set to another cannot be carried out by merely removing then inserting the record. A special case of the *modify* statement makes such a transfer possible. Later specifications as well as some implementations propose a specific statement for this operation.

Figure 3 shows, in an arbitrary procedural pseudocode, a fragment that processes the orders of customer *C400* and another fragment that creates an *ORDER* record for the same customer.

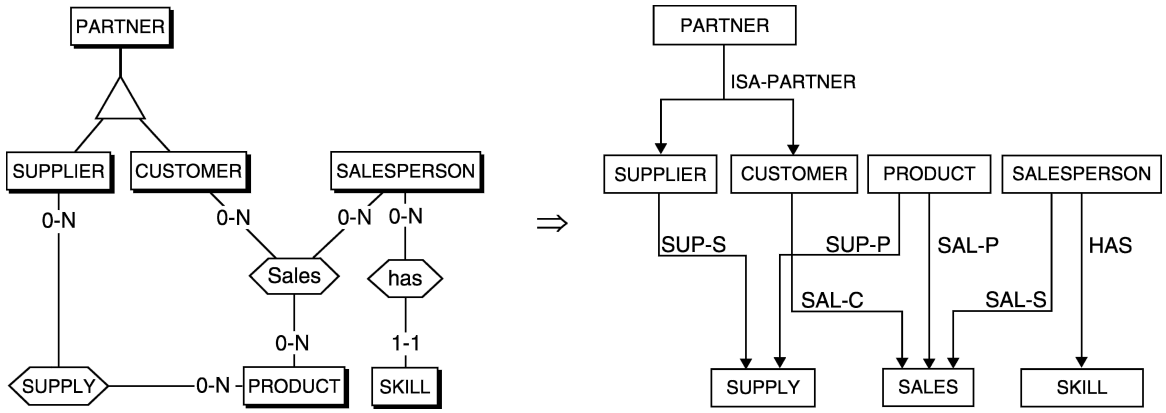
Entity-relationship to Network Mapping

Among the many DBMS data models that have been proposed since the late sixties, the network model is probably the closest to the Entity-Relationship model [5]. As a consequence, network database schemas tend to be more readable than those expressed in any other DBMS data model, at least for simple schemas. Each entity type is represented by a record type, each attribute by a field and each simple relationship type by a set type. Considering modern conceptual formalisms, the network model suffers from several deficiencies,

```
CUSTOMER.CUST-NO := "C400"                    /* store search key value in the UWA
find CUSTOMER record                          /* find CUSTOMER record "C400"
find first ORDER record of FROM set           /* find first ORDER record owned by this CUSTOMER record
while ERROR-COUNT = 0
    get ORDER                                 /* get item values of current ORDER record
    <process item values of current ORDER>
    find next ORDER record of FROM set        /* find first ORDER record
end-while

CUSTOMER.CUST-NO := "C400"
find CUSTOMER record                          /* make CUSTOMER record "C400" the current of FROM
if ERROR-COUNT = 0 then
    ORDER.ORD-NO := 30183                     /* store value of ORD-NO in the UWA
    ORDER.CUST-DATE := "2008/08/19"           /* store value of ORD-DATE in the UWA
    store ORDER                               /* store ORDER record and insert it in the current FROM set
end-if
```

Network Data Model. Figure 3. Two examples of data manipulation code.



Network Data Model. Figure 4. Partial translation of a representative Entity-relationship schema (left) into a network schema (right).

notably the lack of generalization-specialization (is-a) hierarchies and the fact that relationship types are limited to the 1:N category. Translating an Entity-relationship schema into the network model requires the transformation of these missing constructs into standard structures.

Is-a hierarchies. Three popular transformations can be applied to express this construct in standard data management systems, namely one record type per entity type, one record type per supertype and one record type per subtype. Representing each entity type by a distinct record type and forming a set type *S* with each super-type (as owner of *S*) and all its direct subtypes (as members of *S*) is an appropriate implementation of the first variant.

1:1 relationship type. This category is a special case of 1:N and can be expressed by a mere set type, together with dynamic restriction on the number of members in each set. However, merging both record types when one of them depends on the other one (e.g., as an automatic, mandatory member) is also a common option.

Complex relationship type. In most implementations, n-ary and N:N relationship types as well as those with attributes must be reduced to constructs based on 1:N relationship types only through standard transformations. A complex relationship type *R* is represented by a relationship record type *RT* and by as many set types as *R* has roles. The attributes of *R* are translated into fields of *RT*. Cyclic relationship types, if necessary, will be translated in the same way.

Figure 4 illustrates some of these principles.

Discussion

The network model offers a simple view of data that is close to semantic networks, a quality that accounts for much if its past success. The specifications published in the 1971 and 1973 reports exhibited a confusion between abstraction levels that it shared with most proposals of the seventies and that was clarified in later recommendations, notably the 1978 report and X3H2 NDL. In particular, the DDL includes aspects that pertain to logical, physical and procedural layers.

Though they were not implemented in most commercial DBMSs, the CODASYL recommendations included advanced features that are now usual in database technologies such as database procedures, derived fields, check and some kind of triggers.

Key Applications

CODASYL DBMSs have been widely used to manage large corporate databases submitted to both batch and OLTP (On-line Transaction Processing) applications. Compared with hierarchical and relational DBMS, their simple and intuitive though powerful model and languages made them very popular for the development of large and complex applications. However, their intrinsic lack of flexibility in rapidly evolving contexts and the absence of user-oriented interface made them less attractive for decisional applications, such as data warehouses.

Cross-references

► Hierarchical Data Model
► Relational Model

► Database Management System
► Entity-Relationship Model

Recommended Reading

1. Bachman C. Data structure diagrams. ACM SIGMIS Database, 1(2):4–9, 1969.
2. Bachman C. The programmer as navigator. Commun. ACM, 16(11):635–658, 1973.
3. DBTG C. CODASYL data base task group, April 1971 report, ACM, New York, 1971.
4. DDLC C. CODASYL data description language committee, CODASYL DDL Journal of Development (June 1973), NBS Handbook 113 (Jan. 1974), 1973.
5. Elmasri R. and Navathe S. Fundamentals of Database Systems (3rd edn.). Addison-Wesley, 2000. (The appendix on the network data model has been removed from later editions but is now available on the authors' site.)
6. Engels R.W. An analysis of the April 1971 DBTG report. In Proc. ACM SIGFIDET Workshop on Data Description, Access, and Control, 1971, pp. 69–91.
7. Jones J.L. Report on the CODASYL data description language committee. Inf. Syst., 3(4):247–320, 1978.
8. Michaels A., Mittman B., and Carlson C.A. Comparison of relational and CODASYL approaches to data-base management. ACM Comput. Surv., 8(1):125–151, 1976.
9. Olle W. The CODASYL Approach to Data Base Management, Wiley, New York, NY, 1978.
10. Taylor R. and Frank R. CODASYL data-base management systems. ACM Comput. Surv., 8(1):67–103, 1976.
11. Tsichritzis D. and Lochovsky F. Data Base Management Systems, Academic Press, New York, NY, 1977.

Network Database

► Graph Database

Network Topology

► Visualizing Network Data

Neural Networks

Pang-Ning Tan
Michigan State University, East Lansing, MI, USA

Synonyms

Connectionist model
Parallel distributed processing

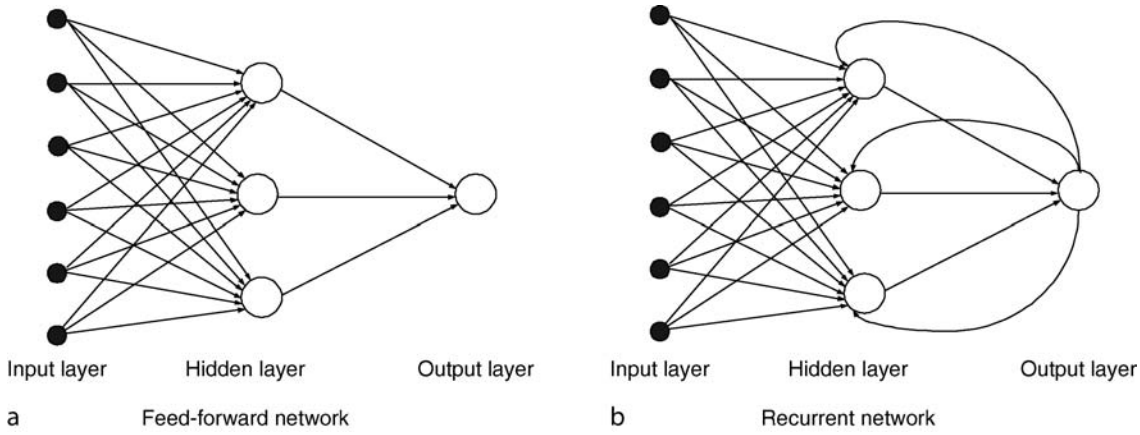
Definition

An artificial neural network (ANN) is an abstract computational model designed to solve a variety of supervised and unsupervised learning tasks. While the discussion in this chapter focuses only on supervised classification, readers who are interested in unsupervised learning using ANN may refer to the literature on vector quantization [6] and self organizing maps [11]. An ANN consists of an assembly of simple processing units called neurons connected by a set of weighted edges (or synapses), as shown in Fig. 1. The neurons are often configured into a feed-forward multi-layered topology, with outputs from one layer being fed into the next layer. The first layer, which is known as the input layer, encodes the attributes of the input data, while the last layer, known as the output layer, encodes the neural network's output. Hidden layers are the intermediary layers of neurons between the input and output layers. A feed-forward ANN without any hidden layer is called a perceptron [16]. Another common ANN architecture is the recurrent network, which allows a neuron to feed its output back into the inputs of other preceding neurons in the network. Such a network topology is useful for modeling temporal and sequential relationships in dynamical systems.

Historical Background

The design of an ANN was inspired by the desire to emulate how a human brain works. Interest in this field began to emerge following the seminal work of McCulloch and Pitts [13], who attempted to understand how complex patterns can be modeled in the brain using a large number of inter-connected neurons. They presented a simplified model of a neuron and showed how a collection of these neurons could be used to represent logical propositions. Nevertheless, they did not provide an algorithm to estimate the weights of the network.

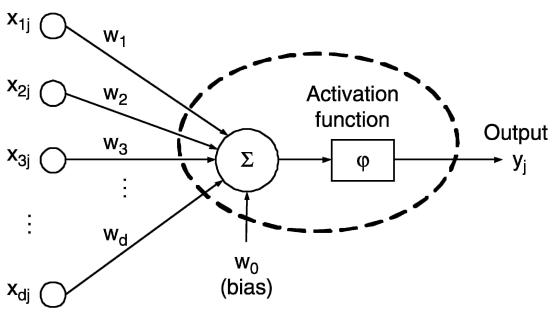
A major step forward in the study of ANN occurred when Hebb [7] formulated a postulate relating the cerebral activities of the brain to the synaptic connections between neurons. Hebb theorized that the process of learning takes place when a pair of neurons is activated repeatedly, thus making their synaptic connection stronger. By strengthening the connection, this enables the network to recognize the appropriate response when the same stimulus is re-applied. This idea forms the basis for what is now known as Hebbian learning.



Neural Networks. Figure 1. Architecture of artificial neural network.

The invention of the perceptron by Rosenblatt [16] marked another major development of the field. Rosenblatt demonstrated that a simple perceptron can be trained to recognize visual patterns by modifying its weights using an iterative error correcting algorithm. A theorem was subsequently proposed to show the guaranteed convergence of the perceptron training rule in finite time – a result that subsequently triggered a wave of interest in the field. During this period, new perceptron training algorithms began to emerge, such as the Widrow-Hoff [18] and stochastic gradient descent [2] algorithms. Such interest, however, began to wane as Minsky and Papert [14] showed the practical limitations of perceptrons, particularly in terms of solving non-linearly separable problems such as the exclusive-or (XOR) function. Though multi-layer neural networks may overcome these limitations, there has yet to be any feasible learning algorithms that can automatically adjust the weights of the neurons in the hidden layer.

Interest in ANN was finally rekindled in the 1980s, fueled by the successful development of the Boltzmann machine [1] and the re-discovery of the backpropagation algorithm [17], both of which demonstrated the feasibility of training multi-layer neural networks. Furthermore, as computers become cheaper, this permits more researchers to participate in the field and experiment with the capabilities of neural networks, unlike the situation in the 1960s. Research in ANN continued to flourish with the development of more complex networks such as radial basis function networks [15], Hopfield networks [8], Jordan networks [10], and Elman networks [4].



Neural Networks. Figure 2. Structure of an artificial neuron.

Foundations

Neurons are the elementary processing units of an ANN. The structure of a neuron is shown in Fig. 2. Each neuron consists of a set of weighted edges $\{w_1,w_2,,w_d\}$, a threshold w_0 (known as the bias), an adder Σ that computes the weighted sum of its input, and an activation function φ that transforms the weighted sum into its output value. The computation performed by a neuron can be expressed in the following form:

$$y_j = \varphi\left(\sum_{i=1}^{d} w_i x_{ij} + w_0\right) \equiv \varphi(z_j) \qquad (1)$$

The output of a neuron can be discrete or continuous, depending on the choice of activation function. For example, the Heaviside function can be used to produce a binary-valued output:

$$\varphi(z) = \begin{cases} 1, z \geq 0 \\ 0, \text{otherwise} \end{cases}$$

while the linear function $\varphi(z) = z$ and sigmoid function

$$\varphi(z) = \frac{1}{1 + \exp(-z)}$$

produce a continuous-valued output.

An ANN must be trained to determine the appropriate set of weights for a given classification task. Training is accomplished by processing the training examples one at a time and comparing the predicted class of a training example to its actual class. The error in prediction is then used to modify the weights of the network. This process is repeated until a stopping criterion is satisfied.

Table 1 shows the activation functions and weight update formula employed by several perceptron learning algorithms. Notice that the amount of weight adjustment is proportional to the difference between the true output y_j and the predicted output $\varphi(z_j)$ of a neuron. Computing the error term $[y_j - \varphi(z_j)]$ in the weight update formula is straightforward for perceptrons, but is more challenging for multi-layer networks because the true output of each hidden neuron is unknown. Furthermore, it is unclear the extent to which each of the neuron's weight contributes to the overall network error, an issue that is known as the credit assignment problem. The backpropagation algorithm attempts to overcome this problem by employing a differentiable activation function (such as the sigmoid function) so that a gradient descent strategy can be used to derive the weight update formula. In the backpropagation algorithm, the weight update step is decomposed into two phases. During the forward phase, the outputs of the neurons are computed one layer at a time, starting from neurons in the first hidden layer, followed by those in the next layer until the neurons in the output layer are computed. Next, during the backward phase, the errors are propagated in the reverse direction, starting from neurons in the output layer, followed by those in the preceding layer until the errors of neurons in the first hidden layer are computed.

Neural Networks. Table 1. Perceptron training rules

Perceptron training rule	Activation function	Weight update formula
Rosenblatt's perceptron	Heaviside function	$w_i \leftarrow w_i + \eta[y_j - \varphi(z_j)]x_{ij}$
Widrow-Hoff	Linear function	$w_i \leftarrow w_i + \eta \sum_j [y_j - \varphi(z_j)]x_{ij}$
Stochastic gradient descent	Linear function	$w_i \leftarrow w_i + \eta[y_j - \varphi(z_j)]x_{ij}$

There are several issues that must be considered when building an ANN model. First, the structure of the network must be appropriately chosen to avoid model overfitting. To determine the structure with the right model complexity, model selection approaches such as cross-validation may be used. Regularization methods are also applicable to penalize network structures that are overly complex. Another possibility is to apply the Bayesian approach, which provides a principled framework for handling the model complexity problem. A review of the Bayesian approach for neural networks can be found in [16]. The cascade correlation algorithm [17] is another useful approach to grow the network dynamically from a structure with no hidden layers. Hidden nodes are then added one at a time until the residual error of the network falls within some acceptable level.

Since the error function of a multi-layer neural network is not convex, convergence of the backpropagation algorithm to a local minimum is another potential issue to consider. One way to mitigate the problem is by adding a momentum term to the weight update formula to escape from the local minimum. Alternative strategies include repeating the network simulation with different initial weights and applying a stochastic gradient descent method to estimate the network weights.

Instead of learning a discriminant function to distinguish instances from different classes, an ANN can also be trained to produce posterior probabilities of their class memberships. This is desirable because a probabilistic framework provides a natural way to compensate for imbalanced class distributions, to incorporate the cost of misclassification into decision making, and to fuse the outputs from multiple networks. An ANN can be designed to generate probabilistic outputs by training its weights to minimize the cross-entropy error function [12] instead of the sum-of-square error function.

As ANN encodes a sophisticated mathematical function, the ability to explain how it makes its prediction is crucial for a number of applications, particularly in the medical and legal domains. Algorithms have been developed to extract symbolic representations or rules from a trained network [18]. Such algorithms generally fall into two categories. The first category generates rules based on features constructed from the weights of the network connections. The

second category uses the trained network to label a data set and then applies rule extraction algorithms on the labeled data set.

In general, multi-layer neural networks are universal approximators, allowing them to fit any type of function. Furthermore, since it is an abstract computational model, it is also applicable to a variety of supervised, unsupervised, and semi-supervised learning tasks.

Key Applications

ANN has been successfully applied to various applications including pattern recognition (face, handwriting, and speech recognition), finance (bankruptcy prediction, bond rating, and economic forecasting), manufacturing (process control, tool condition monitoring, and robot scheduling), and computer aided diagnosis (arrhythmia identification and tumor detection in biological tissues).

Future Directions

As ANN is an abstract computational model that is applicable to a wide spectrum of learning tasks, much of the future advances in this field are tied to progress in the areas of supervised, unsupervised, and semi-supervised learning.

Experimental Results

The experimental evaluation often depends on the learning tasks. For supervised classification, some of the typical evaluation criteria include model accuracy, specificity, sensitivity, F-measure, and run-time for model building.

Data Sets

A large collection of data sets are available at the UCI data mining and machine learning repository. For a more comprehensive list of data sets and other information about neural networks, go to http://www.neural-forecasting.com/.

Url to Code

A comprehensive set of functions for implementing feed-forward and recurrent neural networks is available in MATLAB toolbox.

Cross-references

► Decision Tree Classification
► Data Clustering

Recommended Reading

1. Ackley D.H., Hinton G.E., and Sejnowski T.J. A learning algorithm for Boltzmann machines. Cogn. Sci., 9(1) 147–169, 1985.
2. Amari S. A theory of adaptive pattern classifiers. IEEE Trans. Electronic Comput., 16, 299–307, 1967.
3. Craven M. and Shavlik J.W. Learning symbolic rules using artificial neural networks. In Proc. 10th Int. Conf. on Machine Learning, 1993, pp. 73–80.
4. Elman J.L. Finding structure in time. Cogn. Sci., 14, 179–211, 1990.
5. Fahlman S.E. and Lebiere C. The CASCADE-CORRELATION learning architecture. Adv. Neural Inform. Process. Syst., pp. 524–532, 1989.
6. Haykin S. Neural networks – a comprehensive foundation, 2nd edn. Prentice-Hall, Englewood, Cliffs, NJ, 1998.
7. Hebb D. The Organization of Behaviour, Wiley, New York, 1949.
8. Hopfield J.J. Neural networks and physical systems with emergent collective computational abilities. In Proc. National Academy of Sciences, 79, 2554–2558, 1982.
9. Hopfield J.J. Learning algorithms and probability distributions in feed-forward and feed-back networks. In Proc. National Academy of Sciences, 84, 8429–8433, 1987.
10. Jordan M.I. Attractor dynamics and parallelism in a connectionist sequential machine. In Proc. 8th Annual Conf. of the Cognitive Science Society, 1986, pp. 531–546.
11. Kohonen T. Self-Organizing Maps, Springer, Berlin. 2001.
12. Lampinen J. and Vehtari A. Bayesian approach for neural networks – review and case studies, Neural Networks, 14(3): 7–24, 2001.
13. McCulloch W.S. and Pitts W. A logical calculus of the ideas immanent in nervous activity. Bull. Math. Biophys., 5:115–133, 1943.
14. Minsky M. and Papert S. Perceptrons: An Introduction to Computational Geometry. MIT, Cambridge, MA, 1969.
15. Powell M. Radial Basis Functions for Multivariable Interpolation : A Review. In Algorithms for Approximation. Mason J.C. and Cox M.G. (eds.). pp. 143–167, 1987.
16. Rosenblatt F. Principles of Neurodynamics. Spartan Books, New York, 1959.
17. Rumelhart D.E., Hinton G.E., and Williams R.J. Learning representations by backpropagating errors. Nature, 323, 533–536, 1986.
18. Widrow B. and Hoff M.E. Jr. Adaptive switching circuits. IRE WESCON Convention Record. 1960, pp. 96–104.

New Media Metadata

► Multimedia Metadata

NEXI

► Narrowed Extended XPath I

NFS

► Storage Protocols

NF-SS

► Normal Form ORA-SS Schema Diagrams

N-Gram Models

Djoerd Hiemstra
University of Twente, AE Enschede, The Netherlands

Definition

In language modeling, *n*-gram models are probabilistic models of text that use some limited amount of history, or word dependencies, where *n* refers to the number of words that participate in the dependence relation.

Key Points

In automatic speech recognition, *n*-grams are important to model some of the structural usage of natural language, i.e., the model uses word dependencies to assign a higher probability to "how are you today" than to "are how today you," although both phrases contain the exact same words. If used in information retrieval, simple unigram language models (*n*-gram models with $n = 1$), i.e., models that do not use term dependencies, result in good quality retrieval in many studies. The use of bigram models (*n*-gram models with $n = 2$) would allow the system to model direct term dependencies, and treat the occurrence of "New York" differently from separate occurrences of "New" and "York," possibly improving retrieval performance. The use of trigram models would allow the system to find direct occurrences of "New York metro," etc. The following equations contain respectively (1) a unigram model, (2) a bigram model, and (3) a trigram model:

$$P(T_1, T_2, \cdots T_n|D) = P(T_1|D)P(T_2|D) \cdots P(T_n|D) \quad (1)$$

$$P(T_1, T_2, \cdots T_n|D) = P(T_1|D)P(T_2|T_1, D) \cdots P(T_n|T_{n-1}, D) \quad (2)$$

$$P(T_1, T_2, \cdots T_n|D) = P(T_1|D)P(T_2|T_1, D)P(T_3|T_1, T_2, D) \cdots P(T_n|T_{n-2}, T_{n-1}, D) \quad (3)$$

The use of *n*-gram models increases the number of parameters to be estimated exponentially with *n*, so special care has to be taken to smooth the bigram or trigram probabilities. Several studies have shown small but significant improvements of using bigrams if smoothing parameters are properly tuned [2,3]. Improvements of the use of *n*-grams and other term dependencies seem to be bigger on large data sets [1].

Cross-references

► Language Models
► Probability Smoothing

Recommended Reading

1. Metzler D. and Bruce Croft W. A Markov random field model for term dependencies. In Proc. 31st Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval, 2005, pp. 472–479.
2. Miller D.R.H., Leek T., and Schwartz R.M. A hidden Markov model information retrieval system. In Proc. 22nd Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval, 1999, pp. 214–221.
3. Song F. and Bruce Croft W. A general language model for information retrieval. In Proc. 22nd Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval, 1999, pp. 4–9.

NIAM

► Object-Role Modeling

NN Classification

► Nearest Neighbor Classification

NN Query

► Nearest Neighbor Query
► Nearest Neighbor Query in Spatio-temporal Databases

NN Search

► Nearest Neighbor Query in Spatio-temporal Databases

Node

► Activity

Noise Addition

Josep Domingo-Ferrer
Universitat Rouira i Virgili, Tarragona, Catalonia

Synonyms

Additive noise

Definition

Noise addition is a masking method for statistical disclosure control of numerical microdata that consists of adding random noise to original microdata.

Key Points

The noise addition algorithms in the literature are:

1. *Masking by uncorrelated noise addition.* The vector of observations x_j for the jth attribute of the original dataset X_j is replaced by a vector

$$z_j = x_j + \epsilon_j$$

where ϵ_j is a vector of normally distributed errors drawn from a random variable $\varepsilon_j \sim N(0, \sigma^2_{\varepsilon_j})$, such that $Cov(\varepsilon_t, \varepsilon_l) = 0$ for all $t \neq l$. This does not preserve variances nor correlations.
2. *Masking by correlated noise addition.* Correlated noise addition also preserves means and additionally allows preservation of correlation coefficients. The difference with the previous method is that the covariance matrix of the errors is now proportional to the covariance matrix of the original data, i.e., $\varepsilon \sim N(0, \Sigma_\varepsilon)$, where $\Sigma_\varepsilon = \alpha\Sigma$.
3. *Masking by noise addition and linear transformation.* In Kim [1], a method is proposed that ensures by additional transformations that the sample covariance matrix of the masked attributes is an unbiased estimator for the covariance matrix of the original attributes.
4. *Masking by noise addition and nonlinear transformation.* An algorithm combining simple additive noise and nonlinear transformation is proposed in Sullivan [2]. The advantages of this proposal are that it can be applied to discrete attributes and that univariate distributions are preserved. Unfortunately, the application of this method is very time-consuming and requires expert knowledge on the data set and the algorithm.

See Brand [3] for more details on specific algorithms.

Cross-references

► Inference Control in Statistical Databases
► Microdata
► SDC Score

Recommended Reading

1. Kim J. J. A method for limiting disclosure in microdata based on random noise and transformation. In Proc. Section on Survey Research Methods, American Statistical Association. Alexandria VA, 1986, pp. 303–308.
2. Sullivan G.R. The Use of Added Error to Avoid Disclosure in Microdata Releases. PhD Thesis, Iowa State University, 1989.
3. Brand R. Microdata protection through noise addition. In Inference Control in Statistical Databases, J. Domingo-Ferrer (ed.). LNCS, Volume 2316. Springer, 2002, pp. 97–116.

Non-Clustering Index

► Secondary Index

Non-Dense Index

► Sparse Index

Nonidentifiability

► Anonymity

Nonlinear Magnification

► Distortion Techniques

Non-Metric Temporal Reasoning

► Qualitative Temporal Reasoning

Nonparametric Data Reduction Techniques

Rui Zhang
University of Melbourne, VIC, Australia

Definition

A nonparametric data reduction technique is a data reduction technique that does not assume any model for the data.

Key Points

Nonparametric data reduction (NDR) techniques is opposite to parametric data reduction (PDR) techniques. A PDR technique must assume a certain model for the data. Parameters of the model are determined before the data reduction is performed. A NDR technique does not assume any model and is applied to the data directly. The data reduction effectiveness of a PDR technique heavily depends on whether the model suits the data well. If well-suited, good accuracy as well as substantial data reduction can be achieved; otherwise, both cannot be achieved at the same time. A NDR technique yields more uniform effectiveness irrespective of the data, but it may not achieve as high data reduction as a well-suited PDR technique.

Popular NDR techniques include histograms, clustering and indexes. Histograms are used to approximate data distributions. An equidepth histogram can adjust itself to the data distribution and always gives good approximation of the data, no matter how the data are distributed. Clustering techniques also find the cluster centers automatically, irrespective of the data distribution. Although parameters are used in some clustering algorithms, such as the value of k in a k-means algorithm, k is a given parameter instead of a parameter determined by the actual data. Conceptually, indexes are similar to clustering and they also adjust themselves to the data distribution. No parameter needs to be determined from the data for indexes. A summary of data reduction techniques including NDR techniques can be found in [1].

Cross-references

► Clustering
► Data Deduction
► Histogram
► Parametric Data Reduction Techniques

Recommended Reading

1. Barbará D., DuMouchel W., Faloutsos C., Haas P.J., Hellerstein J.M., Ioannidis Y.E., Jagadish H.V., Johnson T., Ng R.T., Poosala V., Ross K.A., and Sevcik K.C. The New Jersey data reduction report. Q. Bull. IEEE TC on Data Engineering, 20(4):3–45, 1997.

Non-Perturbative Masking

► Non-Perturbative Masking Methods

Non-Perturbative Masking Methods

Josep Domingo-Ferrer
The Public University of Tarragona, Tarragona, Spain

Synonyms

Non-perturbative masking

Definition

Non-perturbative masking methods are SDC methods for microdata protection which do not alter data; rather, they produce partial suppressions or reductions of detail in the original dataset. Sampling, global recoding, top and bottom coding and local suppression are examples of non-perturbative masking methods.

Key Points

1. Sampling is a non-perturbative masking method for statistical disclosure control of microdata. Instead of publishing the original microdata file, what is published is a sample S of the original set of records. Sampling methods are suitable for categorical microdata, but for continuous microdata they should probably be combined with other masking methods. The reason is that sampling alone leaves a continuous attribute V_i unperturbed for all records in S. Thus, if attribute V_i is present in an external administrative public file, unique matches with the published sample are very likely: indeed, given a continuous attribute V_i and two respondents o_1 and o_2, it is

highly unlikely that V_i will take the same value for both o_1 and o_2 unless $o_1 = o_2$ (this is true even if V_i has been truncated to represent it digitally).

2. Global recoding or generalization is a masking method for statistical disclosure control of microdata. For a categorical attribute V_i, several categories are combined to form new (less specific) categories, thus resulting in a new V'_i with $|D(V'_i)| < |D(V_i)|$ where $|\cdot|$ is the cardinality operator. For a numerical attribute, global recoding means replacing V_i by another attribute V'_i which is a discretized version of V_i. In other words, a potentially infinite range $D(V_i)$ is mapped onto a finite range $D(V'_i)$. This technique is more appropriate for categorical microdata, where it helps disguise records with strange combinations of categorical attributes. Global recoding is used heavily by statistical offices. Global recoding is implemented in the μ-Argus package. In combination with local suppression, it can be used to achieve *k*-anonymity.
3. Top coding and bottom coding are special cases of the global recoding masking method for statistical disclosure control of microdata. Their operating principle is that top values (those above a certain threshold), respectively bottom values (those below a certain threshold), are lumped together to form a new category. Top and bottom coding can be used on attributes that can be ranked, that is, numerical or categorical ordinal.
4. Local suppression or blanking is a masking method for statistical disclosure control of microdata. Certain values of individual attributes are suppressed with the aim of increasing the set of records agreeing on a combination of key values. Local suppression is implemented in the *μ*-Argus package. Ways to combine local suppression and global recoding are discussed in DeWaal and Willenborg (1995). In fact, the combination of both methods can be used to attain *k-anonymity*. If a numerical attribute V_i is part of a set of key attributes, then each combination of key values is probably unique. Since it does not make sense to systematically suppress the values of V_i, it can be asserted that local suppression is rather oriented to categorical attributes.

Cross-references

- ▶ Inference Control in Statistical Databases
- ▶ *k*-Anonymity
- ▶ Microdata

Recommended Reading

1. DeWaal A.G. and Willenborg L.C.R.J. Global recodings and local suppressions in microdata sets. In Proc. Statistics Canada Symposium'95, 1995, pp. 121–132.
2. Hundepool A., Domingo-Ferrer J., Franconi L., Giessing S., Lenz R., Longhurst J., Schulte-Nordholt E., Seri G., and DeWolf P.-P. Handbook on Statistical Disclosure Control (Version 1.0). Eurostat (CENEX SDC Project Deliverable), 2006. http://neon.vb.cbs.nl/CENEX/
3. Hundepool A., Van de Wetering A., Ramaswamy R., Franconi F., Polettini S., Capobianchi A., DeWolf P.-P., Domingo-Ferrer J., Torra V., Brand R., and Giessing S. *μ*-ARGUS User's Manual Version 4.1 February 2007. http://neon.vb.cbs.nl/CASC.
4. Willenborg L. and DeWaal T. Elements of Statistical Disclosure Control. Springer-Verlag, New York, 2001.

Non-Pipelineable Operator

▶ Stop-&-Go Operator

Nonsequenced Semantics

Michael H. Böhlen[1], Christian S. Jensen[2], Richard T. Snodgrass[3]

[1]Free University of Bozen-Bolzano, Bozen-Bolzano, Italy

[2]Aalborg University, Aalborg, Denmark

[3]University of Arizona, Tucson, AZ, USA

Synonyms

Nontemporal semantics

Definition

Nonsequenced semantics guarantees that query language statements can reference and manipulate the timestamps that capture the valid and transaction time of data in temporal databases as regular attribute values, with no built-in temporal semantics being enforced by the query language.

Key Points

A temporal database generalizes a non-temporal database and associates one or more timestamps with database entities. Different authors have suggested temporal query languages that provide advanced support for formulating temporal statements. Results of these efforts include a variety of temporal extensions of SQL,

temporal algebras, and temporal logics that simplify the management of temporal data.

Languages with built-in temporal support are attractive because they offer convenient support for formulating a wide range of common temporal statements. The classical example of advanced built-in temporal support is *sequenced semantics*, which makes it possible to conveniently interpret a temporal database as a sequence of non-temporal databases. To achieve built-in temporal support, the timestamps are viewed as implicit attributes that are given special semantics.

Built-in temporal support, however, may also limit the expressiveness of the language when compared to the original non-temporal language where timestamps are explicit attributes. Nonsequenced semantics guarantees that statements can manipulate timestamps as regular attribute values with no built-in temporal semantics being enforced. This ensures that the expressiveness of the original language is preserved.

The availability of legacy statements with the standard non-temporal semantics is also important in the context of migration where users can be expected to be well-acquainted with the semantics of their non-temporal language. Nonsequenced semantics ensures that users are able to keep using the paradigm they are familiar with and to incrementally adopt the new features. Moreover, from a theoretical perspective, any variant of temporal logic, a well-understood language that only provides built-in temporal semantics, is strictly less expressive than a first order logic language with explicit references to time [1,4].

Each statement of the original language has the potential to either be evaluated with temporal or non-temporal semantics. For example, a count query can count the tuples at each time instant (this would be temporal, i.e., sequenced, semantics) or count the tuples actually stored in a relation instance (this would be non-temporal, i.e., nonsequenced, semantics).

To distinguish the two semantics, different approaches have been suggested. For instance, TempSQL distinguishes between so-called current and classical users. ATSQL and SQL/Temporal offer so-called statement modifiers that enable the users to choose between the two semantics at the granularity of statements. Below, statement modifiers are used for illustration. Specifically the ATSQL modifier `NSEQ VT` signals standard SQL semantics with full control over the timestamp attributes of a valid-time database.

The illustrations assume a database instance with three relations:

Employee

ID	Name	VTIME
1	Bob	5–8
3	Pam	4–12
4	Sarah	1–5

Salary

ID	Amt	VTIME
1	20	4–10
3	20	6–9
4	20	6–9

Bonus

ID	Amt	VTIME
1	20	1–6
1	20	7–12
3	20	1–12

and the following queries

```
NSEQ VT
   SELECT COUNT(*) FROM Bonus;

NSEQ VT
   SELECT E.ID
   FROM Employee AS E, Salary AS S
   WHERE VTIME(E) PRECEDES VTIME(S) AND
   E.ID = S.ID;
```

Both queries are nonsequenced, i.e., the valid time is treated as a regular attribute without any special processing going on. The first query determines the number of bonuses that have been paid. It returns the number of tuples in the `Bonus` relation, which is equal to three. Note that if the sequenced modifier was used (cf. *sequenced semantics*) then the time-varying count had been computed. With the given example, the count at each point in time would be two. The second query joins `Employee` and `Salary`. The join is not performed at each snapshot (cf. *sequenced semantics*). Instead it requires that the valid

time of Employee precedes the valid time of Salary. The result is a non-temporal table.

Nonsequenced statements offer no built-in temporal support, but instead offer complete control. This is akin to programming in assembly language, where one can do everything, but everything is hard to do. The query language must provide a set of functions and predicates for expressing temporal relationships (e.g., PRECEDES [2]) and performing manipulations and computations on timestamps (e.g., VTIME). The resulting new query-language constructs are relatively easy to implement because they only require changes at the level of built-in predicates and functions. Instead of using functions and predicates on timestamps, the use of temporal logic with temporal connectives has also been suggested.

Cross-references

- ▶ Allen's Relations
- ▶ Sequenced Semantics
- ▶ SQL-Based Temporal Query Languages
- ▶ Temporal Database
- ▶ Valid Time

Recommended Reading

1. Abiteboul S., Herr L., Van den Bussche J. Temporal versus first-order logic to query temporal databases. In Proc. 15th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems, 1996, pp. 49–57.
2. Allen J.F. Maintaining knowledge about temporal intervals. Commun. ACM, 16(11):832–843, 1983.
3. Böhlen M.H., Jensen C.S., and Snodgrass R.T. Temporal statement modifiers. ACM Trans. Database Syst., 25(4):48, December 2000.
4. Toman D. and Niwiński D. First-order queries over temporal databases inexpressible in temporal logic. In Advances in Database Technology, Proc. 5th International Conf. on Extending Database Technology, 1996, pp. 307–324.

Nontemporal Semantics

- ▶ Nonsequenced Semantics

Non-Uniform Distribution

- ▶ Data Skew

Normal Form ORA-SS Schema Diagrams

Gillian Dobbie[1], Tok Wang Ling[2]
[1]University of Auckland, New Zealand
[2]National University of Singapore, Singapore, Singapore

Synonyms

NF-SS; Normalizing ORA-SS diagrams

Definition

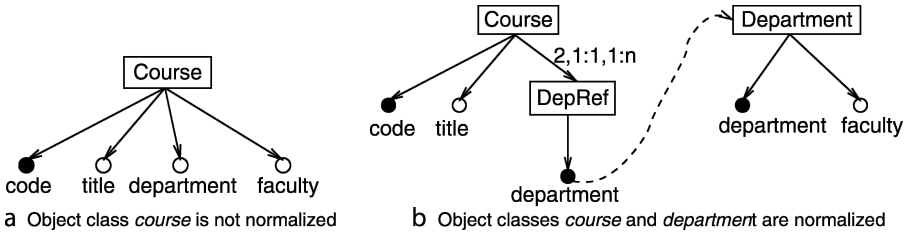
Normal forms have been defined for data models, such as the relational data model, the nested relational data model, the object-oriented data model and more recently, the semi-structured data model, to recognize (and remove) certain kinds of redundant data. A normal form that has been defined for semi-structured data, based on the ORA-SS data model, is described.

Key Points

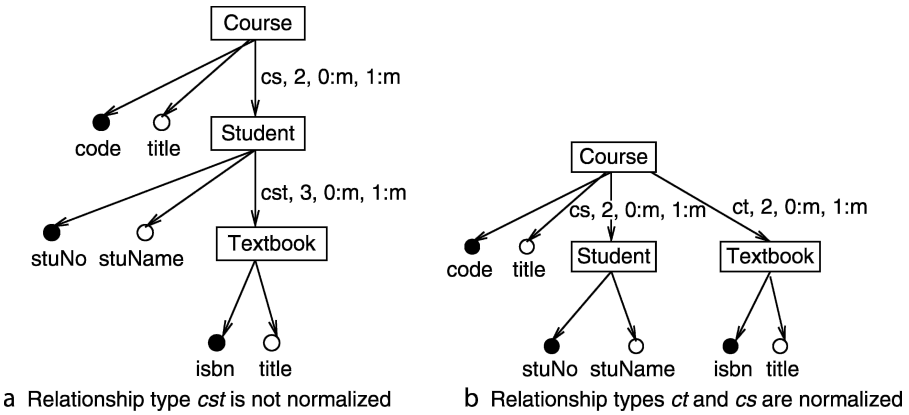
The definition of a normal form for ORA-SS (Object-Relationship-Attribute data model for Semi-structured data) diagrams is based on the definition of a normal form for the nested relational data model as described by Ling and Yan [1], recognizing the similarity between the nesting relationship in the nested relational data model and hierarchies in the semi-structured data model.

The definition for NF ORA-SS (normal form for the ORA-SS data model) can be broken into three main parts. The first part ensures that every object class is normalized. The second part ensures that every relationship type is normalized. The third part ensures that there is no data that can be derived from other data in the diagram. Examples of ORA-SS schemas that are not in NF ORA-SS are shown below, along with schemas that are in NF ORA-SS that capture the same information.

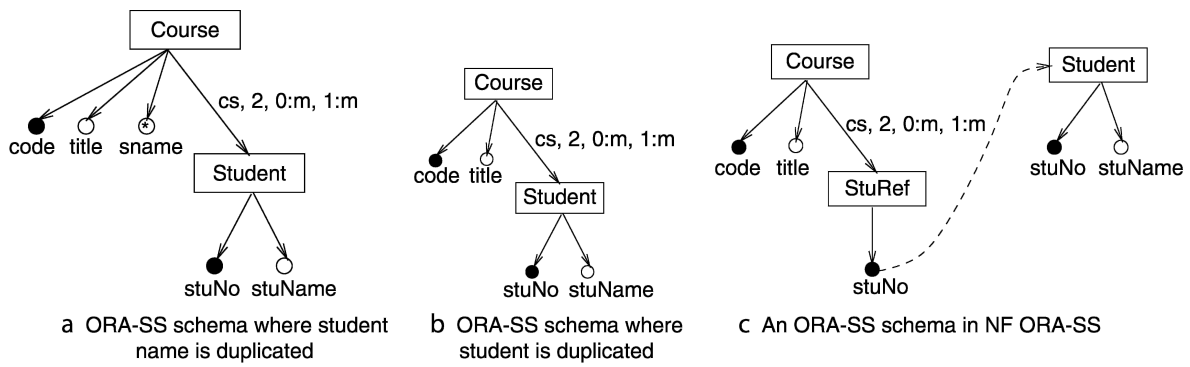
Figure 1a represents an object class *Course*. The attributes of course are *code, title, department* and *faculty*. This object class is not normalized since *code* determines *title* and *department*, but *department* determines *faculty*. What this means is that a department belongs to only one faculty, and this information needs to be stored once rather than once for every course. Figure 1b has object classes *Course*, *DepRef* and *Department*. Object class *DepRef* has a reference to object class *Department*, so the information about the faculty a department belongs to is stored only once. The object classes in Fig. 1b are normalized.



Normal Form ORA-SS Schema Diagrams. Figure 1. Example of object class normal form.



Normal Form ORA-SS Schema Diagrams. Figure 2. Example of relationship type normal form.



Normal Form ORA-SS Schema Diagrams. Figure 3. Example of ORA-SS normal form (NF ORA-SS).

Figure 2a shows a relationship type *cst*, which is a ternary relationship type among the object classes *Course*, *Student*, and *Textbook*. Intuitively it makes sense because courses have students and textbooks. However, students and textbooks are independent of each other. A course has a certain set of students, irrespective of the textbooks for the course, and a course has a certain set of textbooks irrespective of the students taking the course. In the schema in Fig. 2a, the textbooks for a given course are repeated for each student taking the course. Figure 2b has two relationship types *cs* and *ct*, between *Course* and *Student*, and *Course* and *Textbook*, respectively. Both of these relationship types are normalized. However, the object classes *Student* and *Textbook* in Fig. 2b are still not normalized.

Figure 3a shows two object classes *Course* and *Student*. Object class *Course* has attributes *code*, *title* and *sname*. Attribute *sname* is a multivalued attribute and has the names of the students taking the course. This information is repeated in the attribute *stuName* that belongs to object class *Student*, so the ORA-SS schema in Fig. 3a is not in NF ORA-SS. If the student

names are stored only in the attribute *stuName* of object class *Student*, as shown in Fig. 3b that redundancy is removed. Of course the many-to-many relationship type leads to the students name being repeated in every course the student takes. Using a reference to represent this relationship removes this redundancy and the resulting ORA-SS diagram in Fig. 3c is in NF ORA-SS.

Cross-references

► Normal Forms and Normalization
► Object Relationship Attribute Data Model for Semi-structured Data
► Semi-structured Database Design

Recommended Reading

1. Ling T.W. and Yan L.L. NF-NR: a practical normal form for nested relations. J. Syst. Int., 4:309–340.

Normal Forms and Normalization

Marcelo Arenas
Pontifical Catholic University of Chile, Santiago, Chile

Definition

A normal form defines a condition over a set of data dependencies, or semantic constraints, that has been specified as part of a database schema. This condition is used to check whether the design of the database has some desirable properties (for example, the database does not store redundant information), and if this is not the case, then it can also be used to convert the poorly designed database into an equivalent well-designed one.

Historical Background

Information is one of the most – if not the most – valuable assets of a company. Therefore, organizations need tools to allow them to structure, query and analyze their data, and, in particular, they need tools providing simple and fast access to their information.

During the last 30 years, relational databases have become the most popular computer application for storing and analyzing information. The simplicity and elegance of the relational model, where information is stored just as tables, has largely contributed to this success.

To use a relational database, a company first has to think of its data as organized in tables. How easy it is for a user to understand this organization and use the database depends on the design of these relations. If tables are not carefully selected, users can spend too much time executing simple operations, or may not be able to extract the desired information.

Since the beginnings of the relational model, it was clear for the research community that the process of designing a database is a nontrivial and time-consuming task. Even for simple application domains, there are many possible ways of storing the data of interest. Soon the difficulties in designing a database became clear for practitioners, and the design problem was recognized as one of the fundamental problems for the relational technology.

During the 70s and 80s, a lot of effort was put into developing methodologies to aid in the process of deciding how to store data in a relational database. The most prominent approaches developed at that time – which today are a standard part of the relational technology – were the entity-relationship and the normalization approaches. In the former approach, a diagram is used to specify the objects of an application domain and the relationships between them. The *schema* of the relational database, i.e., the set of tables and column names, is then automatically generated from the diagram. In the normalization approach, an already designed relational database is given as input, together with some semantic information provided by a user in the form of relationships between different parts of the database, called *data dependencies*. This semantic information is then used to check whether the design has some desirable properties, and if this is not the case, it can also be used to convert the poor design into an equivalent well-designed database.

Foundations

In the relational model, a database is viewed as a collection of relations or tables. For instance, a relational database storing information about courses in a university is shown in Fig. 1. This relation consists of a time-varying part, the data about courses, and a part considered to be time independent, the *schema* of the relation, which is given by the name of the relation (*Course*) and the names of the attributes of this relation.

Usually, the information contained in a database satisfies some *semantic* restrictions. For example, in the

Course	Number	Title	Section	Room
	CSC258	Computer organization	1	LP266
	CSC258	Computer organization	2	GB258
	CSC258	Computer organization	3	LM161
	CSC258	Computer organization	3	GB248
	CSC434	Data management systems	1	GB248

Normal Forms and Normalization. Figure. 1. Example of a relational database.

CourseInfo	Number	Title
	CSC258	Computer organization
	CSC434	Data management systems

CourseTerm	Number	Section	Room
	CSC258	1	LP266
	CSC258	2	GB258
	CSC258	3	LM161
	CSC258	3	GB248
	CSC434	1	GB248

Normal Forms and Normalization. Figure. 2. Example of a normalized relational database.

relation *Course* shown in Fig. 1, it is expected that only one title is associated with each course number. These restrictions are called *data dependencies*, and they are expressed by using suitable languages. For example, the previous constraint corresponds to a functional dependency, which is an expression of the form $X \rightarrow Y$, where X and Y are sets of attributes. A relation satisfies $X \rightarrow Y$ if for every pair of tuples t_1,t_2 in it, if t_1 and t_2 have the same values on X, then they have the same values on Y. Thus, for example, the relation shown in Fig. 1 satisfies functional dependency *Number* $\rightarrow$ *Title* since each course has only one title.

In [2], Codd showed that a database containing functional dependencies may exhibit some anomalies when the information is updated. For example, consider again the relational database shown in Fig. 1, which includes functional dependency *Number* $\rightarrow$ *Title*. This database is prone to three different types of anomalies. First, if the name of the course with number CSC258 is changed to Computer Organization I, then four distinct cells need to be updated. If any of them is not updated, then the information in the database becomes inconsistent. This anomaly was called an *update anomaly* by Codd [2], and it arises because the instance is storing redundant information. Second, if the information is updated because a new semester is starting, and the course with number CSC434 is not given in that semester, then the last tuple of the instance is deleted and no information about CSC434 appears in the updated instance. This has the additional effect of deleting the title of the course, which will be the same the next time that CSC434 is offered. This anomaly was called a *deletion anomaly* by Codd [2], and it arises because the relation is storing information that is not directly related; the sections of a course vary from one term to another while its title is likely not to be changed from one semester to the next one. This can also lead to *insertion anomalies* [2]; if a new course (CSC336, Numerical Methods) is created, then it cannot be added to the database until at least one section and one room is assigned to the course.

To avoid updates anomalies, Codd introduced three increasingly restrictive *normal forms* [2,3], which specify some syntactic properties that the set of functional dependencies in a database must satisfy. The most restrictive among them is known today as Boyce-Codd Normal Form (BCNF). Informally, a database schema S including a set Σ of functional dependencies is in BCNF, if there is no set of attributes $X \cup \{A\} \cup \{B\}$ such that A, $B \notin X$, $X \rightarrow A$ holds in S but $X \rightarrow B$ does not hold in S, that is, if there are no attributes in S with different levels of association between them according to Σ (If $X \rightarrow A$ holds in S but $X \rightarrow B$ does not hold in S, then for every value of X, there exists only one value of A in the database but possibly many values of B.). For example, the database shown in Fig. 1 is not in BCNF since *Number* $\rightarrow$ *Title* holds in this database, while *Number* $\rightarrow$ *Section* does not hold.

In [2], Codd also introduced the first *normalization algorithm*, that is, a procedure that takes as input a relational schema that includes some data

dependencies and does not satisfy some particular normal form, and produces a new schema that conforms to this normal form. For example, if S is the relational schema *Course* (*Number*, *Title*, *Section*, *Room*) and Σ is the set of functional dependencies {*Number* $\rightarrow$*Title*}, then the standard normalization algorithm for BCNF [1] produces a new schema where *Course* is split into two tables: *CourseInfo*(*Number*, *Title*) and *CourseTerm*(*Number*, *Section*, *Room*). In Fig. 2, it is shown how the information in the initial database in Fig. 1 is stored under the new schema. It should be noticed that the new schema is not prone to any of the anomalies mentioned at the beginning of this section:

- If the name of the course with number CSC258 is changed to Computer Organization I, then only one cell needs to be updated.
- If the information is updated because a new semester is starting, and the course with number CSC434 is not given in that semester, then the last tuple of relation *CourseTerm* is deleted. This does not have the additional effect of deleting the title of the course (this information is kept in the relation *CourseInfo*).
- If a new course (CSC336, Numerical Methods) is created, then it can be added to the database even if no section has been created for this course (tuple (CSC336, Numerical Methods) is included only in the relation *CourseInfo*).

A normalization algorithm takes as input a relational schema and generates a database schema in some particular normal form. It is desirable that these two are as similar as possible, that is, they should contain the same data and the same semantic information. These properties have been called *information losslessness* and *dependency preservation* in the literature, respectively.

Let S_1,S_2 be two database schemas. Intuitively, two instances I_1 of S_1 and I_2 of S_2 contain the same information if it is possible to retrieve the same information from them, that is, for every query Q_1 over I_1 there exists a query Q_2 over I_2 such that $Q_1(I_1) = Q_2(I_2)$, and vice versa. To formalize this notion, a query language has to be chosen. If this query language is relational algebra, then this notion is captured by the notion of calculously dominance introduced by Hull [7]. Schema S_2 *dominates* S_1 *calculously* if there exist relational algebra expressions Q over S_1 and Q' over S_2 satisfying the following property: For every instance I of S_1, there exists an instance I' of S_2 such that $Q(I) = I'$ and $Q'(I') = I$. Thus, every query Q_1 over I can be transformed into an equivalent query $Q_2 = Q_1 \circ Q'$ over I', since $Q_2(I') = Q_1(Q'(I')) = Q_1(I)$, and, analogously, every query Q_2 over I' can be transformed into an equivalent query $Q_1 = Q_2 \circ Q$ over I, since $Q_1(I) = Q_2(Q(I)) = Q_2(I')$.

Normalization algorithms try to achieve the goal of information losslessness; if any of them transforms a database schema S into a database schema S', then S' should dominate S calculously. The standard normalization algorithm for BCNF uses only the projection operator to transform a schema [1] and, thus, calculously dominance is defined in terms of this operator and its inverse, the join operator. More precisely, the normalization algorithm mentioned in this section takes as input a relation R and a set of functional dependencies Σ, and uses the projection operator to transform it into a database schema in BCNF that is composed by some relations R_1,...,R_n. Then R_1,...,R_n is a *lossless decomposition* of R if for every instance I of R there is an instance I' of R_1,...,R_n such that:

- For every $i \in \{1,...,n\}$, it holds that $I'_i = \pi_{U_i}(I)$, where I'_i is the R_i-relation of I' and U_i is the set of attribute of R_i, and
- $I = I'_1 \bowtie I'_2 \bowtie ... \bowtie I'_n$, where each I'_i is the R_i-relation of I'.

That is, every instance I of S can be transformed into an instance I' of S' by using the projection operator, and I can be reconstructed from I' by using the join operator. For example, the relation *CourseInfo* in Fig. 2 can be obtained by projecting the relation *Course* in Fig. 1 over the set of attributes {*Number*,*Title*}, while the relation *CourseTerm* in Fig. 2 can be obtained by projecting the relation *Course* in Fig. 1 over the set of attributes {*Number*, *Section*, *Room*}. Moreover, the relation *Course* in Fig. 1 can be obtained by joining relations *CourseInfo* and *CourseTerm* in Fig. 2. Given that this holds for every instance I of the initial schema *Course* (*Number*,*Title*,*Section*,*Room*), the new schema *CourseInfo* (*Number*,*Title*), *CourseTerm* (*Number*, *Section*, *Room*) is said to be a lossless decomposition of the initial one.

Normalization algorithms also try to achieve the goal of dependency preservation; if any of them transforms a database schema S including a set Σ of data dependencies, into a database schema S' including a set Σ' of data dependencies, then Σ should be equivalent to Σ' (no semantic information is lost). In the running example,

the standard normalization algorithm for BCNF produces a new schema *CourseInfo*(*Number*,*Title*) and *CourseTerm*(*Number*, *Section*, *Room*), and also includes dependency *Number* → *Title* in the relation *CourseInfo*. Thus, the new schema is a dependency preserving decomposition of the initial one.

The normalization approach was proposed in the early 70s by Codd [2]. Since then, many researchers have studied the normalization problem for relational databases and other data models, and today it is possible to find normal forms for many different types of data dependencies: 3NF [2] and BCNF [3] for functional dependencies, 4NF [4] for multivalued dependencies, PJ/NF [5] and 5NFR [8] for join dependencies, and DK/NF [6] for general constraints. These normal forms, together with normalization algorithms for converting a poorly designed database into a well-designed database, can be found today in every database textbook.

Key Applications

Normal forms and normalization algorithms are essential to schema design, redundancy elimination, update anomaly prevention and efficient storage.

Cross-references

- ▶ Boyce-Codd Normal Form
- ▶ Fourth Normal Form
- ▶ Second Normal Form (2NF)
- ▶ Third Normal Form

Recommended Reading

1. Abiteboul S., Hull R., and Vianu V. Foundations of Databases. Addison-Wesley, Reading, MA, USA, 1995.
2. Codd E.F. Further normalization of the data base relational model. In Data base systems. Prentice-Hall, Englewood Cliffs, NJ, USA, 1972, pp. 33–64.
3. Codd E.F. Recent investigations in relational data base systems. In IFIP Congress. North-Holland, Amsterdam, 1974, pp. 1017–1021.
4. Fagin R. Multivalued dependencies and a new normal form for relational databases. ACM Trans. Database Syst., 2(3):262–278, 1977.
5. Fagin R. Normal forms and relational database operators. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1979, pp. 153–160.
6. Fagin R. A normal form for relational databases that is based on domians and keys. ACM Trans. Database Syst., 6(3):387–415, 1981.
7. Hull R. Relative information capacity of simple relational database schemata. SIAM J. Comput., 15(3):856–886, 1986.
8. Vincent M. A corrected 5NF definition for relational database design. Theor. Comput. Sci., 185(2):379–391, 1997.

Normalized Discounted Cumulated Gain (nDCG)

▶ Discounted Cumulated Gain

Normalizing ORA-SS Diagrams

▶ Normal Form ORA-SS Schema Diagrams

Now in Temporal Databases

Curtis E. Dyreson[1], Christian S. Jensen[2], Richard T. Snodgrass[3]
[1]Utah State University, Logan, Utah, USA
[2]Aalborg University, Aalborg, Denmark
[3]University of Arizona, Tucson, AZ, USA

Synonyms

Current time; Current date; Current timestamp; Until changed

Definition

The word *now* is a noun in the English language that means "at the present time." This notion appears in databases in three guises. The first use of *now* is as a function within queries, views, assertions, etc. For instance, in SQL, `CURRENT_DATE` within queries, etc., returns the current date as an SQL `DATE` value; `CURRENT_TIME` and `CURRENT_TIMESTAMP` are also available. These constructs are nullary functions.

In the context of a transaction that contains more than one occurrence of these functions, the issue of which time value(s) to return when these functions are invoked becomes important. When having these functions return the same (or consistent) value, it becomes a challenge to select this time and to synchronize it with the serialization time of the transaction containing the query.

The second use is as a database variable used extensively in temporal data model proposals, primarily as timestamp values associated with tuples or attribute values in temporal database instances. As an example, within transaction-time databases, the stop time of data that has not been logically deleted and thus is current is termed "until changed." A challenging aspect of supporting this notion of *now* has been to contend

with instances that contain this variable when defining the semantics of queries and modification and when supporting queries and updates efficiently, e.g., with the aid of indices.

The third use of *now* is as a database variable with a specified offset (a "now-relative value") that can be stored within an implicit timestamp or as the value of an explicit attribute. Challenges include the specification of precise semantics for database instances that contain these variables and the indexing of such instances.

Historical Background

Time variables such as *now* are of interest and indeed are quite useful in databases, including databases managed by a conventional DBMS, that record time-varying information, the validity of which often depends on the current-time value. Such databases may be found in many application areas, such as banking, inventory management, and medical and personnel records.

The SQL nullary functions `CURRENT_DATE`, `CURRENT_TIME`, and `CURRENT_TIMESTAMP` have been present since SQL's precursor, SEQUEL 2 [1]. The transaction stop time of *until changed* has been present since the initial definition of transaction-time databases (e.g., in the early work of Ben-Zvi [2]).

The notion of *now* and the surprisingly subtle concerns with this ostensibly simple concept permeate the literature, and yield quite interesting solutions.

Foundations

The three notions of "now" in temporal databases are explored in more detail below.

SQL Nullary Functions

The following example illustrates the uses and limitations of *now*, specifically the SQL `CURRENT_DATE`, `CURRENT_TIME`, and `CURRENT_TIMESTAMP`, in conventional databases. Banking applications record when account balances for customers are valid. Examine the relation `AccountBalance` with attributes `AccountNumber`, `Balance`, `FromDate`, and `ToDate`. To determine the balance of account 12345 on January 1, 2007, one could use a simple SQL query.

```
SELECT Balance
FROM AccountBalance
WHERE Account = 12345
  AND FromDate <= DATE '2007-01-01'
  AND DATE '2007-01-01' < ToDate
```

To determine the balance of that account *today*, the nullary function `CURRENT_DATE` is available.

```
SELECT Balance
FROM AccountBalance
WHERE Account = 12345
  AND FromDate <= CURRENT_DATE
  AND CURRENT_DATE < ToDate
```

Interestingly, in the SQL standard, the semantics of the function are implementation-dependent, which opens it to various interpretations: "If an SQL-statement generally contains more than one reference to one or more <datetime value functions>s then all such references are effectively evaluated simultaneously. The time of evaluation of the <datetime value function > during the execution of the SQL-statement is implementation-dependent." (This is from the SQL:1999 standard.) So an implementation is afforded considerable freedom in choosing a definition of "current," including perhaps when the statement was presented to the system, or perhaps when the database was first defined.

Transactions take time to complete. If a transaction needs to insert or modify many tuples, it may take minutes. However, from the point of view of the user, the transaction is atomic (all or nothing) and serializable (placed in a total ordering). Ideally "current" should mean "when the transaction executed," that is, the instantaneous time the transaction logically executed, consistent with the serialization order.

Say a customer opens an account and deposits $200. This transaction results in a tuple being inserted into the `AccountBalance` table. The transaction started on January 14 at 11:49 P.M. and committed at 12:07 A.M. (that is, starting before midnight and completing afterwards). The tuple was inserted on 11:52 P.M. Another user also created a second account with an initial balance of $500 in a transaction that started on January 14 at 11:51 P.M. and committed on 11:59 P.M., inserting a tuple into the `AccountBalance` relation at 11:52 P.M. If the system uses the transaction start time, the following two tuples will be in the relation.

According to these two tuples, the sum of balances on January 14 is $700. But note that though both transactions began on January 14, only the second transaction committed by midnight (also note that the second transaction is earlier in the serialization order since it commits first). Hence, the actual aggregate balance on January 14 was never $700: the balance started at $0 (assuming there were initially no other

tuples), then changed to $500. Only on January 15 did it increase to $700.

Suppose that instead of using the time at which the entire transaction began, the time of the actual insert statement (e.g., for the first transaction, January 14, 11:52 P.M.) is used; then the same problem occurs.

What is desired is for a time returned by CURRENT_DATE to be consistent with the serialization order and with the commit time, which unfortunately is not known when CURRENT_DATE is executed. Lomet et al. [3] showed how to utilize the lock manager to assign a commit time in such a way that it is consistent with the serialization order as well as with dates assigned as values to prior CURRENT_DATE invocations. Specifically, each use of CURRENT_DATE, etc. defines or narrows a *possible period* during which the commit time of the transaction must reside. For CURRENT_DATE this period is the particular 24 hours of the day returned by this function. Read-write conflicts with other transactions further narrow the possible period. For example, if a particular transaction reads values written by another transaction, the transaction time of the reader must be later than that of the writer. At commit time, it is attractive to assign the earliest instant in the possible period to the transaction. Alternatively, if the possible period ever becomes empty, the transaction must be aborted. Lomet et al. outline important optimizations to render such calculations efficient.

Now in End or Stop Columns

The above discussion concerned how to determine what to store in the begin time of a tuple (e.g., in the FromDate column for the two example tuples in the table displayed above) and how to make this time consistent with that returned by CURRENT_DATE. We now consider what to store in the end time of a tuple (e.g., in the ToDate column). The validity of a tuple then starts when a deposit is made and extends until the current time, assuming no update transactions are committed. Thus, on January 16, the balance is valid from January 14 until January 16; on January 17, the balance is valid from January 14 until January 17; etc.

It is impractical to update the database each day (or millisecond) to correctly reflect the valid time of the balance. A more promising approach is to store a variable, such as *now*, in the ToDate field of a tuple, to indicate that the time when a balance is valid depends on the current time. In the example, it would be recorded on January 14 that the customer's balance of $200 is valid from January 14 through *now*. The CURRENT_DATE construct used in the above query cannot be stored in a column of an SQL table. All major commercial DBMSs have similar constructs, and impose this same restriction. The user is forced instead to store a specific time, which is cumbersome and inaccurate (Clifford et al. explain these difficulties in great detail [4]).

The solution is to allow one or more free, current-time variables to be stored in the database. Chief among these current-time variables is "*now*" (e.g., [5]), but a variety of other symbols have been used, including "–" [2], "∞" [6], "@" [7], and "*until-changed*" [8]. Such stored variables have advantages at both the semantic and implementation levels. They are expressive and space efficient and avoid the need for updates at every moment in time.

As an example, consider a variable database with the tuple ⟨*Jane*, *Assistant*, [*June 1*, *now*]⟩, with *now* being a variable. The query "List the faculty on June 15," evaluated on June 27, results in {⟨*Jane*, *Assistant*⟩}.

Now-Relative Values

A now-relative instant generalizes and adds flexibility to the variable *now* by allowing an offset from this variable to be specified. Now-relative instants can be used to more accurately record the knowledge of Jane's employment. For example, it may be that hiring changes are recorded in the database only 3 days after they take effect. Assuming that Jane was hired on June 1, the definite knowledge of her employment is accurately captured in the tuple ⟨*Jane*, *Assistant*, [*June 1*, *now* − 3 *days*]⟩ . This tuple states that Jane was an Assistant Professor from June 1 and until three days ago, but it contains no information about her employment as of, e.g., yesterday.

A now-relative instant thus includes a displacement, which is a signed duration, also termed a span, from *now*. In the example given above, the displacement is minus 3 days. Now-relative variables can be extended to be indeterminate [4], as can regular instants and the variable *now*.

AccountNumber	Balance	FromDate	ToDate
121345	200	2007-01-14	...
543121	500	2007-01-14	...

The semantics of now variables has been formalized with an *extensionalization* that maps from a *variable database level* containing variables as values to an *extensional database level*. The extensional database exhibits three key differences when compared to the variable database level. First, no variables are allowed – the extensional level is fully ground. Second, timestamps are instants rather than intervals. Third, an extensional tuple has one additional temporal attribute, called a *reference time attribute*, which may be thought of as representing the time at which a meaning was given to the temporal variables in the original tuple. Whereas the variable-database level offers a convenient representation that end-users can understand and that is amenable to implementation, the mathematical simplicity of the extensional level supports a rigorous treatment of temporal databases in terms of first order logic.

When a variable database is queried, an additional problem surfaces: what to do when a variable is encountered during query evaluation. In the course of evaluating a user-level query, e.g., written in some dialect of SQL, it is common to transform it into an internal algebraic form that is suitable for subsequent rule or cost-based query optimization. As the query processor and optimizer are among the most complex components of a database management system, it is attractive if the added functionality of current-time-related timestamps necessitates only minimal changes to these components.

Perhaps the simplest approach to supporting querying is that, when a timestamp that contains a variable is used during query processing (e.g., in a test for overlap with another timestamp), a ground version of that timestamp is created and used instead. With this approach, only a new component that substitutes variable timestamps with ground timestamps has to be added, while existing components remain unchanged.

Put differently, a *bind* operator can be added to the set of operators already present. This operator is then utilized when user-level queries are mapped to the internal representation. The operator accepts any tuple with variables. It substitutes a ground value for each variable and thus returns a ground (but still variable-level) tuple. Intuitively, the *bind* operator sets the perspective of the observer, i.e., it sets the reference time. Existing query languages generally assume that the temporal perspective of an observer querying a database is the time when the observer initiates the query.

Torp et al. has shown how to implement such variables within a database system [9].

With *now* as a database variable, the temporal extent of a tuple becomes a non-constant function of time. As most indices assume that the extents being indexed are constant in-between updates, the indexing of the temporal extents of now-relative data poses new challenges.

For now-relative transaction-time data, one may index all data that is not now-relative (i.e., has a fixed end time) in one index, e.g., an R-tree, and all data that is now-relative (i.e., the end time is *now*) in another index where only the start time is indexed.

For bitemporal data, this approach can be generalized to one where tuples are distributed among four R-trees. The idea is again to overcome the inabilities of indices to cope with continuously evolving regions, by applying transformations to the growing bitemporal extents that render them stationary and thus amenable to indexing. Growing regions come in three shapes, each with its own transformation. These transformations are accompanied by matching query transformations [10].

In another approach, the R-tree is extended to store *now* for both valid and transaction time in the index. The resulting index, termed the GR-tree, thus accommodates bitemporal regions and uses minimum bounding regions that can be either static or growing and either rectangles or stair shapes. This approach has been extended to accommodate bitemporal data that also have spatial extents [11].

Key Applications

The notion of *now* as a nullary function representing the current time is common in database applications. For relational database management, SQL offers `CURRENT_DATE`, `CURRENT_TIME`, and `CURRENT_TIMESTAMP` functions that return an appropriate SQL time value. In other kinds of database management systems, such as native XML database management systems, similar constructs can be found. For example, XQuery has a `fn:current-time()` function that returns a value of XML Schema's `xs:time` type. XQuery also has a `fn:current-date()` function.

Less common in existing database applications are the other notions of now: as a variable stored in a database to represent the ever-changing current time or as a time related to, but displaced from, the current time.

Future Directions

The convenience of using now variables poses challenges to the designers of database query languages.

The user-defined time types available in SQL-92 can be extended to store now-relative variables as values in columns. The TSQL2, language [12] does so, and also supports those variables for valid and transaction time. In TSQL2, the "bind" operation is implicit; `NOBIND` is provided to store variables in the database. However, such variables have yet to be supported by commercial DBMSs. It may also be expected that at least one of the three uses of *now* will re-emerge as part of a temporal extension of XQuery or a language associated with the Semantic Web.

The impact of stored variables on database storage structures and access methods is a relatively unexplored area. Such stored variables may present optimization opportunities. For example, if the optimizer knows (through attribute statistics) that a large proportion of tuples has a "to" time of *now*, it may then decide that a sort-merge temporal join will be less effective. Finally, new kinds of variables, such as *here* for spatial and spatio-temporal databases, might offer an interesting extension of the framework discussed here.

Cross-references

- ▶ Bi-Temporal Indexing
- ▶ Bitemporal interval
- ▶ Supporting Transaction Time Databases
- ▶ Temporal Concepts in Philosophy
- ▶ Temporal Query Languages
- ▶ Temporal Strata
- ▶ Temporal XML
- ▶ Time Period
- ▶ Transaction Time
- ▶ TSQL2
- ▶ Valid Time

Recommended Reading

1. Ben-Zvi J. The Time Relational Model. Ph.D. Dissertation, University of California, Los Angeles, 1982.
2. Bliujūtė R., Jensen C.S., Šaltenis S., and Slivinskas G. Lightweight indexing of bitemporal data. In Proc. 12th Int. Conf. on Scientific and Statistical Database Management, 2000, pp. 125–138.
3. Chamberlin D.D., Astraham M.M., Eswaran K.P., Griffiths P.P., Lorie R.A., Mehl J.W., Reisner P., and Wade B.W., SEQUEL 2: a unified approach to data definition, manipulation, and control. IBM J. Res. Dev. 20(6):560–575, 1976.
4. Finger M. Handling database updates in two-dimensional temporal logic. J. Appl. Non-Classical Logics, 2(2):201–224, 1992.
5. Clifford J., Dyreson C.E., Isakowitz T., Jensen C.S., and Snodgrass R.T. On the semantics of "now." ACM Trans. Database Syst., 22(2):171–214, June 1997.
6. Clifford J. and Tansel A.U. On an algebra for historical relational databases: two views. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1985, pp. 247–265.
7. Lomet D., Snodgrass R.T., and Jensen C.S. Exploiting the lock manager for timestamping. In Proc. Int. Conf. on Database Eng. and Applications, 2005, pp. 357–368.
8. Lorentzos N.A. and Johnson R.G. Extending relational algebra to manipulate temporal data. Inf. Syst., 13(3):289–296, 1988.
9. Montague R. Formal Philosophy: Selected Papers of Richard Montague. Yale University Press, New Haven, 1974.
10. Šaltenis S., and Jensen C.S. Indexing of now-relative spatio-bitemporal data. VLDB J., 11(1):1–16, August 2002.
11. Snodgrass R.T. The temporal query language TQuel. ACM Trans. Database Syst., 12(2):247–298, June 1988.
12. Snodgrass R.T. (ed.). The TSQL2 Temporal Query Language. Kluwer, Norwell, MA, USA, 1995.
13. Torp K., Jensen C.S., and Snodgrass R.T. Modification semantics in now-relative databases. Inf. Syst., 29(78):653–683, 2004.
14. Wiederhold G., Jajodia S., and Litwin W. Integrating temporal data in a heterogeneous environment. In Temporal Databases: Theory, Design, and Implementation, Chap. 22, A. Tansel, J. Clifford, S.K. Gadia, S. Jajodia, A. Segev, R.T. Snodgrass (eds.). Benjamin/Cummings, Redwood City, CA, USA, 1993, pp. 563–579.

n-Tier Architecture

- ▶ Multi-Tier Architecture

Null Values

Leopoldo Bertossi
Carleton University, Ottawa, ON, Canada

Definition

Null values are used to represent *uncertain data values* in a database instance.

Key Points

Since the beginning of the relational data model, *null values* have been investigated, with the intention of capturing and representing data values that are uncertain. Depending on the intuitions and cases of uncertainty, different kinds of null values have been proposed, e.g., they may represent information that is withheld, inapplicable, missing, unknown, etc. Thus, in principle, it could be possible to find in a hypothetical database diverse classes of null values, and also several null values of the same class. However, in

commercial relational DBMSs and in the SQL Standard, only a single constant, NULL, is used to represent the missing values.

Many semantic problems appear when null values are integrated with the rest of the relational data model, which essentially follows the semantics of predicate logic. Among them, (i) the interpretation of nulls values (for a particular intuition); (ii) the meaning of relational operations when applied to both null values and certain data values; and (iii) the characterization of consistency of databases containing null values.

Different formal semantics for null values have been proposed. A common and well-studied semantic for *incomplete databases* uses null values to represent unknown or missing values. Each null value in the database represents a whole set of possible values from the underlying data domain. The combination of concrete values that null values might take generates a class of alternative instances containing certain values. This *possible worlds semantics* makes true whatever is true in every alternative instance. However, the usage of null values in the SQL Standard and commercial DBMS still lacks a clear and complete formal semantic.

Cross-references

► Incomplete Information

Recommended Reading

1. Grahne G. The Problem of Incomplete Information in Relational Databases. LNCS 554, Springer-Verlag, Secaucus, NJ, USA, 1991.
2. Levene M. and Loizou G. A Guided Tour of Relational Databases and Beyond, Chap. 5. Springer, London, UK, 1999.
3. Van der Meyden, R. Logical approaches to incomplete information: a survey. In Logics for Databases and Information Systems, J. Chomicki, G. Saake (eds.). Kluwer, Boston, MA, USA, 1998, pp. 307–356.

Numeric Association Rules

► Quantitative Association Rules

Numerical Fact

► Measure

Nymity

► Pseudonymity