D2I

Integrazione, Warehousing e Mining di sorgenti eterogenee

Programma di ricerca (cofinanziato dal MURST, esercizio 2000)

Methods for clustering, meta–querying, approximate and similarity search (Studio di metodi di mining per: clustering, ricerche approssimate, metaquerying)

Paolo Ciaccia, Giovambattista Ianni, Stefano Lodi,
Luigi Palopoli, Marco Patella, Claudio Sartori

D3.R3 **30 novembre 2001**

Sommario

In this report we propose a set of data mining methods and algorithms for metaquerying, fully dynamic clustering and approximate similarity queries. We discuss the combined and data complexity of metaquerying both in the general case and for subclasses of queries and its implementation. We propose fully dynamic clustering algorithms based on dissimilarity transformation and dynamic graph algorithms and discuss their accuracy and efficiency. Finally we extend the PAC approach to approximate similarity queries to the case of range queries and general k-NN queries and devise sequential algorithms to solve PAC queries deriving cost formulae which express the quality of result/search time tradeoff. Moreover, we propose index based algorithms to solve PAC queries and an optimal schedule for accessing index nodes during a k-NN search.

Tema	Tema 3: Data Mining
Codice	D3.R3
Data	30 novembre 2001
Tipo di prodotto	Rapporto tecnico
Numero di pagine	48
Unità responsabile	BO
Unità coinvolte	BO, CS
Autore da contattare	Claudio Sartori Dipartimento di Elettronica, Informatica e Sistemistica Università degli Studi di Bologna Viale Risorgimento, 2, 40136 Bologna, Italia csartori@deis.unibo.it

Methods for clustering, meta–querying, approximate and similarity search
(Studio di metodi di mining per: clustering, ricerche approssimate, metaquerying)

Paolo Ciaccia, Giovambattista Ianni, Stefano Lodi,
Luigi Palopoli, Marco Patella, Claudio Sartori

30 novembre 2001

Abstract

In this report we propose a set of data mining methods and algorithms for metaquerying, fully dynamic clustering and approximate similarity queries. We discuss the combined and data complexity of metaquerying both in the general case and for subclasses of queries and its implementation. We propose fully dynamic clustering algorithms based on dissimilarity transformation and dynamic graph algorithms and discuss their accuracy and efficiency. Finally we extend the PAC approach to approximate similarity queries to the case of range queries and general k-NN queries and devise sequential algorithms to solve PAC queries deriving cost formulae which express the quality of result/search time tradeoff. Moreover, we propose index based algorithms to solve PAC queries and an optimal schedule for accessing index nodes during a k-NN search.

Contents

1 Introduction

Data mining is an active research field whose goal is to design methods and algorithms to explore and analyze large bodies of data. In this report we propose a set of data mining methods and algorithms for metaquerying, fully dynamic clustering and approximate similarity queries. We briefly review the basics of the techniques as well as the research problems investigated in this report. A detailed review can be found in [12].

The goal of metaquerying is to find classes of patterns from a relational database, linking information from several tables. For instance, a metaquery has the form of a logical rule implication between literal $T \leftarrow L_1, \ldots, L_m$, where T and L_i are literal schemes which can be instantiated, or answered, by substituting the predicate variables with predicate (i.e. relation) names from the database. Note that unlike many other mining tools, patterns discovered using metaqueries can link information from several tables in databases. Moreover, these patterns are all relational, while most machine-learning systems can only learn propositional patterns and work on a single relation.

Clustering algorithms find groups, or clusters, in data, according to some notion of similarity. Ideally, the application of a clustering method yields maximal similarity between members of the same group and to minimal similarity between members of different groups. More recently, cluster analysis has emerged as an important data mining technique. In this environment, large repositories of data subject to frequent changes can be analyzed more efficiently by algorithms which recompute the clustering for the updated data from the previous clustering and the set of updates.

Similarity queries are an exploratory technique originally proposed in multimedia databases, where querying is usually performed iteratively through refinement. Moreover, similarity queries are often executed within clustering algorithms. In these applications, not only the accuracy of the result is important, but also the execution cost of the query. Since similarity queries can often be expensive, approximate methods providing the user with a quality/time tradeoff often enable the user to achieve dramatic improvements in execution time when an exact answer is not needed, provided a tangible cost/error trade-off can be guaranteed.

The goal of this report is to present the results of investigations in metaquerying, clustering, and approximate similarity queries, which we summarize below.

Metaquerying. In part I the semantics of metaquerying is defined and its complexity is analyzed. Concerning combined complexity, the general problem is intractable unless $P = NP$. However, a subset of metaquerying problems defined by acyclic metaqueries is in $LOGCFL$ a class of highly parallelizable problems. Data complexity is TC^0 in the general case and in AC^0 for a set of interesting subcases.

Clustering. In part II fully dynamic clustering algorithms are proposed. The algorithms are derived from known static algorithms based on dissimilarity transformation, by decomposing the update stage into two phases: A first phase where the transformation result is updated, and a second phase in which the clustering is updated by means of fully dynamic graph algorithms.

Similarity queries. In part III, the PAC (probably approximately correct) approach is extended for solving similarity queries with an arbitrary number of neighbours, i.e., k-NN queries, with $k > 1$. Sequential algorithms for approximate queries are devised and cost formulae expressing the quality of result/search time tradeoff. Finally, provably correct index-based algorithms for solving PAC queries and an optimal schedule for accessing index nodes are presented.

Part I

Metaquerying (*Giovanbattista Ianni and Luigi Palopoli*)

2 Introduction

Companies and organizations often posses information resources and databases containing a vast amount of data waiting to prove useful one day. The expanding datamining research area is supposed to provide tools for the discovery of valuable knowledge in these huge data resources [24].

Metaquerying [42] is a promising approach for datamining in relational and deductive databases. Metaqueries serve as a generic description of a class of patterns a user is willing to discover. Unlike many other mining tools, patterns discovered using metaqueries can link information from several tables in databases. These patterns are all relational, while most machine-learning systems can only learn propositional patterns and work on a single relation. Metaqueries can be specified by human experts or alternatively, they can be automatically generated from the database schema.

Intuitively, a metaquery has the form

$$T \leftarrow L_1, ..., L_m \tag{1}$$

where T and L_i are literal schemes $Q(Y_1, ..., Y_n)$ and Q is either an ordinary predicate name or a predicate variable. In this latter case, $Q(Y_1, ..., Y_n)$ can be instantiated to an atom with predicate symbol denoting a relation in the database. An answer to a metaquery is an (ordinary) rule obtained by consistently substituting second order predicates with relation names.

Shen *et al.* [40] are, to best of our knowledge, the first who have presented a framework that uses metaqueries to integrate inductive learning methods with deductive database technology.

As an example (taken from [42]), let P,Q, and R be predicate variables and **DB** be a database, then the metaquery

$$R(X,Z) \leftarrow P(X,Y), Q(Y,Z).$$

specifies that the patterns to be discovered are relationships of the form

$$r(X,Z) \leftarrow p(X,Y), q(Y,Z).$$

where p, q, and r are relations from **DB**. For instance, for an appropriate database **DB**, one possible result of this metaquery could be a rule:

$$speaks(X,Z) \leftarrow citizen(X,Y),\ language(Y,Z). \tag{2}$$

A rule which serves as an answer to a metaquery is usually accompanied by indices, that indicate its "plausibility degree". In [9], for example, each rule in the answer is supplied with *support* and *confidence*. The *support* indicates how frequently the body of the rule is satisfied, and the *confidence* measures what fraction of the tuples that satisfy the body, also satisfy the head. A confidence of 0.93, for instance, for the rule (2) means that out of all pairs (X, Z) that satisfy the body of the rule, 93% also satisfy the head. Admissibility thresholds for support and confidence is usually provided by the user. Support and confidence have been also defined for other datamining techniques, such as association rules [3].

Similar to the case of association rules, the plausibility indices are used with two major purposes:

1. to avoid presenting negligible information to the user,

2. to cut off the search space by early detection of plausibility values.

Formal definition of metaqueries and indices are given in the next section.

Metaquerying was implemented in some datamining systems [19, 18, 42] and was arguably very useful in knowledge discovery [26, 39, 47, 42]. However, despite the practical success of metaqueries, their theoretical foundations appear to be quite unclear. A careful reading of research manuscripts on the subject reveals that the semantics is not always well defined. In addition, except a very preliminary *NP*-hardness result [9], a thorough analysis of the involved complexities was never performed.

This report provides a study of some of the theoretical aspects about metaquerying. First, in Section 3, we provide formal definitions for the syntax and semantics of metaqueries. We define several types of metaqueries and indices that encompass, as far as we know, all variants defined in the literature.

In Section 4 we study both the combined complexity and the data complexity of all the metaqueries variants that we define. We show that, under the combined complexity measure, metaquerying is intractable in general (unless $P{=}NP$), but we are able to single out a tractable interesting metaquerying case (whose combined complexity is *LOGCFL*-complete). The tractable class is a subset of metaquerying problems which we call acyclic metaqueries. Acyclicity of metaqueries is a generalization of the analogous concept defined for conjunctive queries [8].
As for the data complexity of metaquerying, we prove that, in general, this is in TC^0, but lies within AC^0 in some interesting cases.

In Section 5 we discuss algorithms to implement metaquerying in a reasonably efficient manner.

Finally, in section 6, we draw some conclusions and report a summary table containing all the complexity results we prove throughout.

3 Metaquerying

3.1 Syntax and semantics

Let U be a countable domain of constants. A database **DB** is $(D, R_1, ..., R_n)$ where $D \subseteq U$ is finite, and each R_i is a relation of arity $a(R_i)$ such that $R_i \subseteq D^{a(R_i)}$.

As stated above, a metaquery **MQ** is a second-order template describing a pattern to be discovered [42]. Such a template has the form

$$T \leftarrow L_1, ..., L_m \tag{3}$$

where T and L_i are *literal schemes*. Each literal scheme T or L_i has the form $Q(Y_1, ..., Y_n)$ where Q is either a *predicate (second order) variable* or a relation symbol, and each Y_j $(1 \leq j \leq n)$ is an ordinary (first order) variable. If Q is a predicate variable, then $Q(Y_1, ..., Y_n)$ is called a *relation pattern* of arity n, otherwise it is called an *atom* of arity n. The right-hand-side $L_1, ..., L_m$ is called the *body* of the metaquery, while T is called the *head* of the metaquery. A metaquery is called *pure* if each two relation patterns with the same predicate variable has the same arity.

Intuitively, given a database instance **DB**, answering a metaquery **MQ** on **DB** amounts to finding all substitutions σ of relation patterns appearing in **MQ** by atoms having as predicate names relations in **DB**, such that the Horn rule $\sigma(\mathbf{MQ})$ (obtained by applying σ to **MQ**) encodes a dependency between the atoms in its head and body, which rule holds in **DB** with a certain degree of plausibility. The plausibility is defined in terms of *indices* which we will formally define shortly.

Let **MQ** be a metaquery and **DB** a database. Let $pv(\mathbf{MQ})$, $ls(\mathbf{MQ})$, and $rep(\mathbf{MQ})$ denote the set of predicate variables, the set of literal schemes, and the set of relation patterns occurring in $\mathbf{MQ}$, respectively (note that $rep(\mathbf{MQ}) \subseteq ls(\mathbf{MQ})$). Moreover, let $rel(\mathbf{DB})$ denote the set of relation names of $\mathbf{DB}$ and $ato(\mathbf{DB})$ denote the set of all the atoms of the form $p(T_1, \ldots, T_k)$ where $p \in rel(\mathbf{DB})$, k is the arity of p, and each T_i is an ordinary variable.

Semantics is defined via types of *metaquery instantiations*: an instantiation type specifies how relation patterns can be instantiated, turning a metaquery to an ordinary Horn rule over the given database. Next, we define three different types of metaquery instantiations, that we call *type-0, type-1* and *type-2*, respectively. In the literature, metaquerying semantics has not been always precisely defined, even if a kind of type-2 semantics is usually assumed (see, e.g.,[40]).

Definition 3.1 Let $\mathbf{MQ}$ be a metaquery and $\mathbf{DB}$ a database. An *instantiation* (on $\mathbf{MQ}$ and $\mathbf{DB}$) is a mapping $\sigma : rep(\mathbf{MQ}) \rightarrow ato(\mathbf{DB})$, whose restriction $\sigma' : pv(\mathbf{MQ}) \rightarrow rel(\mathbf{DB})$ is functional.

The condition above says that predicate names of $\mathbf{MQ}$ are *consistently* substituted with relation names from $\mathbf{DB}$.

In general, given a set S of literal schemes and an instantiation σ, $\sigma(S)$ will denote a set of atoms generated from S by applying σ to each relation pattern belonging to S. Let $\mathbf{DB}$ be a database, and $\mathbf{MQ}$ be a metaquery. In the following, $\sigma(\mathbf{MQ})$ will denote the Horn rule generated by applying σ to each relation pattern of $\mathbf{MQ}$. Whenever we want to remark that σ is from $rep(\mathbf{MQ})$ to $ato(\mathbf{DB})$, we will employ the notation $\sigma_{\mathbf{DB}}^{\mathbf{MQ}}$.

Definition 3.2 Let $\mathbf{MQ}$ be a pure metaquery and $\mathbf{DB}$ a database. An instantiation $\sigma_{\mathbf{DB}}^{\mathbf{MQ}}$ is *type-0* if for any relation pattern L and atom A, $\sigma(L) = A$ implies that L and A have the same list of arguments.

That is, under type-0 semantics, each predicate variable is always matched to a relation with the same arity and ordinary variables are left untouched. As an example, consider the database $\mathbf{DB}_1$ shown in Figure 1 and the metaquery

$$R(X, Z) \leftarrow P(X, Y), Q(Y, Z) \tag{4}$$

A possible type-0 instantiation for $\mathbf{MQ}$ is

$$\sigma = \{\langle R(X, Z), \mathrm{UsPT}(X, Z)\rangle, \langle P(X, Y), \mathrm{UsCa}(X, Y)\rangle, \langle Q(Y, Z), \mathrm{CaTe}(Y, Z)\rangle\}$$

which yields the following Horn rule when applied to $\mathbf{MQ}$:

$$\mathrm{UsPT}(X, Z) \leftarrow \mathrm{UsCa}(X, Y), \mathrm{CaTe}(Y, Z)$$

Definition 3.3 Let $\mathbf{MQ}$ be a pure metaquery and $\mathbf{DB}$ a database. An instantiation $\sigma_{\mathbf{DB}}^{\mathbf{MQ}}$ is *type-1* if for any relation pattern L and atom A, $\sigma(L) = A$ implies that the arguments of A are obtained from arguments of L by permutation.

UsCa

User	Carrier
John K.	Omnitel
John K.	Tim
Anastasia A.	Omnitel

CaTe

Carrier	Technology
Tim	ETACS
Tim	GSM 900
Tim	GSM 1800
Omnitel	GSM 900
Omnitel	GSM 1800
Wind	GSM 1800

UsPt

User	Phone Type
John K.	GSM 900
John K.	GSM 1800
Anastasia A.	GSM 900

Figure 1: The relations UsCa, CaTe, and UsPT of $\mathbf{DB}_1$

UsPt

User	Phone Type	Model
John K.	GSM 900	Nokia 6150
John K.	GSM 1800	Nokia 6150
Anastasia A.	GSM 900	Bosch 607

Figure 2: The new relation *UsPT*

With type-1 instantiations, variable ordering within relation patterns "does not matter". As an example, under this semantics, from metaquery (4) and $\mathbf{DB}_1$, among others, the following Horn rules could be obtained:

$$\mathrm{UsPT}(X, Z) \leftarrow \mathrm{UsCa}(X, Y), \mathrm{CaTe}(Y, Z)$$
$$\mathrm{UsPT}(X, Z) \leftarrow \mathrm{UsCa}(Y, X), \mathrm{CaTe}(Y, Z)$$

The third type of instantiation takes a step further by allowing a relation pattern of arity k to be matched with an atom of arity k', with $k' \geq k$, padding "remaining" arguments to free variables:

Definition 3.4 Let $\mathbf{MQ}$ be a metaquery and $\mathbf{DB}$ a database. An instantiation $\sigma_{\mathbf{DB}}^{\mathbf{MQ}}$ is *type-2* if for any relation pattern L and atom A, $\sigma(L) = A$ implies the following:

- the arity k' of A is greater-than or equal-to the arity of L;
- k of the arguments of A coincide with the k arguments of L, possibly occurring in different positions;
- the remaining $k' - k$ arguments of A are variables not occurring elsewhere in the instantiated rule.

With type-2 instantiations we can express interesting patterns ignoring how many extra attributes a physical relation may have. Should the relation *UsPT* be defined with an additional attribute, as in Figure 2, the metaquery (4) can be instantiated, using a type-2 instantiation, to

$$\mathrm{UsPT}(X, Z, T) \leftarrow \mathrm{UsCa}(Y, X), \mathrm{CaTe}(Y, Z)$$

Note that a type-0 instantiation is a type-1 instantiation where the chosen permutation of relations's attributes is the identity, whereas a type-1 instantiation is a type-2 instantiation, where the arity of the atoms matches the arity of the relation patterns they are substituted for.

Note, moreover, that type-2 instantiations may apply to any metaquery, while type-0 and type-1 instantiations require pure metaqueries.

3.2 Plausibility Indices

In datamining applications, one is generally interested in discovering plausible patterns of data that represent significant knowledge in the analyzed data sets. In other words, it is not typically required for a discovered rule to entirely characterize the given data set, but a significant subset thereof. This idea is embedded in the usage of *plausibility indices.* In the literature, several definitions of plausibility indices are found, like, e.g., *support*, *confidence*, *base* and *strength* [3, 9, 40] (in fact, "support" is similar to "base" and "confidence" is similar to "strength" [9]). In the analysis that follows, we shall use support, confidence and another useful index that we call *cover.* We next provide formal definitions of the indices that we use.

For any set F, let $|F|$ denote its size. In the following, we assume the reader is familiar with basic notions regarding relational algebra and Datalog (see [51] for an excellent source of material about these subjects). Unless otherwise specified, a predicate name p will denote its corresponding underlying relation as well. In particular, an atom $p(\mathbf{X})$ will denote the corresponding database relation, where the list of arguments $\mathbf{X}$ is used, as in Datalog, to positionally refer to p's columns. For a set of atoms R, $att(R)$ is the set of all the variables (attribute) of all the atoms in R, and $\mathbf{J}(R)$ is the natural join of all the relations corresponding to atoms in R.

Definition 3.5 A plausibility index, or index in short, is a function which maps a database instance $\mathbf{DB}$ and a Horn rule $h(\mathbf{X}) \leftarrow b_1(\mathbf{X}_1), ..., b_n(\mathbf{X}_n)$ (defined over $\mathbf{DB}$) to a rational number within $[0, 1]$.

Given an index I, a database $\mathbf{DB}$, and a rule r defined over $\mathbf{DB}$, unless confusion may arise, we will employ the notation $I(r)$ as a shortcut for $I(r, \mathbf{DB})$.

Definition 3.6 Let R and S be two sets of atoms. Then the fraction of R in S, denoted $R \uparrow S$, is

$$\frac{|\pi_{att(R)}(\mathbf{J}(R) \bowtie \mathbf{J}(S))|}{|\mathbf{J}(R)|}.$$

In particular, whenever $|\pi_{att(R)}(\mathbf{J}(R) \bowtie \mathbf{J}(S))| = 0$, $R \uparrow S$ is defined equal to 0.

Definition 3.7 Let $\mathbf{DB}$ be a database, and r be a Horn rule defined on $\mathbf{DB}$. Let $h(r)$ and $b(r)$ denote the sets of atoms occurring in the head and in the body of r, respectively. Then

- the *confidence* of r on $\mathbf{DB}$ is $cnf(r) = b(r) \uparrow h(r)$,
- the *cover* of r on $\mathbf{DB}$ is $cvr(r) = h(r) \uparrow b(r)$,
- the *support* of r on $\mathbf{DB}$ is $sup(r) = \max_{a \in b(r)}(\{a\} \uparrow b(r))$.

Intuitively, $sup(r)$ measures how much the body (or part of it) of an instantiation contains satisfying tuples. When an instantiation scores an high support, a pattern search algorithm may conclude that it is worth to further consider such an instantiation, because there is at least one relation with a high percentage of its tuples satisfying the instantiated body.

When an instantiation scores a high confidence, we can conclude that a high percentage of the assignments which satisfy the body also satisfy the head relation. Hence, confidence tells how much valid the rule is over the given database. Given a rule r, the indices $cnf(r)$ and $sup(r)$ are equivalent to *confidence* and *support* defined in [9]. In [9], there is also a discussion on the motivations underlying various definition of support and confidence.

To conclude, cover tells which is the percentage of implied tuples belonging to the head relation. This latter index that we define here, is useful in those application where it is necessary to decide if it is worth to store the head relation or to compute it in the form of a reasonably matching view, such as in reengineering applications.

As an example, consider the metaquery

$$I(X) \leftarrow O(X)$$

the (type-2) instantiation (on $\mathbf{DB}_1$)

$$\mathrm{UsCa}(X, Z) \leftarrow \mathrm{UsPt}(X, H)$$

scores a cover of 1, which tells that the content of the first attribute of UsPt completely encodes the content of the first attribute of UsCa.

In the sequel, the set of plausibility indices $\{cnf, cvr, sup\}$ will be denoted $\mathbf{I}$.

4 Complexity of metaquerying

In this section we present several results regarding the complexity of metaquerying. We assume the reader is familiar with basic notions regarding complexity classes and, in particular, the definition of the polynomial hierarchy [49]. Next, we recall the definition of the classes AC^0, TC^0, $\#AC^0$, $GapAC^0$, PAC^0, and $LOGCFL$, which will be used in the sequel.

4.1 Preliminaries

Definition 4.1 Let $\mathcal{C}$ be a complexity class, then $NP^{\mathcal{C}}$ is the class of problems decidable in NP using an oracle solving a problem in $\mathcal{C}$ (see [54]).

Definition 4.2 A *conjunctive query* is a set of atoms $\{r_1(\mathbf{X}_1), \ldots, r_n(\mathbf{X}_n)\}$, where $\mathbf{X}_1, \ldots, \mathbf{X}_n$ are lists of variables and/or constants. Let **DB** be a database instance. The problem of satisfying a conjunctive query (Boolean Conjunctive Query satisfaction problem, or BCQ) is the problem of deciding if there exists a substitution ρ for variables in $X_i, 1 \leq i \leq n$ such that, for each i, $1 \leq i \leq n$, $r_i(\rho(\mathbf{X}_i)) \in \mathbf{DB}$. The set $\{r_i(\rho(\mathbf{X}_i)), 1 \leq i \leq n\} \equiv \{r_i(\mathbf{x}_i), 1 \leq i \leq n\}$ is called *ground instance* of $\{r_1(\mathbf{X}_1), \ldots, r_n(\mathbf{X}_n)\}$.

The BCQ problem was proved to be NP-complete in [13].

Definition 4.3 MAJORITY gates are unbounded fan-in gates (with binary input and output) that output 1 if and only if more than half of their inputs are non-zero.

Definition 4.4 A family $\{C_i\}$ of boolean circuits, such that C_i accepts strings of size i, is uniform if there exists a Turing machine T which, on input i produces the circuit C_i. $\{C_i\}$ is said to be *logspace uniform* if T carries out its work using $O(\log i)$ space. Define AC^0 (resp. TC^0) as the class of decision problems solved by logspace uniform families of circuits of polynomial size and constant depth, with AND, OR, and NOT (resp. MAJORITY and NOT) gates with unbounded fan-in [2, 7, 46].

In the following, unless otherwise specified, we will assume to deal with logspace uniform families of circuits. The class TC^0 is of special interest, since it characterizes the computational complexity of such important operations as multiplication, division, and sorting, as well as being a computational model for neural nets. TC^0 was also characterized as being the class of languages that arises in several ways from counting the number of accepting subtrees of AC^0 circuits [2].

Definition 4.5 For any $k > 0$, $\#AC^0_k$ is the class of functions $f : \{0,1\}^* \to \mathbf{N}$ computed by depth k, polynomial size uniform families of circuits with $+,\times$-gates (the usual arithmetic sum and product in $\mathbf{N}$) having unbounded fan-in, where each value incoming into the circuit can be either constant (where the allowed constant values are 1 and 0) or being an input value in the form x_i or $1 - x_i$ (where the allowed input values are 1 and 0). Let $\#AC^0 = \bigcup_{k>0} \#AC^0_k$ [2].

Thus, $\#AC^0$ circuits accept the values 1 and 0 as inputs, but they are considered as natural numbers.

Definition 4.6 $GapAC^0$ is the class of all functions $f : \{0,1\}^* \to \mathbf{N}$ that can be expressed as the difference of two functions in $\#AC^0$ [2, 4].

Definition 4.7 PAC^0 is the class of languages $\{A \mid \exists f \in GapAC^0, x \in A \iff f(x) > 0\}$ [2].

Proposition 4.8 Under logspace uniformity, $PAC^0 = TC^0$ [2, 4].

Definition 4.9 *LOGCFL* coincides with the class of decision problems logspace-reducible to a context free language [46].

Definition 4.10 Let $H(\mathbf{x},\mathbf{y})$ be a formula with two free variable lists $\mathbf{x}$ and $\mathbf{y}$ and let k be a natural number. The *counting quantifier* C is defined as

$$\overset{k}{\underset{\mathbf{y}}{\mathrm{C}}}\, H(\mathbf{x},\mathbf{y}) \equiv |\{\mathbf{y} : H(\mathbf{x},\mathbf{y}) \text{ is true}\}| \geq k$$

[57].

As an example

$$\overset{k}{\underset{\mathbf{y}}{\mathrm{C}}}\, [\mathbf{y} \text{ encodes an hamiltonian path on the graph } \mathbf{x}]$$

is true iff the number of Hamiltonian paths on $\mathbf{x}$ is at least k. A polynomial-bounded version of the counting quantifier is defined as follows.

Definition 4.11 Let $\mathcal{L}$ be a class of languages. A language L' belongs to the class $C\mathcal{L}$ iff there exists a language $L \in \mathcal{L}$, a polynomial time computable function f, and a polynomial p such that

$$x \in L' \leftrightarrow \overset{f(x)}{\underset{\substack{y \\ |y| \leq p(|x|)}}{\mathrm{C}}} \; [(x,y) \in L]$$

The polynomial-bounded counting quantifier embeds the polynomial-bounded existential quantifier as follows: A language L' belongs to the class $\exists\mathcal{L}$ iff there exists a language $L \in \mathcal{L}$, and a polynomial p such that

$$x \in L' \leftrightarrow \overset{1}{\underset{\substack{y \\ |y| \leq p(|x|)}}{\mathrm{C}}} \; [(x,y) \in L]$$

Definition 4.12 The language $\exists\mathbf{C}$-SAT, includes all tuples $\langle F(\mathbf{x}_1,\mathbf{x}_2), m\rangle$ (where $\mathbf{x}_1$ and $\mathbf{x}_2$ are lists of propositional variables) such that F is a boolean expression in conjunctive normal form and

$$\overset{1}{\underset{\mathbf{x}_1}{\mathrm{C}}}\,\overset{m}{\underset{\mathbf{x}_2}{\mathrm{C}}}\, F(\mathbf{x}_1,\mathbf{x}_2) \text{ is true}$$

Theorem 4.13 *$\exists$**C**-SAT and $\exists$**C**-3SAT are $\exists$CP-complete [57].*

Let *PP* be the set of languages $\mathcal{L}$ recognized by a nondeterministic Turing Machine M as follows: $x \in \mathcal{L}$ iff at least one half of the computation paths of $M(x)$ ends in an accepting state. It is known that $\exists CP$ coincides with NP^{PP} [50]. $\exists CP$ belongs to the so called polynomial counting hierarchy: it is contained in *PSPACE*, and (may be properly) contains Σ_2^P and *CP* [57].

Definition 4.14 A *counting Turing Machine (CTM) CM* is an ordinary nondeterministic Turing Machine whose output, on the input x, is the number of accepting computations of $CM(x)$. The time complexity of a CTM is defined to be $f(n)$ if the longest accepting computation associated with the set of all inputs of size n takes $f(n)$ steps (see [38]).

Definition 4.15 Define $\#P$ to be the set of all functions that are computable by polynomial-time CTMs.

Definition 4.16 A problem X is $\#P$-hard if there are polynomial-time Turing reductions from all problems in $\#P$ to X. If X is $\#P$-hard and $X \in \#P$, then X is said to be $\#P$-complete (see [53]).

Definition 4.17 A *parsimonious transformation* [38] is a polynomial transformation f from a problem X to a problem Y such that, if $\#(X, x)$ is defined to be the number of solutions that instance x has in problem X, then $\#(X, x) = \#(Y, f(x))$.

4.2 Complexity Measures

As for the case of ordinary queries, complexity of metaqueries can be defined according to two complexity measures: *combined complexity* and *data complexity* [55]. Complexity measures are defined next.

Let $T \in \{0, 1, 2\}$ be an instantiation type and let I be a plausibility index. Let **DB** denote a database instance, **MQ** a metaquery, and k a rational threshold value such that $0 \leq k < 1$. Then:

1. The *combined* complexity of $\langle \mathbf{DB}, \mathbf{MQ}, I, k, T \rangle$ is the complexity, measured in the size of **DB**, **MQ** and k, of deciding if there exists a type-T instantiation $\sigma_{\mathbf{DB}}^{\mathbf{MQ}}$ such that $I(\sigma(\mathbf{MQ})) > k$.

2. Assuming that a database schema $DS = (D, R_1, ..., R_n)$ has been fixed in advance, the *data* complexity of the metaquery problem $\langle \mathbf{DB}, \mathbf{MQ}, I, k, T \rangle$ is the complexity, measured in the size of **DB**, of deciding if there exists a type-T instantiation $\sigma_{\mathbf{DB}}^{\mathbf{MQ}}$ such that $I(\sigma(\mathbf{MQ})) > k$, where **DB** is a database with schema DS.

In the literature it is usually assumed that one looks for rules that satisfy some metaquery and have two plausibility indices, usually *support* and *confidence*, above some given threshold [9, 40]. Here we split the metaquery problem so that it relates to one plausibility index at a time. The rationale is that, this allows us to single out more precisely complexity sources and, at the same time, complexity measures for problems involving more than one index can be obtained fairly easily from metaquerying problems involving only one index.

4.3 Combined Complexity

It immediately follows from the proof of Theorem 2.2 in [9] that the combined complexities of $\langle \mathbf{DB}, \mathbf{MQ}, \mathit{sup}, 0, 1 \rangle$ and $\langle \mathbf{DB}, \mathbf{MQ}, \mathit{cnf}, 0, 1 \rangle$ are NP-hard. In this section, we generalize these results by stating the combined complexities of metaquerying for various indices and instantiation types.

A first characterization of the problem is given by the following result.

Proposition 4.18 *Let I be an index, and $\mathcal{C}$ be the complexity of deciding the following question: "Given a Horn rule r, a database* **DB** *and a finitely represented rational value $k \in [0, 1)$, is $I(r) > k$ over* **DB***?". Then, for any instantiation type T, the complexity of $\langle \mathbf{DB}, \mathbf{MQ}, I, k, T \rangle$ is in $NP^{\mathcal{C}}$.*

In many cases, we are able to state tighter bounds than those implied by Proposition 4.18. We begin by analyzing the cases when the threshold value k is set to 0. This kind of metaquerying problems have a limited practical usefulness. However, they provide interesting lower bounds for the complexity of metaquerying problems involving thresholds larger than 0.

Definition 4.19 Let $\mathbf{DB}$ be a database, and $r \equiv b_1(\mathbf{X}_1) \leftarrow b_2(\mathbf{X}_2), ..., b_n(\mathbf{X}_n)$ be a Horn rule defined over $\mathbf{DB}$. Let I be an index, and let $A_r = \{b_1(\mathbf{X}_1) \leftarrow b_2(\mathbf{X}_2), ..., b_n(\mathbf{X}_n)\}$ the set of atoms occurring in r. A *certifying set* for I is a set $S \subseteq A_r$ such that there exists a ground instance s of S which is satisfied in $\mathbf{DB}$ iff $I(r) > 0$.

Proposition 4.20 *Consider a rule* $r \equiv h(\mathbf{X}) \leftarrow b_1(\mathbf{X}_1), ..., b_n(\mathbf{X}_n)$*. Then,* $S_{\mathrm{cvr}} \equiv \{h(\mathbf{X}), b_1(\mathbf{X}_1), ..., b_n(\mathbf{X}_n)\}$, $S_{\mathrm{sup}} \equiv \{b_1(\mathbf{X}_1), ..., b_n(\mathbf{X}_n)\}$ *and* $S_{\mathrm{cnf}} \equiv S_{\mathrm{cvr}}$ *are certifying sets for* cvr, sup *and* cnf, *respectively.*

Theorem 4.21 *Let* $I \in \mathbf{I}$ *be a plausibility index. The combined complexity of* $\langle \mathbf{DB}, \mathbf{MQ}, I, 0, T\rangle$ *is NP-complete, for any instantiation type* $T \in \{0, 1, 2\}$.

Next, we consider the combined complexity of metaquerying when the fixed threshold k is such that $0 \leq k < 1$.

Proposition 4.22 *The combined complexity of* $\langle \mathbf{DB}, \mathbf{MQ}, I, k, T\rangle$ *is NP-hard for any plausibility index* $I \in \mathbf{I}$.

Theorem 4.23 *The combined complexity of* $\langle \mathbf{DB}, \mathbf{MQ}, I, k, T\rangle$*, with* $0 \leq k < 1$*,* $T \in \{0, 1, 2\}$ *and* $I \in \{\mathrm{cvr}, \mathrm{sup}\}$ *is in NP.*

As for confidence, it is not difficult to show a *PSPACE* upper bound on its combined complexity and we can deduce, from Theorem 4.21, that the problem is *NP*-hard as well.

A more in-depth analysis shows that the issue of measuring confidence of a given metaquery instantiation turns out to be actually more complex than to measuring the other indices. This is due to the need of computing the *exact* count of tuples satisfying the body of an instantiation, whereas for other indices this is not required. In fact, the problem of deciding if confidence exceeds a given threshold on some instantiation of a metaquery is related to problems where the question concerns *counting* the exact number of solutions of a given instance, as we show below.

Given a boolean formula F (resp. a boolean formula F with at most three literals per clause) in conjunctive normal form, let $\#SAT$ (resp. $\#3SAT$) be the problem of finding the number of satisfying assignments for F.

Theorem 4.24 $\#SAT$ *and* $\#3SAT$ *are* $\#P$*-complete [48, 52].*

Many problems are known to be $\#P$-complete [52, 20], some of which are the counting counterparts of *NP*-complete problems. However, it is also known that counting solutions for some of the problems in *P*is as hard as counting solutions of *NP*-complete problems [53]. A useful tool to show $\#P$ completeness for counting versions of *NP*-complete problems is that of *parsimonious transformations* (defined in Definition 4.17).

Proposition 4.25 *Let* Q *be a conjunctive query* $\{r_1(\mathbf{X}_1), \ldots, r_n(\mathbf{X}_n)\}$*, where* $\mathbf{X}_1, \ldots, \mathbf{X}_n$ *are list of variables and/or constants. Let* $\#BCQ$ *be the problem of counting how many substitutions* ρ *for variables* $\mathbf{X}_i$*,* $1 \leq i \leq n$*, are there such that, for each* i*,* $1 \leq i \leq n$*,* $\rho(r_i(\mathbf{X}_i)) \in \mathbf{DB}$ *(cf. Definition 4.2).* $\#BCQ$ *is* $\#P$*-complete.*

Theorem 4.26 *The combined complexity of* $\langle \mathbf{DB}, \mathbf{MQ}, \mathrm{cnf}, k, T\rangle$ *is in* NP^{PP}.

Theorem 4.27 *The combined complexity of* $\langle \mathbf{DB}, \mathbf{MQ}, \mathrm{cnf}, k, 0\rangle$ *is* NP^{PP} *complete.*

Theorem 4.28 *The combined complexity of* $\langle \mathbf{DB}, \mathbf{MQ}, \mathrm{cnf}, k, 1\rangle$ *and* $\langle \mathbf{DB}, \mathbf{MQ}, \mathrm{cnf}, k, 2\rangle$ *is* NP^{PP} *complete.*

4.4 The complexity of acyclic metaqueries

We have shown above that, as far as the combined complexity measure is concerned, metaquerying is intractable. Next we discuss some tractable subcases. We recall some definitions first.

Definition 4.29 A *Hypergraph* $H = \langle V, E \rangle$ is a set V of vertices, and a set $E \subseteq 2^V$ of (hyper)edges. An *ear* for a hypergraph $H = \langle V, E \rangle$ is an edge $e \in E$ such that for some distinct edge $w \in E$, called the *witness* of e, no vertex of $e - w$ is in any other edge. We say that H is acyclic if its derived hypergraph $GYO(H)$ is empty, where $GYO(H)$ is built by applying the following steps until there are no ears (see [1]) :

1. remove from E all isolated edges, i.e., edges sharing no vertex with other edges.
2. choose an ear e of H.
3. remove e from H by deleting it from E and by deleting from V vertices of e not appearing elsewhere in E.

The following notion of acyclicity for metaqueries is a little bit different from the usual one employed for ordinary conjunctive queries [8]. In particular, the hypergraph associated with a metaquery will contain both ordinary and predicate variables as nodes.

Definition 4.30 Let s be an atom, a relation pattern, or a metaquery. The set of (both predicate and ordinary) variables of s is denoted $var(s)$, whereas the set of ordinary variables of s is denoted $var_o(s)$. Let $\mathbf{MQ}$ be a metaquery, define the *hypergraph* $H(\mathbf{MQ}) = \langle V, E \rangle$ associated with $\mathbf{MQ}$ as follows. $V = var(\mathbf{MQ})$, whereas E contains an edge $e_i = var(R_i(\mathbf{X}_i))$ for each literal scheme $R_i(\mathbf{X}_i)$ in $\mathbf{MQ}$. We say that $\mathbf{MQ}$ is *acyclic* if $H(\mathbf{MQ})$ is acyclic. Similarly, define the *semi-hypergraph* $SH(\mathbf{MQ}) = \langle V', E' \rangle$ associated with $\mathbf{MQ}$ as follows. $V' = var_o(\mathbf{MQ}))$, E' contains an edge $e_i = var_o(R_i(\mathbf{X}_i)$ for each literal scheme $R_i(\mathbf{X}_i)$ in $\mathbf{MQ}$. We say that $\mathbf{MQ}$ is *semi-acyclic* if $SH(\mathbf{MQ})$ is acyclic.

For example, the metaquery

$$\mathbf{MQ}_1 = P(X, Y) \leftarrow P(Y, Z), Q(Z, W)$$

is acyclic, whereas the slightly different metaquery

$$\mathbf{MQ}_2 = P(X, Y) \leftarrow Q(Y, Z), P(Z, W)$$

is cyclic. Finally, the metaquery

$$\mathbf{MQ}_1 = N(X) \leftarrow N(Y), E(X, Y)$$

is semi-acyclic, but it is not acyclic. It is straightforward to show that an acyclic metaquery is semi-acyclic as well.

Theorem 4.31 *Let* $\mathbf{MQ}$ *be an acyclic metaquery and* $I \in \mathbf{I}$ *a plausibility index. The combined complexity of* $\langle \mathbf{DB}, \mathbf{MQ}, I, 0, 0 \rangle$ *is LOGCFL-complete under logspace reductions.*

However, acyclicity is not sufficient to guarantee tractability in general, as shown next for instantiation types other than type-0.

Theorem 4.32 *Let* $I \in \mathbf{I}$ *be a plausibility index and* $T \in \{1, 2\}$ *and let* $\mathbf{MQ}$ *be an acyclic metaquery. The combined complexity of* $\langle \mathbf{DB}, \mathbf{MQ}, I, 0, T \rangle$ *is NP-complete.*

Theorem 4.33 *Let $I \in \mathbf{I}$ be a plausibility index and $T \in \{1,2\}$ and let* $\mathbf{MQ}$ *be an acyclic metaquery. The combined complexity of* $\langle \mathbf{DB}, \mathbf{MQ}, \mathrm{sup}, k, T\rangle$ *and* $\langle \mathbf{DB}, \mathbf{MQ}, \mathrm{cvr}, k, T\rangle$ *is NP-complete.*

One can ask if disregarding predicate variables, and thus referring to semi-acyclic metaqueries, instead of considering acyclic ones would be sufficient in order to give a polynomial evaluation algorithm as far as type-0 metaqueries are concerned. The next result shows that, unfortunately, the evaluation of semi-acyclic type-0 metaqueries is not simpler than evaluating general metaqueries.

Theorem 4.34 *Let $I \in \mathbf{I}$ be a plausibility index and let* $\mathbf{MQ}$ *be a semi-acyclic metaquery. The combined complexity of* $\langle \mathbf{DB}, \mathbf{MQ}, I, 0, 0\rangle$ *is NP-complete.*

4.5 Data complexity

In this section we discuss data complexity of metaquerying problems. Similarly to most query languages, the data complexity is much lower than the combined complexity. In particular, in some interesting case, it lies very low in the complexity hierarchy, as proven next.

Theorem 4.35 *Under the data complexity measure (fixed metaquery and threshold value, variable database),* $\langle \mathbf{DB}, \mathbf{MQ}, I, 0, T\rangle$ *is in AC^0, for $I \in \mathbf{I}$ and for $T \in \{0,1,2\}$.*

In the general case, the data complexity of metaquerying is within TC^0.

Theorem 4.36 *Let $T \in \{0,1,2\}$. Let* $\mathbf{MQ}$ *be a metaquery,* $\mathbf{DB}$ *be a database, $I \in \mathbf{I}$ a plausibility index, and $0 \leq k < 1$. The data complexity of the metaquerying problem* $\langle \mathbf{DB}, \mathbf{MQ}, I, k, T\rangle$ *is in TC^0.*

5 Algorithms for metaquery answering

In this section we address the problem of efficiently computing all the instantiations $\sigma_{\mathbf{DB}}^{\mathbf{MQ}}$ of a fixed metaquery $\mathbf{MQ}$ on an input database $\mathbf{DB}$, such that the values $cnf(\sigma(\mathbf{MQ}))$, $cvr(\sigma(\mathbf{MQ}))$, and $sup(\sigma(\mathbf{MQ}))$ are greater than user-provided thresholds, k_{cnf}, k_{cvr}, and k_{sup}, resp.
We decompose our answering problem into the following three subproblems:

1. Find all the partial instantiations σ_b defined on $body(\mathbf{MQ})$ such that the support obtained by applying σ_b to $\mathbf{MQ}$ is greater than k_{sup} in $\mathbf{DB}$.
2. For each partial instantiation σ_b found in step 1, find all the partial instantiations σ_h defined on $head(\mathbf{MQ})$ such that $\sigma = \sigma_h \circ \sigma_b$ is an instantiation defined on $\mathbf{MQ}$, and $cvr(\sigma(\mathbf{MQ})) > k_{cvr}$.
3. Return as a solution the instantiations σ found in step 2 such that $cnf(\sigma(\mathbf{MQ})) > k_{cnf}$.

The rationale behind this choice is as follows. First, we note that a high-support body is potentially shared by various rules, having different heads. Furthermore, by definition, the computation of the support requires to reduce, one after another, the relations in the rule's body, that is, to compute $\pi_{att(r_i)}(\mathbf{J}(body(r)))$, for each $r_i \in body(r)$; reducing a relation r_i w.r.t. a set of relations S can be sometimes (i.e., when some acyclicity criterion is met) performed

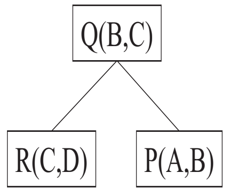

Figure 3: The join tree of Example 5.3.

without explicitly computing $\mathbf{J}(S)$, hereby gaining in efficiency. Similarly, the computation of the cover (step 2) requires only the reduction of the relation associated to the head. Finally, the computation of the overall join (step 3) becomes less expensive if the involved relations are reduced (see [51]).

Next, we recall and/or extend some definitions related to computing conjunctive queries. For simplicity of notation, as we have also done above, in the following we refer to atoms and to associated relations interchangeably, unless ambiguity arises.

Definition 5.1 Let $r_1, \ldots, r_n$ be a set of relations. We say that r_i, $i \in \{1,...,n\}$, is *reduced* w.r.t. $r_1, \ldots, r_n$, if $r_i = \pi_{att(r_i)}(r_1 \bowtie \ldots \bowtie r_n)$. We say that $r_1, \ldots, r_n$ is *reduced* if r_i is reduced w.r.t. $r_1, \ldots, r_n$, for each $i = 1, \ldots, n$.

Definition 5.2 A *join tree* for a set of literal schemes Q is a tree $\mathcal{T}$, whose vertices are the literal schemes of Q, such that whenever the same ordinary variable X occurs in two literal schemes L_1 and L_2, then X occurs in each literal scheme on the unique path linking L_1 and L_2 (see [1]).

Example 5.3 *Let Q be the set $\{P(A,B), Q(B,C), R(C,D)\}$ of literal schemes. Then a join tree for Q is reported in Figure 3.*

It is then easy to generalize analogous results proved for conjunctive queries, to show that a metaquery is semi-acyclic iff its set of literals has a join tree [1, 29].

Definition 5.4 Let $Q = \{r_1, \ldots, r_n\}$ be a set of atoms on a database $\mathbf{DB}$. A *semijoin step* is an expression of the form $r_i := r_i \ltimes r_j$, with $1 \leq i, j \leq n$. A *semijoin program* is a sequence of semijoin steps. A semijoin program is called a *full reducer* for Q if, after executing that program, each r_i is reduced, independently of the initial values of the relations (see [1]).

A full reducer is a way to efficiently reduce a set of atoms. Unfortunately, not every set of atoms has a full reducer. Indeed, it is proved in [11] that a set of atoms has a full reducer iff it is semi-acyclic.

For a semi-acyclic set of atoms Q, a full reducer consists of a sequence of two semijoin programs of the same length, called *first-half* and *second-half* respectively. Let $\mathcal{T}$ be a rooted join tree for Q. The first-half is obtained by performing a bottom-up visit of $\mathcal{T}$: let r_i be the current node in the visit, then for each child r_j of r_i in $\mathcal{T}$, add $r_i := r_i \ltimes r_j$ as next step of the sequence. The second-half is then obtained from the first-half, by reversing the sequence and exchanging the relations, i.e. from $r_i := r_i \ltimes r_j$ we obtain $r_j := r_j \ltimes r_i$.

Example 5.5 *Consider the set $Q = \{p(A,B), p(B,C), r(C,D)\}$. Let $\mathcal{T} = \langle Q, \{\langle q(B,C), p(A,B)\rangle, \langle q(B,C), r(C,D)\rangle\}\rangle$ be an associated join tree. Suppose $\mathcal{T}$ is rooted at $q(B,C)$. A full reducer for Q is:*

$$\left.\begin{array}{l} q(B,C) := q(B,C) \ltimes r(C,D) \\ q(B,C) := q(B,C) \ltimes p(A,B) \end{array}\right\} \textit{first-half}$$
$$\left.\begin{array}{l} p(A,B) := p(A,B) \ltimes q(B,C) \\ r(C,D) := r(C,D) \ltimes q(B,C) \end{array}\right\} \textit{second-half}$$

In general, to compute the value of support, we need to reduce a generic set of atoms, not necessarily a semi-acyclic one. To achieve this goal efficiently we exploit the concept of *hypertree decomposition* of a conjunctive query [28] (see also [29] for a detailed discussion on the concept of "degree of cyclicity" of a conjunctive query).

Definition 5.6 Let Q be a set of literal schemes. A *hypertree* for Q is a triple $\langle \mathcal{T}, \chi, \lambda \rangle$, where $\mathcal{T}$ is a rooted tree, and χ and λ are labeling functions which associate to each vertex p of $\mathcal{T}$ two sets $\chi(p) \subseteq var_o(Q)$ and $\lambda(p) \subseteq Q$.

Definition 5.7 Let $\mathcal{T}$ be a rooted tree. We denote by $vertices(\mathcal{T})$ the set of nodes of $\mathcal{T}$. Define $\chi(\mathcal{T})$ as $\cup_{p \in vertices(\mathcal{T})} \chi(p)$. For any $p \in vertices(\mathcal{T})$, $\mathcal{T}_p$ denotes the subtree of $\mathcal{T}$ rooted at p. A *hypertree decomposition* [28] of a set of literal schemes Q is an hypertree for Q such that

1. for each literal scheme $L \in Q$, there exists $p \in vertices(\mathcal{T})$ such that $var_o(L) \subseteq \chi(p)$;
2. for each ordinary variable $Y \in var_o(Q)$, the set $\{p \in vertices(\mathcal{T}) \mid Y \in \chi(p)\}$ induces a (connected) subtree of $\mathcal{T}$;
3. for each vertex $p \in vertices(\mathcal{T})$, $\chi(p) \subseteq var_o(\lambda(p))$;
4. for each vertex $p \in vertices(\mathcal{T})$, $var_o(\lambda(p)) \cap \chi(\mathcal{T}_p) \subseteq \chi(p)$.

An hypertree decomposition $\langle \mathcal{T}, \chi, \lambda \rangle$ of Q is *complete* if, for each $L \in Q$, there exists $p \in vertices(\mathcal{T})$ such that $var_o(L) \subseteq \chi(p)$ and $L \in \lambda(p)$.

We refer to [28] for a complete discussion about the hypertree decomposition of conjunctive queries.

Example 5.8 *Let Q^{ex} be the set $\{P(A,B), Q(B,C), R(C,D), S(B,D)\}$ of literal schemes. Let $\mathcal{T}^{ex}$ be the hypertree $\langle \{\langle p_1, p_2\rangle, \langle p_2, p_3\rangle\}, \chi^{ex}, \lambda^{ex}\}\rangle$ rooted at p_1, where $\chi^{ex}(p_1) = \{A, B\}$, $\chi^{ex}(p_2) = \{B, C\}$, $\chi^{ex}(p_3) = \{B, C, D\}$, $\lambda^{ex}(p_1) = \{P(A,B)\}$, $\lambda^{ex}(p_2) = \{Q(B,C)\}$, and $\lambda^{ex}(p_3) = \{R(C,D), S(B,D)\}$. Then $\langle \mathcal{T}^{ex}, \chi^{ex}, \lambda^{ex}\rangle$ is a hypertree decomposition of Q^{ex}.*

The *width* of an hypertree decomposition $\langle \mathcal{T}, \chi, \lambda \rangle$ is defined as $\max_{p \in vertices(\mathcal{T})} |\lambda(p)|$. The *hypertree-width* $hw(Q)$ of a set of literal schemes Q is defined as the minimum width over all its hypertree decompositions.

The notion of bounded hypertree-width generalizes the notion of semi-acyclicity for meta-queries. Indeed, a set of literal schemes Q is semi-acyclic iff $hw(Q) = 1$.

Proposition 5.9 Let $\mathbf{MQ}$ be a metaquery and $\mathbf{DB}$ a database. Let $\langle \mathcal{T}, \chi, \lambda \rangle$ be a hypertree decomposition of width c for $\mathbf{MQ}$, and let $\sigma^{\mathbf{MQ}}_{\mathbf{DB}}$ be an instantiation. Define and $\lambda'(p) = \sigma(\lambda(p))$ for each $p \in vertices(\mathcal{T})$. Then $\langle \mathcal{T}, \chi, \lambda' \rangle$ is an hypertree decomposition of width c.

Example 5.10 *The width of $\langle \mathcal{T}^{ex}, \chi^{ex}, \lambda^{ex}\rangle$ of Example 5.8 is 2. As Q^{ex} is not semi-acyclic, it follows that 2 is also the hypertree-width of Q^{ex}.*

Let Q be a set of atoms on a database $\mathbf{DB}$, and let $\langle \mathcal{T}, \chi, \lambda \rangle$ be a hypertree decomposition of Q. It is known that we can build from Q, $\mathbf{DB}$ and $\langle \mathcal{T}, \chi, \lambda \rangle$, a set of atoms Q', a database $\mathbf{DB}'$, and a join tree $\mathcal{T}'$ for Q', such that Q' is semi-acyclic and $\mathbf{J}(Q)$ on $\mathbf{DB}$ is equal to $\mathbf{J}(Q')$ on $\mathbf{DB}'$ [28]. Denote the above triplet $\langle Q', \mathbf{DB}', \mathcal{T}' \rangle$ as $acy(Q, \mathbf{DB}, \langle \mathcal{T}, \chi, \lambda \rangle)$. $\mathcal{T}'$ has the same tree shape as $\mathcal{T}$. For each vertex p of $\mathcal{T}$, there is precisely one vertex p' in $\mathcal{T}'$, and one relation r' in $\mathbf{DB}'$. p' is an atom having $\chi(p)$ as arguments and r' is set to $\pi_{\chi(p)}(\mathbf{J}(\lambda(p)))$. Q' contains all the atoms corresponding to vertices of $\mathcal{T}'$.

function FINDRULES(**DB**, **MQ**, k_{cnf}, k_{cvr}, k_{sup}, T) : set of instantiations;
var Σ : set of instantiations;
    r, s : **array** $[1..n]$ **of** relation;
    procedure FINDHEADS(σ_b : an instantiation);
        function ENOUGHSUPPORT : **boolean**;
        begin
            for each literal scheme $a \in body(\mathbf{MQ})$ **do begin**
                let i such that $var_o(a) \subseteq \chi(p_{\nu(i)})$ and $a \in \lambda(p_{\nu(i)})$;
                let r_a be the relation associated to $\sigma_b(a)$ in **DB**;
                if $|\pi_{var_o(a)}(s[i]) \bowtie r_a|/|r_a| > k_{sup}$ **then return true**;
            end;
            return false;
        end; { ENOUGHSUPPORT }
    begin
        if ENOUGHSUPPORT **then begin**
            compute from s the relation $b = \mathbf{J}(\sigma_b(body(\mathbf{MQ})))$;
            for each type T instantiation σ_h for $head(\mathbf{MQ})$ that agree with σ_b **do begin**
                Set h to be the relation associated to $head(\mathbf{MQ})$ by σ_h;
                $h' := h \ltimes b$;
                if $|h'|/|h| > k_{cvr}$ **and** $|b \ltimes h'|/|b| > k_{cnf}$ **then** $\Sigma := \Sigma \cup \{\sigma_h \circ \sigma_b\}$;
            end
        end
    end; { FINDHEADS }
    procedure FINDBODIES(i : **integer**; σ_b : an instantiation);
    begin
        if $i \leq n$ **then** (* first half *)
            for each type T instantiation σ_i for $\lambda(p_{\nu(i)})$ that agree with σ_b **do begin**
                $r[i] := \pi_{\chi(p_{\nu(i)})}(\mathbf{J}(\sigma_i(\lambda(p_{\nu(i)}))))$;
                for each child p_j of $p_{\nu(i)}$ **do** $r[i] := r[i] \ltimes r[\nu^{-1}(j)]$;
                if $r[i] \neq \emptyset$ **then** FINDBODIES($i+1$, $\sigma_b \circ \sigma_i$);
            end
        else begin (* second half *)
            $s[n] := r[n]$;
            for $j := n-1$ **downto** 1 **do** $s[j] := r[j] \ltimes s[\nu(father(\nu^{-1}(j)))]$;
            FINDHEADS(σ_b);
        end;
    end; { FINDBODIES }
begin
    compute c, the hypertree-width of $body(\mathbf{MQ})$;
    compute $\langle \mathcal{T}, \chi, \lambda \rangle$, a complete hypertree decomposition for $body(\mathbf{MQ})$ of width c;
    compute a bottom-up visit of $\mathcal{T} = (\{p_1, \ldots, p_n\}, E)$ and encode it as a permutation ν of $\{1, \ldots, n\}$;
    $\Sigma := \emptyset$;
    FINDBODIES(1, $\emptyset$);
    return Σ;
end { FINDRULES };

Figure 4: Algorithm FindRules

Example 5.11 *Consider the set of atoms Q^{ex} of Example 5.8, the associated hypertree decomposition $\langle \mathcal{T}^{ex}, \chi^{ex}, \lambda^{ex} \rangle$, and the database $\mathbf{DB}^{ex} = \{p, q, r, s\}$. Let $p'_1 = p(A, B)$, $p'_2 = q(B, C)$, and $p'_3 = t(B, C, D) = r(C, D) \bowtie s(B, D)$, let $\mathcal{T}'^{ex}$ be the tree $\langle \{p'_1, p'_2, p'_3\}, \{\langle p'_1, p'_2 \rangle, \langle p'_2, p'_3 \rangle\} \rangle$, let $Q'^{ex} = \{p(A, B), q(B, C), t(B, C, D)\}$, and let $\mathbf{DB}'^{ex} = \{p, q, t\}$. Then $acy(Q^{ex}, \mathbf{DB}^{ex}, \langle \mathcal{T}^{ex}, \chi^{ex}, \lambda^{ex} \rangle) = \langle Q'^{ex}, \mathbf{DB}'^{ex}, \mathcal{T}'^{ex} \rangle$.*

Theorem 5.12 *Let $\mathbf{r}$ be a Horn rule and let $\mathbf{DB}$ be a database, and let d be the size of the largest relation within $\mathbf{DB}$. Then $sup(\mathbf{r})$ can be computed in time $d^c \log d$, where c is the hypertree-width of $body(\mathbf{r})$.*

Definition 5.13 Let S be a set of literal schemes. We denote by $pv(S)$ the set of predicate variables occurring in S. Let $\sigma_{\mathbf{DB}}^{S_1}$ and $\mu_{\mathbf{DB}}^{S_2}$ be two instantiations defined on two set of literal schemes S_1 and S_2 over a database $\mathbf{DB}$, respectively. We say that σ and μ *agree* if

1. $\sigma(S) = \mu(S)$, where $S = S_1 \cap S_2$, and
2. $\sigma'(V) = \mu'(V)$, where,
   - σ' and μ' are the restrictions of σ and μ on the sets $pv(S_1)$ and $pv(S_2)$ of predicate variables of S_1 and S_2, respectively, and
   - $V = pv(S_1) \cap pv(S_2)$.

Clearly, if σ and μ agree, then $\sigma \circ \mu$ is an instantiation on the set $S_1 \cup S_2$.

Figure 4 shows Algorithm FINDRULES that, given in input a database $\mathbf{DB}$, a metaquery $\mathbf{MQ}$, three rational numbers $0 \leq k_{sup}, k_{cvr}, k_{cnf} < 1$, and $T \in \{0, 1, 2\}$, computes all the type-T instantiations $\sigma_{\mathbf{DB}}^{\mathbf{MQ}}$ such that $sup(\sigma(\mathbf{MQ})) > k_{sup}$, $cvr(\sigma(\mathbf{MQ})) > k_{cvr}$, and $cnf(\sigma(\mathbf{MQ})) > k_{cnf}$.
Within the main procedure, the algorithm computes the hypertree decomposition $\langle \mathcal{T}, \chi, \lambda \rangle$ of $body(\mathbf{MQ})$. By Proposition 5.9, this decomposition can be employed for each instantiation generated. Then, the procedure FINDBODIES performs a bottom-up visit of $\mathcal{T}$ that generates the instantiations σ_b of the body of the metaquery by composing the instantiations σ_i of the visited literal schemes. Also it computes the databases $\mathbf{DB}'$ and $\overline{\mathbf{DB}'}$ (see the proof of Theorem 5.12, steps (3) and (4)), and mantains in a data structure (an array r) the portions of the database $\overline{\mathbf{DB}'}$ shared by the partial instantiations generated during the visit, reusing them. When the root of the tree $\mathcal{T}$ is attained, the first half of a full reducer for $\sigma_b(body(\mathbf{MQ}))$ has been calculated. Hence the procedure FINDBODIES computes the second half of this full reducer (storing the reduced relations into an array s) and calls the procedure FINDHEADS. A backtracking on previously generated local substitutions is finally performed.

The procedure FINDHEADS then verifies if $\sigma_b(body(\mathbf{MQ}))$ has large enough support (this is done by calling the function ENOUGHSUPPORT) and, in the positive case, it computes the associated relation, and searches for heads such that the instantiated metaquery has large enough cover and confidence.

As for the time-complexity analysis of algorithm FINDRULES, in the data complexity setting, we note that during the bottom-up visit of $\mathcal{T}$ it performs a semijoin step for each edge of the visited tree, and that the number of edges visited by the algorithm is upper bounded by the number of instantiations of the body of the metaquery.

Then we can compute all the instantiations $\sigma_{\mathbf{DB}}^{\mathbf{MQ}}$ such that $sup(\sigma(\mathbf{MQ})) > k_{sup}$ in $n^{m-1} d^c \log d$ steps for $T \in \{0, 1\}$, and in $(nb^a)^{m-1} d^c \log d$ steps for $T = 2$, where n is the number of relations within $\mathbf{DB}$, m the number of relation patterns of the metaquery $\mathbf{MQ}$, a the maximum arity of any relation pattern in $\mathbf{MQ}$, b the maximum arity of any relation in $\mathbf{DB}$, and c is the hypertree-width of $body(\mathbf{MQ})$. Note that a, m and $1 \leq c \leq m - 1$ are constants. Moreover ma

is an upper bound for the metaquery size, nbd is an upper bound for the number of attribute values in the database **DB**, and usually holds that $d \gg nb$.

The additional steps needed to search for instantiations with high cover and confidence requires a total amount of $(nd)^m$ steps for $T \in \{0,1\}$, and $(nb^a d)^m$ steps for $T = 2$.

6 Conclusions

In this part we have formally defined the semantics of metaqueries and analyzed their computational complexity. In general, as far as the combined complexity is concerned, metaquerying is intractable (unless P=NP). Therefore, we have defined the class of acyclic metaqueries and shown that a subset of metaquerying problems defined by acyclic metaqueries is in $LOGCFL$ and, as such, is highly parallelizable. Moreover, we have studied the data complexity of metaquerying problems and showed that, in general, it lies within TC^0. Even in this case, we were able to single out a subset of metaquerying problems, for which the data complexity is in AC^0. Finally, we have discussed metaquery implementation. The complexity analysis of metaquerying problems presented here, and which is summarized in Figure 5, is not complete, though. We are working towards establishing other results. In particular, it seems to be interesting to extend our formal framework to more general metaquery forms, such as allowing negation and/or disjunction to occur in metapatterns.

Complexity measure	**Problem type**	**Instantiation type**	**Indices**	**Threshold**	**Complexity**
Combined Complexity	General	$0, 1, 2$	**I**	$k = 0$	NP-Complete (Th. 4.21)
Combined Complexity	General	$0, 1, 2$	cvr and sup	$0 \leq k < 1$	NP-Complete (Th. 4.23)
Combined Complexity	General	$0, 1, 2$	cnf	$0 \leq k < 1$	NP^{PP}-Complete (Th. 4.27, 4.28)
Combined Complexity	Acyclic	0	**I**	$k = 0$	LOGCFL-Complete (Th. 4.31)
Combined Complexity	Acyclic	$1, 2$	**I**	$k = 0$	NP-Complete (Th. 4.32)
Combined Complexity	Acyclic	0	cvr and sup	$0 \leq k < 1$	Open
Combined Complexity	Acyclic	$1, 2$	cvr and sup	$0 \leq k < 1$	NP-complete (Th. 4.33)
Combined Complexity	Acyclic	$0, 1, 2$	cnf	$0 \leq k < 1$	Open
Combined Complexity	Semi-acyclic	$0, 1, 2$	**I**	$k = 0$	NP-Complete (Th. 4.34,Th. 4.32)
Data Complexity	General	$0, 1, 2$	**I**	$k = 0$	AC_0 (Th. 4.35)
Data Complexity	General	$0, 1, 2$	**I**	$0 \leq k < 1$	TC_0 (Th. 4.36)

Figure 5: Summary of complexity results.

Part II
Data clustering (*Stefano Lodi and Claudio Sartori*)

7 Introduction

Cluster analysis is a method for data exploration, originally proposed by statisticians, whose aim is to group data according to some notion of similarity [34]. In the optimal case, this leads to a maximal similarity between members of the same group and to minimal similarity between members of different groups. More recently, cluster analysis has been proposed as a *data mining* activity [24], named *data clustering.* The methods proposed for data clustering can efficiently analyze large data bases still keeping (and sometimes improving) cluster quality, with respect to many traditional methods.

Up to now, little attention has been devoted to dynamic data, i.e., data which are frequently modified and, therefore, require frequent updates of the group composition. An example of this situation is found in *data warehousing.* A data warehouse usually materializes views from a set of base (operational) tables and performs on them some processing, such as clustering. Usually a data warehouse is rebuilt during the periods of inactivity of the operational systems [43], but problem size can easily lead to situations where the inactivity time is not sufficient for re-computing clusters from scratch.

A more efficient solution could reuse old clustering information and consider separately new (or updated) data, obtaining new clusters from the old ones. A dynamic algorithm must perform a *pre-processing stage*, followed by a number of *update stages* [41]. The pre-processing stage reads an initial instance of the problem and computes the initial state which will be used in the iterations and, possibly, an initial solution. An update stage reads an update of the current instance and generates a new state and a new solution, starting the current state.

A few dynamic clustering methods have been proposed in the literature: IncrementalDB-SCAN [22], derived from the static algorithm DBSCAN [23], BIRCH [60] and BUBBLE [27]. The dimensions to consider in order to evaluate and compare their capabilities include effectiveness, efficiency, dynamic constraints and structure of the data space. An optimal solution should deal with fully dynamic data (including both insertions and deletions), metric spaces, categorical data, and clusters of arbitrary shape.

None of the above algorithms is completely satisfactory with respect to at least one of the above-mentioned dimensions. On the other hand, we can observe that these kinds of problems have been studied (and partly solved) in the environments of statistics and pattern recognition. In this work we will discuss how the methods proposed in these environments can be transformed into dynamic clustering algorithms.

The structure of this part is the following. In Section 8 we present, using a common framework and formalism, three clustering methods known in the literature. In Section 9 we derive fully dynamic algorithms for each method of Section 8, and discuss a two-layered architecture for fully dynamic clustering in internal and external memory. In Section 10 we briefly review existing dynamic algorithms and finally, in Section 11, we draw some conclusions and discuss future work.

8 Static Clustering Methods

In the following we will introduce a common formalism for the description of a clustering problem and describe three methods proposed in [30, 37, 58].

Let (X, d) be a metric space and S a finite subspace of X. Let $x \in X$, $y \in X$. By *rank* $r_x^S(y)$ of y w.r.t. x and S we mean the number $i \in \mathbb{N}$ such that exactly $i-1$ points of S are closer to x than y. (By convention, when $x \in S$, the 0th closest point to x is x itself.)

We assume that ties are broken by any method that assigns ranks consistently to any pair of points w.r.t. any point, thereby preserving relative rankings under insertion and deletions. If y has rank i w.r.t. x, we also say y is the ith nearest neighbour of x, written $x_{(i)}(S)$. By *k-neighbourhood* of x in S, written $\mathtt{NN}_S(x,k)$, we mean the set of the k nearest neighbours of x defined by $\mathtt{NN}_S(x,k) = \{y \in S : r_x^S(y) \leq k\}$, i.e., the set of points in S having rank at most k w.r.t. x. Let $d_i(x,S) = d(x,y)$, where $r_x^S(y) = i$. Finally, for $x \in S$, we define $\mathtt{NN}_S^{-1}(x,k) = \{y \in S : x \in \mathtt{NN}_S(y,k)\}$, i.e., the set of all points in S having x among their k nearest neighbours. We will omit the subspace symbol when it is understood from the context. In the following, let $K \in \mathbb{N}$ be a parameter of each algorithm.

The three methods can be generalized by the following schema:

1. For each point y of S compute the K nearest neighbours, i.e., $\mathtt{NN}(y,K)$.
2. Compute a dissimilarity function δ, such that δ_{xy} depends only on $\mathtt{NN}(x,K)$ and $\mathtt{NN}(y,K)$.
3. Compute the connected components of the graph whose vertices are the points of the problem instance and whose weighted edges are the point pairs having dissimilarity smaller than θ. Such components are the clusters of the solution.

The definition of dissimilarity and the number of nearest neighbours separate the methods from one another, as follows:

A. Gowda and Krishna [30]:

$$\delta_{xy} = \begin{cases} r(x,y) + r(y,x), & \text{if } x \in \mathtt{NN}(y,K) \text{ and } y \in \mathtt{NN}(x,K), \\ \infty, & \text{otherwise.} \end{cases} \tag{5}$$

K is a constant in the number of tens.

B. Jarvis and Patrick [37]:

$$\delta_{xy} = \begin{cases} K - |\mathtt{NN}(x,K) \cap \mathtt{NN}(y,K)|, & \text{if } x \in \mathtt{NN}(y,K) \text{ and } y \in \mathtt{NN}(x,K), \\ \infty, & \text{otherwise.} \end{cases} \tag{6}$$

K is a constant in the number of tens.

C. Wong and Lane [58]:

$$\delta_{xy} = \begin{cases} \frac{1}{2}[d_K(x) + d_K(y)], & \text{if } x \in \mathtt{NN}(y,K) \text{ or } y \in \mathtt{NN}(x,K), \\ \infty, & \text{otherwise.} \end{cases} \tag{7}$$

$K = K(N)$ is a function such that $\lim_{N\to\infty} \frac{K(N)}{\ln N} = \infty$, $\lim_{N\to\infty} \frac{K(N)}{N} = 0$.

We briefly summarize the properties of the methods above.

Many traditional hierarchical algorithms show a preference for particular cluster shapes. For instance, the single linkage method tends to join clusters connected by thin dense lines of points (the well known "chaining effect"), thus showing a bias towards continuity, whereas the complete linkage method tends to join elongated clusters whose separation is less than their diameter, thus showing a bias towards globular clusters. The measures of similarity chosen both by method A and B reduce the chaining effect because they take into account the K-neighbourhood of the points. Moreover, since the size of a K-neighbourhood is generally orders of magnitude smaller than the data set, the ability to recognize clusters of arbitrary shape is preserved [30][36][37].

Evaluating similarity as the number of common neighbours in method B also increases robustness in categorical clustering. The following simple example, related to transactional clustering, can be found in [32]. Groups of customers of large retailing companies buy in each transaction only a small number of items from a much larger set of items that defines the cluster. In this environment, the motivation for the method is explained by observing that the similarity of transactions within a cluster is often smaller than the similarity between transactions in different clusters. However, transactions in the same cluster have a larger number of common neighbours than transactions in different clusters.

Method C uses the average neighbourhood radius to estimate similarity of the density of two points. It is shown in [58] that a property holds for this method which is important for incremental approximate clustering: If the data set S is a sample drawn from an underlying probability density function, then the cluster tree produced by the method converges to the cluster tree of the distributions as the number of points increases. The condition ensuring this property is that the number of neighbours K be a superlogarithmic function of N. Practically, in most cases this number will be less than a hundred.

9 Fully Dynamic Clustering

We describe in this section an approach to fully dynamic clustering versions of the algorithms described in the previous section. Let S be a data set which is the initial instance of the problem.

Between two updates, we store the following information in the state of the algorithm:

1. An undirected weighted dissimilarity graph on S, where two points in S are joined by an edge with weight δ if and only if the initialization stage computed a finite dissimilarity δ between them: $G_\delta = (S, E_\delta)$, where $(x, y, \delta_{xy}) \in E_\delta \iff \delta_{xy} \neq \infty$. Notice that, for fixed K, G_δ is a sparse graph satisfying $|E_\delta| = \Theta(N)$.
2. The connected components of the graph obtained from G_δ by removing all edges weighing more than θ.
3. Optionally, two directed graphs G_K and G_K^{-1}, storing, for each point $x \in S$, the K nearest neighbours of x and all the points y such that x is among the y's K nearest neighbours, respectively: $G_K = (S, E_K)$, $G_K^{-1} = (S, E_K^{-1})$, where $(x, y) \in E_K \iff y \in \mathtt{NN}_S(x, K)$, $(x, y) \in E_K^{-1} \iff y \in \mathtt{NN}_S^{-1}(x, K)$. For fixed K, G_K and G_K^{-1} are sparse graphs satisfying $|E_K| = \Theta(N)$, $|E_K^{-1}| = \Theta(N)$.

The initialization stage computes G_K, G_K^{-1}, G_δ and the output clustering as the set of connected components of the graph obtained by deleting from G_δ all edges having weight greater than θ.

The architecture of the update stage is structured into two layers. Roughly, the outer layer performs an analysis of the K-neighbourhoods influenced by the insertion or deletion and determines how to update dissimilarities, that is, which updates (edge insertion or deletion, weight update) must be done in order to bring G_δ, G_K, G_K^{-1} into the new state. The inner layer executes the requested updates to the graphs and calls a dynamic algorithm to re-compute the connected components from the current components and the requested update.

9.1 Updating Dissimilarities

We now describe the outer layer of the update stage for every method in the previous section and discuss its implementation. In the following, let $x^* \in X$ a new point to be inserted into S, or a point to be deleted from S.

Algorithm 1 Method A: Dissimilarity update under single point insertion

```
for y ∈ S : r_y^S(x*) ≤ K do
    if r_{x*}^S(y) ≤ K then insert(y, x*, r_y^S(x*) + r_{x*}^S(y)) fi
    for z ∈ S : r_y^S(z) ≤ K do
        if r_y^S(z) = K
          then if r_z^S(x*) > K
                  then if r_z^S(y) ≤ K then delete(y, z) fi
               elsif r_z^S(y) = K ∧ y < z then delete(y, z) fi
        elsif r_z^S(x*) > K
             then if r_z^S(y) ≤ K then if r_y^S(x*) < r_y^S(z)
                                           then δ_yz := δ_yz + 1 fi
                  fi
        elsif r_z^S(y) < K ∧ y < z
             then if r_y^S(x*) < r_y^S(z) then δ_y := 1 fi
                  if r_z^S(x*) < r_z^S(y) then δ_z := 1 fi
                  if δ_y + δ_z > 0 then δ_yz := δ_yz + δ_y + δ_z fi
        fi
    od
od
```

9.1.1 Method A

We first select the points whose K-neighbourhood is influenced by the insertion or removal of x^*. Let y be a selected point. For both insertion and deletion, we check if the selected point y and x^* are mutual neighbours. In that case, we must delete an edge from or add an edge with appropriate dissimilarity to G_δ. Then we retrieve the K-neighbourhood of y, in case of insertion, or $K+1$-neighbourhood in case of deletion. The points in the result set are candidate to be at the other end of a new edge, or an edge to be deleted, or an updated edge. In fact, when inserting, an edge to the furthest neighbour of y, say z, which is also a mutual neighbour, must be deleted, because x^* is inserted in the rank order before z, and z is no more a mutual neighbour of y. When deleting, an edge between y and its $(K+1)$th neighbour, say z, must be inserted if the K-neighbourhood of z contains y or will contain y after the insertion of x^*. (Note that we have to check for distinct sub-cases in order to avoid duplicate edge insertion or deletion, due to symmetrical properties holding for y and z.) For points z that are non-furthest neighbours, the corresponding edge weight must be incremented once or twice, depending on the rank of the new point x^* with respect to y and z. (Again, we check for symmetrical situations; moreover, we must not cover cases of edge deletion, which have already been taken care of.)

9.1.2 Method B

The cases for inserting or deleting edges in G_δ are similar to method A. Updating edge dissimilarities is slightly more involved. For insertion, in order to determine if the number of neighbours of rank at most K shared by y and z changes, we need to compute the rank of the Kth neighbour of each with respect to the other, since only Kth neighbours may be excluded from K-neighbourhoods after the insertion of x^*. Moreover, the insertion of x^* in both neighbourhoods may replace a lost Kth neighbour, leaving thus the dissimilarity unchanged.

Algorithm 2 Method A: Dissimilarity update under single point deletion

<u>**for**</u> $y \in S : r^S_y(x^*) \leq K$ <u>**do**</u>
    <u>**if**</u> $r^S_{x^*}(y) \leq K$ <u>**then**</u> $delete(y, x^*)$ <u>**fi**</u>
    <u>**for**</u> $z \in S : r^S_y(z) \leq K+1 \ \wedge \ z \neq x^*$ <u>**do**</u>
        <u>**if**</u> $r^S_y(z) = K+1$
            <u>**then**</u> <u>**if**</u> $r^S_z(x^*) > K$
                    <u>**then**</u> <u>**if**</u> $r^S_z(y) \leq K$ <u>**then**</u> $insert(y, z, r^S_z(y) + K)$ <u>**fi**</u>
                <u>**elsif**</u> $r^S_z(y) = K+1 \ \wedge \ y < z$ <u>**then**</u> $insert(y, z, 2K)$ <u>**fi**</u>
        <u>**elsif**</u> $r^S_z(x^*) > K$
            <u>**then**</u> <u>**if**</u> $r^S_z(y) \leq K$ <u>**then**</u> <u>**if**</u> $r^S_y(x^*) < r^S_y(z)$
                                    <u>**then**</u> $\delta_{yz} := \delta_{yz} - 1$ <u>**fi**</u>
                <u>**fi**</u>
        <u>**elsif**</u> $r^S_z(y) < K \ \wedge \ y < z$
            <u>**then**</u> <u>**if**</u> $r^S_y(x^*) < r^S_y(z)$ <u>**then**</u> $\delta_y := 1$ <u>**fi**</u>
                <u>**if**</u> $r^S_z(x^*) < r^S_z(y)$ <u>**then**</u> $\delta_z := 1$ <u>**fi**</u>
                <u>**if**</u> $\delta_y + \delta_z > 0$ <u>**then**</u> $\delta_{yz} := \delta_{yz} - \delta_y - \delta_z$ <u>**fi**</u>
        <u>**fi**</u>
    <u>**od**</u>
<u>**od**</u>

9.1.3 Method C

Again we select the points y whose K-neighbourhood is influenced by the insertion or removal of x^*. In this case, however, note that the method does not require two points to be mutual neighbours to assign a non-infinite dissimilarity to their edge. Therefore, we always insert or delete a new edge between the point x^* and each y. We also compute the new K neighbourhood radius ry to use in updating the dissimilarity. When inserting, if x^* is the furthest neighbour, then the ry is distance to x^*. Otherwise, it is the distance to the current $(K-1)$th neighbour. During deletion, the K-neighbourhood of y will include one more neighbour after the removal of x^*, therefore ry is the distance to the $(K+1)$th neighbour.

Then we collect all points z which might change their dissimilarity to y. We do this in two steps. In the first step we process all points that are currently in the neighbourhood of y (insertion), or will be in its neighbourhood afterward (deletion). In the second step we process points having y in their neighbourhood, making sure they have not been already processed in the first step.

For points processed in the first step, we must compute the new K neighbourhood radius rz to be used in updating dissimilarities. Then we check the ranks of y and z with respect to each other and the rank of x^* to determine the update actions to call. When inserting, either the edge from y to z is removed (if y and z will be outside the K-neighbourhood of each other) or its weight is updated. When deleting, either a new edge from y to z is added or its weight is updated. (As in previous methods, we have to check for symmetrical situations.)

9.1.4 Implementation of the Updates

We discuss briefly the practical implementation and execution cost of the proposed update algorithms. We will examine both in-memory and external memory implementations.

We denote by F the worst-case amount of time used to retrieve $\mathtt{NN}_S(x, K)$, and by B the worst-case amount of time used to retrieve $\mathtt{NN}_S^{-1}(x, K)$. We will name henceforth $\mathtt{NN}_S(x, K)$, $\mathtt{NN}_S^{-1}(x, K)$ *forward* and *backward* K nearest neighbour query, respectively.

The structure of the algorithms consists of nested loops (at most two levels of nesting) and

Algorithm 3 Method B: Dissimilarity update under single point insertion

```
for y ∈ S : r_y^S(x*) ≤ K do
    if r_{x*}^S(y) ≤ K
       then insert(y, x*, K − |(NN_S(y, K) ∪ {x*}) ∩ NN_S(x*, K)|) fi
    for z ∈ S : r_y^S(z) ≤ K do
        if r_y^S(z) = K
           then if r_z^S(x*) > K
                   then if r_z^S(y) ≤ K then delete(y, z) fi
                elsif r_z^S(y) = K ∧ y < z then delete(y, z) fi
        elsif r_z^S(x*) > K
            then if r_z^S(y) ≤ K then if r_z^S(y_(K)) ≤ K
                                         then δ_yz := δ_yz + 1 fi
                 fi
        elsif r_z^S(y) < K ∧ y < z
            then if r_y^S(z_(K)) < K ∧ r_z^S(y_(K)) < K
                    then δ_yz := δ_yz + 1 fi
                 if r_y^S(z_(K)) > K ∧ r_z^S(y_(K)) > K
                    then δ_yz := δ_yz − 1 fi
        fi
    od
od
```

conditional statements within the loops. The outer loop body applies to the set of data points y having x^* among their K nearest neighbours, i.e. the result of a query $\mathtt{NN}_S^{-1}(x^*, K)$. The body of the first inner loop in method C and the (only) inner loop in methods A and B applies to at most $K+1$ nearest neighbours z of y. In method C, the second inner loop applies to a subset of $\mathtt{NN}_S^{-1}(y, K)$, i.e. the points z having y among their K nearest neighbours. All other statements appearing in the algorithms can be evaluated in time $O(K)$ once we know the sets of points selected by the loop statements, plus at most their $(K+1)$-neighbourhood.

First we consider in-memory implementation. If the graph G_K and G_K^{-1} are not part of the algorithm's state, then the time needed for a single insertion or deletion is bounded by the time spent by the nested query $\mathtt{NN}_S(\mathtt{NN}_S(\mathtt{NN}_S^{-1}(x^*, K), K+1), K+1)$ for methods A, B and $\mathtt{NN}_S(\mathtt{NN}_S(\mathtt{NN}_S^{-1}(x^*, K), K+1), K+1)$ + $\mathtt{NN}_S(\mathtt{NN}_S^{-1}(\mathtt{NN}_S^{-1}(x^*, K), K), K+1)$ for method C, plus the required scans of the results of the queries to test ranks and distances. Thus:

$$T_{A,B} = O\big(\mathsf{B} + K \cdot |\mathsf{B}| \cdot (\mathsf{F} + K)\big) \tag{8}$$

$$T_C = O\Big(|\mathsf{B}| \cdot \big(K \cdot (\mathsf{F} + K + |\mathsf{B}|) + \mathsf{B} + |\mathsf{B}| \cdot \mathsf{F}\Big) \tag{9}$$

If the graphs G_K and G_K^{-1} are kept in the algorithm's state, then any forward or backward nearest neighbour query at a point in S can be answered by one look-up in either graph. To this cost we must add the time to update G_K and G_K^{-1}. This can be done executing both a forward and a backward K nearest neighbour query at x^* and updating G_K and G_K^{-1} in time proportional to the size of the query result. Notice that every entry in G_K is an ordered list of neighbours whose size is bounded by $K+1$. Notice also that the size of an entry in G_K^{-1} is bounded only by N, however G_K^{-1} is only needed in method C, and we need not maintain it as an ordered list, since backward nearest neighbour queries are not used to retrieve ranks or distances by rank, but only sets of points.

For methods A and B the time cost of a single insertion or deletion is bounded by the time to execute $\mathtt{NN}_S(x^*, K)$ and $\mathtt{NN}_S^{-1}(x^*, K)$, plus the time needed for scanning the entries in the

Algorithm 4 Method B: Dissimilarity update under single point deletion

```
for y ∈ S : r_y^S(x*) ≤ K do
    if r_{x*}^S(y) ≤ K then delete(y, x*) fi
    for z ∈ S : r_y^S(z) ≤ K + 1 ∧ z ≠ x* do
        if r_y^S(z) = K + 1
          then if r_z^S(x*) > K
                 then if r_z^S(y) ≤ K
                        then insert(y, z, K − |(NN_S(y, K + 1) − {x*}) ∩
                                     NN_S(z, K)|) fi
               elsif r_z^S(y) = K + 1 ∧ y < z
                 then if r_z^S(y) ≤ K
                        then insert(y, z, K − |(NN_S(y, K + 1) − {x*}) ∩
                                     (NN_S(z, K + 1) − {x*})|) fi
               fi
        elsif r_z^S(x*) > K
          then if
                 r_z^S(y) ≤ K then if r_z^S(y_(K+1)) ≤ K
                                     then δ_yz := δ_yz − 1 fi
               fi
        elsif r_z^S(y) ≤ K ∧ y < z
          then if r_y^S(z_(K+1)) < K + 1 ∧
                  r_z^S(y_(K+1)) < K + 1
                  then δ_yz := δ_yz − 1 fi
               if r_y^S(z_(K+1)) > K + 1 ∧
                  r_z^S(y_(K+1)) > K + 1
                  then δ_yz := δ_yz + 1 fi
        fi
    od
od
```

graphs. For method C, the second inner loop requires a further computation of one backward nearest neighbour query for every y in $\mathtt{NN}_S(x^*, K)$. Therefore:

$$T_{A,B} = O\big(\mathsf{B} + \mathsf{F} + K^2 \cdot |\mathsf{B}|\big) \tag{10}$$

$$T_C = O\Big(\mathsf{B} + |\mathsf{B}| \cdot \mathsf{F} + |\mathsf{B}| \cdot K + K^2\Big) \tag{11}$$

If the data set S is a metric space, then the algorithms above can be implemented efficiently using dynamic metric access methods. For instance, the M-tree [16] supports efficient execution of queries of the form $\mathtt{NN}_S(x, k)$. Unfortunately, queries like $\mathtt{NN}_S^{-1}(x, k)$ are, to the best of our knowledge, not supported by such methods. However, if the access method supports storing of objects defined by a point and a radius, then a simple workaround is to construct a structure capable of answering efficiently inverse k-neighbourhood queries for all $k \leq K$, for fixed K, in an indirect way as follows: After loading all points of S in a temporary structure, a new structure is constructed containing an object $(x, d_K(x, S))$ for every point $x \in S$. $d_K(x, S)$ can be obtained executing a query $\mathtt{NN}_S(x, k)$ in the temporary structure. (Implementations of the nearest neighbour query usually return the distance of all neighbours in the result set.) In such structure, a *point query* x obviously returns the set $\mathtt{NN}_S^{-1}(x, k)$. Of course, the structure must be updated whenever a point x^* is inserted into or deleted from S: for all points in $\mathtt{NN}_S^{-1}(x, k)$, the corresponding object $(x, d_K(x, S))$ must be deleted and reinserted with updated radius.

Algorithm 5 Wong and Lane's approach: Dissimilarity update under single point insertion

for $y \in S : r_y^S(x^*) \leq K$ **do**
  if $r_y^S(x^*) = K$ **then** $ry := d(y, x^*)$ **else** $ry := d_{K-1}(y)$ **fi**
  $insert(y, x^*, \frac{1}{2}[d_K(x^*) + ry])$
  for $z \in S : r_y^S(z) \leq K$ **do**
    if $r_z^S(x^*) = K$ **then** $rz := d(z, x^*)$
    elsif $r_z^S(x^*) < K$ **then** $rz := d_{K-1}(z)$
      else $rz := d_K(z)$
    fi
    if $r_y^S(z) = K$
      then if $r_z^S(y) > K$ **then** $delete(y, z)$
        elsif $r_z^S(y) = K \ \wedge \ r_z^S(x^*) \leq K \ \wedge \ y < z$
          then $delete(y, z)$
        fi
    elsif $r_z^S(y) \geq K$ **then** $\delta_{yz} := \frac{1}{2}[ry + rz]$
    elsif $y < z$ **then** $\delta_{yz} := \frac{1}{2}[ry + rz]$
    fi
  od
  for $z \in S : r_z^S(y) \leq K \ \wedge \ r_z^S(x^*) > K \ \wedge \ r_y^S(z) \geq K$ **do**
    $\delta_{yz} := \frac{1}{2}[ry + d_K(z)]$
  od
od
for $y \in S : r_{x^*}^S(y) \leq K \ \wedge \ r_y^S(x^*) > K$ **do**
  $insert(y, x^*, \frac{1}{2}[d_K(x^*) + d_K(y)])$
od

9.2 Updating Connected Components

The proposed algorithms perform insertions, deletions and updates of edge weights, in the form of assignments to δ and the calls to the *insert* and *delete* procedures. Besides actually updating the graph G_δ, the called procedures must invoke a *dynamic graph algorithm* to update the connected components of G_δ.

First we consider clustering in internal memory. Fully dynamic connectivity algorithms have received considerable attention in the past. (See [21] for a survey on dynamic graph algorithms.) Algorithms in this class strike a balance between the asymptotic costs of the two operations allowed: *edge update* and *connectivity query*. The well-known deterministic approach of Frederickson has worst-case complexity $O(m^{1/2})$ per update and $O(1)$ per query, where m is the number of edges in the graph. Since the dissimilarity graph G_δ is sparse with $|E_\delta| = \Theta(N)$, Frederickson's algorithm would run in $O(N^{1/2})$ worst-case time in our application. Recently Holm et al. have presented an algorithm achieving worst-case complexity $O(\log^2 n)$ for updates and $O(\log n / \log \log n)$ for connectivity queries, where n is the number of vertices in the graph. For our application, we believe the trade-off provided by the latter approach to be more appropriate: The update cost matches well the expected running time of nearest neighbour queries in practice and the query time is small enough to guarantee that the clusters of the solution can be accessed adequately fast by front-end routines, e.g. for the computation of summaries and visualization.

We consider now clustering in external memory. Chiang et al. [14] introduce a technique to design I/O efficient graph algorithms and proposes algorithms for several fundamental graph problems. The algorithms are derived by a simulation technique from algorithms for the Parallel Random Access Machine (PRAM) model. Efficiency with respect to I/O from external memory

Algorithm 6 Wong and Lane's approach: Dissimilarity update under single point deletion

for $y \in S : r_y^S(x^*) \leq K$ **do**
  $ry := d_{K+1}(y)$
  $delete(y, x^*)$
  for $z \in S : r_y^S(z) \leq K+1 \ \wedge \ z \neq x^*$ **do**
    if $r_z^S(x^*) \leq K$ **then** $rz := d_{K+1}(z)$
      else $rz := d_K(z)$
    fi
    if $r_y^S(z) = K+1$
      then if $r_z^S(y) > K+1$ **then** $insert(y, z, \frac{1}{2}[d_{K+1}(y) + rz])$
        elsif $r_z^S(y) = K+1$
          then if $r_z^S(x^*) > K$ **then** $insert(y, z, \frac{1}{2}[d_{K+1}(y) + rz])$
            elsif $y < z$ **then** $insert(y, z, \frac{1}{2}[d_{K+1}(y) + rz])$
            fi
        fi
    elsif $r_z^S(y) > K$ **then** $\delta_{yz} := \frac{1}{2}[d_{K+1}(y) + rz]$
    elsif $y < z$ **then** $\delta_{yz} := \frac{1}{2}[d_{K+1}(y) + rz]$
    fi
  od
  for $z \in S : r_z^S(y) \leq K \ \wedge \ r_z^S(x^*) > K \ \wedge \ r_y^S(z) \geq K+1$ **do**
    $\delta_{yz} := \frac{1}{2}[ry + d_K(z)]$
  od
od
for $y \in S : r_{x^*}^S(y) \leq K \ \wedge \ r_y^S(x^*) > K$ **do**
  $delete(y, x^*)$
od

is defined in terms of number of I/O operation in the *parallel disk model* (PDM) [56]. In this model, external memory hardware is represented as an array of D devices capable of transferring in parallel B items each during a single I/O operation, and the global address space is assumed to be striped across the devices. Data are transferred to and from an internal memory of capacity M items. Therefore in PDM, the worst-case complexity of an algorithm is function of B, D, M, and the number of input items. I/O efficient algorithms can often be expressed in terms of two primitive operations, *scanning* and *sorting*. The asymptotic upper bounds in I/O of these operations for x input items are denoted as $scan(x)$ and $sort(x)$, and are be shown to be

$$scan(x) = \frac{x}{DB} \tag{12}$$

$$sort(x) = \frac{x}{DB} \log_{M/B} \frac{x}{B} \tag{13}$$

We recall a useful result from [14].

Theorem 9.1 ([14]) *Let A be a PRAM algorithm that solves a problem of size N by using N processors and $O(N)$ space, and that after each of $O(\log N)$ stages, each of time T, both the number of active processors and the number of memory cells that will ever be used again are reduced by a constant factor. Then A can be simulated in external memory in $O(T \cdot sort(N))$ I/O operations.*

This result easily implies upper bounds on the complexity of a fully dynamic algorithm for connected components, once we know the complexity of the problem for the PRAM model satisfying the hypothesis of the theorem.

Ferragina [25] recalls a simple fully dynamic algorithm for maintaining a minimum spanning forest under edge updates running in $O(\log n)$ worst-case time using $O(m/\log n)$ processors, which satisfies the hypothesis of theorem 9.1. In fact, the algorithm applies two well-known parallel techniques on graphs, Euler tour on trees and list ranking, on inputs of length at most m. In [14] algorithms for Euler tour an list ranking of I/O complexity $O(sort(n))$, for inputs of length n, are given as consequences of theorem 9.1.

In our application, the graph G_δ has $O(N)$ edges. Hence, updating a minimum spanning forest for G_δ under edge updates takes $O(sort(N))$ I/Os in the worst case. Obviously, the connected components and of the forest are exactly the connected components of the graph G_δ and constitute an implicit representation of the updated clusters. However, if clusters must be stored in explicit form as rooted stars, in order to guarantee that connectivity queries can be answered in $O(1)$ I/Os, connected components in such explicit representation can be computed from the minimum spanning forest again in $O(sort(N))$ I/Os using the Euler tour technique.

10 Related Work

A detailed discussion of data clustering methods can be found in [12]. Here we recall only the works which deal with dynamic data.

IncrementalDBSCAN [22] is the dynamic version of the clustering algorithm DBSCAN [23]. IncrementalDBSCAN provides external memory clustering of metric data and is capable of recognizing clusters of arbitrary shape. Since it is based on the distance between points, it is hard to utilize in the kind of applications with categorical data mentioned in section 8.

The BUBBLE [27] algorithm implements a search tree in which leaves represent subclusters and points are iteratively inserted and placed into the leaf which minimizes some homogeneity function. This approach differs from ours in that it is only partially dynamic (it supports only insertions). Moreover, the homogeneity function is the distance from the metric center of a subcluster, thus inducing a bias towards finding globular clusters.

The BIRCH [60], algorithm is a precursor of BUBBLE for vector spaces. Therefore its applicability is confined to numerical data.

11 Conclusions

Although the data clustering problem in data mining has received considerable attention recently, most of the proposed algorithms are static. Dynamic algorithms are either partially dynamic algorithms, or fully dynamic algorithms which are not entirely satisfactory in important applications such as categorical clustering and approximate incremental clustering. In this report we have studied the applicability of three clustering methods from pattern recognition and statistics to the fully dynamic clustering problem. For each method, we have devised algorithms for updating clusters under single point insertions and deletions, and derived upper bounds to the time for computing each update, both in internal and in external memory.

In the future we plan to investigate the practical efficiency and the relative speedup attainable with respect to static implementations of the same methods.

Part III
Similarity queries (*Paolo Ciaccia and Marco Patella*)

12 Introduction

Similarity queries are frequently used in data exploration and mining. Often, in these applications, the user is not interested in a *correct* result, but he/she can accept an error on the result if such result can be (very) quickly obtained, thus trading the quality of the result for the search time. In [12, Part III], we reviewed and classified several solutions for solving *approximate similarity queries*, i.e. similarity queries where an error on the result is allowed: among all the considered solutions, we have elected the PAC approach [15] as the most promising one, most prominently because it is the only one which is able to offer a tangible cost/error trade-off, whereas other solutions are only able to give (non-tight) bounds on the quality of the result. The major limitation of the PAC approach, as presented in [15], is that it only deals with 1-NN queries: In this Part, we will extend the PAC approach to the most common types of similarity queries, namely k-NN, with $k > 1$, and range queries.

13 Background and General Definitions

In this Section we provide basic terminology and definitions. Relevant notation is summarized in Table 1.

The scenario we consider can be concisely described through the concept of *similarity workload*, first introduced in [44]:

Definition 1 (Similarity Workload) *A similarity workload is a quadruple, $\mathcal{W} = (\Omega, d, \mu, X)$, where:*

- *Ω is a (possibly infinite) domain of values;*
- *$d : \Omega \times \Omega \to \Re^+$ is a metric over Ω – a non-negative and symmetric function which satisfies the triangle inequality, $d(p_i, p_j) \leq d(p_i, p_k) + d(p_k, p_j)$ $\forall p_i, p_j, p_k \in \Omega$ – which measures the dissimilarity of points in Ω;*
- *μ is a probability measure over the metric space (Ω, d), which reflects the distribution of query points;*
- *X is a (finite) subset of Ω, called the data set.*

□

It has to be remarked that the class of metric spaces strictly includes that of vector spaces equipped with an L_p norm (e.g., L_2 is the Euclidean distance) or a quadratic-form distance function.

A *random query* $\mathbf{q}$[1] is a random variable, distributed according to μ, whose possible realizations are query points $q \in \Omega$. For a given query $q \in \Omega$, the minimum distance between q and a set $A \subseteq \Omega$ is defined as usual as:

$$d_A(q) = \inf\{d(q,p) \mid p \in A\}$$

Note that $d_A(q) = 0$ if $q \in A$ and that $A \subseteq B$ implies $d_B(q) \leq d_A(q)$. The (not necessarily unique) point of A for which the minimum distance is attained is called the *nearest neighbor* (NN) of q in A, denoted $p^1_A(q)$.

[1]Random variables are written in bold throughout the paper.

Symbol	Description
Ω	domain of values
d	distance function
μ	probability measure for queries
$X \subset \Omega$	finite data set (finite set of points from Ω)
$\mathcal{T}$	tree indexing the X data set
$\mathbf{q}$	random query point
$q \in \Omega$	query point (realization of $\mathbf{q}$)
$\mathcal{B}_r(q)$	r-ball of point q
$p^j_X(q)$	j-th nearest neighbor of point q in X
$\widetilde{p}^j_X(q)$	approximate j-th nearest neighbor of point q in X
N_i	i-th node of the index tree $\mathcal{T}$
Ω_i	data region spanned by node N_i
X_i	subset of X reachable from node N_i
$d_A(q)$	minimum distance between q and points in a set A
$d^-_A(q)$	lb on the minimum distance between q and points in A (also denoted as $\textsc{MinDist}(q, A)$)
$d^+_A(q)$	ub on the minimum distance between q and points in A (also denoted as $\textsc{MinMaxDist}(q, A)$)
Err	error of an approximate result
$cost$	cost of an algorithm (expressed as number of computed distances)
ϵ	accuracy parameter (relative error)
δ	confidence parameter
r^i_δ	i-th δ-radius for a k-NN query
k_δ	δ-cardinality of a range query

Table 1: Summary of relevant notation.

Our approach here is slightly different from that used in [15], in that there the probabilistic framework relied on the randomness of the distribution of points within the X data set (thus considering query points as fixed points), whereas here we completely ignore the *data* distribution and only take into account the distribution of query points [44].

13.1 Index Trees

We consider the case where the data set X is indexed by means of an index tree $\mathcal{T}$. The specific structure of the index tree is not relevant to what follows, in that our approach can be applied to *any* either multi-dimensional or metric hierarchical index structure that satisfies the following generic description.

Each *node* N_i (typically mapped to a disk page) in the tree $\mathcal{T}$ corresponds to a *data region*, $\Omega_i \subseteq \Omega$. Node N_i stores a set of entries, where each entry consists (at least) of a pointer to a child node N_c and of a specification of Ω_c, the data region of the child node. All indexed objects are stored in the leaf nodes of the tree, and the set X_i consisting of all the objects reachable from (a path starting from) node N_i is guaranteed to be contained in the data region Ω_i, i.e. $X_i \subseteq \Omega_i$.

Given a node N_i of $\mathcal{T}$, the *local* nearest neighbor of q in N_i is the point $p^1_{X_i}(q)$; its distance from q satisfies the inequality:

$$d_{\Omega_i}(q) \leq d_{X_i}(q) = d(q, p^1_{X_i}(q))$$

since $X_i \subseteq \Omega_i$. Given a description of Ω_i it is possible to bound from below and above the value of $d_{X_i}(q)$. For instance, if data regions are *balls*, i.e.:

$$\Omega_i = \mathcal{B}_{r_i}(c_i) \stackrel{\text{def}}{=} \{p \in \Omega \mid d(c_i, p) \leq r_i\} \tag{14}$$

then it is immediate to derive from the triangle inequality that $d(q, c_i) - r_i \leq d_{X_i}(q) \leq d(q, c_i) + r_i$.

In the following we denote with $d^-_{\Omega_i}(q)$ ($d^+_{\Omega_i}(q)$) the greatest lower bound (least upper bound, respectively) on $d_{X_i}(q)$ which can be derived given the specific description of Ω_i. Summarizing, we have:

$$d_{X_i}(q) \in [d^-_{\Omega_i}(q), d^+_{\Omega_i}(q)] \tag{15}$$

We will refer to $d^-_{\Omega_i}(q)$ and $d^+_{\Omega_i}(q)$ as the local NN distance lower bound and local NN distance upper bound, respectively.

The root of $\mathcal{T}$ is a node, conventionally denoted N_0, for which it is $X_0 \equiv X$. Clearly, the local NN of query q in N_0 is also the "global" NN of q in the X data set. Without loss of generality we assume that $d^-_{\Omega_0}(q) = 0$ and use d^+ as a shorthand for $d^+_{\Omega_0}(q)$. In practice, d^+ is a, possibly infinite, upper bound on the maximum value of the NN distance, considering all possible queries q. Practically, d^+ can be set to a finite value when the metric space is *bounded*, that is $d(q, p) \leq d^+ < \infty$ holds $\forall q, p \in \Omega$.

Example 13.1 *The R-tree [33] family of spatial access methods organizes D-dimensional points by means of Minimum Bounding Rectangles (MBRs). This principle is applied at all levels of the tree, so that each data region Ω_i is a (hyper-)rectangle in a D-dimensional Cartesian space. Details to compute both $d^-_{\Omega_i}(q)$ and $d^+_{\Omega_i}(q)$ are given in [45], where such bounds are respectively termed* MinDist(q, Ω_i) *and* MinMaxDist(q, Ω_i). □

Example 13.2 *The M-tree [16] access method can index objects over generic metric spaces. Given a distance function d, the region of a node N_i is a metric ball (see Equation 14) whose description is $[c_i, r_i]$, i.e. the "center point" of the ball and the "radius" of the ball.*[2] *Thus, all the objects reachable from node N_i are within distance r_i from c_i. Note that the actual "shape" of data regions depends on the specific metric space (Ω, d). For instance, regions are "diamonds" in $(\Re^2, L_1)$, circles in $(\Re^2, L_2)$, and squares in $(\Re^2, L_\infty)$.*

Lower and upper bounds on $d_{X_i}(q)$ are computed as follows. For the lower bound we have:

$$d^-_{\Omega_i}(q) = \min\{0, d(q, c_i) - r_i\} \tag{16}$$

which always holds due to triangle inequality. It has to be remarked that although in many cases $d_{\Omega_i}(q) = d^-_{\Omega_i}(q)$ holds, this is not true *in general. For instance, if one considers normalized D-dimensional vectors, that is, the unit sphere $\Omega \equiv S^{D-1} = \{p \in \Re^D \mid |p| = 1\}$, and $d \equiv L_2$, one has $d_{\Omega_i}(q) > d^-_{\Omega_i}(q)$ whenever $r_i > 0$. In general equality holds only for so-called* path metric spaces *(p-MS), which include vector spaces equipped with L_p norms [31].*

For the upper bound two cases can be considered. When no deletions occur, *a center coincides with (is a copy of) an indexed database object [16], thus:*

$$d^+_{\Omega_i}(q) = d(q, c_i)$$

On the other hand, in the general case where objects can be deleted, since there is no further guarantee that c_i still coincides with an existing object, the correct upper bound is:

$$d^+_{\Omega_i}(q) = d(q, c_i) + r_i$$

□

[2] The center point is also called a "routing object" in [16].

13.2 Similarity Queries

We consider two basic types of queries that can be issued within a Similarity Workload (see Definition 1): range and (k-)nearest neighbor queries.

Definition 2 (Range Query) *Given a similarity workload $\mathcal{W} = (\Omega, d, \mu, X)$, let $\mathbf{q}$ a random variable distributed according to μ, q be a realization of $\mathbf{q}$, and r a real value, a* range query $\texttt{range}_X(q, r)$ *selects all those $p \in X$ such that $d(q, p) \leq r$.* □

Definition 3 (k-Nearest Neighbor (k-NN) Query) *Given a similarity workload $\mathcal{W} = (\Omega, d, \mu, X)$, let $\mathbf{q}$ a random variable distributed according to μ, q be a realization of $\mathbf{q}$, and k an integer value, a* k-nearest neighbor (k-NN) query $\texttt{NN}_X(q, k)$ *selects the k points in X having minimum distance d from q.* □

14 The PAC approach

Approximate evaluation of similarity queries requires the definition of an error measure [12, Part III]. The PAC approach differs from correct (where no error is allowed) and approximately correct (where the error cannot exceed a specified bound) search because it allows to exceed the error bound ϵ with a certain probability δ:

Definition 4 *Given a similarity query, an accuracy parameter ϵ, and a confidence parameter $\delta \in [0, 1)$, an algorithm is PAC (probably approximately correct) if, given a (suitable) definition of error, Err, it is guaranteed that*

$$\boxed{\Pr\{\mathbf{Err} > \epsilon\} \leq \delta} \tag{17}$$

□

It has to be noted that the above definition encompasses both correct and approximately correct search:

- When $\epsilon = 0$ and $\delta = 0$ we obtain correct search, since we do not allow to exceed an error $Err = 0$.
- When $\epsilon > 0$ and $\delta = 0$ we obtain approximately correct search, since we accept an error $\epsilon > 0$, but do not allow to exceed it.
- Finally, when $\epsilon = 0$ and $\delta > 0$, we allow an error only in a fraction δ of the cases, on the average.

The specification of PAC algorithms, therefore, relies on the possibility of defining a suitable measure of error and on finding a way to limit such error in a probabilistic way.

Example 14.1 *In [15], PAC 1-NN queries are considered, with error defined as*

$$\boxed{Err = \frac{d\left(q, \widetilde{p}_X^1(q)\right)}{d\left(q, p_X^1(q)\right)} - 1} \tag{18}$$

where $p_X^1(q)$ is the nearest neighbor of q in X and $\widetilde{p}_X^1(q)$ is the result provided by the PAC algorithm. The probability to exceed the maximum error is bounded by using information on the distance distribution of the nearest neighbor [17] (see also Section 14.1.1). □

Algorithms for solving PAC queries are similar to those used for solving correct similarity queries, but are able to terminate the search over the data set as soon as a "sufficiently good" solution is found. In particular, the algorithm can be stopped whenever the probability of exceeding the error bound is not higher than δ: This *stopping condition*, that is used to avoid continuing the search when it is unlikely to found better solutions (i.e. lower errors), is determined by considering probabilistic information on how the expected result is distributed.

Since we are dealing with generic metric spaces, a reasonable choice is to exploit the distribution of distances with respect to the query object [17]. For example, such information was used in [15] to derive a suitable stopping condition for PAC 1-NN search (see Section 14.1.1). Formally, let $F(\cdot)$ be the distance distribution corresponding to distance d, that is

$$\boxed{F(x) = \Pr\{d(\mathbf{q}, p) \leq x\}} \tag{19}$$

where p is a point of X and $\mathbf{q}$ is a random point in Ω distributed according to μ.[3] In this light, PAC algorithms can therefore be devised whenever it is possible to use $F()$ to characterize the (distribution of) error **Err**.

14.1 PAC Queries

In this Section we will show how the PAC approach can be applied to the types of similarity queries defined in Section 13.2.

14.1.1 1-NN Queries

This basic case has been dealt in [15], where the error for a given query $\mathtt{NN}_X(q,1)$ is defined as in Equation 18. This appears to be a reasonable definition of error, and it is also used by other approximately correct approaches [6, 59]. The stopping condition is derived by considering the distance distribution of the nearest neighbor of q, which is given as [17]

$$G^1(x) \stackrel{\text{def}}{=} \Pr\left\{d\left(\mathbf{q}, p_X^1(\mathbf{q})\right) \leq x\right\} = 1 - (1 - F(x))^N \tag{20}$$

where $N = |X|$ is the number of objects in the X data set. $G^1(x)$ represents the probability that the nearest neighbor of q in X is at a distance not higher than x from q, or, equivalently, the probability that at least 1 point from X is at a distance not higher than x from q. Considering $G^1(\cdot)$, we can compute the so-called *δ-radius* as:

$$r_\delta^1 \stackrel{\text{def}}{=} sup\left\{r \middle| \Pr\left\{d\left(\mathbf{q}, p_X^1(\mathbf{q})\right) \leq r\right\} \leq \delta\right\} = sup\left\{r \middle| G^1(r) \leq \delta\right\} \tag{21}$$

Obviously, if $G^1(\cdot)$ is invertible, it is $r_\delta^1 = (G^1)^{-1}(\delta)$. r_δ^1 is the maximum distance r for which the probability that the NN of q is at a distance lower than r is not higher than δ. Therefore, if a point p' is found at a distance lower than r_δ^1, the probability that this is not the correct NN for q is not higher than δ. By scaling the δ-radius by a factor $(1+\epsilon)$, we accept an error of ϵ with probability lower than δ. The 1-NN search can thus be stopped as soon as we find an object p' whose distance from q satisfies $d(q,p') \leq (1+\epsilon)r_\delta^1$.

14.1.2 k-NN Queries

In order to generalize the previous approach to the case of k-NN queries, with $k > 1$, we first have to find a suitable definition of error for which Equation 18 is a particular case. Of course, such definition should also allow us to easily find a probabilistic stopping condition based on

[3]For simplicity, here and in the following the dependence of probability distributions on the X data set is understood in the notation.

$F(\cdot)$. If $p_X^i(q)$ is the i-th nearest neighbor of q in X, and $\widetilde{p}_X^i(q)$ is the approximate i-th NN provided by the PAC algorithm, the error on the i-th nearest neighbor is defined as

$$Err_i = \frac{d\left(q, \widetilde{p}_X^i(q)\right)}{d\left(q, p_X^i(q)\right)} - 1$$

Reasonable definitions of error for a k-NN query include:

- The maximum between the errors Err_i, $1 \leq i \leq k$.
- The average of errors Err_i.
- The error Err_k on the k-th nearest neighbor only.

To derive a suitable stopping condition for the k-NN PAC algorithm, we need to compute the distribution of the k $\mathbf{Err}_i$ random variables. In particular, given a vector of values $\{x_1, \ldots, x_k\}$, with $x_i \leq x_j, \forall i < j$, the probability that, for each i, $d\left(\mathbf{q}, p_X^i(\mathbf{q})\right) \leq x_i$ has to be found. The *combined k-nearest neighbor distribution* $G^{1,\cdots,k}(x_1, \ldots, x_k)$ expresses the probability that each i-th nearest neighbor is at a distance from q not higher than x_i:

$$\boxed{G^{1,\cdots,k}(x_1, \ldots, x_k) \stackrel{\text{def}}{=} \Pr\left\{d\left(\mathbf{q}, p_X^1(\mathbf{q})\right) \leq x_1, \ldots, d\left(\mathbf{q}, p_X^k(\mathbf{q})\right) \leq x_k\right\}} \tag{22}$$

Intuitively, $G^{1,\cdots,k}(x_1, \ldots, x_k)$ represents the probability of finding at least i objects at a distance not higher than x_i from q, $\forall i = 1, \ldots, k$ (see Figure 6).

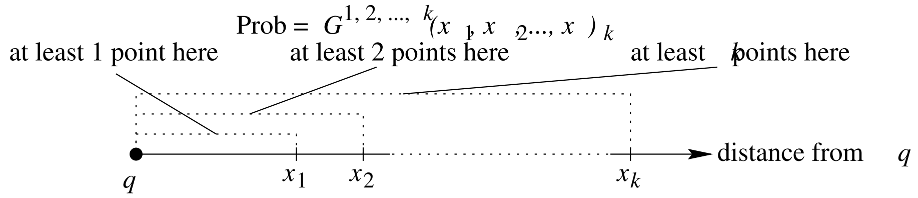

Figure 6: The combined k-nearest neighbor distribution $G^{1,\cdots,k}(x_1, \ldots, x_k)$.

Given a definition of error
error, to evaluate if a vector $\{x_1, \ldots, x_k\}$ satisfies Equation 17 we have to find the density of $G^{1,\cdots,k}(x_1, \ldots, x_k)$, $g^{1,\cdots,k}(x_1, \ldots, x_k) = \frac{\partial G^{1,\cdots,k}(x_1,\ldots,x_k)}{\partial x_1 \cdots \partial x_k}$, and integrate it over the possible domain of values. Analytical computation of $G^{1,\cdots,k}(x_1, \ldots, x_k)$, however, is impractical for large values of k; here we only give the analytical formula for the cases $k = 2$ and $k = 3$.

Lemma 14.2 *For $k = 2$ it is*

$$G^{1,2}(x_1, x_2) = 1 - (1 - F(x_1))^N - \binom{N}{1} F(x_1)(1 - F(x_2))^{N-1}$$

whereas, for $k = 3$, we obtain

$$\begin{aligned} G^{1,2,3}(x_1, x_2, x_3) = 1 - (1 - F(x_1))^N - \binom{N}{1} F(x_1)(1 - F(x_2))^{N-1} \\ - \binom{N}{2} F(x_1)(2F(x_2) - F(x_1))(1 - F(x_3))^{N-2} \end{aligned}$$

Proof: Omitted. $\square$

The impracticability of expressing $G^{1,\cdots,k}(x_1,\ldots,x_k)$ prevents us from using arbitrary definitions of *Err* for k-NN queries. However, for "simple" definitions, something can still be done: In particular, when using the maximum error, it is possible to derive a simple stopping condition not involving the joint distribution. Suppose that we have the following definition of error

$$\boxed{Err = \max_{i=1}^{k}\{Err_i\} = \max_{i=1}^{k}\left\{\frac{d\left(q,\widetilde{p}_X^i(q)\right)}{d\left(q,p_X^i(q)\right)} - 1\right\}} \tag{23}$$

Let us define the distance distribution of the i-th nearest neighbor $G^i(x)$ as

$$G^i(x) \stackrel{\text{def}}{=} \Pr\left\{d\left(\mathbf{q},p_X^i(\mathbf{q})\right) \le x\right\} = 1 - \sum_{j=0}^{i-1}\binom{N}{j}F(x)^j(1-F(x))^{N-j} \tag{24}$$

$G^i(x)$ expresses the probability to find at least i objects at a distance not higher than x from q. Of course, if $i \ge j$ it is $G^i(x) \le G^j(x)$.

Given a value of δ, we can compute the i-th δ-radius, r_δ^i as

$$r_\delta^i \stackrel{\text{def}}{=} sup\left\{r|\Pr\left\{d\left(\mathbf{q},p_X^i(\mathbf{q})\right) \le r\right\} \le \delta\right\} = sup\left\{r|G^i(r) \le \delta\right\} \tag{25}$$

From the monotonicity of $G^i(x)$ in i, it immediately follows that if $i \le j$ it is $r_\delta^i \le r_\delta^j$. Again, if $G^i(\cdot)$ is invertible, it is $r_\delta^i = (G^i)^{-1}(\delta)$. Each r_δ^i value represents the maximum r value for which the probability that at least i objects have a distance not higher than r from q is not higher than δ (see Figure 7).

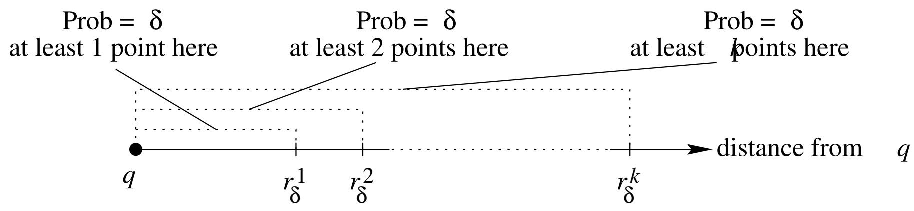

Figure 7: The meaning of the δ-radius, r_δ^i, for a given value of confidence δ.

Since we want to limit the maximum error to ϵ, each δ-radius should not be exceeded more than $(1+\epsilon)$, thus we can stop the search as soon as we find i objects at a distance not higher than $(1+\epsilon)r_\delta^i$, $\forall i = 1,\ldots,k$. In order to demonstrate the correctness of the stopping condition, we have the following preliminary result.

Lemma 14.3 *Given a value x, the probability $G^{1,\cdots,k}(x,\ldots,x)$ is equal to the probability $G^k(x)$:*

$$G^{1,\cdots,k}(x,\ldots,x) = G^k(x)$$

whereas the probability $G^{1,\cdots,k}\left(\underbrace{x,\ldots,x}_{i},\underbrace{d^+,\ldots,d^+}_{k-i}\right)$ *is equal to the probability $G^i(x)$:*

$$G^{1,\cdots,k}\left(\underbrace{x,\ldots,x}_{i},\underbrace{d^+,\ldots,d^+}_{k-i}\right) = G^i(x)$$

Proof: $G^{1,\cdots,k}(x,\ldots,x)$ is the probability that at least 1 object is at a distance not higher than x from $\mathbf{q}$, *and* at least 2 object are at a distance not higher than x from $\mathbf{q}$, *and* ..., *and* at least k object are at a distance not higher than x from $\mathbf{q}$. Such probability is trivially equal to the probability that at least k object are at a distance not higher than x from $\mathbf{q}$. The first result immediately follows from the definition of $G^k(x)$ (see Equation 24). Similar arguments are applied to derive the second result, since we are considering only the i objects closer to $\mathbf{q}$. □

The following Theorem expresses the stopping condition for k-NN queries when the error is defined as in Equation 23.

Theorem 14.4 *If the definition of error for k-NN queries is given by Equation 23, then, given the query* $\mathtt{NN}_X(q,k)$, *the search can be stopped as soon as we find k objects* $\widetilde{p}_X^1(q),\ldots,\widetilde{p}_X^k(q)$, *with* $d\left(q,\widetilde{p}_X^1(q)\right) \le d\left(q,\widetilde{p}_X^2(q)\right) \le \ldots \le d\left(q,\widetilde{p}_X^k(q)\right)$, *such that*

$$d\left(q,\widetilde{p}_X^i(q)\right) \le (1+\epsilon)r_\delta^i \tag{26}$$

Proof: First, let $\epsilon = 0$. In this case, let $x_i = d\left(q,\widetilde{p}_X^i(q)\right)$: The probability to find a closer point (i.e. to have an error $\mathbf{Err}_i$ higher than $\epsilon = 0$) is therefore given as $G^{1,\cdots,k}\left(\underbrace{x_i,\ldots,x_i}_{i},\underbrace{d^+,\ldots,d^+}_{k-i}\right)$ which, for Lemma 14.3, is equal to $G^i(x_i)$. The result immediately follows from the definition of δ-radius, r_δ^i, in Equation 25, since we are considering only the maximum between the k error values Err_i. When $\epsilon > 0$, it is sufficient to appropriately scale the i-th δ-radius. □

Theorem 14.4 gives us the stopping condition for k-NN queries when the error is defined through Equation 23. However, this requires the computation of the k δ-radii, r_δ^i for each $i = 1,\ldots,k$, and to check k conditions to interrupt the search. A simpler definition of error, which leads to a single stopping condition, is the following:

$$\boxed{Err = \frac{d\left(q,\widetilde{p}_X^k(q)\right)}{d\left(q,p_X^k(q)\right)} - 1} \tag{27}$$

which means that we only take into account the error on the k-th nearest neighbor, ignoring the errors on each i-th nearest neighbor for $i < k$. The probabilistic stopping condition can be found by considering the distance distribution of the k-th nearest neighbor of $\mathbf{q}$, $G^k(\cdot)$, which is given in Equation 24. From this, the corresponding δ-radius, r_δ^k, can be computed as in Equation 25, and the search can be stopped as soon as the distance between q and its k-th nearest neighbor found so far is not higher than $(1+\epsilon)r_\delta^k$.

14.1.3 Range Queries

In k-NN queries the cardinality of the result set is given by the user, thus the error obtained by an approximate algorithm has to take into account differences in distance values. For range queries, on the other hand, the (maximum) distance is set by the user, thus the error should consider the number of returned objects. Denoting the number of objects returned by a range query $\mathtt{range}_X(q,r)$ as $k_X(q,r) = |\{p \in X | d(q,p) \le r\}|$, the error is thus defined as

$$\boxed{Err = 1 - \frac{\widetilde{k}_X(q,r)}{k_X(q,r)}} \tag{28}$$

From Definition 4 and Equation 28 it is clear that the search can be stopped as soon as we find a number of objects k such that

$$\Pr\left\{k_X(\mathbf{q},r) > \frac{k}{1-\epsilon}\right\} \le \delta$$

Thus, to find a suitable stopping condition, we have to find the distribution of the random variable $k_X(\mathbf{q}, r)$. This can be easily obtained by considering the search as a repetition of N Bernoulli trials, each with a probability of success equal to $F(r)$. The probability that the number of retrieved object is k is therefore given as:

$$p_k \stackrel{\text{def}}{=} \Pr\{k_X(\mathbf{q}, r) = k\} = \binom{N}{k} F(r)^k (1 - F(r))^{N-k} \tag{29}$$

Thus it is

$$\Pr\{k_X(\mathbf{q}, r) \leq k\} = \sum_{i=0}^{k} p_i = \sum_{i=0}^{k} \binom{N}{i} F(r)^i (1 - F(r))^{N-i}$$

and the search can be stopped as soon as

$$\Pr\{k_X(\mathbf{q}, r) \leq k\} = \sum_{i=0}^{k} \binom{N}{i} F(r)^i (1 - F(r))^{N-i} \geq 1 - \delta \tag{30}$$

It has to be noted that we can compute the minimum value of k such that Equation 30 is satisfied: Such value, which is similar to the δ-radius of k-NN search and that we call *δ-cardinality*, denoted k_δ, can be defined as

$$\begin{aligned} k_\delta &\stackrel{\text{def}}{=} \inf\{k \mid \Pr\{k_X(\mathbf{q}, r) > k\} \leq \delta\} \qquad (31) \\ &= \inf\left\{k \left| \sum_{i=0}^{k} \binom{N}{i} F(r)^i (1 - F(r))^{N-i} \geq 1 - \delta \right.\right\} \end{aligned}$$

and can be computed before starting the search. Any PAC algorithm can therefore stop the search whenever $\widetilde{k}_X(q, r) \geq (1 - \epsilon) k_\delta$.

It has to be noted that our definition of PAC range queries does not encompass the case of approximate range queries defined in [5], since we consider the error on the cardinality of the result, whereas in [5] this is the output and the error is committed by appropriately enlarging or reducing the query region.

15 PAC Algorithms

As said in previous Section, PAC algorithms are nothing but correct algorithms equipped with a stopping condition that allows us to interrupt the search whenever the actual result is "good enough", i.e. when the probability to find a better result (a nearest objects for k-NN queries or more relevant objects for range queries) is lower than the confidence parameter δ. The stopping conditions expressed in Sections 14.1.2 and 14.1.3 for k-NN and range queries respectively can be used to this purpose.

15.1 Sequential PAC Algorithms

Sequential algorithms can be used whenever the data set X is stored in a simple sequential file and no index is available. PAC algorithms, however, are able to stop the file scan as soon as the stopping condition is verified. The expected cost, measured as the number of distance computations (*probes*), is estimated by considering a *random sampling process with repetitions* (i.e. a point can be probed more than once). This is an adequate model as long as there is no correlation between the distances of the points to q and their positions in the file, N is large, and the estimated cost is (much) lower than N. On the other hand, when the analysis derives that the cost is comparable to N, then predictions deviate from the actual performance and only provide a (non-tight) upper bound of the cost.

15.1.1 Sequential PAC Range Algorithm

Given a range query $\texttt{range}_X(q,r)$, the sequential algorithm solves it by scanning the entire file and checking, for each accessed object, if the query predicate is satisfied: If this is the case, the object is put in the result set. Since the probability of success is $F(r)$, the number of trials needed to have an expected number of successes equal to $(1-\epsilon)k_\delta$ is equal to $(1-\epsilon)k_\delta/F(r)$. Therefore, for the PAC range sequential algorithm, the expected cost is

$$\boxed{E\left[\mathbf{cost}\right] = \frac{(1-\epsilon)k_\delta}{F(r)}} \tag{32}$$

Equation 32 exhibits the fact that the cost is not directly related to values of ϵ and δ, thus varying such values will not influence the search cost as long as the value of $(1-\epsilon)k_\delta$ stays constant. Of course, reducing the value of ϵ or of δ would increase the value of $(1-\epsilon)k_\delta$, and thus of searching costs, as it can be intuitively argued.

15.1.2 Sequential PAC k-NN Algorithm

Given a k-NN query $\texttt{NN}_X(q,k)$, a sequential algorithm would scan the entire file and maintain an (ordered) array of the best k objects encountered so far. If we use the definition of error given by Equation 27, then the search is stopped as soon as we find k objects having a distance from q not higher than $(1+\epsilon)r_\delta^k$, with r_δ^k given by Equation 25. The probability of success for each probe is now $F\left((1+\epsilon)r_\delta^k\right)$ and we have to find exactly k successes, therefore the expected cost is

$$\boxed{E\left[\mathbf{cost}\right] = \frac{k}{F\left((1+\epsilon)r_\delta^k\right)}} \tag{33}$$

which is consistent with results obtained in [15] for the case $k=1$. Reducing the value of ϵ or of δ will reduce the value of $(1+\epsilon)r_\delta^k$, thus increasing search costs, since $F(\cdot)$ is a non-decreasing function and appears at the denominator in Equation 33.

If the definition of error is given as in Equation 23, the sequential scan can be stopped as soon as all the k stopping conditions are verified. However, now it is not easy to find a simple cost formula like Equation 33, since the probability of success of each probe depends on the combined k-NN distance distribution $G^{1,\cdots,k}(x_1,\ldots,x_k)$ which, as said in Section 14.1.2, cannot be easily found for arbitrary values of k.

15.2 Solving PAC Queries with Index Trees

Sequential algorithms for solving PAC queries are not amenable when the number of objects in the data set N is very large, especially when ϵ and δ have both small values. To reduce search times with respect to the sequential case for correct similarity queries, index trees are commonly used. In the following we provide search algorithms to solve PAC queries with any index tree $\mathcal{T}$ satisfying the description in Section 13.1.

15.2.1 Index PAC Range Algorithm

The PAC range algorithm recursively descends the tree only accessing those nodes that can lead to objects relevant for the $\texttt{range}_X(q,r)$ query. The search starts at the root node and ends whenever enough objects are found (PAC stopping condition of Equation 30), or when no node for which $d_{\Omega_i}^-(q) \le r$ can be found. The $\texttt{PAC-range}$ algorithm in Figure 8 is therefore applied to the root node N_0 of $\mathcal{T}$ and returns a set of objects $\mathcal{P}$ for which is guaranteed that $\Pr\{k_X(\mathbf{q},r) > |\mathcal{P}|/(1-\epsilon)\} \le \delta$.

Correctness of the $\texttt{PAC-range}$ index algorithm of Figure 8 is demonstrated by the following

Algorithm `PAC-range`

Input: query point q, query radius r, node N, error bound ϵ, confidence δ;
Output: set of points $\mathcal{P}$, $(1+\epsilon)$-approximate result of $\texttt{range}_X(q,r)$ with probability $\geq 1-\delta$;

If N is a leaf then:
    For each point p in N do:
        If $d(q,p) \leq r$ then:
            Add p to the result set $\mathcal{P}$;
            If $|\mathcal{P}| \geq (1-\epsilon)k_\delta$ STOP;
else: // N is an internal node
    For each child node N_i of N do:
        If $d^-_{\Omega_i}(q) \leq r$ then:
            READ(N_i);
            INVOKE `PAC-range`$(q, r, N_i, \epsilon, \delta)$ and add the result of to the result set;
End.

Figure 8: The `PAC-range` algorithm.

Theorem 15.1 (`PAC-range` correctness) *Algorithm `PAC-range` returns a number of object for which the error is lower than ϵ with probability at least $1-\delta$.*

Proof: If `PAC-range` stops because it has completely explored the tree, then the result is guaranteed to be correct since no probability is involved at all. Therefore, let us assume that we stopped the search because we found a number of objects k which is greater than or equal to $(1-\epsilon)k_\delta$: We have to show that $\Pr\{k_X(\mathbf{q},r) > k/(1-\epsilon)\} \leq \delta$. Since $k \geq k_\delta(1-\epsilon)$ holds, then it is $\Pr\{k_X(\mathbf{q},r) > k/(1-\epsilon)\} \leq \Pr\{k_X(\mathbf{q},r) > k_\delta\}$. The result follows from the definition of k_δ in Equation 31. □

15.2.2 Index PAC k-NN Algorithm

The PAC k-NN index algorithm follows the same steps of the correct NN index algorithm. Such algorithm was first presented for the PMR-Quadtree [35] and then generalized in [10] to work with *any* index tree which respects the generic description provided in Section 13.1. The algorithm uses a priority queue, `PQ`, whose entries have type

$$[d^-_{\Omega_i}(q), d^+_{\Omega_i}(q), ptr(N_i)]$$

where $ptr(N_i)$ is a pointer to node N_i. The generality of the algorithm stems from the fact that the only part that depends on the specific index at hand is the computation of $d^-_{\Omega_i}(q)$ and $d^+_{\Omega_i}(q)$ bounds.

Any (correct or approximate) algorithm for solving nearest neighbor queries by means of an index tree $\mathcal{T}$ has necessarily to implement a *scheduling policy* to determine the order according to which `PQ` is ordered, i.e. how index nodes have to be visited.[4] It is clear that, for a given query, different policies can lead to different performance. In general, we say that a policy is *optimal* for a given query if no other policy can do better for that query, where performance is measured by the number of read nodes.

A *schedule* Π is a policy for sequencing the retrieval of nodes. For the moment assume that all nodes are leaves, i.e. the tree consists of a root and of M leaves.[5] Since the root is not a

[4] This does not apply to range queries, because in that case the resulting objects are not ordered.

[5] This is only required to prove optimality of the schedules. PAC k-NN algorithms work for trees having height > 1.

concern for scheduling purposes, a schedule Π can be simply modelled as a permutation of the set $\{1, \ldots, M\}$:

$$\Pi = (\Pi_1, \Pi_2, \ldots, \Pi_i, \ldots, \Pi_M) \quad (34)$$

where N_{Π_i} is the leaf that schedule Π will fetch at step i (provided it does not stop earlier). Two well-known policies, first introduced in [45], are based on the values of $d^-_{\Omega_i}(q)$ and $d^+_{\Omega_i}(q)$.

Definition 5 (MinDist scheduling policy) *The* MinDist *scheduling policy sorts nodes by increasing values of $d^-_{\Omega_i}(q)$, thus it first accesses that node for which $d^-_{\Omega_i}(q)$ is minimal.*

Definition 6 (MinMaxDist scheduling policy) *The* MinMaxDist *scheduling policy sorts nodes by increasing values of $d^+_{\Omega_i}(q)$, thus it first accesses that node for which $d^+_{\Omega_i}(q)$ is minimal.*

Intuitively, the MinDist is an "optimistic" scheduling policy since it privileges nodes that may contain objects closer to q, whereas using MinMaxDist those nodes are favored that surely contain objects closer to q. In [10] it is proved that the MinDist scheduling policy is optimal for correct k-NN search provided that $d_{\Omega_i}(q) = d^-_{\Omega_i}(q)$. If, however, only $d_{\Omega_i}(q) \geq d^-_{\Omega_i}(q)$ holds, no guarantee of optimality can be given. The `PAC-k-NN` algorithms presented in this Section are parametric in the scheduling policy Π, so that the best schedule for the case at hand can be chosen at query time, depending on the information available (see Section 15.3).

Let us first consider the definition of error of Equation 27. The $\texttt{PAC-k-NN}_k$ in Figure 9 can be used to solve PAC k-NN queries: The r variable represents the "current search radius", and it is decreased whenever a point closer to q than its k-th NN is found. As in the `PAC-range` algorithm, any node N_i whose region Ω_i is distant from q at least r can be safely pruned from the search since it cannot lead to better objects.

Correctness of the $\texttt{PAC-k-NN}_k$ index algorithm of Figure 9 is demonstrated by the following

Theorem 15.2 ($\texttt{PAC-k-NN}_k$ correctness) *Algorithm $\texttt{PAC-k-NN}_k$ returns k objects for which the error is lower than ϵ with probability at least $1 - \delta$.*

Proof: If $\texttt{PAC-k-NN}_k$ stops because `PQ` becomes empty, then the result is guaranteed to be $(1+\epsilon)$-approximately correct since no probability is involved at all. Therefore, let us assume that we stopped the search because we found k objects, $p^1, \ldots, p^k$, such that $d\left(q, p^k\right)$ is lower than or equal to $(1+\epsilon)r^k_\delta$: According to Equation 24, we have to show that $\Pr\left\{d\left(\mathbf{q}, p^k_X(\mathbf{q})\right) < r/(1+\epsilon)\right\} \leq \delta$. Since $r \leq (1+\epsilon)r^k_\delta$ holds, then it is $\Pr\left\{d\left(\mathbf{q}, p^k_X(\mathbf{q})\right) < r/(1+\epsilon)\right\} \leq \Pr\left\{d\left(\mathbf{q}, p^k_X(\mathbf{q})\right) < r^k_\delta\right\}$. The result follows from the definition of r^k_δ in Equation 25. □

Now, suppose that the definition of error for a k-NN query is given as in Equation 23. The algorithm should now take into account the k δ-radii and the search can be stopped only when all the k stopping conditions are met. The $\texttt{PAC-k-NN}_{max}$ in Figure 10 differs from the $\texttt{PAC-k-NN}_k$ algorithm in that the stopping condition consists in comparing the distances between q and each i-th NN with the δ-radii (line 10. of the algorithm).

Correctness of the $\texttt{PAC-k-NN}_{max}$ index algorithm of Figure 10 is demonstrated by the following

Corollary 15.3 ($\texttt{PAC-k-NN}_{max}$ correctness) *Algorithm $\texttt{PAC-k-NN}_{max}$ returns k objects for which the error is lower than ϵ with probability at least $1 - \delta$.*

Proof: If $\texttt{PAC-k-NN}_{max}$ stops because `PQ` becomes empty, then the result is guaranteed to be $(1+\epsilon)$-approximately correct since no probability is involved at all. On the other hand, if the algorithm stops because the stopping condition holds, then correctness immediately follows from Theorem 14.4. □

Algorithm `PAC-k-NN`$_k$

Input: query point q, query cardinality k, index tree $\mathcal{T}$ with root node N_0, error bound ϵ, confidence δ;

Output: vector of points $\mathcal{P} = \{p^1, \ldots, p^k\}$, $(1+\epsilon)$-approximate result of $\mathtt{NN}_X(q,k)$ with probability $\geq 1-\delta$;

INSERT(PQ, $[0, d^+, ptr(N_0)]$, Π); // Initialize PQ with a pointer to the root of $\mathcal{T}$
Compute r_δ^k; Let $r = d^+$;
While PQ $\neq \emptyset$ do:
    $[d_{\Omega_i}^-(q), d_{\Omega_i}^+(q), ptr(N_i)] =$ POP(PQ); // Extract the first entry from PQ
    READ(N_i);
    If N_i is a leaf node then:
        For each point p in N_i do:
            If $d(q,p) < r$ then:
                INSERT($\mathcal{P}, p$), let $r = d(q, p^k)$;
                If $r \leq (1+\epsilon) r_\delta^k$ then STOP;
                else PRUNE(PQ, $r/(1+\epsilon)$);
    else: // N_i is an internal node
        For each child node N_c of N_i do:
            If $d_{\Omega_c}^-(q) < r/(1+\epsilon)$ then:
                INSERT(PQ, $[d_{\Omega_c}^-(q), d_{\Omega_c}^+(q), ptr(N_c)]$, Π);
                    // Insert N_c's entry in PQ respecting Π policy
End.

Figure 9: The `PAC-k-NN`$_k$ algorithm used when the error is defined as in Equation 27.

15.3 Optimal Scheduling

In the previous Section, we have seen that optimal schedules exist for the case of correct k-NN search. However, for approximate k-NN search *no proof of optimality for* MINDIST *or other scheduling policies exists.* Furthermore, our experience with high-dimensional data sets shows that in such scenarios so many nodes have $d_{\Omega_i}(q) = 0$ (since $q \in \Omega_i$) that MINDIST policy loses interest, since it cannot discriminate between such regions.

Given a schedule Π, we want to compute the probability, $S_\Pi(j)$, that the search algorithm will stop in no more than j steps, i.e. after having fetched j leaves in the order dictated by schedule Π. This information is then used to compute the expected search cost to be paid if Π is used and, in turn, to investigate the existence of an optimal schedule. In order to compute $S_\Pi(j)$ we need some knowledge about distribution of the nearest neighbors in nodes of $\mathcal{T}$. Since, however, the `PAC-k-NN` algorithms proposed in Section 15.2.2 does not use any *local* information, we need a further assumption to derive a provably optimal schedule for them.[6]

Assumption 1 (Shifting & Scaling - S&S) *All local distributions are "similar", in that they can be obtained just by shifting and scaling a "normalized" distribution, $\mathcal{N}(\cdot)$, defined as:*

$$\boxed{\mathcal{N}(r) \stackrel{def}{=} \Pr\left\{ d_{X_i}(\mathbf{q}) \leq l_i + r(u_i - l_i) \mid d_{\Omega_i}^-(\mathbf{q}) = l_i, d_{\Omega_i}^+(\mathbf{q}) = u_i \right\}} \qquad (35)$$

□

[6]For simplicity, in the following we concentrate on the `PAC-k-NN`$_k$ algorithm.

Algorithm $\texttt{PAC-k-NN}_{max}$

Input: query point q, query cardinality k, index tree $\mathcal{T}$ with root node N_0, error bound ϵ, confidence δ;

Output: vector of points $\mathcal{P} = \{p^1, \ldots, p^k\}$, $(1+\epsilon)$-approximate result of $\texttt{NN}_X(q,k)$ with probability $\geq 1-\delta$;

Insert($\texttt{PQ}$, $[0, d^+, ptr(N_0)]$, Π); // Initialize $\texttt{PQ}$ with a pointer to the root of $\mathcal{T}$
For each $j = 1, \ldots, k$, compute r_δ^j; Let $r = d^+$;
While $\texttt{PQ} \neq \emptyset$ do:
  $[d^-_{\Omega_i}(q), d^+_{\Omega_i}(q), ptr(N_i)] =$ Pop($\texttt{PQ}$); // Extract the first entry from $\texttt{PQ}$
  Read(N_i);
  If N_i is a leaf node then:
    For each point p in N_i do:
      If $d(q,p) < r$ then:
        Insert($\mathcal{P}, p$), let $r = d(q, p^k)$;
        If for each $j = 1, \ldots, k$, $d\left(q, p^j\right) \leq (1+\epsilon) r_\delta^j$ then Stop;
        else Prune($\texttt{PQ}$, $r/(1+\epsilon)$);
  else: // N_i is an internal node
    For each child node N_c of N_i do:
      If $d^-_{\Omega_c}(q) < r/(1+\epsilon)$ then:
        Insert($\texttt{PQ}$, $[d^-_{\Omega_c}(q), d^+_{\Omega_c}(q), ptr(N_c)]$, Π);
          // Insert N_c's entry in $\texttt{PQ}$ respecting Π policy
End.

Figure 10: The $\texttt{PAC-k-NN}_{max}$ algorithm used when the error is defined as in Equation 23.

Thus, the $\mathcal{N}(\cdot)$ distribution acts like a "template" for the local distributions. The probability that node N_i is pruned for the probabilistic condition is given as

$$\Pr\left\{d_{X_i}(\mathbf{q}) \leq (1+\epsilon) r_\delta^k\right\} = \mathcal{N}\left(\frac{(1+\epsilon) r_\delta^k - l_i}{u_i - l_i}\right) \tag{36}$$

Among all nodes in the priority queue, we want to access that maximizing the likelihood of containing a "good" object. The problem becomes thus that of maximizing, at each step, the value of $\mathcal{N}(\cdot)$, that is:

$$\max_i \left\{\mathcal{N}\left(\frac{(1+\epsilon) r_\delta^k - l_i}{u_i - l_i}\right)\right\} \Longrightarrow \max_i \left\{\frac{(1+\epsilon) r_\delta^k - l_i}{u_i - l_i}\right\}$$

Note that the expression on the right *does not require to know* $\mathcal{N}(\cdot)$. Two relevant cases to consider are:

$$l_i = 0 \quad \Longrightarrow \quad \min_i \{u_i\} \qquad \text{(MinMaxDist)}$$

$$\delta = 0 \quad \Longrightarrow \quad \min_i \left\{\frac{l_i}{u_i - l_i}\right\}$$

where the first is likely to occur in high-dimensional spaces for structures which allows data regions to overlap. In conclusion, we have:

Theorem 15.4 *The schedule Π^*, also called* LowerUpperDist*, that at step j chooses a node $N_{\Pi_j^*}$ for which it is:*

$$\frac{(1+\epsilon)r_\delta^k - l_{\Pi*_j}}{u_{\Pi*_j} - l_{\Pi*_j}} \geq \frac{(1+\epsilon)r_\delta^k - l_{\Pi*_i}}{u_{\Pi*_i} - l_{\Pi*_i}} \qquad \forall i > j \tag{37}$$

is optimal for the **`PAC-k-NN`**$_k$ *algorithm.*

Proof: Omitted. □

15.3.1 The Case of General Trees

What derived in previous Section assumes two-levels trees, composed by the root node and a set of leaves. For trees with three or more levels, the problem of deriving optimal schedules has no solution, since at each step only a partial knowledge on the tree structure and on the probabilistic information associated to the nodes is available to the `PAC-k-NN` search algorithms. However, if we consider nodes with a sufficiently large fanout, the number of leaf nodes is far greater than the number of internal nodes, thus the cost of accessing nodes in the higher levels of the tree is not a main concern, and can thus be ignored in determining an optimal schedule.

16 Comments and Extensions

In this Part we have extended the PAC approach for solving similarity queries, first presented in [15], to the case of k-NN (with $k > 1$) and range queries. We have devised sequential algorithms to solve PAC queries and have derived cost formulae that show the quality of result/search time trade-off. Moreover, we have also presented index-based algorithms for solving PAC queries demonstrating their correctness. Finally, we have proposed an optimal schedule for accessing index nodes during a k-NN search.

The PAC algorithms presented in Section 15 all exploit "global" information on the distance distribution to derive the probabilistic conditions that allow stopping the search. In particular, the index-based algorithms of Section 15.2 do not consider specific information of the index tree $\mathcal{T}$ being used, i.e. they are unaware of the existence of data regions. Of course, using "local" information could lead to an improvement of both performance of the algorithms and distribution of error. In fact, by using only the global distance distribution, such algorithms, which we call PAC-$\mathcal{G}$ where the $\mathcal{G}$ stands for "global", can only offer a bound on error in a fraction $(1-\delta)$ of queries, whereas, in the remaining δ fraction, no guarantee on error distribution can be given. In this light, PAC-$\mathcal{G}$ algorithms are only able to limit the fraction of queries for which an error occurs, thus *no control on the actual distribution of error can be given.*

To overcome problems of presented PAC-$\mathcal{G}$ algorithms, we plan to investigate two different approaches that exploit other kind of information:

- The PAC-$\mathcal{L}$ approach, where the $\mathcal{L}$ stands for "local", that maintains information for each node of the tree $\mathcal{T}$ to implement a local pruning rule, taking into account the relationship between node regions and the query point.

- The PAC-$\mathcal{A}$ approach, where the $\mathcal{A}$ stands for "adaptive", that only relies on global information, but where such information is adapted to the case at hand by appropriately taking into account bounds derived for the actual query point.

We plan to devise specific algorithms for both the PAC-$\mathcal{L}$ and the PAC-$\mathcal{A}$ approaches and to investigate the trade-off between search costs and quality of the result (i.e. distribution of error) for both solutions with respect to the PAC-$\mathcal{G}$ case.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases.* Addison-Wesley Publishing Company, 1995.

[2] M. Agrawal, E. Allender, and S. Datta. On TC^0, AC^0 and arithmetic circuits. *Journal of Computer and System Sciences*, 60(2):395–421, 2000.

[3] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data,* Washington, D.C., pages 207–216. ACM Press, 1993.

[4] A. Ambainis, D. M Barrington, and H. LêThanh. On counting AC^0 circuits with negative constants. *In Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science,* Brno, Czech Republic, pages 409–417, 1998.

[5] Sunil Arya and David M. Mount. Approximate range searching. *Computational Geometry*, 17(3-4):135–152, December 2000.

[6] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching. *Journal of the ACM*, 45(6):891–923, November 1998.

[7] D. A. Mix Barrington, N. Immerman, and H. Straubing. On uniformity within NC_1. *Journal of Computer and System Sciences*, 41(3):274–306, 1990.

[8] C. Beeri, R. Fagin, M. Yannakakis, and D. Maier. On the desirability of acyclic database schemas. *Journal of ACM*, 30(3):479–513, 1983.

[9] R. Ben-Eliyahu-Zohary and E. Gudes. Towards efficient metaquerying. *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence,* Stockholm, pages 800–805, 1999.

[10] Stefan Berchtold, Christian Böhm, Daniel A. Keim, and Hans-Peter Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'97)*, pages 78–86, Tucson, AZ, May 1997.

[11] P.A. Bernstein and N. Goodman. The power of natural semijoins. *SIAM J. of Computing*, 10(4):751–771, 1981.

[12] Tiziana Catarci, Valeria Cesari, Paolo Ciaccia, Marianna De Scalzi, Giovambattista Ianni, Stefano Lodi, Luigi Palopoli, Marco Patella, and Claudio Sartori. Analysis and comparison of methods and algorithms for data mining. Technical Report D3.R1, D2I (Integration, Warehousing, and Mining of Heterogeneous Data Sources) - Italian MIUR Project, 2001. Available at URL `http://www.dis.uniroma1.it/~lembo/D2I/`.

[13] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. *Conference Record of the Ninth Annual ACM Symposium on Theory of Computing,* Boulder, pages 77–90, 1977.

[14] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, San Francisco, California, 22–24 January 1995.

[15] Paolo Ciaccia and Marco Patella. PAC nearest neighbor queries: Approximate and controlled search in high-dimensional and metric spaces. In *Proceedings of the 16th International Conference on Data Engineering (ICDE 2000)*, pages 244–255, San Diego, CA, March 2000.

[16] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB'97)*, pages 426–435, Athens, Greece, August 1997.

[17] Paolo Ciaccia, Marco Patella, and Pavel Zezula. A cost model for similarity queries in metric spaces. In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'98)*, pages 59–68, Seattle, WA, June 1998.

[18] C. Domshlak, D. Gershkovich, E. Gudes, N. Liusternik, A. Meisels, T. Rosen, and S. E. Shimony. FlexiMine-homepage. *Ben-Gurion University, Mathematics and Computer Science. URL: www.cs.bgu.ac.il/˜ kdd.*

[19] C. Domshlak, D. Gershkovich, E. Gudes, N. Liusternik, A. Meisels, T. Rosen, and S. E. Shimony. FlexiMine - a flexible platform for KDD research and application construction. *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining (KDD-98),* New York City, 1998.

[20] M. E. Dyer and Alan M. Frieze. On the complexity of computing the volume of a polyhedron. *SIAM Journal on Computing*, 17(5):967–974, 1988.

[21] David Eppstein, Zvi Galil, and Giuseppe Italiano. *CRC Handbook of Algorithms and Theory of Computation*, chapter 22, Dynamic Graph Algorithms. CRC Press, 1999.

[22] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Michael Wimmer, and Xiaowei Xu. Incremental clustering for mining in a data warehousing environment. In *Proc. 24th Int. Conf. Very Large Data Bases, VLDB*, pages 323–333, 24–27 August 1998.

[23] Martin Ester, Hans-Peter Kriegel, Jorg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In Evangelos Simoudis, Jia Wei Han, and Usama Fayyad, editors, *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96)*, page 226. AAAI Press, 1996.

[24] U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy. *Advances in Knowledge Discovery and Data Mining.* AAAI Press/MIT Press, 1996.

[25] Paolo Ferragina. A technique to speed up parallel fully dynamic algorithms for MST. *Journal of Parallel and Distributed Computing*, 31(2):181–189, December 1995.

[26] Y.J. Fu and J.W. Han. Meta-rule-guided mining of association rules in relational databases. *DOOD95 Workshop on Integration of Knowledge Discovery with Deductive and Object Oriented Databases,* Singapore, pages 1–8, 1995.

[27] Venkatesh Ganti, Raghu Ramakrishnan, Johannes Gehrke, Allison Powell, and James French. Clustering large datasets in arbitrary metric spaces. In *Proc. 15th IEEE Conf. Data Engineering, ICDE*, 23–26 March 1999.

[28] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems,*Philadelphia. *To appear in Journal of Computer and System Sciences*, pages 21–32, 1999.

[29] G. Gottlob, N. Leone, and F. Scarcello. The complexity of acyclic conjunctive queries. *Journal of ACM*, 48(3), 2001.

[30] K. Chidananda Gowda and G. Krishna. Agglomerative clustering using the concept of mutual nearest neighbourhood. *Pattern Recognition*, 10(2):105–112, 1978.

[31] Misha Gromov. *Metric Structures for Riemannian and Non-Riemannian Spaces*, volume 152 of *Progress in Mathematics*. Birkhäuser, 1999.

[32] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. Rock: A robust clustering algorithm for categorical attributes. In *Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Austrialia*, pages 512–521. IEEE Computer Society, 1999.

[33] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, Boston, MA, June 1984.

[34] John A. Hartigan. *Clustering algorithms.* Wiley Series in Probability and Mathematical Statistics. John Wiley & Sons., New York, 1975.

[35] Gísli R. Hjaltason and Hanan Samet. Ranking in spatial databases. In *Proceedings of the 4st International Symposium on Advances in Spatial Databases (SSD'95)*, pages 83–95, Portland, ME, August 1995.

[36] R. Jarvis and I. Hofman. Robust and efficient cluster analysis using a shared near neighbours approach. In *ICPR'98, Proc. of the 14th Int'l Conference on Pattern Recognition*, pages 243–247. IEEE Computer Society, 1998.

[37] R. Jarvis and E. Patrick. Clustering using a similarity measure based on shared near neighbors. *IEEE Transactions on Computers*, 22(11):1025–1034, November 1973.

[38] David S. Johnson. *A Catalog of Complexity Classes*, chapter 2, pages 69–161. In [54], 1990.

[39] B. Kero, L. Russell, S. Tsur, and W. M. Shen. An overview of data mining technologies. *DOOD95 Workshop on Integration of Knowledge Discovery with Deductive and Object Oriented Databases,* Singapore, 1995.

[40] B. Leng and W.M. Shen. A metapattern-based automated discovery loop for integrated data mining - unsupervised learning of relational patterns. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):898–910, 1996.

[41] Peter Bro Miltersen, Sairam Subramanian, Jeffrey Scott Vitter, and Roberto Tamassia. Complexity models for incremental computation. *Theoretical Computer Science*, 130(1):203–236, August 1994.

[42] B. G. Mitbander, K. Ong, W.M. Shen, and C. Zaniolo. *Metaqueries for data mining*, chapter 15, pages 375–398. In [24], 1996.

[43] Inderpal Singh Mumick, Dallan Quass, and Barinderpal Singh Mumick. Maintenance of data cubes and summary tables in a warehouse. In Joan Peckham, editor, *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 100–111, Tucson, Arizona, 13–15 June 1997. *SIGMOD Record* 26(2), June 1997.

[44] Vladimir Pestov. On the geometry of similarity search: Dimensionality curse and concentration of measure. Technical Report RP-99-01, School of Mathematical and Computing Sciences, Victoria University of Wellington, New Zealand, January 1999. `http://xxx.lanl.gov/abs/cs.IR/9901004`.

[45] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. Nearest neighbor queries. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 71–79, San Jose, CA, May 1995.

[46] W. L. Ruzzo. On uniform circuit complexity. *Journal of Computer and System Sciences*, 22(3):365–383, 1981.

[47] W. M. Shen. Discovering regularities from knowledge bases. *Intelligent Systems*, 7(7):623–636, 1992.

[48] J. Simon. *On some central problems in computational complexity*. PhD thesis, Dept. of Computer Science, Cornell University, Ithaca, NY, 1975.

[49] L. J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):1–22, 1976.

[50] Jacobo Torán. An oracle characterization of the counting hierarchy. *Proceedings of the 3rd conference on Structure in Complexity Theory,* Washington D.C., pages 213–223, 1988.

[51] Jeffrey D. Ullman. *Principle of database and knowledge-base systems.* Principle of computer science series. Computer Science Press, Inc., 1988.

[52] Leslie G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.

[53] Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[54] Jan van Leeuwen (Ed.). *Handbook of Theoretical Computer Science.* Elsevier and MIT Press, 1990.

[55] M. Y. Vardi. The complexity of relational query languages. *Proceedings of the Fourteenth ACM SIGACT Symposium on Theory of Computing,* San Francisco, pages 137–146, 1982.

[56] Jeffrey Scott Vitter. External memory algorithms and data structures. In James Abello and Jeffrey Scott Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society Press, Providence, RI, 1999.

[57] K. Wagner. The complexity of combinatorial problems with succinct input representation. *Acta Informatica*, 23:325–356, 1986.

[58] M. Anthony Wong and Tom Lane. A kth nearest neighbour clustering procedure. *J. R. Statist. Soc. B*, 45(3):362–368, 1983.

[59] Pavel Zezula, Pasquale Savino, Giuseppe Amato, and Fausto Rabitti. Approximate similarity retrieval with M-trees. *The VLDB Journal*, 7(4):275–293, 1998.

[60] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. BIRCH: An efficient data clustering method for very large databases. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 103–114, Montreal, Quebec, Canada, 4–6 June 1996. *SIGMOD Record* 25(2), June 1996.