# Précis: The Essence of a Query Answer [*]

Georgia Koutrika
*University of Athens*
*koutrika@di.uoa.gr*

Alkis Simitsis
*Nat. Tech. Univ. of Athens*
*asimi@dblab.ntua.gr*

Yannis Ioannidis
*University of Athens*
*yannis@di.uoa.gr*

## Abstract

*Wide spread use of database systems in modern society has brought the need to provide inexperienced users with the ability to easily search a database with no specific knowledge of a query language. Several recent research efforts have focused on supporting keyword-based searches over relational databases. This paper presents an alternative proposal and introduces the idea of précis queries. These are free-form queries whose answer (a précis) is a synthesis of results, containing not only information directly related to the query selections but also information implicitly related to them in various ways. Our approach to précis queries includes two additional novelties: (a) queries do not generate individual relations but entire multi-relation databases; and (b) query results are personalized to user-specific and/or domain requirements. We develop a framework and system architecture for supporting such queries in the context of a relational database system and describe algorithms that implement the required functionality. Finally, we present a set of experimental results that evaluate the proposed algorithms and show the potential of this work.*

## 1. Introduction

"**précis** /ˈpreɪsiː: [(of)] *a shortened form of a piece of writing or of what someone has said, giving only the main points.*" (Longman Dictionary)

A précis is often what one expects in order to satisfy an information need expressed as a question or as a starting point towards that direction. For example, if one asks about 'Woody Allen', a possible response might be in the form of the following précis:

"*Woody Allen was born on December 1, 1935 in Brooklyn, New York, USA. As a director, Woody Allen's work includes Match Point (2005), Melinda and Melinda (2004), Anything Else (2003). As an actor, Woody Allen's work includes Hollywood Ending (2002), The Curse of the Jade Scorpion (2001)."*

Likewise, returning a précis of information in response to a user query is extremely valuable in the context of web accessible databases, which have emerged as libraries, museums, and other organizations publish their electronic contents on the Web. With the abundance of available information, exploring the contents of a web database and finding anything useful is a difficult and often fruitless procedure. However, "end users want to achieve their goals with a minimum of cognitive load and a maximum of enjoyment. ... humans seek the path of least cognitive resistance and prefer recognition tasks to recall tasks" [1]. In addition, they often have very vague information needs or know a few buzzwords. Based on the above, support of free-form queries over databases and generation of answers in the form of a précis comprises an advanced searching paradigm helping users to gain insight into the contents of a database. A précis may be incomplete in many ways; for example, the abovementioned précis of 'Woody Allen' includes a non-exhaustive list of his works. Nevertheless, it provides sufficient information to help someone learn about Allen and identify new keywords for further searching. For example, the user may decide to issue a new query about "*Anything Else*" or follow underlined topics (hyperlinks) to pages containing more relevant information.

Supporting précis queries using a relational database system is not straightforward. Pre-specified queries embedded in user-interface forms are not a realistic approach. Neither should web users be expected to have any knowledge about the relational data model, schemas, structured query languages, or even the schema of a particular database, to form their own structured queries.

In addition, relational queries produce a single relation whose tuples are therefore forced to contain attributes from numerous relations. Such flattened out results are often unnatural and unusable in constructing a meaningful précis. Queries should be able to generate a whole new database, with its own schema, constraints, and contents, derived from their counterparts in the original database. Through the concepts and relationships captured in them, such results will provide the knowledge necessary to derive semantically rich précis for the user.

Database-type query results are very useful in other situations as well. Given large databases, enterprises often need smaller subsets that conform to the original schema and satisfy all of its constraints in order to perform realistic *tests* of new applications before deploying them to production. Likewise, software vendors need such smaller but correct databases to *demonstrate* new software product functionality. Generating such databases with current relational technology, one relation at a time and manually deriving the appropriate constraints, is not acceptable.

**Contributions**. Motivated by the above, this paper presents a comprehensive effort to generalize relational queries in two directions. First, query conditions should be free-form, containing only selection clauses. The system will dynamically decide the joins and other predicates that are relevant to the conditions specified and construct the complete qualifications that the query results should satisfy. Second, the above process will generate sets of queries resulting in sets of interconnected relations that together conform to the appropriate schema constraints. Furthermore, these generalized forms of queries will also be customized to the inquiring user's preferences and requirements, which could be specified at query time or could be retrieved from a user profile. In detail, the following are our main contributions.

- *Précis Queries Framework*. We introduce the concept of précis queries in the context of databases and describe a framework for their support. The major steps for answering précis queries are (*a*) result schema generation, where the database part that contains information related to the query is recognized, and (*b*) result data generation, where tuples are extracted from the database with the use of appropriate SQL queries. Our approach is applicable to other types of (semi-) structured data as well. However, for presentation reasons, we focus on relational data here.

- *System Architecture and Customized Query Processing Algorithms*. We describe the architecture of a system that supports précis queries, and we provide appropriate algorithms for each module of it. We also illustrate a semi-automatic method that translates the relational output of a précis query into a "natural language" synthesis of results.

- *Experiments*. Finally, we present a set of results evaluating each part of the system.

## 2. Related Work

Précis queries are free-form queries. The need for free-form queries has been early recognized in the context of databases. Motro [4] described the idea of using *tokens*, i.e. value of either data or metadata, when accessing information instead of structured queries, and proposed an interface that understands such utterances by interpreting them in a unique way, i.e., complete them to proper queries. BAROQUE [3] used a network representation of a database and defined several types of relationships in order to support functions that scan this network. With the advent of the World Wide Web, the idea has been revisited. In particular, recent approaches on keyword searches in databases [5, 6, 7, 8] extended the idea of tokens to values that may be part of attribute values.

These approaches work on some kind of graph (data graph [5], schema graph [7, 8], dependency graph [4]). Based on this graph, the interpretation for a given set of database tokens is a query that corresponds to a sub-graph connecting their corresponding nodes. An answer to a keyword search is a set of ranked tuples based on some

criterion (the number of joins [8], IR-style answer-relevance ranking [9]). On the other hand, Oracle 9i Text [15], Microsoft SQL Server 2000 [16] and IBM DB2 Text Information Extender [17] create full text indexes on text attributes of relations and then perform keyword queries. Keyword search over XML databases has also attracted interest recently [12, 13, 14].

Our main differences from existing approaches can be summarized as follows. *First*, we work on the database schema graph. However, instead of simply locating and connecting values in tables as other approaches do [7, 8], we also consider information around these values that may be related to them. For example, the answer provided by existing approaches for "Woody Allen" would be in the form of relation-attribute pair, such as (Name, Director). On the contrary, the answer to a précis query might also contain information found in other parts of the database, e.g. movies directed by Woody Allen. This information needs to be "assembled" –in perhaps unforeseen ways– by joining tuples from multiple relations. *Second*, existing approaches return flattened out results. Instead, we generate a whole new database, with its own schema, constraints, and contents, derived from their counterparts in the original database. *In addition*, using the information conveyed by the database graph, which may be properly annotated to further enhance its semantics, we try to construct a close to natural language representation of an answer. *Finally*, we allow generating customized answers in response to a query by making use of weights on the database schema graph that may be specified by a domain expert or each user. This is inspired by work on user preferences [11].

## 3. Framework

### 3.1 Data Model

A *relation schema* $S_i$ is denoted as $R_i(A_{1i}, A_{2i}, ..., A_{ki})$ and consists of a relation name $R_i$ and a set of attributes $A_i = \{A_{ji}: 1 \le j \le k_i\}$. A *database schema* $D$ is a set of relation schemas $\{S_i: 1 \le i \le m\}$. When populated with data, relation and database schemas generate *relations* and *databases*, respectively. We use $R_i$ to denote a relation following relation schema $S_i$ and $D$ to denote a database following database schema $D$.

We consider the *database schema graph* $G(V, E)$ as a directed graph corresponding to a database schema $D$. There are two types of nodes in $V$: (a) *relation nodes, $R$,* one for each relation in the schema; and (b) *attribute nodes*, $A$, one for each attribute of each relation in the schema. Likewise, edges in $E$ are the following: (a) *projection edges*, $\Pi$, each one connects an attribute node with its container relation node, representing the possible projection of the attribute in the system's answer; and (b) *join edges*, $J$, from a relation node to another relation node, representing a potential join between these relations.

COMPUTER SOCIETY

These could be joins that arise naturally due to foreign key constraints, but could also be other joins that are meaningful to a domain expert. Joins are directed for reasons explained later. For simplicity in presentation, we assume (a) that primary keys are not composite; thus, an attribute from a relation joins to an attribute from another relation, and (b) that these attributes have the same name. For convenience, we do not depict the joining attributes in both relations; instead, the common name of the joining attributes is tagged on the respective join edge between the two relations. Therefore, a database graph is defined as a directed graph $G(V,E)$, where: $V = R \cup A$, and $E = \Pi \cup J$.

**Weights.** A *weight*, $w \in [0,1]$, is assigned to each edge of the graph $G$ showing the significance of the bond between the corresponding nodes. $w = 1$ expresses strong relationship: if one node of the edge appears in an answer, then the edge should be taken into account making the other node appear as well. $w = 0$, occurrence of one node of the edge in an answer does not imply occurrence of the other node. Based on the above, two relation nodes could be connected through two different join edges, in the two possible directions, between the same pair of attributes, but carrying different weights. A directed join edge expresses the dependence of the left part of the join on the right part. The left part indicates the relation already considered for the answer and the right one corresponds to the relation that may be included influencing the final result, if the join is taken into account. For simplicity, we assume that there is at most one directed edge from one node to the same destination node.

**Example**. Consider a movies database described by the schema below; primary keys are underlined.

```
THEATRE(tid,name,phone,region)
PLAY(tid,mid,date), GENRE(mid,genre)
MOVIE(mid,title,year,did)
CAST(mid,aid,role)
ACTOR(aid,aname,blocation,bdate)
DIRECTOR(did,dname,blocation,bdate)
```

The database graph is depicted in Figure 1. For instance, observe the two directed edges between MOVIE and GENRE. Movies and genres are related but one may consider that genres are more dependent on movies than the other way around. In other words, an answer regarding a genre should always contain information about related movies, while an answer regarding a movie may not necessarily contain information about its genres. For this reason, the weight of the edge from GENRE to MOVIE is 1, while the weight of the edge from MOVIE to GENRE is 0.9.

Using different weights on graph's edges allows constructing different answers to the same query.

− Weights may be set by the user at query time using an appropriate user interface. This option enables interactive exploration of the contents of a database. In particular, changing weights associated with the underlying database results in a different set of queries executed in order to obtain related tuples from this part of the database and

essentially affects the part of the database explored. The user may explore different regions of the database starting, for example, from those containing objects closely related to the topic of a query and progressively expanding to parts of the database containing objects more loosely related to it.

− Sets of weights may be created by a designer targeting different groups of users. For instance, reviewers and cinema fans have access to a movies database. The former may be typically interested in in-depth, detailed answers; using an appropriate set of weights would enable these users to explore larger parts of the database around a single précis query. Cinema fans usually prefer shorter answers. A different set of weights would allow producing answers containing only highly related objects.

− Finally, multiple sets of weights corresponding to different user profiles may be stored in the system. Using user-specific weights allows generating *personalized* answers. For example, a user may be interested in the region where a theatre is located, while another may be interested in a theatre's phone. As a result, different users may see different answers to the same query.
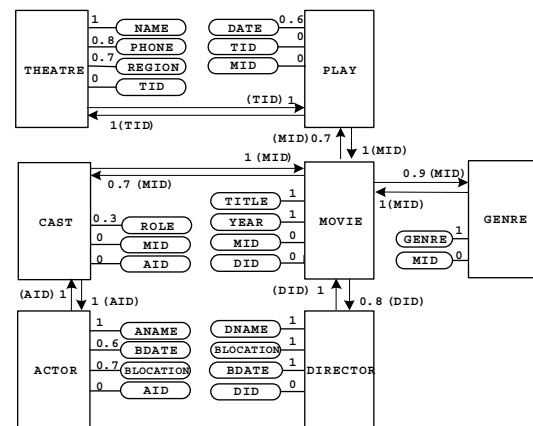


**Figure 1. An example database graph**

### 3.2 Transfer of Weight over Paths in the Graph

A directed path $p$ between two relation nodes, comprising adjacent join edges, represents the "implicit" join between these relations. Similarly, a directed path between a relation node and an attribute node, comprising a set of adjacent join edges and a projection edge represents the "implicit" projection of the attribute on this relation. In correspondence to edges, we call these paths, *transitive join* and *transitive projection paths*, respectively.

The weight of a path is a function of the weight of constituent edges, and should decrease as the length of the path increases [10]. In our implementation, we have chosen multiplication as this function.

**Example**. In Figure 1, the weight of the projection of attribute PHONE over THEATRE equals to 0.8, while its weight with respect to MOVIE is 0.7*1*0.8 = 0.56.

### 3.3 Query model

Consider a database $D$ and a *précis query* $Q$ which is a set of tokens, i.e. $Q=\{k_1, k_2, \ldots, k_m\}$. The result of applying $Q$ on $D$, called *précis*, is a new database $D'$ satisfying these:

1. The set of relation names in the result schema $D'$ is a subset of that in $D$.
2. For each relation name $R_i$ in the result $D'$, its set of attributes $B_i=\{B_{ji}: 1 \le j \le l_i\}$ in $D'$ is a subset of its set of attributes $A_i=\{A_{ji}: 1 \le j \le k_i\}$ in $D$. That is, the result of the query involves some of the attributes of each relation schema present.
3. For each relation name $R_i$ in the result $D'$, the set of tuples in the corresponding relation $R_i'$ is a subset of the set of tuples in the original relation $R_i$ (when projected on the set of attributes $B$ that are present in the result).
4. The result database $D'$ is generated by (foreign-key) join queries starting from the relations where the keywords in $Q$ appear and transitively expanding on the database schema $D$. The final set of relation names, attributes, and tuples in $D'$ are determined by *constraints*.

We define two types of such constraints: (*a*) A *degree constraint* $d$ determines attributes and relations in $D'$, and (*b*) A *cardinality constraint* $c$ determines the number of tuples in $D'$. In order to describe the result of a query $Q$, a pair of constraints, one of each category should be provided. Possible degree constraints could include these:

– The maximum number of attributes in $D'$
– The minimum weight of projection paths in the database schema graph $G$

Constraints on the edge weights of the database schema graph $G$ are more immune to the effects of database normalization or database restructuring. For example, in the example movies database depicted in Figure 1, each movie has only one director. Assume that we want to associate a movie with more than one director. For this purpose, a new relation, DIRECTED_BY(<u>mid</u>, <u>did</u>), would be added. As a result, the length of the path between MOVIE and DIRECTOR would increase, as well as the number of relations required in $D'$, in order to show information about movies and related directors. Consequently, the first and the third constraint from the previous list should be adapted accordingly; on the contrary, the second constraint would remain valid as long as the weight of the path between MOVIE and DIRECTOR would not change. This can be accomplished by assigning appropriate weights on constituent edges.

Possible cardinality constraints could include these:

– The maximum number of tuples in $D'$
– The maximum number of tuples per relation in $D'$

Using different constraints allows generating different answers for the same query and the same set of weights over the edges of the database graph. Similarly to weights, constraints may be specified at query time by the user, or be pre-specified by a designer, or may be stored as part of

a user's profile. For example, in the graph of Figure 1, attributes of THEATRE have different weights. With the use of an appropriate criterion, an answer about a theatre may contain only its name or may also contain information about phone and region.
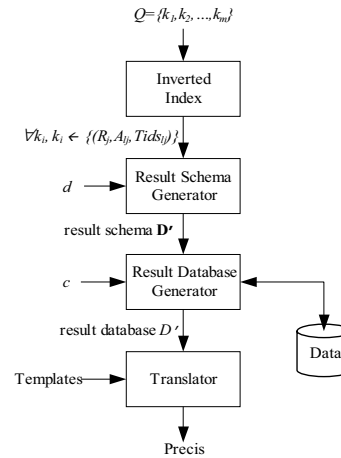


**Figure 2. System Architecture**

## 4. System architecture

A high level representation of the system architecture for answering précis queries is depicted in Figure 2. First, the user submits a précis query $Q=\{k_1, k_2, \ldots, k_m\}$. In order to generate an answer, the following steps are performed.

*Inverted Index*. An inverted index associates each token that appears in the database with a list of occurrences of the token. Each occurrence is recorded as an attribute-relation pair, $(R_j, A_{lj})$. For each such pair, the list $Tids_{lj}$ of ids of tuples from $R_j$ in which $A_{lj}$ includes the token, is also returned. A token may be found in more than one tuples and attributes of a single relation and in more than one relations. We chose to build our own inverted index that allows efficient retrieval of all occurrences of a token. Given a query $Q$, the index returns for each token $k_i$ in $Q$, a list of all its occurrences, i.e. $k_i \rightarrow \{(R_j, A_{lj}, Tids_{lj})\}, \forall k_i$ in $Q$.

*Result Schema Generator*. This step is responsible for finding which part of the database schema may contain information related to $Q$. The output of this step is a result schema $D'$ comprised of relations that contain the tokens of $Q$ and relations transitively joining to the former and a subset of their attributes that should be present in the result according to an input degree constraint $d$.

*Result Database Generator*. This step produces the result database $D'$ that corresponds to the schema $D'$. The result database $D'$ is generated by (foreign-key) join queries starting from the relations where the tokens in $Q$ appear and transitively expanding on the database schema $D'$. The final set of tuples in $D'$ comprises tuples containing the query tokens and tuples joining to these, and its size is

determined by an input cardinality constraint $c$.

*Translator*. This step is required if a précis needs to be transformed into a narrative form. In particular, this step provides a query answer constructed as a proper structured management of individual results, according to certain rules and templates predefined by a designer or the administrator of the database.

# 5. Answering a Précis Query

For illustration purposes, we use this running example.

**Query Example**. Consider the query $Q=\{`Woody\ Allen'\}$, and the constraint that only projections with weight equal to or greater than $0.9$ should be present in the answer (degree constraint) and up to three tuples should be retrieved per relation (cardinality constraint).

## 5.1 Result Schema Generator

The Result Schema Generator is responsible for finding which part of the database schema may contain information *most* related to a given query $Q$. Its output is a result schema $D'$ comprised of all relations containing query tokens and relations transitively joining to the former as well as a subset of their attributes that should be present in the result according to a degree constraint $d$ provided as input. We formulate the problem as follows.

**Problem formulation**. Consider the database schema graph $G$ corresponding to the database schema $D$, and the relation nodes on the graph corresponding to the relations where the query tokens have been found. Furthermore, consider the set $P_n$ of all (transitive) acyclic projection paths in $G$ that are attached to these relations in order of decreasing weight, i.e.

$$P_n = \{p_i \mid i \in [1,n],\ w_{i-1} \geq w_i\}$$

The result schema $D'$ containing most related information to the query according to a given degree constraint $d$ corresponds to the schema graph $G'$. This is a sub-graph of $G$ including the nodes mapping the relations that contain the query tokens and the set of all (transitive) projection paths on $G'$ attached to these nodes is the ordered subset $P_d$ of $P_n$, such that:

$$d = \max(\{t \mid t \in [1,n]: d(P_t)\ \text{holds}\ \})$$

Possible expressions of $d(.)$ are given in Table 1.

### Table 1. Possible degree constraints

| Expression | Description |
|---|---|
| $t \leq r$ | selects up to $r$ top-weighted projections |
| $w_t \geq w_o$ | selects top-weighted projections with weight $\geq w_o$ |
| $length(p_t) \leq l_o$ | selects top-weighted projections with path length $\leq l_o$ |

**Algorithm**. The Result Schema Generator algorithm takes as input the database schema graph $G$, the set of relations where the query tokens have been found, and a degree criterion $d$. It constructs a schema graph $G'$ that represents the result schema for the query according to the degree constraint.

---

**Result Schema Algorithm**

**Input**: database schema graph $G(E,V)$, degree criterion $d(.)$
$\{R_j \mid R_j$ relation containing query tokens$\}$,

**Output**: result schema graph $G'(E',V')$

---

$QP = \varnothing,\ P_d = \varnothing,\ G' = \varnothing$

1. **Foreach** edge $e$ attached to a relation $R_j$: $e(R_j, x) \in E$, $x \in V$
    $QP \leftarrow e$
  **End for**

2. **While** ($QP$ not empty)
  2.1  Get head $p$ from $QP$
  2.2  **If** ($d(P_d \cup \{p\})$ does not holds) **Then** exit while **End if**
  2.3  **If** ($p$ is projection path) **Then**
      $P_d \leftarrow p$
      Update edges, nodes and in-degrees in $G'$ accordingly
    **End if**
  2.3  **If** ($p$ is join path) **Then**
      **Foreach** edge $e \in G$ that is attached to $p$
        $p'$ is the concatenation of $p$ and $e$ and is acyclic
        **If** ($d(P_d \cup \{p'\})$ does not holds) **Then** Exit For **End if**
        $QP \leftarrow p'$
      **End for**
    **End if**
  **End while**

3. Output $G'$

---

**Figure 3. Result Schema Algorithm**

The algorithm, presented in Figure 3, performs a best-first traversal of the database schema graph $G$. The basic idea is to gradually construct projection paths on $G$ attached to the input relations in order of decreasing weight. Paths of equal weight are considered in order of increasing length. In other words, shorter paths are favoured among paths of equal weight based on the intuition that these may connect more closely related entities. If a projection path satisfies the degree constraint, then it is "added" into the sub-graph $G'$, i.e. edges and nodes of the path not already in $G'$ are inserted into it. The algorithm stops when no other projection paths satisfying the constraint can be constructed on graph $G$.

More specifically, the algorithm keeps a queue $QP$ of candidate paths in order of decreasing weight, and a set $P_d$ of projection paths on $G'$ that are attached to the relations containing the query tokens. Initially, $QP$ contains all edges on $G$ attached to those relations. In each round, the algorithm picks from $QP$ the candidate path $p$ with the highest weight. If the degree constraint is not satisfied, then the algorithm stops and returns $G'$. If the constraint is satisfied, then different actions are performed depending on the type of path. If $p$ is a projection path, the result schema graph, $G'$, is updated accordingly, and $p$ is inserted into the set of projections, $P_d$, encountered so far. It is possible that projection paths starting from different relations containing the query tokens share common relations. In order to facilitate subsequent steps, and

primarily result database generation, we mark each relation node on $G'$ that is found in more than one path. In particular, we count the number of input relations whose paths include this relation (in-degree). We will see how this is used during result database generation.

If $p$ is a join path, then it is expanded into longer paths which are placed into QP. A path $p$ is expanded according to the following rule: a new path $p'$ is generated for each edge $e$ that is adjacent to $p$ in the graph $G$ and is the concatenation of $p$ and $e$. Edges are considered in order of decreasing weight. This helps pruning, as explained shortly, and improves the time of insertion of new paths in the ordered QP. A new path $p'$ is pruned if it does not satisfy the degree constraint. Then, the algorithm stops expansion of $p$, since all paths subsequently generated will not satisfy the constraint.
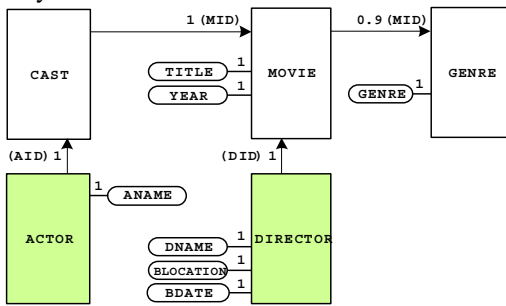


**Figure 4. Result Schema for the query example**

**Query Example (cont'd)**. Given the query Q={'Woody Allen'}, "Woody Allen" is a director and also an actor. Therefore, the inverted index returns relations DIRECTOR and ACTOR. Given the degree constraint that only projections with weight equal to or greater than 0.9 should be present in the answer, Figure 4 shows the result schema graph output by this step with respect to the example database depicted in Figure 1. The input relations are shown in color. On this graph, there are paths from DIRECTOR or ACTOR, some of them having common relations, one such being relation MOVIE. For presentation reasons, this example is kept simple, in the sense that the set of paths arriving to a relation from DIRECTOR and the set of paths arriving to the same relation from ACTOR result in the projection of the same attributes of that relation. In a general case, paths from DIRECTOR to a relation could result in projection of different attributes on this relation than paths arriving to the same relation from ACTOR. We use $P_d$ to keep track of the projection paths departing from each of the input relations. Moreover, observe in the result schema of the figure that MOVIE has an in-degree equal to 2.

In the example above, 'Woody Allen' is found in two different relations. In general, it is possible that a single value may be used to represent different objects (homonyms), e.g., 'Woody Allen' could correspond to two different persons, or different values may be used for the same object (synonyms); e.g., 'W. Allen' and

'Woody Allen' that correspond to the same person. To tackle the former problem, in the absence of any additional knowledge stored in the system, we may return multiple answers, one for each homonym, or obtain additional information through interaction with the user. For the latter problem, there exist approaches [e.g., 19, 20] for cleaning and homogenizing string data, such as addresses, acronyms, names and so forth. However, both these problems are orthogonal to answering précis queries.

### 5.2 Result Database Generator

The Result Database Generator produces a result database $D'$ corresponding to the schema $D'$. The result database is generated by selecting tuples in the relations containing the query tokens and tuples in other relations of $D'$ transitively joining to the former. The final set of tuples in $D'$ is determined by a cardinality constraint $c$. We formulate the problem as follows.

**Problem Formulation**. Given are the database schema graph $G'$ corresponding to the database schema $D'$, the relation nodes containing the query tokens, and a cardinality criterion $c(.)$. In addition, for each of these relations, the list of tuples containing query tokens is considered. Based on the above, this is an initial database $D_o$ corresponding to the database schema $D'$. The set of possible result databases corresponding to schema $D'$ in order of increasing cardinality is:

$$D_1 \leftarrow D_o \bowtie R_1 , D_2 \leftarrow D_1 \bowtie R_2 , \dots , D_{n_j} \leftarrow D_{n_{j-1}} \bowtie R_{n_j}$$

At any point, a relation $R_i$ is joined to $D_{i-1}$ if there is a join edge in $G'$ between this relation and a relation already populated in $D_{i-1}$. If more than one joins may be executed, these are considered in order of decreasing weight. In this way, relations in $D'$ that are most related to the query are populated first. Any relations that may not be eventually populated due to the cardinality constraint would be the most weakly connected to the query. A database $D_i$ contains all tuples also contained in $D_{i-1}$ plus any tuples from that join to those through the corresponding join edge $e_i$. The number of tuples in $D_i$ is card($D_i$). According to the cardinality criterion, the result database $D'$ is a database $D_c$, such that:

$$c = \max( \{ t \mid t \in [0, n_J] : c(D_t) \text{ holds} \})$$

For each relation $R_i$, a subset of its tuples, $R_i'$, is found in the result $D'$, projected on the set of attributes that are present in the result. Possible expressions of $c(.)$ are given in Table 2. A combination of those is also possible.

**Table 2. Possible cardinality constraints**

| Expression | Description |
| --- | --- |
| card($D_t$)$\leq c_o$ | max. total number of tuples in $D'$ is $c_o$ |
| card($R_t$)$\leq c_o$ | max. number of tuples per relation in $D'$ is $c_o$ |

**Algorithm**. The Result Database Generator algorithm has inputs: the result schema graph $G'$ produced in the previous step; a cardinality constraint $c(.)$, and the

relation nodes containing query tokens. In addition, for each of these relations, the list of tuple id's containing query tokens is provided (returned by the inverted index). The output is the result database $D'$ that is an instance of $D'$ corresponding to graph $G'$.

Initially, $D'$ contains all tuples involving query tokens. For each relation $R_j$ with matching tuples, the algorithm retrieves them using their ids provided in $Tids_j$. In relational algebra, the query executed looks like this: $\sigma_{Tids_j}(R_j)[\Pi(R_j)]$, where $\Pi(R_j)$ is the set of attributes of $R_j$ that is projected in the result schema $D'$. If the cardinality constraint allows only a subset of $R_j$'s matching tuples to be selected, then a random subset of those is retrieved using *NaïveQ*, which is described below.

Subsequently, the algorithm loops through the set of join edges of $G'$. In each round, one or more joins may be possibly executed. In this case, as already explained, the join with the highest weight precedes. Execution of joins that depart from relations with an in-degree greater than 1 is postponed. Thus, the algorithm ensures that all tuples that may populate a relation in the result database as result of different joins arriving to it will be produced before moving from this relation further on the database graph. Also, any duplicates are removed. (Which of the tuples collected in a relation are used for subsequently joining tuples from other relations depends on the paths stored in $P_d$. For simplicity, we omit any details regarding this.)

For each directed join $R_i \bowtie R_j$ executed, with a subset of $R_i$, namely $R_i'$, already in $D'$, a subset of $R_j$ is retrieved containing tuples joining to those in $R_i'$. The number of tuples in $R_j'$ is determined by the cardinality criterion. The attributes projected in $R_j'$ are specified by the projections edges attached to the corresponding relation and the join edges that depart from it to other relations in $G'$. In terms of the query executed for this purpose, this does not contain the actual join between the two relations. In relational algebra, the corresponding query is the following: $\sigma_{Ids_j}(R_j)[\Pi(R_j)]$, where $\Pi(R_j)$ is the set of attributes of $R_j$ that should be projected, and $Ids_j$ is the set of values of the attribute of $R_j$ used for joining with $R_i$. These are contained in the corresponding joining attribute of $R_i'$. Again, only a subset of $R_j$'s tuples may be required according to the criterion $c(.)$. There are two possible ways to obtain this subset: *NaïveQ* and *RoundRobin*.

(*NaïveQ*) One way is to submit an SQL query and keep only the top tuples, whose number is determined based on the cardinality constraint. For instance, in Oracle, this can be performed using the pseudo-column $RowNum$.

If the join considered, $R_i \bowtie R_j$, is *to-1*, then the above method selects a random subset of tuples from $R_j$ that join to all tuples in $R_i$ in $D'$ (assuming that each of $R_i$'s tuples in $D'$ joins with one tuple in $R_j$). However, if the join is *to-n*, then there is a risk of selecting a subset of $R_j$'s tuples that join to only a subset of $R_i'$'s tuples in $D'$. As a result

of this selection, there will be tuples in $R_i'$ that will not join to any tuples from $R_j$. To avoid this situation, an approach would be the following: assume that the total number of tuples that may be retrieved from $R_j$ according to the cardinality constraint is $T$, and $t$ is the number of tuples in $R_i'$. Then, the number of tuples retrieved per tuple of $R_i'$ is $T/t$. A set of parameterized queries may be submitted, each one retrieving up to $T/t$ joining tuples from $R_j$ for each tuple in $R_i'$. This method attempts to retrieve tuples from $R_j$ that are uniformly distributed over tuples of $R_i'$. However, since the real distribution in the database may be very different, we have adopted the following round-robin method.

(*Round-Robin*) For each tuple in $R_i'$, a scan of joining tuples from $R_j$ is opened. Each time, only one joining tuple from a scan is retrieved as long as the cardinality constraint holds. If there are no tuples to be retrieved from a scan, this is closed.

Having executed the join $R_i \bowtie R_j$, the in-degree of $R_j$ is reduced accordingly. The algorithm stops execution when either all join edges of $G'$ have been considered or the cardinality constraint does not hold.

---

**Result Database Algorithm**

**Input**: $\{R_j \mid R_j$ relation containing query tokens $\}$,
$\{Tids_j \mid Tids_j$ a set of matching tuple id's in $R_j\}$,
result schema graph $G'$, cardinality criterion $c(.)$

**Output**: Result Database $D'$

1. $D' \leftarrow \{$ naïveQ $(\sigma_{Tids_j}(R_j)[\Pi(R_j)], c(\sigma_{Tids_j}(R_j)[\Pi(R_j)])), \forall R_j\}$
2. **Foreach** applicable *join edge* in $G'$ where destination $R_i$ has in-degree = 1
   2.1 **If** the join is to-n **then**
       $D' \leftarrow$ RoundRobin $(D' \bowtie R_i, c(D' \bowtie R_i))$
   **Else**
       $D' \leftarrow$ naïveQ $(D' \bowtie R_i, c(D' \bowtie R_i))$
   **End if**
   2.2 decrease $R_i$'s in-degree
3. Output $D'$

---

**Figure 5. Result Database algorithm**

**Query Example (cont'd)**. Given the query $Q = \{$'Woody Allen'$\}$, and the cardinality constraint that up to three tuples should be retrieved per relation, a part of the output of this step is depicted in Figure 6. (Attributes required for joins have been also projected in the result, but these will not show in the final answer, since they are not included in the result schema of Figure 4).

## 5.3 Result Database Translator

In this section, we illustrate a semi-automatic method to render the SQL-like response of a précis query to a more user-friendly synthesis of results. In the context of this work, the presentation of a query answer is defined as a proper structured management of individual results, according to certain rules and templates predefined by a designer or the administrator of the database. Clearly, we

do not anticipate the construction of a human-intelligent system; rather, we try to provide a user-friendly response through the composition of simple clauses. We only sketch our approach due to space considerations. More details may be found in [21].

In our framework, in order to describe the semantics of a relation $R$ along with its attributes in natural language, we consider that relation $R$ has a conceptual meaning captured by its name, and a physical meaning represented by the value of at least one of its attributes that characterizes tuples of this relation. We name this attribute the *heading* attribute and we depict it as a hachured rounded rectangle. For example, in Figure 6, the relation MOVIE conceptually represents "movies" in real world; indeed, its name, MOVIE, captures its conceptual meaning. Moreover, the main characteristic of a "movie" is its title, thus, the relation MOVIE should have the TITLE as its heading attribute. By definition, the edge that connects a heading attribute with the respective relation has a weight $1$ and it is always present in the result of a précis query. A domain expert makes the selection of heading attributes.

The synthesis of query results follows the database schema and the correlation of relations through primary and foreign keys. Additionally, it is enriched by alphanumeric expressions called *template labels* mapped to edges of the database schema graph.

A *template label*, label(u,z) is assigned to each edge $e(u,z) \in E$ of the database schema graph $G(V,E)$. This label is used for the interpretation of the relationship between the values of nodes $u$ and $z$ in natural language.

Each projection edge $e \in \Pi$ that connects an attribute node with its container relation node, has a label that signifies the relationship between this attribute and the heading attribute of the respective relation; e.g., the YEAR of a MOVIE (.TITLE). If a projection edge is between a relation node and its heading attribute, then the respective label reflects the relationship of this attribute with the conceptual meaning of the relation; e.g., the TITLE of a MOVIE. Each join edge $e \in J$ between two relations has a label that signifies the relationship between the heading attributes of the relations involved; e.g., the GENRE (.GENRE) of a MOVIE (.TITLE). The label of a join edge that involves a relation without a heading attribute signifies the relationship between the previous and subsequent relations.

We define as the label $l$ of a node $n$ the name of the node and we denote it as $l(n)$. For example, the label of the attribute node TITLE is "title". The name of a node is determined by the designer/administrator of the database.

The template label label(u,z) of an edge $e(u,z)$ formally comprises the following elements: (a) lid, a unique identifier for the label in the database graph; (b) $l(u)$, the name of the starting node; (c) $l(z)$, the name of the ending node; (d) $expr_1, expr_2, expr_3$ alphanumeric expressions. A simple template label has the form:

label(a,b) = $expr_1$ + $l(u)$ + $expr_2$ + $l(z)$ + $expr_3$

where the operator "+" acts as a concatenation operator.

In order to use template labels or to register new ones, we use a simple language for templates that supports variables, loops, functions, and macros.

The translation is realized separately for every occurrence of a token. At the end, the précis query lists all the clauses produced. For each occurrence of a token, the analysis of the query result graph starts from the relation that contains the input token. The labels of the projection edges that participate in the query result graph are evaluated first. The label of the heading attribute comprises the first part of the sentence. It becomes obvious that for multiple attributes of the same relation we have to repeat several times the same subject. To avoid this, a domain expert should have attached suitable expressions in the projections edges, in order to allow the construction of complex sentences that make sense.

After having constructed the clause for the relation that contains the input token, we compose additional clauses that combine information from more than one relation by using foreign key relationships. Each of these clauses has as subject the heading attribute of the relation that has the primary key. The procedure ends when the traversal of the databases graph is complete.
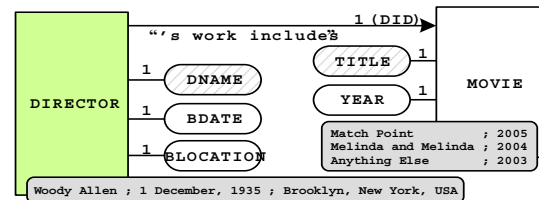


**Figure 6. A part of our example database**

**Query Example (cont'd).** Consider the database instance in Figure 6. At first, we consider the case of "Woody Allen" as a director. We construct the template clause for the DIRECTOR relation:

```
@DNAME + " was born on " +
@BDATE +  " in " + @BLOCATION
```

Afterwards, we built the template clause that derives from the MOVIE relation:

```
@TITLE + " (" + @YEAR + ")"
```

Then, we proceed with the second clause composed by the join relationship between the DIRECTOR and MOVIE relations. The template label of this relationship is represented with the following formula:

```
label(DIRECTOR,MOVIE) =
     expr_1 + @DNAME + expr_2 + MOVIE_LIST
```

The macro MOVIE_LIST and the expressions can be defined as:

```
DEFINE MOVIE_LIST as
  [i<arityOf(@TITLE)]
     {@TITLE[$i$]+" ("+@YEAR[$i$]+"),"}
  [i=arityOf(@TITLE)]
     {@TITLE[$i$]+" ("+@YEAR[$i$]+")."}
  expr_1 ← "As a director,"
  expr_2 ← "'s work includes "
```

Similarly, we proceed with the clause composed by the join relationship between the MOVIE and GENRE relations. Therefore, the result of the précis query for the token "Woody Allen" located in the relation DIRECTOR will be:

*"Woody Allen was born on December 1, 1935 in Brooklyn, New York, USA. As a director, Woody Allen's work includes Match Point (2005), Melinda and Melinda (2004), Anything Else (2003). Match Point is Drama, Thriller. Melinda and Melinda is Comedy, Drama. Anything Else is Comedy, Romance."*

Due to lack of space, we omit the presentation of the case of "Woody Allen" as an actor. As we mentioned before, in absence of any information that both instance values refer to the same physical entity, the answer of the précis query comprises one part for each token occurrence.

## 6. Experimental Results

Experiments were conducted using a prototype system implemented on top of Oracle 9i R2. Our data comes from the Internet Movies Database [18] with information about over 340000 films. We created indexes on all join attributes. Below, we discuss preliminary results of experiments evaluating the algorithms described.

**Evaluation of Result Schema Generator.** The Result Schema Generator is responsible for finding which part of the database schema may contain information most related to Q. Its input is a set of relations containing the query tokens and a degree constraint d. In our experiments, we considered the degree d to be the maximum number of attributes projected in the answer, and we used 20 randomly generated sets of weights for the edges of the database schema graph.
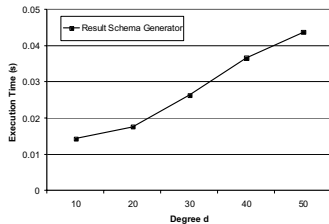


**Figure 7. Result Schema generator execution time**

Figure 7 presents the execution time of the Result Schema Generator as a function of d and considering that query tokens are contained in a single relation $R_o$. We considered 10 different relations as $R_o$. Consequently, each point in this figure represents the average of 200 different experiment runs with the same characteristics. We observe that the execution time of the Result Schema Generator is very small even for large values of d. Overall, (considering also execution times shown in subsequent figures), Result Schema Generator execution time can be considered negligible. This is a safe assumption especially for applications where answers of very large d are not common since these would be far too detailed for serving the purpose of a précis.

**Evaluation of Result Database Generator**. We first present a cost model for the execution time of the Result Database Generator, and then we discuss experimental results that evaluate the generator algorithm as well as the cost model.

*Cost model.* The creation of the result database is performed by submitting to the database a series of selection queries without joins. In particular, having found tuples containing the query tokens, tuples from any other relation $R_j$ are retrieved based on a list of values for the attribute that joins this relation to the result database. Furthermore, it is reasonable to assume that there are indexes on all join attributes, and that the result database fits in memory. Based on the aforementioned, we use the following cost model for the Result Database generator considering only I/O overhead and ignoring the initial overhead for finding the tuples that contain the query keywords:

$$\texttt{Cost(D)} = \sum_{R_i} \texttt{card(R}_i) \cdot (\texttt{IndexTime} + \texttt{TupleTime}) \quad (1)$$

where $\texttt{cost(D)}$ is the cost of generating the result database, $\texttt{card(R}_i)$ is the number of tuples retrieved from relation $R_i$, $\texttt{TupleTime}$ is the time to read a tuple from a relation given its tuple id, and $\texttt{IndexTime}$ is the time to find the tuple id for a given value from its index. Consider that we use a cardinality criterion that specifies a (maximum) number of tuples per relation in D′, i.e. $c_R$, then the previous formula can be written as follows:

$$\texttt{Cost(D)} = c_R \cdot n_R \cdot (\texttt{IndexTime} + \texttt{TupleTime}) \quad (2)$$

where $n_R$ is the number of relations populated in D′.

We now discuss experimental results that evaluate the Result Database generator and its proposed cost model.

Figure 8 presents the execution time of the Result Schema Generator as $c_R$ ranges between 1 and 91 and $n_R$ equals to 4. For this experiment, we have turned off the capability of selecting tuples from a relation in a round-robin fashion. Thus, all queries are executed using *NaïveQ*. We used 10 sets of 4 relations, making sure that there is no relation in any set that does not join with another relation of this set. For each set, we considered each of its relations as the initial relation $R_o$ (containing the query token), and for each $R_o$, we considered 5 random sets of tuples as the seed for producing the instances of the remaining relations. Consequently, each point in the figure represents the average of 200 (10*4*5) different experiment runs with the same characteristics. The figure shows that time increases almost linearly with $c_R$, which seems to be in agreement with Formula (2) as well.

Figure 9 presents the execution time of the Result Database Generator as $n_R$ ranges between 1 and 8 and $c_R$ equals to 5. Each point in the figure was generated following a similar philosophy as above (we do not describe its details for space constraints). Again, we measured the execution time having all queries executed using *NaïveQ*. The figure shows that time increases almost

linearly with $n_R$, which again seems to be in agreement with Formula (2) as well. Based on the experimental results above, Formula (2) seems to be a reasonable approximation of the execution cost of the Result Database Generator.
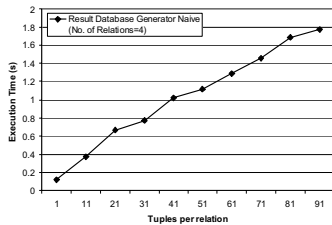


**Figure 8. Result Database generator execution time**

Figure 9 also plots the execution time of the generator, when round-robin is used. The performance of the generator deteriorates with round-robin. However, one has to take into account that, in order to make the execution times of naïve and round-robin comparable, round-robin has been used for every join, not just the *to-n*. In practice, however, the round robin method is applied only wherever required and the naïve is applied everywhere else. Thus, when we have for example 8 relations in the answer, the overall execution time of the generator will be less than the time required to execute round-robin with 8 relations.
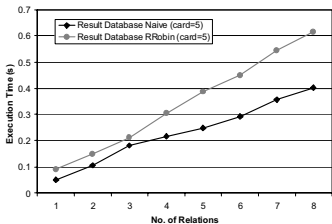


**Figure 9. NaïveQ vs. RoundRobin execution times**

Based on the experimental results above, Formula (2) seems to be a reasonable approximation of the execution cost of the Result Database Generator. Moreover, the Result Database Generator is the most time-consuming part of the system. Consequently, we could define cardinality constraints based on the desired response time of a query, $cost_M$. Then, Formula (2) may be solved for $n_R$ or $c_R$, and this would be a cardinality constraint. E.g.:

$$c_R = \frac{\cos t_M}{n_R(IndexTime + TupleTime)} \qquad (3)$$

## 7. Conclusions and Future Work

In this paper, we introduced the concept of précis queries in the context of databases and presented a framework along with algorithms for their support. Our approach is applicable to other types of (semi-) structured data as well. Also, we presented a semi-automatic method to render the sets of tuples returned in response to a précis query to a "natural language" synthesis of results. Finally, we presented a set of results evaluating each part of the

system implemented showing the potential of this work. The long version of this paper [21] contains further details omitted due to space considerations. In ongoing work, we are investigating the possibility of having weights on data values as well. We are also interested in extending the translator and providing a graphical tool intended for use by a domain expert. Finally, an interesting continuation will be the further optimization of the whole process.

## 8. References

1. Marchionini, G. (1992). Interfaces for End-User Information Seeking. J. of the American Society for Inf. Sci., 43(2), 156-163.
2. Balmin, A., Hristidis, V., Papakonstantinou, Y. (2004). ObjectRank: Authority-Based Keyword Search in Databases. In VLDB'04, 564-575.
3. Motro, A. (1986). BAROQUE: A Browser for Relational Databases. ACM TOIS, 4(2), 4 1986, 164-181.
4. Motro, A. (1986). Constructing Queries from Tokens. In SIGMOD'86, 120-131.
5. Bhalotia, G., Hulgeri A., Nakhe, C., Chakrabarti, S., Sudarshan, S. (2002). Keyword Searching and Browsing in Databases using BANKS. In ICDE'02, 431-440.
6. Masermann, U., Vossen, G. (2000) Design and Implementation of a Novel Approach to Keyword Searching in Relational Databases. J. Stuller et al. (Eds.): ADBIS-DASFAA 2000, LNCS 1884, 171-184.
7. Agrawal, S., Chaudhuri, S., Das, G. (2002). DBXplorer: A System for Keyword-Based Search over Relational Databases. In ICDE'02, 5-16.
8. Hristidis, V. Papakonstantinou, Y. (2002). DISCOVER: Keyword Search in Relational Databases. In VLDB'02, 670-681.
9. Hristidis, V., Gravano, L., Papakonstantinou, Y. (2003). Efficient IR-Style Keyword Search over Relational Databases. In VLDB'03, 850-861.
10. Collins, A., Quillian, M. (1969). Retrieval Time from Semantic Memory. J. of Verbal Learning and Verbal Behaviour, 8, 240-247.
11. Koutrika, G., Ioannidis, Y. (2005). Personalized Queries under a Generalized Preference Model. In ICDE'05.
12. Florescu, D., Kossmann, D., Manolescu, I. (2000). Integrating keyword search into XML query processing. In WWW9.
13. Guo, L., Shao, F., Botev, C., Shanmugasundaram, J. (2003). XRANK: Ranked keyword search over XML documents. In SIGMOD'03, 16-27.
14. Hristidis, V., Papakonstantinou, Y., Balmin, A. (2003). Keyword proximity search on XML graphs. In ICDE'03.
15. Oracle 9i Text. url: www.oracle.com/technology/products/text/index.html
16. MS SQL Server 2000. url: msdn.microsoft.com/library/
17. IBM DB2 Text Information Extender. url: www.ibm.com/software/data/db2/extenders/textinformation/
18. Internet Movies Database. url: at www.imdb.com
19. Dong, X., Halevy, A., Madhavan, J. (2005). Reference Reconciliation in Complex Information Spaces. In SIGMOD'05.
20. Sarawagi, S. (2000). Special Issue on Data Cleaning. Bulletin of the Tech. Committee on Data Eng., Vol. 23, No. 4, 2000.
21. Koutrika, G., Simitsis, A., Ioannidis, Y. (2005). Précis: The Essence of a Query Answer. Extended Version. url: http://www.dblab.ntua.gr/~asimi/publications/KoSI05.pdf