BAROQUE: A Browser for Relational Databases

AMIHAI MOTRO
University of Southern California

The standard, most efficient method to retrieve information from databases can be described as systematic retrieval: The needs of the user are described in a formal query, and the database management system retrieves the data promptly. There are several situations, however, in which systematic retrieval is difficult or even impossible. In such situations exploratory search (browsing) is a helpful alternative. This paper describes a new user interface, called BAROQUE, that implements exploratory searches in relational databases. BAROQUE requires few formal skills from its users. It does not assume knowledge of the principles of the relational data model or familiarity with the organization of the particular database being accessed. It is especially helpful when retrieval targets are vague or cannot be specified satisfactorily. BAROQUE establishes a view of the relational database that resembles a semantic network, and provides several intuitive functions for scanning it. The network integrates both schema and data, and supports access by value. BAROQUE can be implemented on top of any basic relational database management system but can be modified to take advantage of additional capabilities and enhancements often present in relational systems.

Categories and Subject Descriptors: H.2.1 [**Database Management**]: Logical Design—*data models*; H.2.3 [**Database Management**]: Languages—*query languages*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*retrieval models*

General Terms: Design, Human Factors, Languages

Additional Key Words and Phrases: Browsing, database, exploratory search, relational database, user interface

When I look up something in the dictionary, it's never where I look for it first. The dictionary has been a particular disappointment to me as a basic reference work, and the fact that it's usually more my fault than the dictionary's doesn't make it any easier on me. Sometimes I can't come close enough to knowing how to spell a word to find it; other times the word just doesn't seem to be anywhere in the dictionary. I can't for the life of me figure out where they hide some of the words I want to look up. They must be in there someplace.

ANDY ROONEY

1. INTRODUCTION

1.1 Browsing Interfaces

The standard, most efficient method for retrieving information from databases can be described as *systematic retrieval*: The needs of the user are described in a formal query, and the database management system retrieves the data promptly.

Author's address: Department of Computer Science, University of Southern California, Los Angeles, CA 90089.

There are several situations, however, in which systematic retrieval is difficult or even impossible, including the following:

(1) The user is not familiar with the principles employed by the system to organize data (the *data model*).
(2) The user is not familiar with the contents or definition (*schema*) of the particular database to be accessed.
(3) The user is not proficient in the procedures used for the definition and retrieval of the required information (the *data language*).
(4) The user has only a vague retrieval target (e.g., the user is looking for something "interesting" or "suitable").
(5) The user has a clear retrieval target but lacks some of the information necessary to describe it (e.g., the user wants to find out the meaning of a word in a dictionary database, but cannot spell it correctly).

There are many of these situations in real-world environments (e.g., department stores, libraries), and a common solution there is to rely on intuition and embark on an exploratory search. The search often begins at an arbitrary location, and while it is in progress, the person also gains insight into the nature and organization of the searched environment. Eventually, the search either terminates successfully or is abandoned. Such a search technique is often referred to as *browsing*.

This paper describes a new user interface, called BAROQUE (BROwse and QUEry), that implements exploratory searches in relational databases. BAROQUE requires few formal skills from its users. It does not assume knowledge of the principles of the relational data model or familiarity with the organization of the particular database being accessed. It is especially helpful when retrieval targets are vague or cannot be specified satisfactorily.

An interface such as BAROQUE (which we call a *browser*) is expected to increase the usefulness and popularity of relational database management systems. It is not intended to replace systematic retrieval but to be used as a complementary method in any of the situations mentioned above. Such an interface is also useful in conjunction with systematic retrieval, that is, for studying the data before submitting a query or clarifying its failure afterward.

The need for this alternative method of retrieval has already been recognized, and several database management systems have experimented with tools that allow users to explore their environment. Some of these efforts are discussed below.

Cattell [1] designed and implemented an interface to an Entity–Relationship database [4]. The interface features a set of directives for scanning a network of entities and relationships, and presenting each entity, together with its context, in a display called a *frame*. The principles of this interface were carried over to Cypress, a database management system developed by Cattell at Xerox [2]. Cypress starts with a data model based largely on constructs derived from various well-known data models, complementing it with an extensive array of features and tools. In particular, Cypress allows users to browse through the database, displaying its entities and relationships.

Browsing is offered as the principal retrieval method for *loosely structured databases* [16]. Such databases are heaps of facts that do not adhere to any

conceptual design. Since these facts are named binary relationships between data values, the data may be regarded as a network of values. Two styles of browsing, called *navigation* and *probing*, are defined. Both are derived from a standard query language that is based on predicate logic.

In contradistinction to these browsers, the browsing features that have been introduced into relational systems (e.g., SDMS [13], INGRES [19], and DBASE-III [8]) have only limited exploration capabilities. These features are actually tools for scanning relations (including relations that are results of formal queries). Their primary limitation is that browsing is confined to a single relation at a time, and it is not possible to browse across relation boundaries. If a user encounters a value while browsing and wants to know more about it, it is necessary first to determine in what other relations this value may appear (quite difficult), then formulate a standard query, and resume browsing in the new relation. Satisfying questions such as, is x related in any way to y? is impossible without extensively scanning the database.

Although the focus of this paper is on browsing in conventional databases, it is worth noting that exploratory searches have been implemented in related applications. Browsers have been constructed for the Smalltalk programming environment [9] and later for PIE [10], a personal information environment that evolved from Smalltalk. These environments are intended to support the development of software but can be employed to store and manipulate aggregates of data as well. Browsing is also the principal access method in a prototype electronic encyclopedia [23] and in WORDNET [15], a prototype automated English language dictionary, which includes cross-referencing of the entries on the basis of sense relations.

1.2 Design and Implementation of BAROQUE

Three aspects of the design and implementation of BAROQUE are of particular significance:

(1) BAROQUE is designed for a basic relational model, and is implemented "on top" of an existing database management system.
(2) BAROQUE operates on a network view of relational databases that the interface constructs automatically.
(3) BAROQUE requires from its users few formal skills and minimal preparatory knowledge.

These aspects are discussed below in more detail.

Since its introduction, there have been numerous enhancements and extensions of the original relational model [5], and currently there is a great variety of relational database management systems, many with additional features and capabilities (see [6] and [20] for discussion of some of the major extensions). To increase its applicability, BAROQUE assumes the basic model as described by Codd. Nevertheless, certain enhancements, especially those that give more meaning to the attributes of relations, can be used to advantage by the browser for improving its behavior. BAROQUE accesses the stored database through the

query language provided by the existing database management system. Thus, it could be built "on top" of existing systems (an advantage when using commercial products). Actually, the only modification required to the existing database management system is that it must be programmed to update automatically a special new relation (which is used exclusively by the browser) after every update to the other relations. In addition, BAROQUE needs to access the database definition (schema). Such access is now available in many systems.

The browser operates on a *network* view of the given relational database. The relational database is viewed as a network of data items, with named links connecting related items. This representation resembles a semantic binary network [21] and is derived automatically from the given relations. Although the principles of this view could be explained to the user, it is usually unnecessary, since it becomes apparent after some experimentation. One of the common tools provided by database management systems to help users familiarize themselves with the contents and organization of the database is access to the database definition. Although this information may assist users in their explorations, interpreting it requires technical understanding of the relational data model. To avoid this requirement, we incorporate the information present in the definition into the same network view.

The browser provides users with several functions that scan and search this network, allowing them to present items and ask questions such as, What is it? or What is known about it? Such *access by value* is an especially important feature for users with no knowledge of the organization of the database. The browser emphasizes simplicity by using a very simple command language that avoids any relational terminology and by relying on menus. The intention is to provide an interface that can be mastered quickly by naive users. Of course, standard query processors provide more flexible access to the data (but are more complex to use). For sophisticated users who wish to interleave browsing and querying, the interface can switch rapidly between these two activities.

BAROQUE is a prototype system. As such, it implements mostly those features unique to its design. For a complete interface it may be enhanced with additional capabilities; some to consider are listed below. They would provide BAROQUE with abilities to

(1) Recognize *synonyms.* For example, users may ask about `LA` and get information on `Los_Angeles`. This can be achieved through simple modifications of the item dictionary. It will increase system responsiveness, with relatively low overhead and without affecting the database itself.

(2) Accept browsing topics that are *approximate* data values, such as substrings (BAROQUE looks for exact matches only).

(3) Provide *summaries* prior to listing long answers. For example, the answer to `What is known about Mozart`, shown in Section 3.2, would first summarize the list of his compositions with `AUTHOR of COMPOSITION (626 items)`.

(4) Scan long answers flexibly (BAROQUE displays them a windowful at a time).

(5) Accept input via a pointing device (in BAROQUE selection is done by typing) and use graphics to show the current item and its immediate relationships.

Many of these features have been implemented successfully in several database management systems.

1.3 Overview of this Paper

The remainder of this paper is divided into three sections. Section 2 discusses the advantages (for our purposes) of a network view of the data, and shows how such a view can be derived automatically from a given relational database. The design and implementation of BAROQUE are the topics of Section 3. Section 4 concludes with discussions on issues such as semantics, cost, and further possibilities.

2. A NETWORK VIEW OF THE RELATIONAL MODEL

For the purpose of browsing, the tabular representation of the relational data model presents three problems:

(1) *Information about a particular real-world entity may be stored in various places.* In general, users who are unfamiliar with computer data models tend to think in terms of real-world entities and therefore expect all the information pertaining to each entity to be grouped together. However, in relational databases such information may be distributed over several relations. Experienced users who are familiar with the definition of the database and are proficient in a formal query language may be able to extract all this information with a suitable query, but casual browsers are often unable to do so. Some relational database management systems facilitate this task with special mechanisms. For example, the definition of a database may include certain interfile links over common fields; then, when viewing a record in one file, the user may ask to *cross over* to the other file and view the associated records (a typical implementation of such a feature is available in POWER-BASE [17]). A relational browser should be able to assemble automatically all the information from the database that pertains to the browsing topic.

(2) *A tabular representation introduces structural boundaries and lacks explicit links that users may follow.* A common technique for exploratory searches is to start with a known item of information, use it to uncover some additional information, then follow this information to still other information. The tabular representation of relational databases is not particularly suitable for such search processes. When a relational database system delivers an item of interest, there is no immediate way to follow this lead. For example, when a university database returns the name of an instructor as part of its answer to a query about a particular course, finding additional information about this instructor usually requires consulting the definition of the database and formulating another query.

(3) *All requests for information must include references to the definition of the database.* During the course of human interaction, information may be obtained by naming particular entities, as in: What is the population of Los Angeles? To obtain this information from a relational database, it must first be determined in which *relation* and under what *attribute* the *value* `Los_Angeles` may appear. In other words, most relational databases do not support access by value. This limitation can be explained by the observation that relational database management systems implement a mapping from the data dictionary (attribute names)

into the database (values), but not the inverse mapping. Given an attribute, it is possible to retrieve all its values, but, given a value, it is not possible to retrieve the corresponding attributes. For example, one can list all values of the attribute `CITY.NAME`, but it is impossible to list all the attributes of which `Los_Angeles` is a value (i.e., `CITY.NAME`, `UNIVERSITY.LOCATION`, and `OLYMPIAD.SITE`). By permitting access by value, a user interface approximates more closely the style of human interaction (note that such access is an essential feature of natural language interfaces to databases, such as INTELLECT [11] or LADDER [12]).

These problems suggest that, for the purpose of browsing, a network representation of the data may be more satisfactory. In a network representation each real-world entity is modeled with one database item, and specific links are established between related items. Relatively few modifications are necessary to provide relational databases with network views (and make their actual tabular representation transparent). We present our solution in four steps: First, we define simple items; then, we define relationships between items; next, the model is extended to allow composite items; and, finally, the definition of the database is incorporated into the same model.

2.1 Items

All the occurrences of a particular data value v in a relational database are considered collectively to be one abstract item called v. For example, the value `Los_Angeles` may appear in the database under the attributes `CITY.NAME`, `UNIVERSITY.LOCATION`, and `OLYMPIAD.SITE`; together, these occurrences represent an item called `Los_Angeles`.[1]

Assembling this item, of course, requires that all the different occurrences of a value be accessible through the item name. As mentioned earlier, relational databases cannot be accessed with values alone: It is also necessary to provide the attribute names under which these values may be found. To enable such access by value, a mapping from values into attribute names is needed: Given a value, this mapping determines the attributes under which it appears. Such a mapping can be used to correlate all the different occurrences of a value in the database and will therefore serve as an *item directory*. This item directory is implemented with an additional relation that has two attributes: *value* and *attribute*. A pair (v, a) in this relation states that the value v appears under the attribute a. This relation cannot be modified by the users; the system should automatically update it to reflect user updates to the other relations. (This is similar to the way INGRES [18] handles secondary indexes.)

2.2 Item Relationships

After the data items that exist in a given relational database have been defined, the next step is to define the item relationships that are implied by this database.

Item relationships are based on the *functional dependencies* that exist in the database. Each relation embeds several such dependencies, and those that involve the key are known to the database management system. Specifically, in each

[1] Note that when the same data value represents several different real-world entities, the item created will not have clear semantics. This issue is discussed in Section 4.

relation every nonkey attribute is functionally dependent on the key attribute.[2] Consequently, each *value* is related through functional dependencies to other values in the same tuple. Since each data item combines all the occurrences of a particular value in the database, the relationships of this item to other items are based on all the dependencies in which these occurrences participate. Note that this item may be the source of a functional dependency in one relation, and the target of a functional dependency in another.

As an example, consider a database called **MUSIC** with three relations (key attributes are underlined):

```
COMPOSER = (NAME, COUNTRY, PERIOD, YEAR_OF_BIRTH,
            YEAR_OF_DEATH)
COMPOSITION = (TITLE, AUTHOR, TYPE)
PERIOD = (NAME, START_YEAR, END_YEAR)
```

Consider the value **Mozart**. It appears once in the **COMPOSER** relation: (**Mozart, Austria, Classical, 1756, 1791**) and many times in the **COMPOSITION** relation, for example, (**Jupiter, Mozart, Symphony**), (**Magic_Flute, Mozart, Opera**), and (**Hunt, Mozart, Quartet**). On the basis of these tuples, the item **Mozart** is related to seven other items: **Austria, Classical, 1756, 1791, Jupiter, Magic_Flute** and **Hunt**. By concatenating the relation name and the attribute names involved in each functional dependency, meaningful names for the relationships can be obtained. For example, the relationships between **Mozart** and **Jupiter** and between **Mozart** and **Austria** are, respectively, **is-AUTHOR-of-COMPOSITION-having-TITLE**, and **is-NAME-of-COMPOSER-having-COUNTRY**. The complete list of the relationships of **Mozart** is as follows:

```
Mozart  is-NAME-of-COMPOSER-having-COUNTRY  Austria
Mozart  is-NAME-of-COMPOSER-having-PERIOD  Classical
Mozart  is-NAME-of-COMPOSER-having-YEAR_OF_BIRTH  1756
Mozart  is-NAME-of-COMPOSER-having-YEAR_OF_DEATH  1791
Mozart  is-AUTHOR-of-COMPOSITON-having-TITLE  Jupiter
Mozart  is-AUTHOR-of-COMPOSITION-having-TITLE  Magic_Flute
Mozart  is-AUTHOR-of-COMPOSITION-having-TITLE  Hunt
```

A small portion of database **MUSIC** is shown in Figure 1. The network of items that corresponds to this portion (without **YEAR_OF_BIRTH** and **YEAR_OF_DEATH**) is shown in Figure 2. Note that all edges represent two-way relationships. For example, **Mozart** is related to **Jupiter** via **is-AUTHOR-of-COMPOSITION-having-TITLE**, and **Jupiter** is related to **Mozart** via **is-TITLE-of-COMPOSITION-having-AUTHOR**.

Formally, assume a relation $A = (\underline{A_1}, \ldots, A_m)$ and let $(a_1, \ldots, a_m)$ be a tuple in A. The following item relationships are implied by this tuple:

(1) Item a_1 is related to item $a_i (i = 2, \ldots, m)$ via *is-A_1-of-A-having-A_i*.
(2) Item $a_i (i = 2, \ldots, m)$ is related to item a_1 via *is-A_i-of-A-having-A_1*.

[2] For now, we only consider relations that have simple (i.e., single-field) keys. Relations with composite keys are considered later.

COMPOSER

NAME	COUNTRY	PERIOD	YEAR_OF_BIRTH	YEAR_OF_DEATH
Bach	Germany	Baroque	1685	1750
Haydn	Austria	Classical	1732	1809
Mozart	Austria	Classical	1756	1791

COMPOSITION

TITLE	AUTHOR	TYPE
Surprise	Haydn	Symphony
Jupiter	Mozart	Symphony
Hunt	Mozart	Quartet

PERIOD

NAME	START_YEAR	END_YEAR
Baroque	1600	1750
Classical	1750	1800
Romantic	1800	1900

Fig. 1. Portion of database MUSIC.

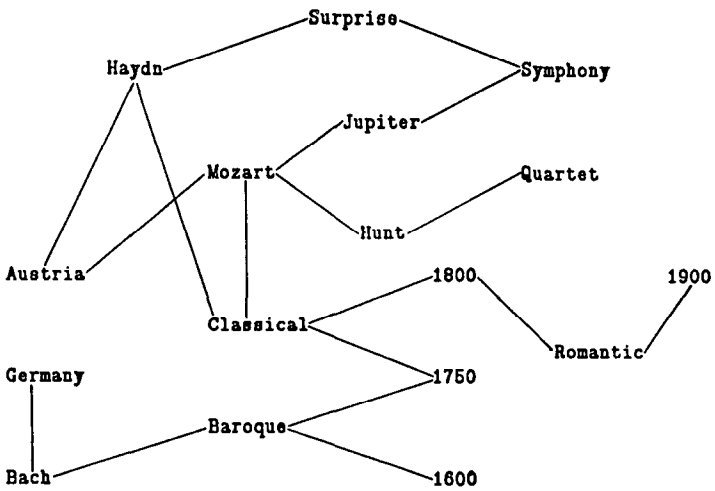

Fig. 2. View of database MUSIC as a network of items.

2.3 Composite Items and Their Relationships

Consider again the relation **COMPOSITION**. Since different composers may have authored compositions with the same title (such as **Symphony_no_1** or **Pathetique**), a more realistic assumption is that **COMPOSITION** has a composite key: **COMPOSITION** = (**TITLE**, **COMPOSER**, **TYPE**). In this case **TYPE** is functionally dependent on a *combination* of **TITLE** and **COMPOSER**. To define item relationships in such cases, we introduce the notion of a *composite* item, which is a combination of items. For example, the items **Pathetique** and **Tchaikovksy** are combined to create the composite item (**Pathetique**, **Tchaikovsky**), whose **TYPE** is **Symphony**. Another composite item is (**Pathetique**, **Beethoven**); its **TYPE** is **Sonata**. A composite item occurs in the database wherever

its components appear *in the same tuple* of some relation under the *key attributes.* Composite items need not have separate entries in the item directory, since they can be located through the entries of their components.

Notice that the individual components of the key are themselves functionally dependent on the key. These so-called *trivial* dependencies are important, since they help establish relationships from components of the key to other values of the tuple. For example, **Pathetique** is related to both (**Pathetique, Tchaikovsky**) and (**Pathetique, Beethoven**), which in turn are related, respectively, to **Symphony** and **Sonata.**

In the previous example, let the attribute **IDENTIFICATION** denote the pair (**TITLE, AUTHOR**). The change in the key of **COMPOSITION**[3] affects the last three relationships of **Mozart**, as follows:

```
Mozart   is-AUTHOR-of-COMPOSITION-having-IDENTIFICATION
         (Jupiter, Mozart)
Mozart   is-AUTHOR-of-COMPOSITION-having-IDENTIFICATION
         (Magic_Flute, Mozart)
Mozart   is-AUTHOR-of-COMPOSITION-having-IDENTIFICATION
         (Hunt, Mozart)
```

Formally, assume a relation $A = (\underline{A_1}, \ldots, \underline{A_p}, A_{p+1}, \ldots, A_m)$, denote $(A_1, \ldots, A_p)$ by α, and let $(a_1, \ldots, a_p, a_{p+1}, \ldots, a_m)$ be a tuple in A. The following item relationships are implied by this tuple:

(1) Item $(a_1, \ldots, a_p)$ is related to item $a_i (i = 1, \ldots, m)$ via *is-α-of-A-having-A_i.*

(2) Item $a_i (i = 1, \ldots, m)$ is related to item $(a_1, \ldots, a_p)$ via *is-A_i-of-A-having-α.*

2.4 Incorporating Schema Elements into the Model

A useful feature of many database management systems is to allow users to retrieve information about the definition (schema) of the database. A database schema describes the structure of the database; in the relational model this description usually includes the name of the database, the names of its relations, the attributes of each relation (including the designation of the key attributes), and the data types of the attributes. Although this information may assist users in their explorations, it requires technical understanding of the relational data model. To avoid this requirement, we incorporate the information present in the schema into the network of items. This uniform representation of schema and data is an important convenience: Virtually all database interfaces perpetuate the dichotomy between schema and data when this distinction may be of concern to database designers, but the distinction is not always clear to casual database users.

For this purpose, a relational schema is perceived as a conceptual hierarchy of three levels: the database, the relations, and the attributes. Each element of this hierarchy is represented by a separate item, and several special relationships are introduced: The relationship *contains-information-on* relates the database item to each relation item, the relationship *is-identified-by* relates every relation item to its key attributes, and the relationship *has-attribute* relates every relation item to its nonkey attributes. When composite items are present, the hierachy is

[3] Of course, key of **COMPOSITION** refers here to database key, not musical key!

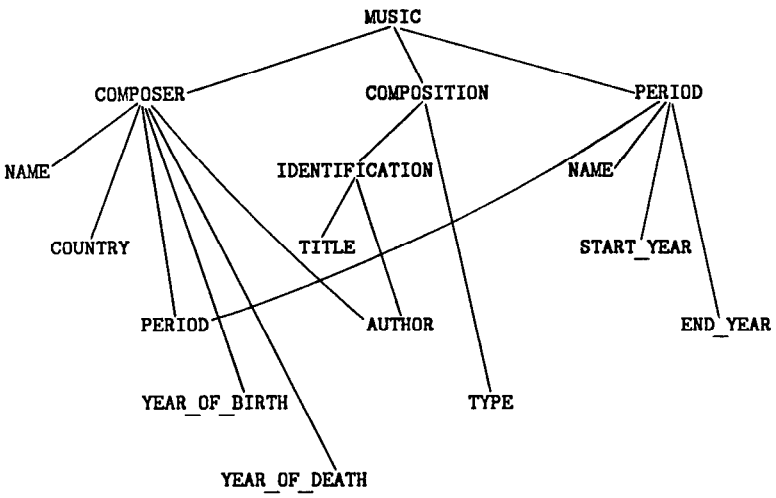

Fig. 3. View of schema of database MUSIC as a network of items.

extended with an additional level: A composite item is related to each of its components with *is-combination-of*.

Consider now the attributes **NAME** (of **COMPOSER**) and **AUTHOR** (of **COMPOSITION**). The values of both attributes are drawn from the same domain, but in relation **COMPOSER** this attribute is a key, while in relation **COMPOSITION** it is not. Usually, this indicates that **COMPOSER** has additional information on each **AUTHOR**. If it is known that these attributes are from the same domain, then a new type of relationship is established between them: *may-have-additional-information-on*.[4] The complete network of items that corresponds to the schema of the relational database **MUSIC** is shown in Figure 3. Every edge in this figure should be labeled with the appropriate relationship name (and every relationship has an appropriately named inverse). Some examples are

```
MUSIC  contains-information-on  COMPOSER
COMPOSER  is-identified-by  NAME
COMPOSER  may-have-additional-information-on  AUTHOR
```

Finally, to connect the schema network with the previous item network, a relationship called *includes* is established between every attribute and its values (its inverse is called *is*). For example:

```
AUTHOR includes Bach, Beethoven, Handel, Haydn, Mozart,
       Tchaikovsky
Austria is COUNTRY
```

Consequently, a relational database is viewed as an integrated network of concepts and values. The most general concept is the name of the database,

[4] The availability of such additional information is discussed in Section 4. An attribute such as **AUTHOR** is sometimes called a *foreign key*. To improve data integrity, some systems can require each value of the foreign key to occur also as a value of the corresponding key. Such a requirement is known as a *referential integrity constraint* [7].

which then leads to relation names. Each relation name leads to its attribute names, which, in turn, lead to their individual values. Individual values lead to other values.

3. BROWSING IN A RELATIONAL DATABASE

The network view of relational databases creates a favorable environment for exploratory searches. Four functions (requests) that scan the network in four different ways are defined. These functions, called: What is it? What is known about it? What is the connection? and Any others like it? accept names of items and return names of other items or names of relationships. First, we describe the functions and then the actual user interface constructed around them, called BAROQUE.

3.1 What Is It?

The special relationships introduced in Section 2.4 (and their inverses) create a classification hierarchy that extends from the name of the database to the data values. The browsing request: What is it? *classifies* items by returning their position in this hierarchy. The information needed to classify data items is readily available from the item directory. For example.[5]

```
>What is Mozart?
 Mozart is NAME of COMPOSER, AUTHOR of COMPOSITION
>What is Jupiter?
 Jupiter is TITLE of COMPOSITION
>What is (Jupiter, Mozart)?
 (Jupiter, Mozart) is IDENTIFICATION of COMPOSITION
```

In order to classify schema items, the stored schema is consulted. The following sequence demonstrates how classification may be used to gain familiarity with the contents of the database. Note that, in the beginning, the user knows only the name of the database (`MUSIC`). Each answer then provides him with a topic for another request for classification.

```
>What is MUSIC?
 MUSIC is the database
 MUSIC includes information on COMPOSER, COMPOSITION, PERIOD
>What is COMPOSITION?
 COMPOSITION is part of database MUSIC
 COMPOSITION is identified by IDENTIFICATION
 COMPOSITION has attribute TYPE
>What is IDENTIFICATION?
 IDENTIFICATION is identifying attribute of COMPOSITION
 IDENTIFICATION is a combination of TITLE, AUTHOR
>What is AUTHOR?
 AUTHOR is part of IDENTIFICATION
 AUTHOR is specified in 1-16 characters
 COMPOSER  may-have-additional-information-on  AUTHOR
```

[5] For clarity, when a relationship name and a source item repeat with different target items, they are listed only once, with all the applicable target items.

3.2 What Is Known about It?

Detailed information on items can be obtained with the browsing request: What is known about it? This request *describes* items by listing all the relationships in which it participates, other than those used to classify it (this list is called a *description*). These relationships are obtained through the item directory and the stored schema. Some examples are

```
>What is known about Mozart?
 Mozart is
 NAME of COMPOSER having
         COUNTRY Austria
         PERIOD Classical
         YEAR_OF_BIRTH 1756
         YEAR_OF_DEATH 1791
 AUTHOR of COMPOSITION having IDENTIFICATION
         (Hunt, Mozart)
         (Jupiter, Mozart)
         (Magic_Flute, Mozart)
>What is known about (Magic_Flute, Mozart)?
 (Magic_Flute, Mozart) is
 IDENTIFICATION of COMPOSITION having TYPE Opera
```

For schema items, the same request returns a list of all the instances of that item. In particular, for an attribute it lists all the data values that appear under it; for a relation, all the tuples in that relation; for a database, all the relations in the database. (Of course, in practice, these requests should be verified before they are performed.) Of these three requests, the most useful is the first, since it enables a smooth transition from the schema subnetwork to the item subnetwork. For example:

```
>What is known about AUTHOR?
 AUTHOR includes Bach, Beethoven, Brahms, Handel, Haydn,
        Mozart, ...
```

3.3 What Is the Connection?

When information cannot be located by navigating on the network, it may be useful to present the browser with *two* items and request that it attempt to establish a connection between them. The browsing request: What is the connection? searches for paths of relationships between the two given items. As an example, consider the request:

```
>What is the connection between Bach and Baroque?
```

Two paths are returned:

```
1. Bach is NAME of COMPOSER having PERIOD Baroque
2. Bach is NAME of COMPOSER having YEAR_OF_DEATH 1750 which is
   END_YEAR of PERIOD having NAME Baroque
```

While the information revealed by first path may seem obvious ("Bach was a composer during the Baroque period"), the second path may be a discovery of sorts: The year Bach died is considered the end of the Baroque period." Note that the two connections use different occurrences of **Baroque**, and that **Bach** and **Baroque** in the latter connection occur in two different relations.

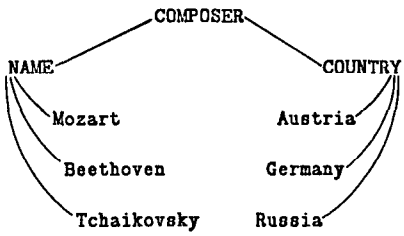

Fig. 4. Portion of the integrated network of items.

Consider now a request to connect **Verdi** and **Opera**. Since Verdi composed many operas, there will be numerous paths, each leading from **Verdi**, to one of his operas, to the type **Opera**. Such parallel paths should be abstracted into a single connection (such paths are easy to detect, since they happen only when the two items occur as values of two nonkey attributes in the same relation).

An item that occurs in two relations establishes connections between all items that occur in either tuple (in the previous example, **1750** occurs in both **COMPOSER** and **PERIOD** and establishes a connection between **Bach** and **Baroque**). With such intermediaries, connections may be established between almost any two items. Searching a path of connections between two items could become very costly as the length of the path increases. However, the significance of the path declines rapidly with its length. Consequently, limiting path length to a small value, such as three or four, will produce most of the significant paths in reasonable time.

Still, some short paths may have little significance. Consider the portion of the item network described in Figure 4. With *includes* relationships, two data items may be connected simply because they are instances of two related attribute items; for example, **Mozart** and **Russia** may be connected through **NAME**, **COMPOSER**, and **COUNTRY**. Similarly, two attribute items may be connected through every relationship between their data items; for example, **NAME** and **COUNTRY** may be connected through **Mozart** and **Austria**, **Beethoven** and **Germany**, and so forth. Paths of both types are avoided if relationships between two schema items are not used for a path between two data items, and relationships between two data items are not used for a path between two schema items.

3.4 Any Others like It?

Occasionally, it may be useful to browse "by an example": the user presents the browser with an item and the browser returns similar items. The browsing request: Any others like it? is different from the previous three in that it may require further user involvement. Given an item, the browser attempts to find other items that have a similar description. Since most items have elements in their descriptions that are unique to them, usually these requests cannot be satisfied without relaxing some of these constraints. This is done by accompanying the failure message with a list of relationships that can be matched individually (and the total number of matched items). The user then selects the relationships that are relevant to the request (or abandons it altogether). Consider, for example, the request:

```
>Any others like Mozart?
```

The uniqueness of the relationships derived from the relation **COMPOSITION** (e.g., only **Mozart** authored a composition entitled (**Jupiter, Mozart**)) results in no matched descriptions. Therefore, the outcome is

```
None. There are
17 other  NAME-of-COMPOSER-having-COUNTRY  Austria
23 other  NAME-of-COMPOSER-having-PERIOD  Classical
```

Assuming the user selects both relationships, the answer would be[6]

```
Others like Mozart: Haydn
```

Of course, selecting relationships that match successfully does not guarantee the successful matching of their conjunction, so the second attempt can fail too. Still, as the example demonstrates, this process crystalizes requests that at first may be quite vague ("Others like Mozart") into specific queries ("Other Austrian composers of the Classical period"). Alternatively, an algorithm for matching descriptions may be used; it lists the data items whose descriptions are *closest* to the description of the topic of the request (i.e., share with it the most data items).

Together, the four browsing requests constitute a simple, yet flexible, tool to search the item network. What is it? can be used as a blind attempt to find the meaning of a value. What is known about it? gives fuller descriptions and provides the basic form of navigation. The other requests are convenient for particular types of searches.

3.5 The User Interface

BAROQUE has three modes of operation: *main*, *query*, and *browse*; each mode is reachable from the other two modes. When invoked, BAROQUE goes into its main mode, where the user selects a database and may examine some of its global parameters, such as *size* or *description*. To interact with the selected database through a standard relational query language, the user switches to query mode. For browsing, the user switches to browse mode. This simple architecture enables rapid switches back and forth between browsing and querying in the selected database. To select a different database, the user returns to main mode.

To store and manipulate the databases, BAROQUE uses the UNIFY [22] database management system. Since UNIFY supports SQL [3] as its primary retrieval language, SQL is also the language used in query mode. In this mode, BAROQUE simply solicits SQL queries from the user, submits them to the SQL processor, and returns the answers (or messages) to the query screen.

Browsing is an iterative process, in which the user supplies a browsing *topic* and a browsing *request*, and BAROQUE returns its *findings*. Accordingly, the browse screen features several windows. They display the current mode, the selected database, the current topic, the current request, and the latest findings. After the user enters a data value in the topic window and selects a request from the request menu (existing topic or request are the default values), BAROQUE fills in the findings window with the appropriate data values and relationships. Typically, the user then selects one of the data values in the findings window as

[6] This answer should not be interpreted too literally!

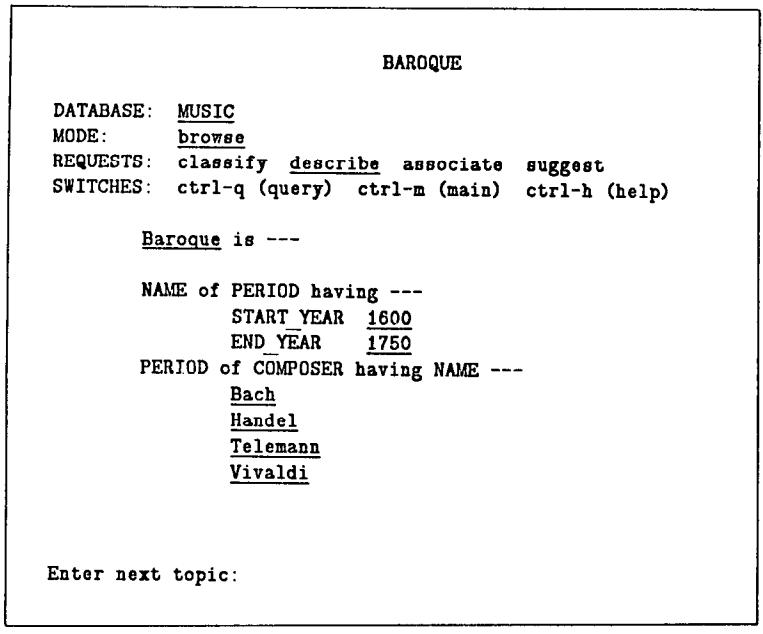

Fig. 5. The browse screen of BAROQUE.

the next browsing topic and selects a new browsing request.[7] Thus, necessary interaction is kept to a minimum, with the combined advantage of simplicity (everybody can learn to browse in a matter of minutes) and efficiency.

A typical snapshot of a browse screen is shown in Figure 5. The requests *classify*, *describe*, *associate*, and *suggest* implement, respectively, the requests: What is it? What is known about it? What is the connection between them? and Any others like it? The database name is **MUSIC**, the mode is **browse**, the request is **describe**, and the topic is **Baroque**. Underlined strings are actually highlighted to provide instant identification of database values.

Upon entry to browse mode, the name of the database is established as the default topic. Thus, this most general value is provided to the user as the end of a thread. By following it, the user may survey the database and ultimately reach every other value.

4. CONCLUSION

A user interface such as BAROQUE may be implemented with relative ease on top of any relational database management system that provides database access from a host language. Note that the database itself is never modified, except for the additional directory relation. All other applications and user interfaces are unaffected by BAROQUE.

Experience gained with BAROQUE indicates that it is mastered very quickly by users without any database experience. In particular, the organization of the

[7] Clearly, this calls for implementation using a pointing device. Currently, the user must retype these items (or leave them unchanged).

data in a relational model is transparent, since it gives the illusion of a network of items and relationships. There are few technical details, and the browsing requests are very intuitive. We note that so far the system has been used with databases of modest size and complexity; its performance with larger volumes of data still needs to be tested.

The cost entailed by the browser, in terms of the additional space to store the item directory and the additional computation for its initialization and its continuous update, is comparable to the cost of a secondary index on every database attribute. If sufficient storage is unavailable, it is possible to implement only part of the item network, by inverting on selected attributes only. For example, values of **YEAR_OF_BIRTH** and **YEAR_OF_DEATH** could be left out of the item directory. These values will be listed while browsing in their neighborhoods (e.g., with requests such as What is known about Mozart?), but they may not become topics of browsing requests. Selective inversion has the interesting effect of distinguishing between actual *items* that participate in relationships and simple *properties* that describe items. In fact, the resulting model resembles the Entity–Relationship approach to data modeling [4]. One possible strategy for selective inversion is to invert only on attributes that are keys or foreign keys. Under this strategy, every item that is assembled occurs at least once as the value of the key in some relation. For effective browsing, the item directory should be implemented with an efficient access method, such as hashing or indexing [7].

Another performance issue is the processing of requests. Consider, for example, the way BAROQUE handles a request to *describe* an item. First, it issues a selection query to the item directory. Then, for each entry found in the item directory, it issues a selection query to a database relation. The answers are then combined to form a description. This process could be speeded up substantially if the item directory contained actual pointers to the tuples that include this item, instead of just references to the attributes under which this item may be found. However, in addition to the increased overhead for maintaining this new item directory, processing of browsing requests could no longer be done through the query language alone, but would require modifications to the underlying database management system.

Our method for assembling data items is based only on identities of data values. Consequently, values that possess different meanings altogether, but are expressed with the same string of characters, are assembled into one item (e.g., the period **Baroque** and the database interface by this name). This weakness, sometimes referred to as the "connection trap," can be attributed to the limited semantic capabilities of the basic relational model, in which the only information available on the meanings of the different attributes are their names and their primitive types (e.g., integer, character). A well-known enhancement to the relational model [14] uses a stronger concept of *abstract domains* to classify the attributes. This enhancement can be readily incorporated into BAROQUE to assemble separate items for values that belong to multiple domains. For example, assume a database with attributes **SALARY**, **PRICE** defined over the domain **DOLLARS**, and attributes **YEAR_OF_BIRTH** and **YEAR_OF_DEATH** defined over the domain **YEARS**. If the value **1685** appears under both **PRICE** and

YEAR_OF_BIRTH, BAROQUE will create two separate items: 1685 DOLLARS and 1685 YEARS.

Another semantic enhancement, available in several systems, is the definition of *referential integrity constraints* in the schema of the database. A referential integrity constraint [7] links an attribute of one relation to the key attribute of another relation, requiring every value of the former attribute to occur also as a value of the latter attribute. Typically, these "cross-references" are used to enhance the integrity of the database. However, like abstract domains, referential integrity constraints provide strong evidence of the similarity of two attributes and can, therefore, be used to guide the assembly of values into items; two occurrences of the same value under two attributes that are related by a referential integrity constraint may be assembled safely into one item. Note that relying exclusively on integrity constraints for assembling items implies that only values of key attributes participate in items. This suggests using integrity constraints in tandem with the selective inversion strategy offered above, which inverts only on keys and foreign keys.

Notice, however, that even with the current approach, the names of relationships in which 1685 participates provide different *interpretations* for this item. For example, BAROQUE, will *classify* the topic 1685 as both PRICE of ITEM and YEAR_OF_BIRTH of COMPOSER. Similarly, it will *describe* it as PRICE of ITEM having ITEM_NO 6710 and YEAR_OF_BIRTH of COMPOSER having NAME Bach. Thus, while the information included in these answers combines different semantics of the item 1685, it is interpreted clearly, and the user can disregard the portion of the answer that is irrelevant, thus avoiding any "traps."

REFERENCES

1. CATTELL, R. G. G. An entity-based database interface. In *Proceedings of ACM-SIGMOD International Conference on Management of Data* (Santa Monica, Calif., May 14–16). ACM, New York, 1980, pp. 144–150.
2. CATTELL, R. G. G. Design and implementation of a relationship-entity-datum data model. CSL-83-4, Xerox Palo Alto Research Center, Palo Alto, Calif., May 1983.
3. CHAMBERLIN, D. D., ET AL. SEQUEL 2: A unified approach to data definition, manipulation, and control. *IBM J. Res. Dev. 20*, 6 (Nov. 1976), 560–575.
4. CHEN, P. P. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst. 1*, 1 (Mar. 1976), 9–36.
5. CODD, E. F. A relational model for large shared data banks. *Commun. ACM 13*, 6 (June 1970), 377–387.
6. CODD, E. F. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst. 4*, 4 (Dec. 1979), 397–434.
7. DATE, C. J. *An Introduction to Database Systems*, vol. I. 3rd ed. Addison-Wesley, Reading, Mass., 1982.
8. DBASE-III. *Reference Manual.* Ashton-Tate, Culver City, Calif., 1984.
9. GOLDBERG, A., AND ROBSON, D. A metaphor for user interface design. In *Proceedings of the 13th Hawaii International Conference on System Science* (Honolulu, Jan. 3–4). Univ. of Hawaii, Honolulu, 1980, pp. 148–157.
10. GOLDSTEIN, I., AND BOBROW, D. Browsing in a programming environment. In *Proceedings of the 14th Hawaii International Conference on System Science* (Honolulu, Jan. 8–9). Univ. of Hawaii, Honolulu, 1981.
11. HARRIS, L. R. Natural language front ends. In *The AI Business*, P. H. Winston and K. A. Prendergast, Eds. MIT Press, Cambridge, Mass., 1984.

12. HENDRIX, G. G., ET AL. Developing a natural language interface to complex data. *ACM Trans. Database Syst. 3*, 2 (June 1978), 105–147.
13. HEROT, C. Spatial management of data. *ACM Trans. Database Syst. 5*, 4 (Dec. 1980), 493–513.
14. MCLEOD, D. J. High level definition of abstract domain in a relational data base system. *Comput. Languages 2*, 3 (July 1977), 61–73.
15. MILLER, G. A. Dictionaries of the mind. In *Proceedings of 23rd Annual Meeting of the Association for Computational Linguistics* (Chicago, July 8–12). Association for Computational Linguistics, Morristown, N.J., 1985, pp. 305–314.
16. MOTRO, A. Browsing in a loosely structured database. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, (Boston, June 18–21). ACM, New York, 1984, pp. 197–207.
17. POWER-BASE. *Reference Manual.* Power-base Systems, New York, 1983.
18. STONEBRAKER, M., ET AL. The design and implementation of INGRES. *ACM Trans. Database Syst. 1*, 3 (Sept. 1976), 189–222.
19. STONEBRAKER, M., AND KALASH, J. TIMBER: A sophisticated database browser. In *Proceedings of the Eighth International Conference on Very Large Data Bases* (Mexico City, Sept. 8–10). VLDB Endowment, Saratoga, Calif., 1982, pp. 1–10.
20. STONEBRAKER, M., AND ROWE, L. A. The design of POSTGRES. UCB/ERL 85/95, Electronics Research Laboratory, College of Engineering, Univ. of California at Berkeley, Nov. 1985.
21. TSICHRITZIS, D. C., AND LOCHOVSKY, F. H. *Data Models.* Prentice Hall, Englewood Cliffs, N.J., 1982.
22. UNIFY. *Reference Manual.* 3rd ed. UNIFY Corporation, Lake Oswego, Oreg., 1983.
23. WEYER, S. A., AND BORNING, A. H. A prototype electronic encyclopedia. *ACM Trans. Off. Inf. Syst. 3*, 1 (Jan. 1985), 63–88.

Received December 1984; revised July 1985; accepted April 1986.