TENTACLE: A Graph-Based Database System

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF SCIENCE
AT THE UNIVERSITY OF CAPE TOWN
IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Marc Gerhard Welz
1999

Academic Supervisor: Associate Professor P. T. Wood
Administrative Supervisor: Dr A. C. M. Hutchison

Abstract

With the advent of large and complex applications and the emergence of semi-structured information repositories such as the World Wide Web, new demands are being made on database systems.

The TENTACLE database system is an experimental database system which provides facilities capable of meeting some of these demands. The distinguishing features of the system are that it:

- uses a graph-based data model (and storage subsystem) to provide a flexible means of representing poorly structured information,
- integrates a path expression-based query language with a general purpose language to query and manipulate the graph structures, thereby eliminating the impedance mismatch encountered in a two language system, and
- provides a programmable database kernel capable of executing the combined query and utility language, allowing the construction of domain specific applications inside the database without the assistance of wrappers or gateways.

As a demonstration of the utility of the system, I have constructed a hypertext server inside the TENTACLE database without making use of external mediators or gateways. Since the hypertext server program is part of the database content, database facilities may be used to assist in the creation and maintenance of the hypertext server itself. In addition, the close integration of hypertext server and database simplifies tasks such as the management of associations between hypertext entities or the maintenance of different document views.

Contents

Chapter 1

Introduction

The goal of the TENTACLE project was to build an alternative database system. The system is intended to be an experimental (yet usable) platform designed to explore a number of unconventional ideas which are currently unlikely to be encountered in more mainstream systems.

Since the system has been built for the sake of exploring a different permutation of design decisions and establishing whether these are viable, the contribution of the system lies more in the exploration of the space of possible database systems and not necessarily in the introduction of a particular capability.

By choosing to build a complete, stand-alone database it has been possible to introduce alternatives in a number of significant database components. Furthermore, the complete implementation makes it possible to apply this experimental system to a real-life problem domain and to acquire empirical results — something which would have been difficult with an on-paper simulation.

When building such an experimental database system it is interesting to select a set of alternative approaches and components in the hope of producing a system which exhibits strengths in areas where conventional databases are weak — this helps to justify the existence of the database system.

1.1 Conventional Database Systems

Conventional databases have been designed for applications such as payroll processing, stock or inventory management, flight reservation systems and banking transactions. These applications tend to exhibit the following properties:

Known Database Schema and Structure: The aspect of the world to be modeled by the database can be described beforehand. This allows the analysis and creation of schema information and typing structures before the system becomes operational. For example, it is usually known in advance that the canonical toy banking system requires operations to withdraw, deposit and transfer funds as well as an operation to make a balance enquiry.

Informed User Population: Users and their agents are aware of the database schema and know where to look for a particular data item. In the case of the toy banking system the software running on the ATM (Automatic Teller Machine) implicitly contains the information to select an account balance entry from the correct table, while the expert intent on discovering ATM usage trends is explicitly aware of the database schema. In other words the users know where a particular piece of information is located.

1.2 Alternative Database Systems

Conventional databases perform very well within the constraints given in the previous section. However, there are a number of application domains which do not exhibit the abovementioned properties of comparative simplicity and static structure amenable to prior analysis. Such applications include hypertext systems, databases containing research results and PIMs (Personal Information Managers). Recently these applications have been described as having a poorly defined structure — they have been called *semi-structured* applications [13, 19].

It appears to be accepted that plain relational databases have difficulties meeting the requirements of such applications [21, 57]. Consequently significant amounts of effort have been directed at finding more suitable systems. Thusfar no definitive database solution seems to have been found[1] — this seems to be borne out by the fact that semi-structured applications still tend to use no more than structured files as their storage subsystems (for example HTML [16]), although it is clear that they would benefit from the more advanced capabilities of complete database systems.

In light of these circumstances it seemed interesting to make provisions in the TENTACLE database systems for semi-structured applications. These provisions take the form of a data model, query language and implementation

[1]Object-orientated databases still tend to require advance analysis of the problem domain to set up class hierarchies.

which may be adapted more easily to a particular problem domain and which seem to be more suited to the task of representing and manipulating a poorly defined structure than a typical relational system. A brief overview of these three features is provided below.

- The data model used by the TENTACLE system is untyped and graph-based and is thus similar to the ones used by LORE [48] or STRUDEL [30]. The advantage of using a graph-based structure is that it is comparatively easy to model associations between entities by representing entities (or attributes of entities) as nodes and their interrelationships as edges.

- The TENTACLE query system makes use of path expressions over the database graph. Path expressions on graphs are similar to regular expressions on strings, where regular expressions match character sequences, path expressions match sequences of nodes and edges. Path expressions appear to be a natural extension of navigational access methods used in hypertext or file systems and should assist the user in traversing a poorly structured environment. A further feature of the query system is that it is integrated with a procedural scripting language, where query expressions may be used within the scripting language and vice versa. This not only guarantees that the query system is sufficiently expressive, but also removes the classical impedance mismatch between poorly coupled query and general purpose language systems as encountered in many of the more popular relational databases (eg SQL/PL1, SQL/C,C++).

- The TENTACLE database implementation takes the form of a programmable database kernel which provides a native graph storage system. Programs written in the combined query and scripting language are uploaded into the database kernel, stored as part of the database graph and executed on request. The benefit of such an approach is that it is possible to create an entire, self-contained application within the database — it is possible to do away with the helpers and gateways typically required by conventional databases. Furthermore, the capabilities of the database may be used to manage the construction of the domain specific application itself.

Since these capabilities are not typical of a database, it is not only interesting to build such a system, but also to apply it to a problem domain to discover how the combination of capabilities may be deployed — this provides an indication of the utility and performance of the system.

1.3 Application

The World Wide Web has been chosen as the example application to exercise the TENTACLE implementation. It provides an opportunity to demonstrate the capabilities of the TENTACLE database system since it is a domain which does not satisfy the properties of typical database applications[2]: Information encountered on the World Wide Web is semi-structured and the user is initially unaware of the interrelationship between entities — these have to be discovered, hence the phrase "to navigate the World Wide Web".

The World Wide Web example application takes the form of a program which has been uploaded into and runs within the TENTACLE database. This allows the application to program the database server to provide an HTTP (Hypertext Transfer Protocol [17]) interface to the world. In other words, the database becomes a hypertext or web server which services requests submitted by web browsers.

Because the web server executes inside the database, facilities such as querying capabilities are available immediately, making it possible to offer more advanced services such as materialised document views. Currently these features are not commonly encountered on web servers — most web servers do not provide database capabilities but simply store hypertext entities as files on the server file system. In the cases where hypertext entities are indeed stored in a database, the database is unlikely to make provisions for semi-structured data. Instead the database content is no less regular than that of a conventional database. In such a case the World Wide Web (a networked database in its own right) merely serves as a gateway to another database.

1.4 Outline of the Dissertation

The remainder of the dissertation is structured as follows: The next chapter, Chap. 2, supplies a background, describing a selection of graph-based query languages and database systems, of which a number have been applied to semi-structured applications. The background chapter also introduces the example application domain, the World Wide Web and a number of systems which have been used to query it.

Chapter 3 sets out the TENTACLE data model (an untyped graph-based model) while Chap. 4 describes the integrated query and scripting language and provides a number of short examples of how the language may be used.

Thereafter, in Chap. 5, the implementation is documented. This chapter consists of a system design overview, followed by a description of the im-

[2]See the itemised properties listed earlier in this chapter.

plementation of the database components, from the lowest layers (physical storage organisation) to the higher layers (query and scripting interpreter).

Chapter 6 explains how the system was used to build the example application. It shows how the database system has been programmed to provide a Hypertext Transfer Protocol server and how the built-in query system can be used in this environment.

Chapter 7 concludes the dissertation with a discussion of the results, as well as a description of potential extensions to and further applications of the system.

Chapter 2

Background

2.1 Overview

The TENTACLE project relates to a number of database subtopics. It is an alternative database system and makes use of a graph-based model and query language. These are related to other systems in Sect. 2.2. TENTACLE, like a number of other graph-based systems, is intended to be used in semi or poorly structured application domains. This topic will be introduced in Sect. 2.3. One of the most significant examples of a semi-structured application (and the one chosen as example in this project) is the World Wide Web — it will be described in Sect. 2.4.

2.2 Data Models and Query Languages

As a first approximation data models may be grouped into two categories (See also Fig. 2.1):

Value-based systems where entities are accessed using keys, where a key is a set of distinguishing properties or attributes.

Identity-based systems where entities are referenced by means of pointers or (object) identifiers.

The primary example of a value-based data model is the relational model. It was introduced by Codd [24] and has become the dominant model used in commercial systems (such as Informix [6] or Oracle [9]). Query languages associated with the relational model include SQL, Quel and Query by Example. Most introductory database textbooks include a description of the relational model and associated query languages including [42, 58].

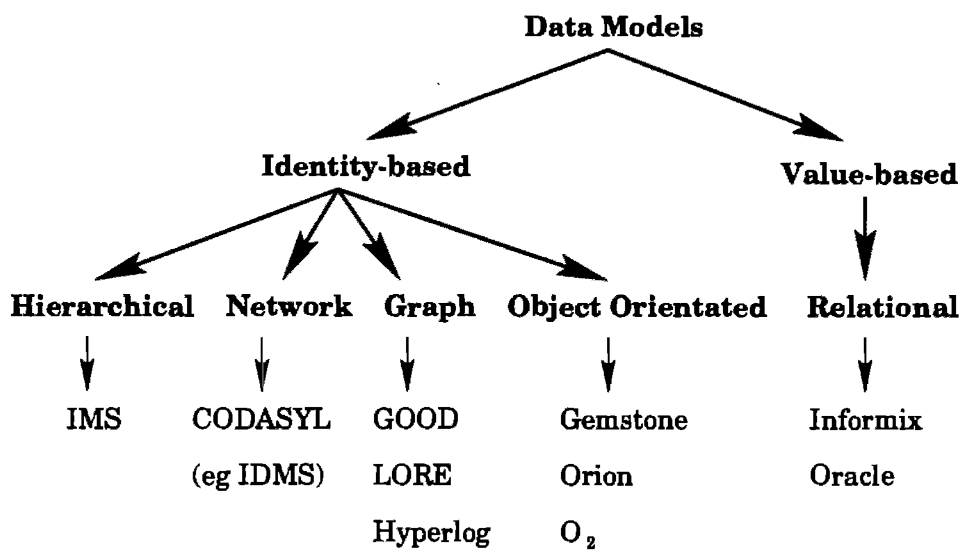

Figure 2.1: Classification of selected data models into identity and value-based systems

Identity-based data models include network, hierarchical, object orientated and graph-based models. Network and hierarchical models are earlier systems which have mostly been displaced by relational systems, while newer databases using the object-oriented model are in ascendancy.

Databases using the network model are typically CODASYL-based systems such as IDMS, while IMS [5] is an example of a hierarchical database. Well-known object-oriented databases include O_2 [27], Orion [41] and Gemstone [45]. There are a number of query languages in use in object-orientated systems, the most prominent one is OQL [22], a derivative of SQL. [29] contains a survey of object-orientated systems, while [44] explains selected object-orientated data models and languages.

Graph-based models are less well known identity-based data models. Essentially graph-based systems model entities as a set of nodes and their interrelationships as edges between nodes. Apart from their more recent use in semi-structured applications, graph-based databases have also been used in GIS (Geographic Information Systems) as well as visual specification and query languages. Examples of graph-based systems include GRAS [39], GraphDB [34], GraphLog [26], GOOD [35] and Hyperlog [53]. Note that some of the systems, while graph-based at a conceptual level, are implemented on top of a non-native storage subsystem (eg: GraphLog uses Datalog, GOOD uses a relational database). The TENTACLE system provides its own graph-based storage subsystem.

GRAS (GRAph Storage) is an operational graph-based database system initially developed for a software engineering application (IPSEN) and has subsequently been used in a number of other structure-oriented environments. The GRAS data model and implementation was designed to support entities which change size and structure dynamically. The GRAS system is implemented as a kernel (a manager of complex data items which may vary in size and structure) surrounded by several layers which provide extended services such as change management of individual items or schema/attribute management of the entire database. The GRAS kernel is also used by PROGRES (PROgrammed Graph REwriting System). PROGRES [55] provides a very high level language based on graph grammars which provides facilities for graph rewriting and transformation.

GraphDB is a data model and query language designed to be used in spatial databases in order to better represent the connectivity between entities (and not merely their spatial geometry). Coupled with specialised graph traversal operations and the ability of GraphDB to store paths in the database explicitly it is possible to formulate, amongst others, reachability queries. GraphDB's querying system provides an SQL-like construct as well as graph rewriting facilities and the capability to represent and manipulate heterogenous collections.

GraphLog is a visual database query language. Queries in the language are formulated as graphs. These so-called query graphs are approximately equivalent to conventional logic rules. In such a query graph a distinguished edge approximates the head of the logic rule (specifies the "output" of the query) whilst the remaining graph components are the equivalent of the rule body, where the rule body is matched against the database graph.

GOOD is a graph-orientated object database model which represents both the database schema as well as data instances as graphs. GOOD was intended to provide a minimal set of graph operations which could serve as the foundation for more complex operations or transformations. GOOD, like GraphLog, is a visual environment.

Hyperlog is a graph-based language. It uses the hypernode data model which is an extension of a conventional graph-based model where nodes may themselves consist of graph structures. In this regard the hypernode model may be thought of as a nested graph-based model. As in GraphLog, Hyperlog queries take the form of rules which are matched against the data graph. The body of a hyperlog rule serves as a set of templates matched against the graph, while the rule head may be used to update the graph. Negated rule heads are interpreted by Hyperlog as deletion requests.

2.3 Semi Structured Data

Traditionally databases have been deployed in applications such as payroll management, analysis of census data, inventory management or flight reservation systems. Such systems are tightly controlled and highly structured environments modeling only a comparatively small part of the world — although the volume of data handled by such systems may be very large, the schema information is usually comparatively simple and amenable to prior analysis. Typically there exist only a small number of types, operators and constraints, and it is feasible to declare these before the database becomes operational and retain them for the life of the database.

It has been long recognised that database systems designed for such traditional applications are difficult to apply to both more complex as well as less structured application domains. When extending databases to non-traditional applications, the emphasis has usually been focused on the former — providing an environment supportive of more complex and sophisticated tasks, with a lesser emphasis on supporting less structured applications.

In particular, the more advanced systems which are reaching commercial maturity, namely object-orientated and object-relational systems, have been developed to provide, amongst other features, larger and more complex typing systems. In the case of the object-orientated system, these take the form of user-defined class hierarchies and member functions, while object-relational systems tend to provide pre-written modules (commercially known as data blades [6] or data cartridges [9]) which provide a domain-specific set of data types and associated operations.

Only recently (and probably as a consequence of the explosive growth of the World Wide Web) has an emphasis been placed on supporting less structured applications. This is the field of semi-structured data (introductions to which can be found in [13, 19]). Semi-structured data (sometimes also described as poorly structured or schemaless data) is data characterised by the absence of an explicit, well-defined schema. Instead it is left to the user to discern schema information from the structure of the data.

This can be thought of as a reversal of the conventional approach of managing information — in a conventional database system a schema is defined beforehand and data inserted into the system has to conform to the schema or be discarded, whilst in a semi-structured system the schema information is derived or deduced from the data as it is added to the system.

This alternative approach results in schema information of a different quality. A classical predefined database schema is designed to be regular (to facilitate database manipulation) and tends to be small (in order to make prior analysis tractable), as well as complete and accurate (in the sense that all data entities are fully specified and non-conforming entities rejected). The schema information contained in a semi-structured system is weaker — it tends to be part of the structure of data and may be difficult to discern from instance information. It may only provide a partial indication of the database structure, serving more as a guide or set of hints to the user.

Clearly the classical schema, when available, provides significant assistance when querying and manipulating data. However, there exist situations when such a schema is unavailable or of reduced effectiveness.

For example, genuinely unstructured or irregular data may only derive minimal benefit from a conventional schema — the schema may be expensive to construct and maintain, and be itself irregular and complex, effectively treating each data instance as a special case.

Another example would be an application domain where very little is known about the data before it is inserted into the database. Such systems include databases which hold research results (for example AceDB [28] which stores information related to molecular biology). In such a situation a conventional schema might have to redesigned with each new data entry.

A further example would be a heterogenous, decentralised environment where it is not possible to impose a global schema, or where the schema is unreliable. The World Wide Web exhibits these characteristics — no central authority can impose a schema, and schema information, to the extent present, has been known to be falsified[1].

There even exist a few cases where a semi-structured system may be useful even though a well-formulated schema for the given domain is available. These include situations where a casual user wishes to browse the database content without having to be aware of or learn the underlying schema. It may also be useful to employ a system supporting semi-structured data when integrating or interchanging data of environments employing divergent data models.

Systems designed for semi-structured environments include LORE [48] and UnQL [20].

[1]For example, some authors include commonly searched-for phrases in the keyword fields of hypertext documents in order to achieve a greater exposure in the listing of index servers, see [12].

LORE (Lightweight Object REpository) was initially intended to function as a private workspace or intermediate store for the mediators of the TSIMMIS (The Stanford/IBM Manager of Multiple Information Sources) project, but has been subsequently extended to function as a database in its own right.

LORE makes use of OEM (the Object Exchange Model) to represent information. Each object within this model consists of an identifier, a label and a value. The value can either be a simple entity or a set of references to further objects. This model may be thought of as representing data as a node-labelled graph.

This graph structure can then be queried using the LORE query language, LOREL [14, 54]. The language attempts to deal with irregular structures by performing type coercion, permitting wild-cards in queries and not differentiating between tests for equality against a single value and tests for existence within a set.

UnQL (UNstructured Query Language) is a query language and associated calculus (UnCal) for semi-structured data. UnQL models data as an edge-labelled tree or graph, storing information only at edge labels — unlike OEM, the UnQL model does not associate an identity with a node. UnQL attempts to augment the conventional relational operations which tend to operate on flat structures with operations which are capable of manipulating deep or cyclic structures.

2.4 World Wide Web

The World Wide Web is the largest networked hypertext system. It was introduced in 1990 and has, at the time of writing grown to over 3 million participating hosts or web servers [8].

The World Wide Web is a client/server system where clients (known as user agents) contact the servers (referred to as web servers) to access named hypertext entities. Entities are identified by their URL (Uniform Resource Locator [18]) and are related to each other via references called hyperlinks.

Hyperlinks are tags embedded in hypertext documents which refer to other hypertext entities. Conceptually hyperlinks are the extended electronic successors of footnotes or bibliographic references as encountered in printed texts.

The hypertext documents (also known as web pages) of the World Wide Web are usually written in HTML [16], the Hypertext Markup Language, an application of SGML (Standard Generalized Markup Language). In addition to providing a means of inserting hyperlinks, HTML also provides more conventional markup tags to declare elements such as headings, tables or quoted texts. Apart from HTML documents, the World Wide Web also makes provisions for a large number of other data entities making it possible to refer to items such as audio or video clips, images, executables or compressed archives from hypertext documents.

Web server and user agent interact via HTTP (the Hypertext Transfer Protocol [17]). HTTP was intended to be a high-level, simple and stateless protocol. It specifies a text-based request/response dialog between client and server (initiated by the client) where the client requests an operation (such as the retrieval) on a particular hypertext entity where after the server returns a response (in the case of a retrieval request this might contain the requested entity). HTTP was designed to be used atop any conventional reliable connection-based transport protocol, but is currently deployed almost exclusively atop TCP/IP (the reliable connection-based transport protocol of the Internet).

Conventional web servers tend to store hypertext entities either on the local file system or generate hypertext entities by invoking CGI [3] programs (programs conforming to the Common Gateway Interface). Since web servers usually do not provide their own database or querying facilities, CGI scripts or programs are also used when database capabilities are required. This process involves starting a CGI program to query a third-party database server, adding hypertext markup to the query results and returning the output to the web server. Attempts have been made to reduce the costs of invoking a gateway program for each client request by optimizing the interface (eg FastCGI [4]) or by moving the gateway program into the web server. The latter is usually achieved by including a scripting interpreter in the hypertext server (such as `mod_perl` or `mod_php` in the case of the Apache [2] web server).

The World Wide Web is an interesting system because it lacks a controlling entity which can impose and enforce a schema or structure. As a consequence the associations between hypertext entities are unconstrained. For example, in a classical database modeling a part of a university, it might be possible to enforce the constraint that members of a department (listed in the departmental members relation) have to be employees of the university (listed in the employees relation). Such constraints are unlikely to be enforced on the World Wide Web — while one departmental web page may indeed contain hyperlinks to all its employed members, another department may only list the secretary as contact person, a third department may link to its research groups instead, while a fourth might list staff, students and the departmental cat.

The absence of a global schema makes it difficult to use traditional query systems to extract information from the World Wide Web. Instead a number of alternative approaches are in use. Currently the most popular approach to query the web is to use index servers (also known as search engines). Index servers occur in two variants, those where potentially interesting documents are selected and categorised by humans (such index servers include Yahoo [11]) and those which are generated automatically and allow the user to search for matching string expressions or phrases present in the indexed documents (an example of such an index server is Altavista [1]).

In addition there exist a number of research projects which attempt to provide query and or database management facilities which are suited to the semi-structured domain of the World Wide Web. Not all of these projects approach this task from the semi-structured data perspective — for example, some attempt to transform selections of the web into more regular structures whilst others develop specialised hypertext models or extensions to existing models. A small sample of these different approaches is briefly described below:

ARANEUS [15] attempts to query the World Wide Web by formulating a schema for an existing set of hypertext documents, extracting information from these documents (using the EDITOR language to parse the documents) and inserting this information into a conventional relational system (using the ULIXES language). Once the information has been inserted, the facilities of the relational database can then be used to create different views of the information in the form of new hypertext documents (this phase is specified using the PENELOPE language). Essentially the ARANEUS project translates semi-structured information sources into an intermediate, highly structured form which may then in turn be used to construct semi-structured views of the information source.

RAW [31] (Relational Algebra for the Web) augments the classical relational algebra with operators and types (domains) designed to make it possible to apply the algebra to the World Wide Web. In particular RAW introduces types to access URLs, sequences of URLs (paths) and fragments of hypertext documents. RAW also adds operators (SCAN and INDEXSCAN) to retrieve documents from the web and insert these into a suitable tuple structure which may then be accessed by other relational operators. In other words RAW is a domain specific extension to the relational algebra which makes it possible to traverse the World Wide Web using relational operators.

WebSQL [49] is a SQL derivative designed to query the web. The system models the web as a relation of hypertext documents and a relation of hyperlinks, both computed on demand. These form the basis for the *virtual graph* which is used by the query system. WebSQL augments SQL with constructs to perform string searching (`MENTIONS` and `CONTAINS`) as well as facilities to formulate path-based queries using regular expressions. WebSQL is able to distinguish between hyperlinks to the current document, to documents on the same host and hyperlinks to a different, remote host. This capability enables the system to calculate the cost of a query and may be used to optimise it.

WebLog [43] is a logic and query language for the World Wide Web. WebLog introduces the *rel-infon*, a fragment of an HTML document delimited by a user-selected HTML tag (an example would be paragraphs if the user selects the paragraph delimiting HTML tag <p>). Hyperlinks, rel-infons and entire HTML pages may be used in query expressions which resemble DATALOG rules and these queries may be used to generate restructured or derived web pages. WebLog provides a number of domain specific builtin predicates for matching string subexpressions and accessing web pages. The set of builtin predicates may be extended by making the functionality of external programs available as a new builtin.

Hyperwave [47] (previously HyperG) provides a data model developed specifically for hypermedia systems. It defines a graph by means of S-collections. An S-collection can either be an atomic node or be a structure consisting of a number of S-collections and associated directed edges. S-collections bear some resemblance to hypernodes as encountered in systems such as Hyperlog [53]. An interesting aspect of the model is that it attempts to impose a typing structure on graphs by categorising the S-collections into specific types (lists, trees as well as a catch-all) in an attempt to model common hypertext structures. For example a sequence of hypertext pages constituting the chapters of a book might be represented as the list S-collection type.

STRUDEL [30] is web-site management and query system. It uses a graph-based data model similar to OEM (the Object Exchange Model of the LORE system), and like LORE, STRUDEL is capable of integrating data from a number of sources via wrappers and mediators. STRUDEL also provides its own native data graph repository. The query language associated with STUDEL is STRUQL (Site Transformation Und Query Language). STRUQL is used both in the definition of an integrated view of several information sources, as well as in the querying of the unified data graph, where it may be used to define site graphs (analogs to database views) which are used by an HTML generator to create a web site. As the acronym indicates, STRUQL provides constructs to generate and restructure graphs. Furthermore STRUQL allows the user to formulate powerful path expressions which may include builtin as well as user-defined predicates.

Of the systems listed above, STRUDEL bears the closest resemblance to TENTACLE. Both STRUDEL and TENTACLE use a graph-based data model and provide sophisticated path expressions to traverse the database graph. Both systems have been applied to the domain of the world-wide web.

However there also exist a number of differences between the two systems: Where STRUDEL has been designed specifically for the management of web sites, TENTACLE is intended for use in semi-structured applications in general. STRUQL as well as LORE and OQL use path expressions as an adjunct to more conventional query clauses which bear some resemblance to the `SELECT ...FROM ...WHERE ...` of SQL, while TENTACLE attempts to investigate the feasibility of using path expressions as the only query construct. TENTACLE allows the user to embed output formatting information in a query expression, while STRUDEL, like ARANEUS, appears to use a separate HTML generator module to markup the query output.

Chapter 3

Data Model

3.1 Overview

The TENTACLE database system uses a weakly typed graph-based model to represent information. The following features of a graph-based model may make it attractive for use in both poorly structured as well as complex applications:

1. It is possible to traverse the database structure without having to refer to a schema. The user merely follows links from known entities to unknown ones. This process is relatively simple and inexpensive — no join operation is required.

2. It is relatively easy to represent associations between entities. A graph-based model allows the user to relate entities to each other by simply creating a link between two nodes. The addition is reasonably inexpensive.

3. A graph-based model is capable of modeling complex structures directly. Such structures may be arbitrarily deep or cyclic. Like object-orientated models, graph-based models tend to provide a means to distinguish between references to the same entity (node/object identity) or references to entities merely possessing the same attributes.

That such a model can be useful is supported by the fact that the World Wide Web may be viewed as a graph or network based database system. Its phenomenal growth and popularity can partly be credited to the ease with which new information can be added to the system and related to other, already existent information. In this regard it differs fundamentally from relational or even most object-orientated systems which might require costly schema modifications.

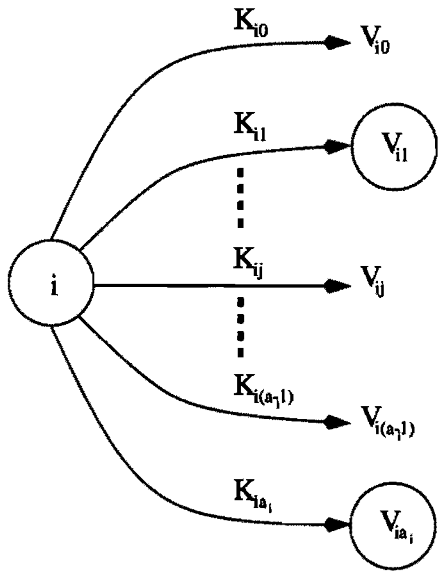

Figure 3.1: Graphical representation of node N_i with a_i attributes where each attribute consists of a key K_{ij} and value V_{ij}. Note that values V_{i1} and V_{ia_i} are references to other nodes.

3.2 Model Definition

The graph-based model used by the TENTACLE system represents the database as a set of objects or nodes, where each node possesses a unique identifier as well as zero or more attributes. An attribute takes the form of a key/value pair. The key may be used in lookup operations within the scope of the node to return the attribute value component. The value is either a reference to a node or an atomic entity.

Viewed as a graph, the key of a node attribute may be thought of as being a directed labelled edge originating at that node, while the attribute value may be thought of as being the target node. Where the attribute value is an atomic entity, the target node may be thought of as a special case possessing no attributes of its own (meaning that it has to be a leaf node) and having an identifier which corresponds to the atomic attribute value.

More formally: The TENTACLE data model represents information as a set of nodes $\{N_0, ..., N_n\}$ and a set of atomic entities $\{M_0, ..., M_m\}$ where each node N_i consists of a unique identifier i and a set of attributes $\{A_{i0}, ..., A_{ia_i}\}$. Each attribute $A_{ij}, j \in [0, a_i]$ consists of a key/value pair (K_{ij}, V_{ij}) where the key K_{ij} is an atomic entity $M_p, p \in [0, m]$, while the value is either an atomic entity $M_r, r \in [0, m]$ or a reference to a node $N_s, s \in [0, n]$.

Itemised definition:

D database,	$D = N \cup M,\ N \cap M = \emptyset$
N node set,	$N = \{N_0, ..., N_n\}$
M atomic value set,	$M = \{M_0, ..., M_m\}$
N_i node with a_i attributes,	$N_i \in N, N_i = (i, \{A_{i0}, ..., A_{ia_i}\})$
A_{ij} attribute of node N_i,	$A_{ij} = (K_{ij}, V_{ij})$
K_{ij} key of attribute A_{ij},	$K_{ij} \in M$
V_{ij} value of attribute A_{ij},	$V_{ij} \in N \cup M$

Note the deliberate distinction between the node set N and atomic values M. Elements in the set M are assigned by and of meaning to the user, while the identifiers of the node set N are opaque, immutable surrogates meeting the requirements for strong identity as set out by [38].

The TENTACLE data model is similar to those used by other systems designed to deal with semi-structured data. For example it differs only slightly from OEM (the Object Exchange Model of LORE [48]) in that that OEM uses a different object or node representation — an OEM node consists of an identifier, a single label and a set of references to other nodes whilst TENTACLE models a node as an identifier and a set of attributes (each attribute consisting of a label and a reference). In other words OEM labels nodes, while the TENTACLE data model labels edges.

To explain the TENTACLE data model in more familiar terms, one can use a file system analogy: A node in the graph-based model can be thought of as a directory in the file system, where attributes (key/value pairs) are directory entries. The key component corresponds to the name of the directory entry, while the value is either the content of a file or another directory. However, unlike a file system, the graph based structure has no intrinsic notion of a parent directory — after all, the data is not modeled as a hierarchy, but as a graph, thus a node can be referenced by zero or more other nodes (using the directory analogy this means zero or more "parents").

It should be noted that a graph is a generalization of a hierarchy or tree (a file system has a hierarchical structure), since any tree is a special case of a graph which has been restricted to a non-cyclic structure where a single node (the root) has no parent node and all others have exactly one parent. For example if one were emulate a file system structure using the TENTACLE data model, one could use distinguished keys for the purpose of denoting references to the current node and its parent (the key . and .. would seem appropriate) and enforce an acyclicity constraint.

This fact that trees are special cases of graphs should make it possible for the TENTACLE database to interact with or emulate the functionality of information repositories which use a hierarchy as their data model (such systems would include some text retrieval systems, networked file systems or directory servers such as LDAP [60]).

3.3 Summary

The TENTACLE system uses a simple, untyped, graph-based data model. Such a model is capable of representing complex associations between entities directly and allows the user to explore (or navigate) the data without having to refer to a schema. Similar models have been used in other systems intended to query and manipulate semi-structured information.

Chapter 4

Language

The TENTACLE system provides an integrated query and scripting language. The query component is based on path expressions over the database graph, while the scripting component resembles conventional general purpose programming languages. The former will be described in the next section (Sect. 4.1), whereafter a brief overview of the scripting language will be given (Sect. 4.2). That section will be followed by an explanation of the integration of the two components (Sect. 4.3) and a section (Sect. 4.4) of example queries phrased in the combined language.

4.1 Query Component

Query languages are an essential feature of any database; without a facility for formulating queries, a database is likely to become a write-only storage system. Relational databases introduced a number of high-level query languages including QBE, QUEL and, the best known, SQL. These declarative languages make it significantly easier to access the database to the extent that even people with limited programming skills are able to query a database system.

This success of systems which offer a declarative and easy to use query interface suggests that these aspects should also be made part of the requirements of the TENTACLE query language.

In addition it is desirable to make provisions in the TENTACLE language for querying semi-structured or schema-less data, since cases may arise where the structure of the stored data may not (yet) be known, or where a casual user may simply be unaware of the schema. In such cases the query language should assist in the browsing the data and possibly even assist in the discovery of its structure.

Several attempts have been made to modify SQL to be used in non-relational and semi-structured applications (for example OQL [22] appears to become the most popular query interface to object-orientated databases, while LOREL [14] employs an SQL-like syntax to query the semi-structured LORE system).

However, using an already familiar syntax with different semantics can cause confusion, and since the TENTACLE database system is a deliberate attempt to explore alternative database designs, it was decided to follow a different approach. In particular, the system which serves as a point of departure for the design of the TENTACLE query language is that of regular expressions. Regular expressions occur in a number of user applications such as shell interpreters and advanced editors and should be familiar to non-programmers, thus presumably meeting the requirement of being reasonably easy to use.

Conventional regular expressions are template strings which are matched against a stream of characters. The TENTACLE query language applies a similar principle but matches sequences of nodes and edges instead of sequences of individual characters. To avoid confusion with the usual regular expressions, these expressions have been termed *path expressions.*

Put simply: A regular expression matches a character string, while a path expression matches a path in the database, where a database path is a sequence of nodes and edges which allows the user to move from an initial node to another entity in the database graph. In this respect TENTACLE paths are not dissimilar to the paths encountered in object hierarchies or file systems. Examples of these include the path expression:

```
ship.hold[2].container[4].owner
```

which allows the user of an object database to locate the owner of the fourth container stored in the second hold of a given freighter, while a file system path of the form:

```
/usr/bin/vim
```

allows the user to descend from the root directory / to the file `vim`.

Path expressions seem to be useful in semi-structured problem domains since they permit the user to start at a known point and then gradually explore adjacent entities.

4.1.1 Terminology

In order to explain the TENTACLE query language it is useful to introduce two terms which can be used to describe the components of path expressions. Consider the object path expression

```
ship.hold[2].container[4].owner
```

This expression is specified as a concatenation of delimited entities. For this dissertation such entities shall be referred to as *segments* while their delimiters shall be referred to as *separators*. Thus the first segment (left to right ordering can be assumed) of the example path expression would be **ship** and the second segment would be `hold[2]`[1]. The separator in this example is the period (.) — other environments may make use of different separators (for example file systems tend to use / as separator).

Observe that the first or *initial* segment specifies the point at which the path starts, while subsequent segments are used to constrain the possible paths emanating from this point of entry. In the above example, the first segment **ship** might be a variable containing a reference to a freighter object, while the next segment `hold[2]` indicates that only the second member of the hold attribute needs to be considered when following this path. In other words, the initial segment specifies the starting or *input* entity of the traversal. This traversal ends at the final or *result* entity. In the above case the result entity is a reference to the owner (of the fourth container).

Note that the terminology has been introduced using an example from an object-orientated programming language or database. However, the terms can easily be extended to TENTACLE path expressions. Where segments of an object path are matched against objects and their members, TENTACLE segments are matched against nodes and their attributes. Similarly the input and result entities are extended to refer to sets of graph components (nodes, attribute keys and attribute values, see Chap. 3 for their definition).

4.1.2 Operators

So far a separator has been presented as a means to delimit the segments of a path. However, a separator can also be thought of as a binary infix operator, in the case of the above example the separator . is associated with the concatenation operation.

The TENTACLE query language extends this notion to provide its core functionality. In particular it introduces three additional separators which

[1]Note that an individual segment may have a composite structure.

provide a means of specifying alternation, conjunction and closure. These may then be used to construct more complex path expressions to be matched against database paths. The three separators are | to denote alternation, & to denote conjunction (requiring a match for both alternatives in a branch of the graph, where alternation requires only a single one) and * for closure[2].

To relate these operators back to conventional regular expressions across sequences of characters: In regular expressions the concatenation operator is left implicit (assumed between all plain characters) while both alternation and closure are specified explicitly. An operator to denote conjunction is usually not required since strings are linear structures (where a particular position in the string is uniquely determined) unlike the branching structures of graphs (where a particular branch might have to match one subexpression, *and* another branch might have to match another).

4.1.3 Syntax

The grammar defining the syntax of TENTACLE path expressions is reasonably compact and given below:

```
PATH → SEGMENT
     → SEGMENT '.'  PATH
     → '[' PATH ']' '*'
     → '[' PATH ']'
     → PATH '&' PATH
     → PATH '|' PATH
```

LEGEND: Uppercase strings denote nonterminals, singly quoted characters denote terminals.

The structure of a segment will be explained later and the complete grammar of the combined query and scripting language is given in Appendix A.

Since TENTACLE path expression make use of several separators, it becomes necessary to define their precedence: Alternation has the lowest precedence followed by conjunction followed by concatenation. Square brackets are used to override default precedence and thus have the highest precedence. The closure operator incorporates square brackets and has thus an equally high precedence.

[2]The closure operator is an unary postfix operator.

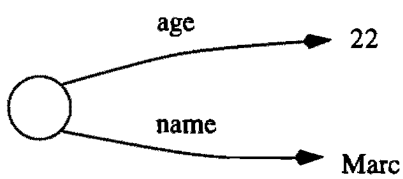

Figure 4.1: Simple example database graph

4.1.4 Introductory Examples

This section provides a number of simple TENTACLE path expressions. More complex examples will be given in Sect. 4.3 and Sect. 4.4, while a detailed semantics of the path expressions can be found in Appendix B.

Consider the TENTACLE database graph given in Fig 4.1 consisting of a single node possessing two attributes, with keys `age` and `name` respectively.

Given a reference to the node in the variable `me` it is possible to retrieve the value of the age attribute using the following path expression:

```
me.age.
```

Here the first segment, `me`, specifies the point of entry into the system (ie the *input set*) while `age` selects the age attribute. The expression contains a third segment, the empty or *null* segment (hence the second `.` separator). When occurring as any other than the initial segment, the null segment matches any graph component — it is the equivalent of a wildcard or don't care match. In the example the null segment matches the attribute value 22. This is also where the path expression finishes, returning the attribute value as its result set.

If the null segment occurs in place of a conventional initial segment, then the default input set is used. It contains a reference to a single node. This is node defined as the global database entry point and is the persistent root of the database. If the node given in Fig. 4.1 is designated the database root, then the above path expression may be replaced by a shorter equivalent:

```
.age.
```

An example of a path expression which makes use of the alternation construct is given below. The expression requests either the age or birthdate value:

```
me.age.|me.birthdate.
```

Again, the result set contains a single reference to the 22 of the age attribute (the example node does not possess a birthdate attribute). By making use of the square brackets to override default precedence, it is possible to rewrite the above expression as:

```
me.[age|birthdate].
```

The final introductory example expression illustrates the use of the closure operator:

```
me.[]*
```

The result set of this expression contains all the graph components reachable from the graph component referenced by the variable `me`[3]. Here this set contains four members — references to the keys and values of the node attributes (key `name` and value `Marc` for the first attribute and `age` and 22 for the second).

4.2 Scripting Language

The TENTACLE language includes a general purpose programming (here referred to as *scripting*) component.

It is somewhat uncommon to encounter scripting languages in database systems. Usually the reason advanced for their omission is that general purpose programs may consume an unpredictable amount of resources (time, storage). This problem is solved somewhat crudely in the TENTACLE system by setting resource limits which, if exceeded, result in the termination of the script.

The inclusion of a scripting component guarantees the computational completeness of the TENTACLE language, thus making it unnecessary to use host or wrapper languages (as is the case in a number of other query languages such as SQL). The elimination of host languages removes the impedance mismatch usually encountered in two language systems.

The scripting component is also used as data definition language, new data items may be added to the system by calling a function to link the new item to a node reachable from the database root (in other words data persistence is by reachability).

In addition the scripting language component makes it easier to add active-database capabilities to the system such as dynamically computed

[3] Although not illustrated in this example, the TENTACLE closure operator is capable of dealing with graph cycles.

data or triggers. The TENTACLE system takes advantage of this by providing a trigger which is executed as soon as a client connection to the database is established. Such a facility makes it possible to program the interface presented by the database to the client, in other words, it is possible to adapt or interface the database to its problem domain or application without having to make use of external gateway or mediator programs.

Like the path expressions, the scripting component of the TENTACLE language has been kept simple deliberately — had either component been overly complex their combination would, in all likelihood, have become unreadable and their interaction unmanageable. Hence the scripting component resembles a small, simple subset of a procedural language such as C [37] or PHP [10].

Most complex syntactic constructs have been omitted — equivalent functionality is provided by a set of builtin functions. For example where *C* would use a construct such as `X&&Y` to denote conjunction, the TENTACLE scripting language uses `and(X,Y)`[4]. This reduces the tokens reserved by the scripting component of the language to the following:

`if else`	conditional evaluation
`while`	iteration
`var`	variable declaration
`{}`	block delimiters
`()`	parameterisation of functions, loops and conditions
`,`	parameter separator
`[]`	path expression delimiters
`""` and `''`	quoting of literals
`.`	program terminator

The complete grammar of the language is given in Appendix A.

The canonical "Hello World" program expressed in the TENTACLE scripting component looks like this:

```
write(connection,"Hello World").
```

In this example the function `write()` appends the value of its second argument (The quoted literal `"Hello World"`) to its first argument (the network connection handle referenced by the variable `connection`).

Another example shows how the scripting language may be used to insert data into the system. The script given below will generate the graph in Fig. 4.1 and make the node the root of the database graph:

[4]Note that this changes the semantics of `and()` from those of *C* to those of *PASCAL*.

```
var me
me=newid()
link(me,"age","22")
link(me,"name","Marc")
root(me).
```

The first line declares the variable `me`, the second line requests the database to allocate a new node and assigns a reference to this node to the variable `me`. The two calls to the `link()` function associate two attributes with the node, while the last line informs the system that the newly allocated node should become the new entry point into the system.

Further examples of the scripting language will be given in the next section (Sect. 4.3) and Chap. 6.

4.3 Integration

The integration of the query and scripting components of the TENTACLE language is achieved by allowing path expressions to appear in place of R-values in the scripting component (here termed *R-value substitution*) and by permitting R-values of the scripting language to appear as segments in the path expressions of the query component (referred to as *segment substitution*). In other words parts of the scripting component may occur in path expressions and vice versa. This mutual nesting may be arbitrarily deep.

4.3.1 R-value Substitution

Path expressions, when enclosed in brackets `[]`, may appear in place of the more usual R-values (literals, variables and function calls) of the scripting component. When a path expression occurs as an R-value, its value is the result set of the expression[5]. Consider the path expression `me.age.` matched against the database graph depicted in Fig. 4.1. The result set of this expression is a reference to the attribute value 22. Thus when the expression is used as the third R-value in the script:

```
write(connection,"My age is ",[me.age.]).
```

the output will be the text `My age is` 22.

[5]Where the caller expects only a single value, the first element of the set is used.

4.3.2 Segment Substitution

Any R-value (ie literal, variable or function call) of the scripting component may be used as a segment in a path expression. For example in the first path expression `me.age.`, it is given that the initial segment `me` is a variable which references the node in Fig. 4.1, while `age` is a literal matching itself.

The example may be modified by replacing the age literal with a variable holding the value `age`:

```
var property
property=age
write(connection,"My ",property," is ",[me.property.]).
```

The output of this script is identical to that of the previous example, namely `My age is 22`. Note that, like in all the previous examples, the variable `me` is assumed to have been declared and initialised previously. Also note that the quoting of literals containing only plain characters is optional, there is no distinction between `property=age` and `property="age"` provided `age` has not been previously declared as a variable (in which case `property=age` would assign the dereferenced value of the variable `age` to the variable `property`). Defensive coding practice suggests enclosing all literals in quotes, including those occurring in path expressions (for example `me."age"`). However, for the short examples given in this dissertation, this appears unnecessary and only reduces readability.

Function calls may be used in a similar manner to variables. When a function call occurs as initial segment, its return value is used as input set for the path expression. When used in another position a match succeeds if the function call returns true (ie non-null — the TENTACLE language provides a distinct null value). Thus the path `.age.` can be thought of as being a shortened version of `root().age.true()`, since `root()` returns a reference to the entry point of the graph, while `true()` always succeeds.

Another example of a path expression using a function call is:

```
[me.age.write(connection,"My age is ",here())].
```

Again the output of this expression is `My age is 22`. The nested function call `write(connection,"My age is ",here())` has been included as the third segment in the path expression and the side effect of its evaluation results in output. Note the special function `here()` returns the value of the current matching component of the database graph (22 in the example)[6].

[6]The `here()` function bears a limited similarity to the `this` pointer as encountered in C++.

The last example of path expression and scripting integration illustrates how functions and path expressions may be nested several layers deep:

```
[me.write(connection,"My ",here()," is ",[here().],".")].
```

The resultant output is `My age is 22. My name is Marc..` The top-level path expression consists of two segments, where the second segment is the nested function call `write(connection,"My ",here()," is ",[here().])`. This function call is evaluated for each attribute key of the node (`name` and `age`) and accesses the current graph component by using the `here()` function as well as the immediate neighbour using the path expression `[here().]` which uses the current position as input set.

The above examples of formulating similar queries in a number of different ways demonstrate the flexibility of the integrated querying and scripting language, even though the language consists of a comparatively small number of building blocks. By providing a flexible querying system, it is hoped that users will be able to construct queries in formats which are convenient and which map naturally to a given application domain.

4.4 Further Example Queries

This section presents three more complex queries. They are derivatives of queries presented in the literature, and show how one might use the TENTACLE query language to approach some of the issues identified by the authors.

4.4.1 The Movie Database

> `Query`: Find the scriptwriters of movies directed and produced by the same person.

This query has been adapted from an application domain introduced by [19] (the Internet Movie Database [7], where users may browse a large body of movie-related information). The query is an attempt to compare TENTACLE path expressions and path expressions which might form part of more conventional `SELECT ...WHERE ...` clauses.

Given a database graph such as depicted in Fig. 4.2 which represents selected details of two movies, the path expression which would return the set of scriptwriter names is:

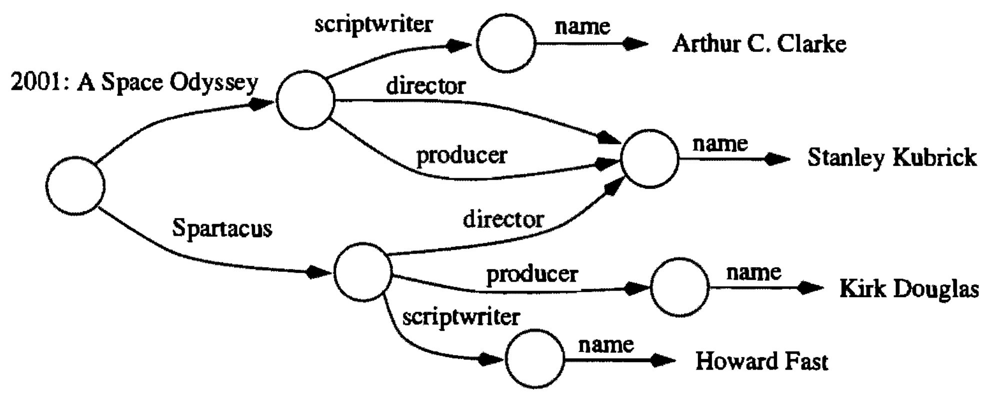

Figure 4.2: The Movie Database

```
[..equal([here().director.],[here().producer.])
   .scriptwriter..name.]
```

This path expression (which has been split over two lines for the sake of readability) consists of seven segments, where the third segment is a complex subexpression. The subexpression evaluates to true if the result sets of the two path expressions `[here().director.]` and `[here().producer.]` are equal. Since evaluation of subsequent segments only proceeds if previous segments have been matched, the third segment serves to eliminate movie nodes which do not possess equal director and producer attributes. The remaining four segments (`scriptwriter..name.`) traverse the graph from a movie node to the name attribute of the scriptwriter.

An advantage of TENTACLE path expression syntax over more conventional `SELECT ...WHERE ...` clauses is that the relationship between the result (the `SELECT` clause) and the constraint (the `WHERE` clause) can be given directly. Consider a naive attempt at splitting the above path expression into a `SELECT` and a `WHERE` clause[7]:

```
SELECT [...scriptwriter..name.]
WHERE [...director.]  = [...producer.]
```

Clearly the clauses as given above are insufficient since they do not specify how the three path expressions are related to each other, ie how much of the path expressions should be the same — if left unspecified the paths `[...director.]` and `[...producer.]` might refer to an entirely different movies. This difficulty has been identified by [19]. The author suggests using

[7]This is a hypothetical example. The TENTACLE language does not provide a `SELECT ...WHERE ...` construct.

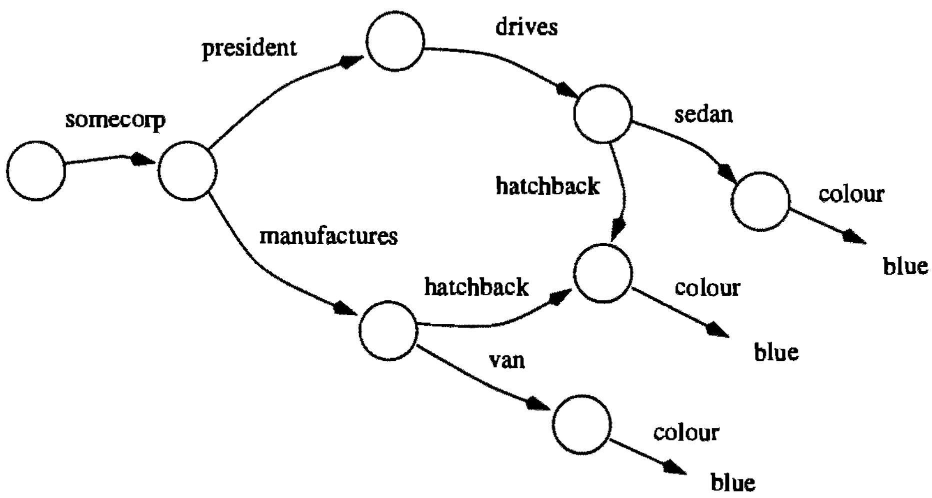

Figure 4.3: The Blue Cars

variables to indicate how the different path expressions relate to each other. Such a query clause could possibly be written as:

```
SELECT [..$movie.scriptwriter..name.]
WHERE [..$movie.director.]  = [..$movie.producer.]
```

This is the approach which has been chosen for query languages such as OQL. The TENTACLE approach of combining all clauses into a single path expression appears to be more compact, since common segments of the path need only be given once, and variables are not required.

4.4.2 The Blue Cars

> `Query`: Find all blue vehicles driven by the president of the company that manufactured them.

This example has been taken from [40]. The query given has, what the author calls, a type-n cycle in its query graph. Such a query is difficult to express in SQL.

A fragment of a database graph which matches the query is given in Fig. 4.3. In the figure a company[8] is represented as a node (anchored at the database root) which contains a president attribute (a reference to the president node) and a manufactures attribute. The president node references

[8]For the sake of brevity only a single company `somecorp` has been shown where otherwise several would have been given.

the set of vehicles driven by him, while the manufactures node references the set of vehicles manufactured.

The TENTACLE path expression which retrieves the vehicles meeting the requested criteria is given below:

```
[...[president..drives...&
     manufactures...equal([here().colour.],blue)]]
```

The expression consist of four segments, where the first three are null segments which traverse the graph from an entry point to a company node. The fourth segment consists of a conjunction subexpression, where the result set of the conjunction is the intersection of the result sets of its two components. The first component returns a result set of vehicles which are driven by the president, while the second returns a set of vehicles which are manufactured by the company and have the colour blue.

4.4.3 The Restaurant Guide

`Query`: Find cheap restaurants.

This example has been derived from [14]. The query is intended to illustrate how information may be extracted from a semi-structured database.

In the example this database takes the form of a restaurant guide. The entries of this guide do not conform to a regular structure. Some restaurants may be described by a brief text, other entries may contain a listing of courses, yet other restaurants might be described by key fields which classify the restaurant according to criteria such as price or cuisine.

Figure 4.4 provides a small part of this hypothetical guide. The figure shows three restaurants, `Amigos`, `Melissa's` and `The Squirrel`. `Amigos` is only described by a short text, `The Squirrel` contains greater detail about individual courses (`The Squirrel` provides a wide selection of cheap starter courses of a reasonable quality[9]), while `Melissa's` is described by attributes indicating opening times, cuisine and price.

Selecting cheap restaurants from this guide is achieved by searching (recursively) for attributes of restaurants which contain the substring `cheap`.

It should be clear that this query might not retrieve all cheap restaurants (some cheap restaurants might be described as having a `reasonable` price). The query might also retrieve expensive restaurants (for example, those which contain the substring `not cheap`). This illustrates the tradeoff made by a semi-structured system — the benefit of being able to store complex and

[9]Descriptions of the other courses have been omitted to simplify the figure.

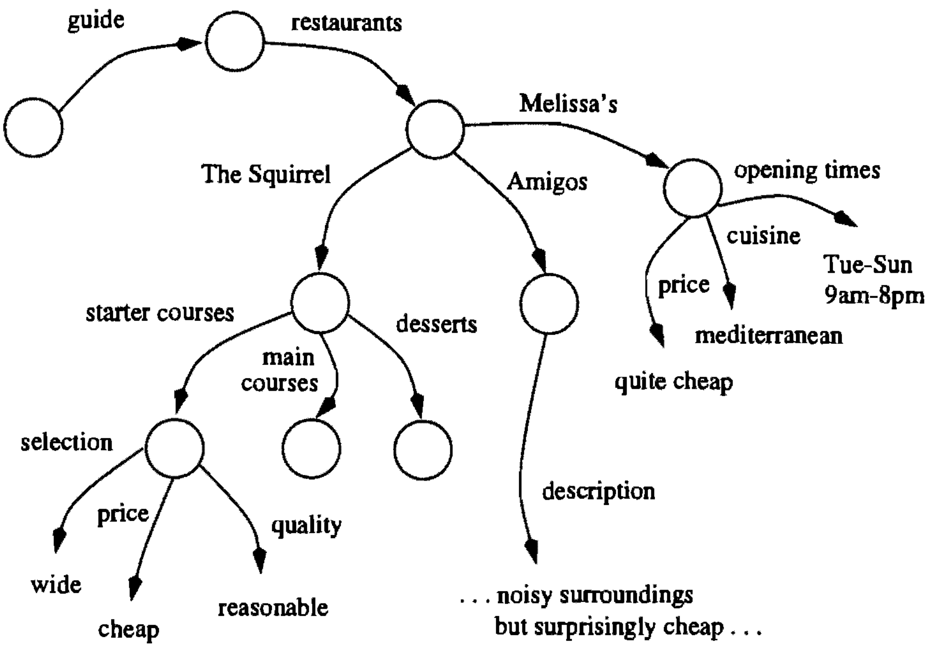

Figure 4.4: The Restaurant Guide

unconstrained data without having to formulate an exact schema is paid for by either reduced query accuracy or increased query complexity. In many cases this tradeoff is acceptable — the casual user browsing the guide in search of a cheap meal is probably prepared to accept an incomplete list of cheap restaurants, and is likely to examine individual restaurant descriptions before visiting the restaurant.

The query to select cheap restaurants can be written as the TENTACLE path expression:

```
[.guide..restaurants..setsubstring("cheap",[here().[]*])]
```

The segments `[.guide..restaurants.]` traverse the graph from its root to the node which contains a set of references to restaurant nodes. The path expression `[.guide..restaurants..]` would select the keys of all of these references, ie *all* restaurant references. In order to constrain the result set, the function call `setsubstring("cheap",[here().[]*])` is used in place of the last empty segment.

`setsubstring()` returns true if the first argument is a substring of one of the elements of its second argument. In the example, the second argument is a path expression which returns the set of all graph components reachable from the current position (the current position being the key of a reference to a particular restaurant node). Thus `setsubstring("cheap",[here().[]*])`

only succeeds if a graph component which contains the substring cheap is reachable from the current position.

Chapter 5

Implementation

5.1 System Overview

The core of the TENTACLE database system is a single server process which fields requests from multiple clients via a network interface. The server takes the form of a programmable database kernel which interprets instructions written in the combined querying and scripting language. It is thus one of the tasks of the server to map instructions in the language to low-level storage operations.

This mapping can be decomposed into several stages, making it possible to partition the database into several layers (see Fig. 5.1) where each layer can make use of the functionality provided by lower layers.

This section explains how the TENTACLE server was partitioned into its layers. The description progresses upwards from the lowest layer.

The lowest layer of the TENTACLE database system is called the *block manager*. It accesses secondary storage via seek, read and write system calls to the host operating system. Since disk devices, device drivers and file systems typically use a fixed block size for their internal operation (common sizes are $1K$ or $4K$), it was decided that the block manager request fixed (instead of variable) size blocks, thus avoiding the penalty of having the operating system merge or split variable size blocks.

It is the task of the block manager to keep track of free and used blocks and service requests submitted by the other database layers for new and existing blocks from its local set of cached fixed-size pages.

The layer above the block manager is called the *graph storage manager*. It performs all the basic graph manipulation operations — these comprise operations to create, retrieve, modify and delete nodes and their attributes. This makes the graph storage manager responsible for mapping the nodes

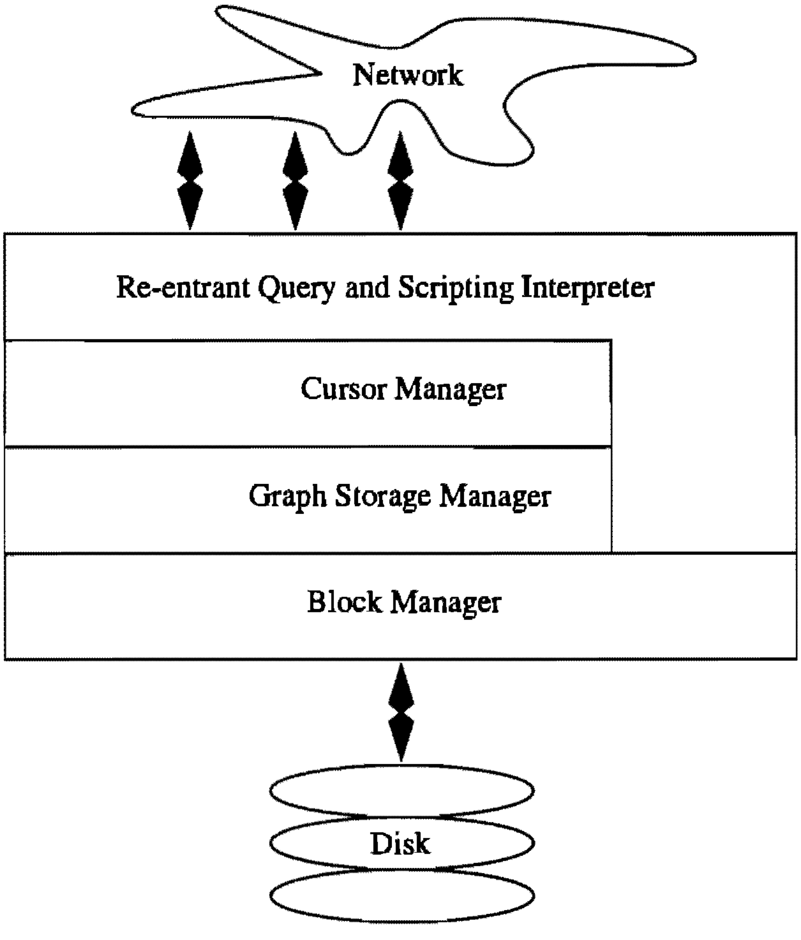

Figure 5.1: Components of the TENTACLE Database

and their attributes onto the pages supplied by the block manager.

In addition the graph storage manager is responsible for maintaining a global node index so that references between nodes can be resolved efficiently, as well as a per node index so that individual attributes of a node can be retrieved efficiently on their key.

The operations provided by graph storage manager are used by the *cursor manager*. It is responsible for maintaining pointers or handles (here termed *cursors*) to nodes and their attributes. This allows the cursor manager to detect access collisions between different clients.

The language module is the topmost layer of the system and co-ordinates the other database components. It provides a re-entrant interpreter for both the query and the scripting component of the TENTACLE language — in other words it is capable of servicing several remote clients simultaneously. This module receives and parses the instruction stream of a client, dispatches the instructions, uses the cursor module to access the database graph and returns output to the client.

This concludes the overview of the database components. The next sections present a more detailed description of the assumptions made, design decisions taken and tradeoffs arrived at for each component implementation.

5.2 Block Manager

The TENTACLE block manager is responsible for reading and writing data blocks or pages from secondary storage on behalf of the other system components. The block manager can use either a single file or disk partition as secondary storage. Writing directly to a partition allows one to bypass the overhead imposed by the file system.

It is assumed that the available secondary storage (disk space) is significantly larger that the available primary storage (Random Access Memory) and that accesses to secondary storage are comparatively expensive operations, but unavoidable since the entire database might not fit into RAM. Thus the block manager maintains a buffer or page cache, so that only active parts of the database need to be resident. An incoming request is compared to the content of the cache. If the block has already been retrieved and is available in one of the buffers, then the address of that buffer is returned to the calling layer (usually the graph storage manager). Otherwise the block at the given file or disk offset is read into the least recently used page and its address returned to the callee.

It may seem counter-intuitive to maintain a block cache when the operating system can maintain a block cache as well. However, there exist three advantages which a user-level cache has over a cache within the operating system:

1. Most conventional operating systems can not easily be modified to adjust their caching strategy (especially on a per file/device basis), and the global caching strategy might not be the best one available for the database.
2. In most operating systems it is difficult to pass information about the importance of a block to the caching component of the operating system — in general there is no way of providing the operating system with hints as to the likelihood that a given block will be requested again.
3. An operating system level cache incurs the overhead of a copy operation, even on a cache hit, since the data needs to be transferred from an internal operating system buffer into the area specified by the user-level program, while a user level cache only needs to pass an address to the calling function.

Since the TENTACLE database is intended to function as an experimental/research platform, it might be desirable to modify the caching strategy at a later stage and investigate the effects of the modifications on the performance of the system (currently such an investigation has not yet been attempted). Thus it seemed prudent to include a user-level cache.

The first two reasons enumerated above also motivate the decision not make use of a memory-mapped file/device interface[1], instead the database interacts with the operating system via the conventional read, write and seek system calls.

5.3 Graph Storage Manager

The graph storage manager is responsible for mapping graph structures onto the block buffers supplied by the block manager — the graph storage manager uses the services of the block manager to provide the operations to store, retrieve and modify the components of a graph.

A decision which influences the set of possible designs of this module (and which needs to be taken in almost all storage system where references

[1]Memory-mapped IO is the process whereby the operating system uses the virtual memory facilities provided by the hardware to place (map) a file into the address space of a process, causing the file to appear like a normal memory area — this removes the overhead mentioned in point 3 above and confers the advantage of having a cache hit or miss detected in hardware.

exist between stored entities) is the choice between implementing a reference between two entities as a direct pointer to the storage location of the entity, or as a reference to a logical identifier which is only later mapped onto the address of the entity.

The direct pointer approach has been used by systems such as O_2 [27]. It offers the advantage of minimizing the lookup costs, but makes moving stored entities difficult, since this involves either updating all pointers on referring entities or keeping a forwarding pointer to the new address at the original location of the (now moved) entity. Updating all referring pointers is expensive, while forwarding pointers fragment the storage space.

The alternate approach of using logical identifiers as references which are mapped onto an address introduces the performance penalty of an extra lookup for each access. The advantage is that it becomes comparatively easy to move an entry (since only the lookup system, instead of all references, needs to be updated). For the same reasons it is also less expensive to compact the holes left by deleted entries in an effort to minimize fragmentation.

The TENTACLE system uses logical identifiers (sometimes also referred to as surrogates), since it was anticipated that nodes within the system are likely to change size relatively frequently and these resize operations may involve the movement of nodes. Logical identifiers also make it possible to support a stronger form of node identity as defined by [38].

This choice means that the graph storage manager layer can be divided into three principal subcomponents: An index mechanism to assign identifiers to nodes and map these identifiers onto addresses, a component which manages the packing of nodes into block buffers and a component which organizes the internal structure of an individual node. These parts are explained in more detail below.

5.3.1 Node Index

The function of this component is to map a logical identifier onto a physical block address (an offset into a file or directly into a disk partition). It was deemed desirable to have the block address of a node independent of its logical identifier, since this has the advantage that the system can select any block which has sufficient space (resulting in far better space utilization) and also makes it possible to implement more sophisticated graph clustering strategies at a later date (dynamic clustering strategies which attempt to cluster groups of nodes which reference each other (see [50])). However, this approach of making physical node locations independent of their logical identifiers has the disadvantage that there has to be an index entry for each node. Since it is conceivable to have large numbers of small nodes, this means that the index structure can be very large. Thus a memory resident structure does not seem to be a viable solution; instead the index resides on disk and only its actively used parts are paged into memory.

Two of the more common approaches used for maintaining such indices are trees (B-trees are common for disk resident structures) or hash tables (although these are more frequently used for memory resident structures).

A hash table (in this case its task would be to hash the logical identifier onto a position in the table containing a pointer to the physical address of its block) offers a very fast lookup mechanism, but has the disadvantage of requiring large amounts of contiguous storage for efficient operation. If the storage space is reserved when the database is created, then it is possible that a significant amount of space may be wasted. On the other hand, if the table is only increased in size when needed (as is the case with extendible hashing), then, because the table would be disk resident, one incurs the expensive overhead of having to reorganize the database in order to create a larger piece of contiguous storage.

A B-tree, while not as fast as a hash table, has the advantage that it does not require a contiguous storage area, and makes efficient use of the storage allocated to it.

As a matter of fact, for the TENTACLE system it is possible to improve the space utilization of the B-tree even further, since one can take advantage of the fact that the system itself generates the node identifiers: If one uses a counter to generate the logical identifiers, it can be guaranteed the identifier of the newest node is always larger than all other existing ones[2]. This means that new entries are only ever inserted at the rightmost side of the tree (see Fig. 5.2). Thus nodes in the B-tree can be filled completely (there is no advantage in reserving space in the nodes for subsequent insertions, since it is certain that none will occur).

If no deletions occur, then only the rightmost nodes of this modified B-tree are ever incompletely filled. For such a case the average space utilisation (u) is better than $1 - (d/N)$ where N is the number of nodes in the B-tree and d is depth of the tree. This is a pleasing, since it means that the average space utilisation tends to unity as the tree increases in size.

Unfortunately this formula only holds if no deletions occur, since deletions, unlike insertions, may occur anywhere in the tree. However, it is possible to compact the tree using a post-order tree traversal which shifts each entry to the left by x positions and decrements the keys of the interior

[2]Admittedly there is the problem of encountering a counter wraparound, but the current implementation does not attempt to deal with that contingency since the default 32 bit counter can sustain one insert per second for 136 years, while a 64 bit counter can sustain a million insertions per second for 584 millennia.

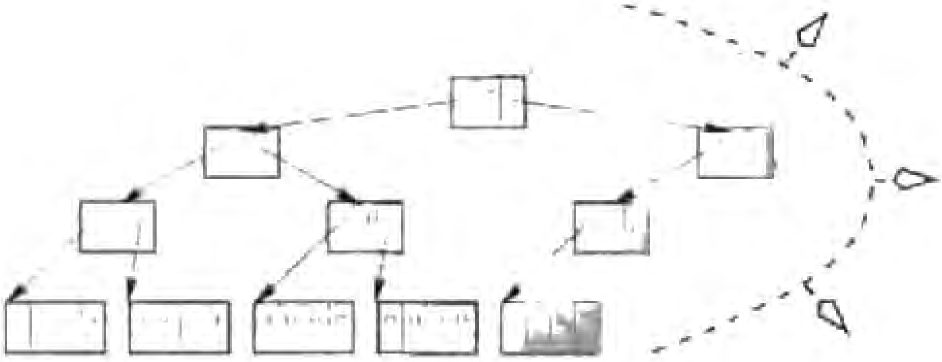

Figure 5.2: Growth of modified B-tree occurs only at right end

nodes by x, where x is the number of deleted entries to the left of the current entry or key.

The cost of this compaction per deletion can be calculated as follows. For a sufficiently large tree $1 - (d/N) \approx 1$. If we wish to keep the space utilisation (u) within some delta (δ) of this ideal value ($u + \delta \approx 1$), then the compaction needs to be run after δT deletions (where T the total number of entries in the index). The cost of a compaction is a tree traversal, or $O(T \ln T)$ accesses, thus the cost of a compaction per deletion is $\frac{O(T \ln T)}{\delta T} = O(\ln T)$. Since a deletion without compaction already costs $O(\ln T)$ (cost of descending the tree and marking the entry as deleted), periodic compaction of the index does not appear to be too exorbitant, since it only increases the cost of a deletion by a constant factor $\frac{1}{\delta}$. Furthermore, the effective cost of a compaction may be reduced by scheduling for it for a period when the database is lightly loaded.

Scheduling periodic compactions makes it possible to guarantee a minimum space utilisation. For example, for a values of $\delta = 0.2$ the worst case space utilisation is $u = 1 - \delta = 0.8$ or 80%.

While the above modified B-tree does make efficient use of storage space, in some cases access speed may be more important than efficient space usage. For this reason the TENTACLE graph storage manager combines a hash table and the above modified B-tree in its node index so that the user can select his or her own particular space-time tradeoff.

At the top level the TENTACLE node index makes use of a fixed size (determined at database creation time), disk resident hash table to map identifiers onto disk blocks. By default[3] the hash function uses the least significant bits of the logical identifier as offset into the table — since the identifiers are generated by a counter this means that the table should be filled sequentially, wrapping around each time the counter reaches a new multiple of the table size.

Collisions are resolved by using a secondary index structure (this method is also known as hashing with buckets), namely the abovementioned modified B-tree. This means that the database does not have to be perform an expensive table expansion operation as would have been the case if extendible hashing had been used.

This combination of hash table and B-tree lets the user of the database make a space-time tradeoff suitable for his or her purposes with a gradation between two extremes (see also Fig. 5.3):

- *Either* a very large hash table where few collisions occur and buckets are unlikely to be any larger than a single block. This ensures that very few disk accesses are required for a lookup, but has the overhead of a large hash table of a constant size even if the database itself contains few or no nodes.

- *Or* a very small hash table where each bucket is a large modified B-tree. This approach requires more disk accesses but makes efficient use of available space — blocks do not have to be contiguous, little space on each block is wasted and empty blocks are returned to the free block pool.

The operation of the entire index system can be illustrated by using a set of examples. Consider the toy index system having the following properties:

- A word size of 1 byte (resulting in a logical identifier space of 8 bits for a total of 256 identifiers).

- A block size capable of holding 8 words.

- A top-level hash table of 2 blocks for a total of 16 words.

The hash table holds 2^4 words with 2^3 words per block, so the least significant 4 bits will be used to hash into the table — the 4th bit is used to determine the block of the hash table while the least significant 3 bits are used to determine the offset within that block.

[3] The hash function may be modified to provide a means of clustering adjacent index entries but this will not be discussed in this dissertation.

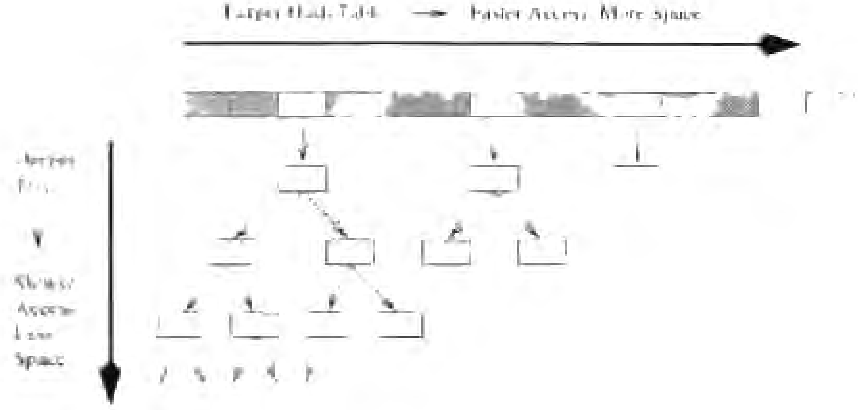

Figure 5.3: Space versus Time tradeoffs are determined by hash table size

The system starts with an empty table into which 12 nodes are inserted; the counter assigns them the logical identifiers 1 to 12 (zero is reserved to mark null or empty fields). The state of the index after these operations is depicted in Fig. 5.4. At this stage a lookup operation requires only one block access. For example, to find the block address of the node with identifier 9 the system examines the 4 least significant bits of 9. These are 1001. The set bit at position 4 informs the system that the second block in the hash table should be requested from the block manager, while the 001 at positions 3,2 and 1 indicates that the address of the node is to be found at offset 1 (2nd word) on this block.

Thereafter a further 5 nodes are inserted having identifiers 13 to 17. At identifier 16 the hash function wraps around, and at identifier 17 the first collision is encountered. This forces the allocation of a bucket.

The body[1] of the bucket (a leaf of the modified B-tree) consists of identifier/address pairs. In the example (see Fig. 5.5) the leaf block contains two such pairs, one for 1 and one for 17. Note that the node identifier can not be implicit as is the case in the hash table where there is only one possible slot for the corresponding address, but rather has to be stored explicitly on the leaf node since a given node identifier/address pair can be stored anywhere in the body of the leaf node.

After inserting a further 16 nodes (18 to 33 inclusive) the above leaf is full; it contains 3 identifier/address pairs for 1, 17, and 33, and on inserting the 49th node the modified B-tree grows from a single leaf to an interior

[1] The leaf node possesses a header signature of two null values to distinguish it from interior nodes of the modified B-tree and data blocks (blocks storing the actual node and thus not part of the secondary index structure).

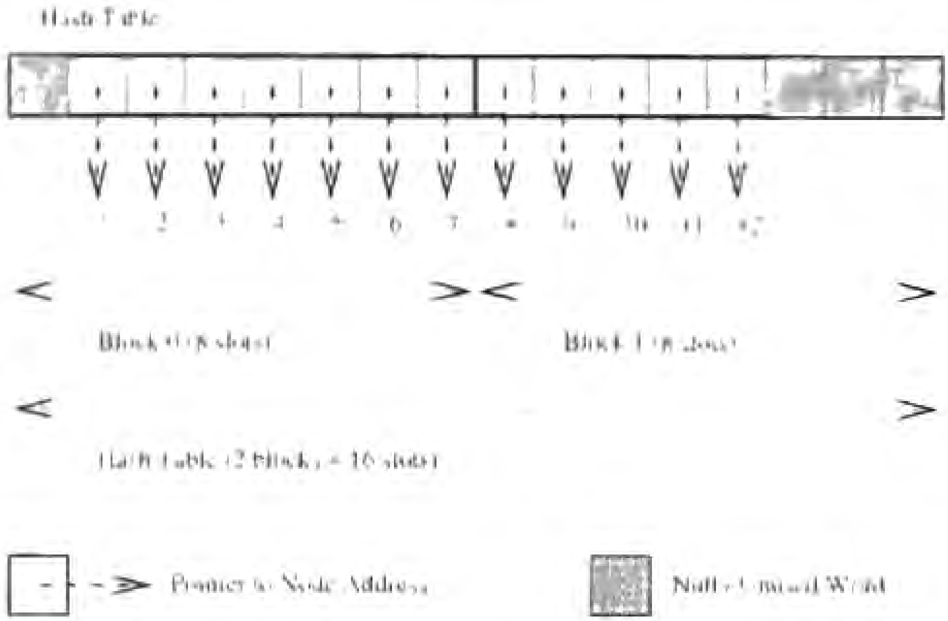

Figure 5.1 Index after 12 Inserts

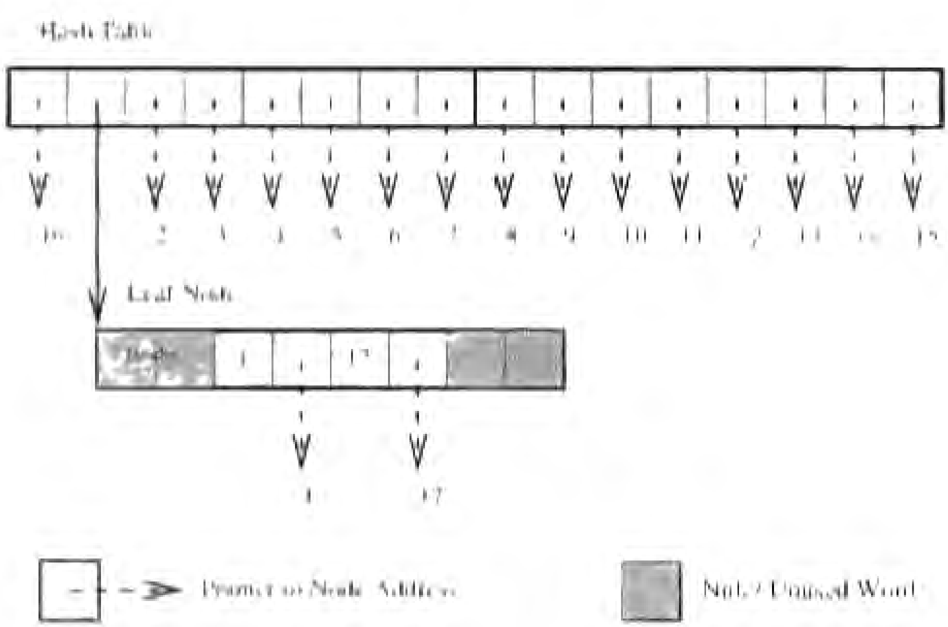

Figure 5.2 Index after 17 Inserts

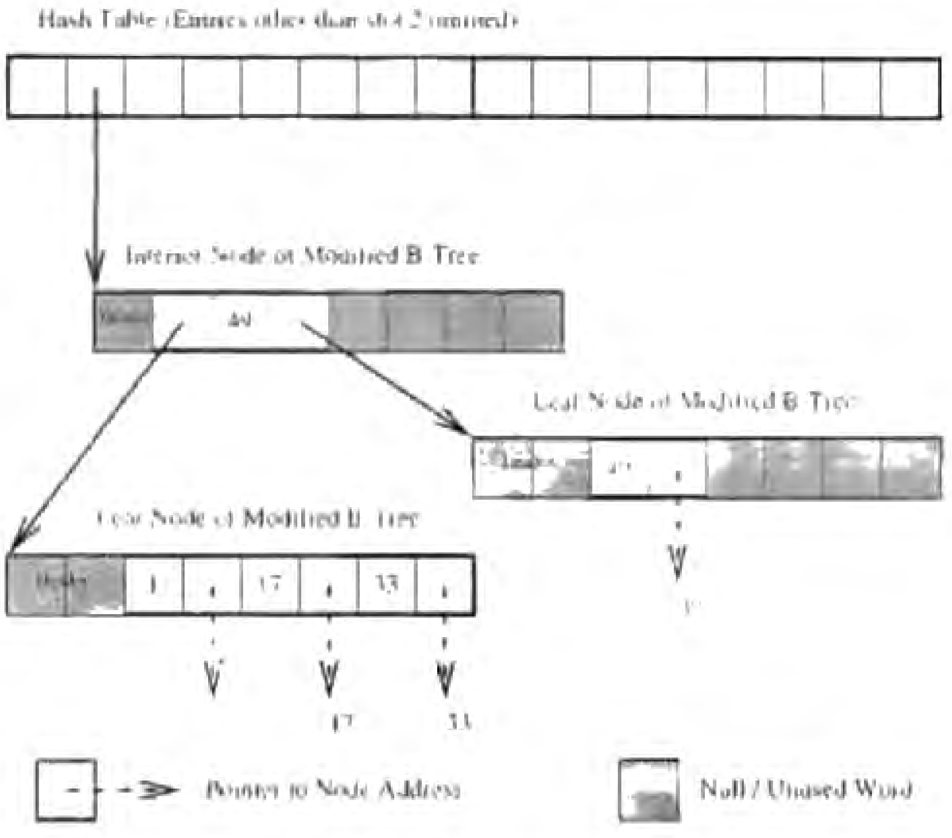

Figure 5.6: Index after 49 Inserts

node[5] and two leaves (illustrated in Fig. 5.6). Note that the left leaf is not split and balanced against the right leaf as required in a conventional B-tree since it is certain that no further entries will be inserted in the left leaf.

At this stage the cost of locating the address of the node having identifier 17 has risen to 3 block accesses. For some applications this cost may be acceptable, while others might need faster access speeds. In such cases the creator of the database may specify a larger hash table — in this toy example a hash table of 8 blocks (for a total of 64 slots) would reduce the access cost to 1 block request. However, these 8 blocks could be underutilized (wasted) if significantly fewer than 64 nodes are present in the database.

[5]Note that the header of the interior node consists of a single null (compared to the two nulls for leaves and a unused first word for data blocks).

5.3.2 Node Packing

The node packing component is responsible for selecting a physical address (disk block and offset into that block) for a new node. As the node changes in size, the packing component may also have to move the node to a new address, find further storage blocks, or may have to move adjacent nodes to ensure that there is sufficient space to accommodate the increase in size.

In other words, the node packing component is responsible for finding a block to store a newly created node as well as finding new blocks if the node requires more storage space.

It should be noted that nodes are written directly into the buffers supplied by the block manager (which writes the content of a buffer without any modifications to its corresponding block on disk). There is thus no structural difference between a memory resident node and one written to disk — there is no swizzling of pointers or references or any other form of unpacking of a disk resident node to transform it into a memory resident structure before it can be accessed or modified. This *zero copy* approach reduces the amount of memory used by the TENTACLE system.

The packing procedure is defined as follows. When a node is created it contains no attributes and is significantly smaller than a block. A block is selected which has sufficient space for a new node (if no such block is available then a new block is requested), and the node is written to it. As attributes are added to the node, it is possible that the node becomes too large for the current block. If the growing node shares the block with other nodes then it is moved into another block which has more free space (that block may have been newly allocated). If the node becomes too large to fit into a single block, then the node is marked as large and additional blocks are allocated to it. Once a node has been marked as large, it does not share blocks with other nodes.

This procedure results in two kinds of blocks storing nodes. The one type contains one or more small nodes and is sharable; the other kind of block is part of a large node and is not shared between nodes. This approach has been chosen since it limits the fraction of space wasted: a large node may require n blocks of which the last may be nearly empty. So the space utilization is $\frac{n-1+\delta}{n}$ where δ is the amount of space used on the last block. By definition a large node occupies more than one block, so in the worst case the fraction of space used is $\frac{2-1+0}{2} \approx 50\%$ — an acceptable value for a worst case, so there does not seem to be sufficient incentive to find a use for the wasted space on the last block. On the other hand, a small node may occupy as little as 196 bytes[6] of a 4096 byte block. That would result in a space utilization of

[6]This is the fixed size overhead imposed by the node header which contains information

$\frac{190}{4096} \approx 5\%$, with 95% of the space on the block going to waste. This value is too high (especially in view of the fact that it is anticipated that most of the nodes in the system may be quite small), and hence the motivation for packing several small nodes onto a block.

5.3.3 Internal Node Organization

Each node consists of three parts: A header, a sorted list of node attributes (key/value pairs) and an index on attribute keys. The attributes are sorted to minimize the number of block requests required when performing a sequential access, while the index is used to minimize the number of block requests required when accessing attributes values on their keys.

The header contains such information as the logical identifier of the node, total node size, a count of attributes, a pointer the node attributes as well as a pointer to the index structure. The header imposes a fixed overhead of slightly less than 200 bytes.

The node attributes are stored as a sorted sequence. There is no limit on the number of attributes a node may have, and both the attribute key and its value may be of arbitrary length[7], where attributes larger than a block are stored in a list of blocks.

The index is a modified binary tree. The modification has to do with the manner in which an attribute key is stored in internal nodes. In a conventional binary search tree the entire key is stored within an internal node. Since the key can be variable in size this means that the interior nodes can also be of variable size. This complicates node management. In order to avoid these complications, the TENTACLE system borrows an idea from Patricia trees [56] which makes it possible to use interior nodes of a fixed size.

In a Patricia tree only the index at which the binary string representations of the keys at left and right subtree differ is stored in an interior node[8] with

such as the logical node identifier, node size, attribute count and flags.

[7]Within reason. Both the number of attributes and their size is constrained by the largest number which can be stored as a system word (currently $2^{32} \approx 4$ billion), although it is unlikely that other limits will be reached before the global block address space is exhausted, since it too is bounded by the size of a system word.

[8]Actually Patricia trees merge (pair) each interior node with a leaf (data) node, but this fact is ignored for the sake of simplifying the comparison since the TENTACLE system does not use this approach. This decision was motivated by the fact that merging data nodes with interior nodes would have reduced the effective density of interior nodes per block (each interior node becomes significantly "larger"), with the result that more block cache misses would occur when accessing a number of keys of a particular node.

the requirement that the indices have to increase with each subsequent branch down the tree.

The TENTACLE tree uses a similar approach, but instead of operating on the binary representation of the key, it uses the conventional (character array) representation of the key. Each interior node of the tree stores the *first* index into the key at which the left and right branches (subtrees) of the node differ, as well as the character at that index which has the lesser value. The system also enforces the constraint that the values of these indices have to increase or remain equal to the previous index (ie may not decrease) as one descends the tree. The next paragraphs describe the lookup, insert and deletion operations on this modified tree.

Let k be the key to be found, $|k|$ be the length of this key and k_i the character at a particular index[8] i of key k. Each interior node contains the index i at which its branches start to differ as well as the character c of the left branch at that position.

A lookup is performed by examining the index i stored in the root node. If it is greater than the length of the key to be found ($i > |k|$) then the left branch of the tree is taken. Otherwise the character of the key k at the index i specified is compared against the character c stored in the interior node. If the character in the key is equal or less than the character stored in the node ($k_i \leq c$), then the left branch of the tree is taken, otherwise the right branch is chosen. This process is repeated on each subsequent node until a leaf node is encountered, where the full keys are compared — if they match then the lookup was successful.

For example, searching the tree depicted in Fig. 3.7 for the key `Alphard` would proceed as follows. At the root node (node 1) the third character of `Alphard` (`k[2]='p'`) is compared against the character `'d'` stored in the node. Since `'p'>'d'`, the right branch is taken. Now `'p'` is compared against `'g'` stored in node 2 and again the right branch is taken. Thereafter `'p'` is compared against `'p'` of node 4 and so the left branch is taken. This points to a leaf node and so the full key `Alphard` is compared to the leaf which results in a match. Note that a full comparison has to be performed, otherwise a search for keys such as `Ilphard` or `Alpha` would return false matches.

The procedure to insert a new key k makes use of the lookup operation. The new key k is used in the comparison against the interior nodes as explained above. Once a leaf l is encountered a full comparison is performed against the leaf. If the key on the leaf is the same as the new key ($l = k$), then the new key k is inserted immediately before the leaf l, effectively forming a

[8]Starting with index 0 at the first position.

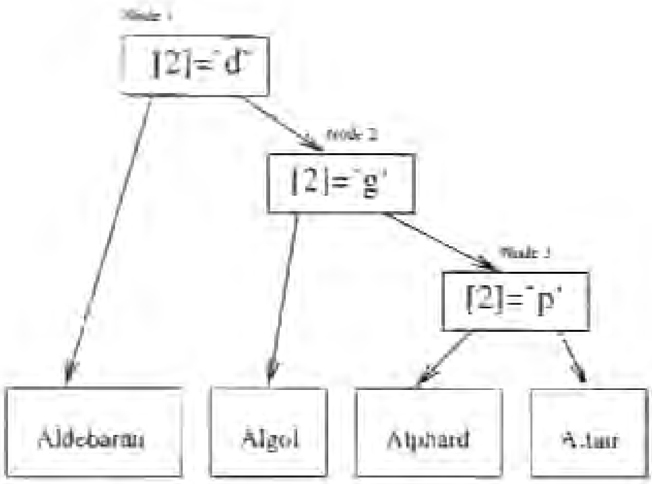

Figure 5.7: Example Index Tree

linked list.

Otherwise the index d at which the first difference between the new key k and the leaf l occurs is used in a second descent of the tree. However, this descent only progresses as far as the deepest interior node p which contains an index i equal or smaller than the computed index ($i \leq d$). Immediately below this node p (with branch b) a new node n is inserted. The one branch of the new n is set to point to the new key k, while the other is a pointer to the branch b which the new node n displaced on its parent p. The index i and character c stored in this new interior node are taken from the comparison of the new key k against the leaf l on the first descent of the tree ($c = min(l[d], k[d])$). A pseudocode explanation of the insert procedure is given below.

```
PROCEDURE TO INSERT KEY K

attempt to lookup key k returning closest leaf l
compare k against l
if k = l then
   insert k before l
else
   find index d where k and l differ
   let b be root node of tree
   while index i of node b smaller or equal than d
         descend as in lookup operation
   let p be parent of b
   let n be a new node
```

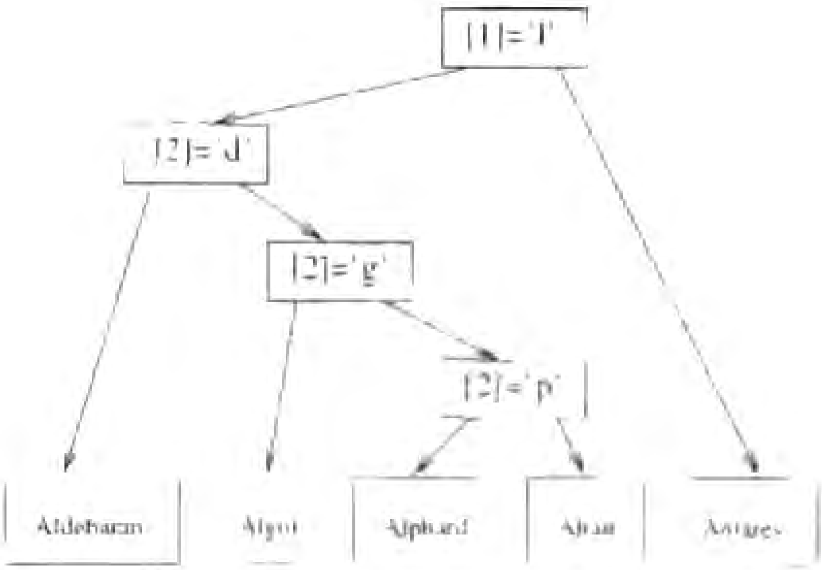

Figure 5.8: Tree after inserting the key Antares

```
insert n below p in place of b
if k[d] < l[d] then
   c = l[d]
   set right pointer of n to x
   set left pointer of n to b
else
   c = k[d]
   set right pointer of n to b
   set left pointer of n to x
set index of node n to d
set character of node n to c
```

To illustrate the insert method the key Antares is inserted into the tree in Fig. 5.7. Using the lookup operation one descends to the leaf Altair (Antares ([2]='t'): 't'>'d', 't'>'g', 't'>'p'). A comparison results in a mismatch at index 1 with 'l'<'n'. This prompts another descent of the tree. Since the index stored at the root node is 2 (which is greater than 1) the new node is inserted above it to replace it as the root of the tree. Since Antares is greater than Altair the right branch of the new node is set to point to the new node while the left points to the old root. The result of this insert is depicted in Fig. 5.8.

Deletion of a key is performed by using the lookup operator to find the matching key. The internal node immediately above this key is deleted and

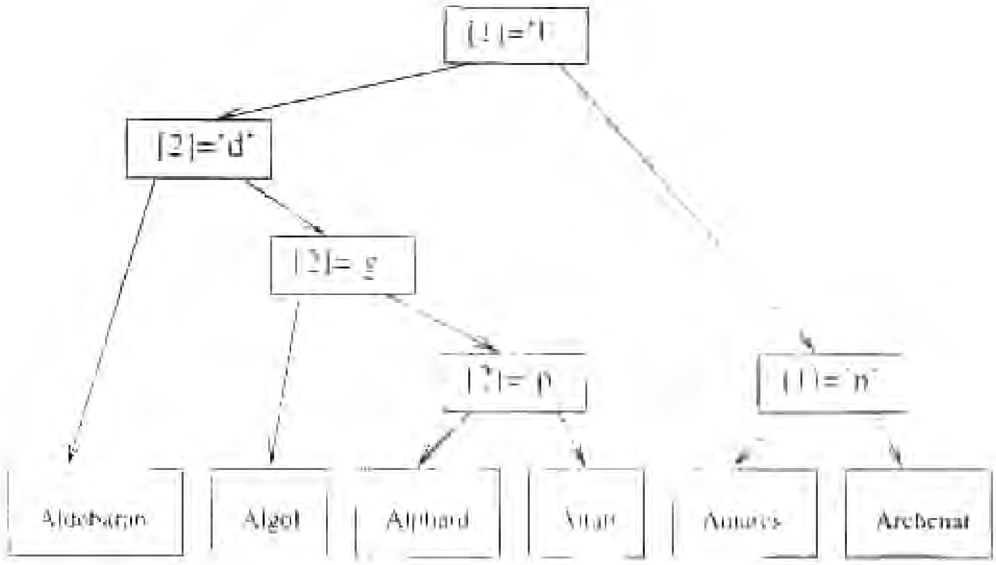

Figure 5.9: Skew, but not completely skew, tree

the pointer on its parent (which points to the (now deleted) interior node) is replaced with a pointer to the remaining branch of the node which was deleted

While insert operations are more expensive in the modified tree than their counterparts in a conventional binary tree, it should be noted that this tree does not require defragmentation of the storage space occupied by internal nodes (since all interior nodes have the same size) and also has the pleasant characteristic of having a reduced probability of constructing pathologically skew trees[10] without needing to introduce expensive rebalancing operations. For example inserting the sorted sequence of keys `Aldebaran`, `Algol`, `Alphard`, `Altair`, `Antares`, `Archernar` into the modified tree results in a structure depicted in Fig. 5.9, which, although skew, is not an expensive linked list as would have been the case in a conventional binary tree.

5.4 Cursor Manager

A *cursor* is a handle which allows a user to access a particular graph component (node, attribute key, attribute value)

[10] A conventional binary tree of n entries could have a worst case depth of n, while the modified tree has a worst case depth of $min(n, a * l)$ where a is the number of characters of the alphabet from which the key string is constructed and l is the number of characters of the longest key

Cursors impose a layer between the graph manager (which provides basic graph manipulation operations) and the users of these operations, namely the programs executed by the combined query and scripting language interpreter.

Since all accesses to the database have to be made through the cursor manager, it is possible to regulate access to the database at this layer. The cursor manager is capable of detecting concurrent accesses to graph components and denying access to those which have been locked.

A cursor is a convenient abstraction. Higher level components do not have to maintain any state other than a reference to a cursor when accessing a graph component — instead the cursor maintains any required state information on their behalf.

Consider a request made by the language interpreter for a particular attribute key of a given node. This request is passed to the cursor manager which establishes if the requested attribute is available and unlocked; if it is, then the cursor manager requests the graph manager to locate the physical address of the node using the global node index. This physical address is cached in the cursor for that access. Given the location of the node, the graph manager can be instructed to use the attribute index of the node to find the key attribute. The address of this attribute is also stored in the cursor. The cursor manager completes the request by returning a reference to this cursor to the language component.

Essentially the cursor manager uses cursors to present graph components in such a way that they resemble conventional variables — thus higher layers have a reasonably uniform means of accessing database components and local variables. The higher levels are shielded from most database maintenance tasks to the extent that a node may be moved or reorganised by the node packing module of the graph manager even while it is accessed. The cursor manager ensures that the resultant address changes are propagated to the cursors which reference the node without disrupting the operation of the language component.

5.5 Language Interpreter

The interpreter of the combined scripting and querying language is the executive of the database, coordinating all the other components. It accepts incoming connections from the network, reads data and interprets instructions sent from remote hosts as well as programs stored in the database itself. It also dispatches graph manipulation requests to the lower layers of the database, awaits their response and passes these back to the requesting party.

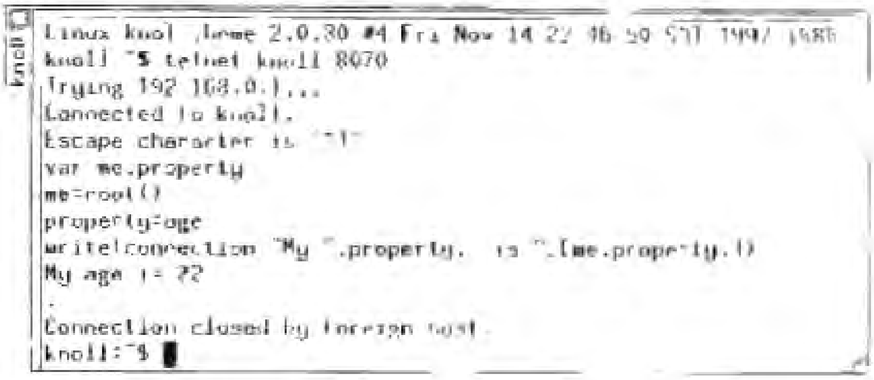

Figure 5.10: Screen dump of a telnet client connecting to the TENTACLE server

All these tasks occur within a single process/thread. Using a single re-entrant interpreter instead of multiple interpreter processes/threads has made the development of the system more complex, but eliminates the comparatively expensive overhead incurred on process (or even thread) creation, intercommunication and synchronization. Furthermore, it is easier to make assertions about the correctness of particular execution path for a single process system than it is for multiple processes or threads.

The database is an experimental system and thus, the first implementation, is intended to run on small workstations. Such platforms tend to be uniprocessor systems with comparatively limited IO bandwidth. In such environments bottlenecks tend to occur at the disk and network IO subsystem before the processor becomes fully utilized, hence the fact that the single database process can not take advantage of a multiprocessor host is not deemed to be a significant disadvantage.

The database interacts with the outside world via a reliable connection-based protocol, TCP/IP [52, 51]. TCP/IP (Transmission Control Protocol over Internet Protocol networks) is in common use on the Internet as well as on private networks, hence the database should be able to communicate with a large number of systems. As a matter of fact, it is possible to use a conventional telnet client to query the database. The screen dump of an actual session is given in Fig. 5.10 (the query is an example from Chap. 4).

The database server operates by listening to an advertised network port. For each incoming client connection, an interpreter context is initialised. The context maintains, amongst others, a set of variables (consisting of variables local to the instance as well as cursors into the database), a stack (to maintain parser state) and administrative information (including the amount of resources consumed by the client).

Once the interpreter context has been set up, the system is ready to execute instructions. Instructions may be read either directly from the network connection, or from the a script stored in database graph. The former allows the client to upload scripts or queries directly, while the latter allows the system to present a preprogrammed interface to the client.

The language interpreter is implemented as a table driven pushdown automaton, where a separate stack is maintained for each client, and where each client is allocated a time slice for execution. Executions are scheduled on a round-robin basis, while I/O is nonblocking — clients waiting for I/O are suspended in favour of those clients which are ready to run.

Queries are evaluated using sets of cursors. A path expression takes as input (and returns as output) a set of cursors. The segments of path expressions are matched against the dereferenced cursors as described in Chap. 4. Alternation and conjunction are computed in the conventional manner by taking the union and intersection of cursor sets, while concatenations of segments are used by the cursor manager to traverse the database graph (see Appendix B). The closure of a path subexpression is evaluated using a method resembling the semi-naive evaluation of DATALOG predicates. The input cursor set for the next evaluation of the subexpression consists of only new cursors, ie those contained in the result set of the previous evaluation but not in any earlier result set. If no new cursors can be found, the closure evaluation terminates. Given an input set of cursors I, a path expression E, the procedure to compute the result set R of the closure [E]* is given by the pseudocode below.

```
PROCEDURE TO EVALUATE [E]*

n = I
R = I
do
    evaluate expression E with
             input set of new elements n
             returning temporary result set T
    n = T set difference R
    R = R union T
while n not null
```

5.6 Summary

This chapter has described selected aspects of the implementation of the TENTACLE system. The system was built over a period of slightly more than one year and is a single person effort[11]. In view of these resource limits, a number of capabilities could not be included, or only implemented incompletely. In particular, the system lacks proper ACID transactions (while concurrent accesses are possible, the system only provides locking facilities, and conflicts have to be resolved at the user level).

Nevertheless, the system does perform reasonably well[12] and provides sufficient functionality to implement an example application which demonstrates the capabilities of the combined query and scripting language as well as the underlying data model. This example application is explained in the next chapter.

[11] The implementation amounts to about 30000 lines of C code.

[12] An elementary performance test (See Appendix C) indicates that the system is capable of inserting about 100 node attributes per second on modest PC hardware.

Chapter 6

Web Server Application

The demonstration application domain of the TENTACLE database system is the World Wide Web, where the TENTACLE system has been used to implement an HTTP or web server[1]. Web servers make data resources (usually hypertext documents) available to their users via the Hypertext Transfer Protocol (HTTP [17]) — as such all resources whose Uniform Resource Locators (URLs [18]) start with the `http:` prefix are retrieved from a web server.

Conventional web servers allow their clients to retrieve resources which are stored as files from a part of the directory hierarchy of the server file system. Note that this process does not involve a database, which seems rather surprising since facilities commonly associated with databases (such as integrity management and querying capabilities) could be useful for such an application.

There exist cases where hypertext entities are indeed generated from the content of a database; however, access to this data is indirect and extensive processing is usually required to transform the data from its internal storage representation to the native format of the World Wide Web. Essentially databases commonly encountered on the World Wide Web are regular databases which are accessible through a web-based interface. In such cases the World Wide Web (itself a networked semi-structured database) simply serves as a gateway to another database (typically a highly structured relational system).

The TENTACLE database system attempts to achieve a tighter integration between web server and database. This provides not only an opportunity to exercise the database system on a nontrivial application, but also the chance to extend the capabilities of web servers by addressing some of the

[1] A simplified description of this application may be found in [59].

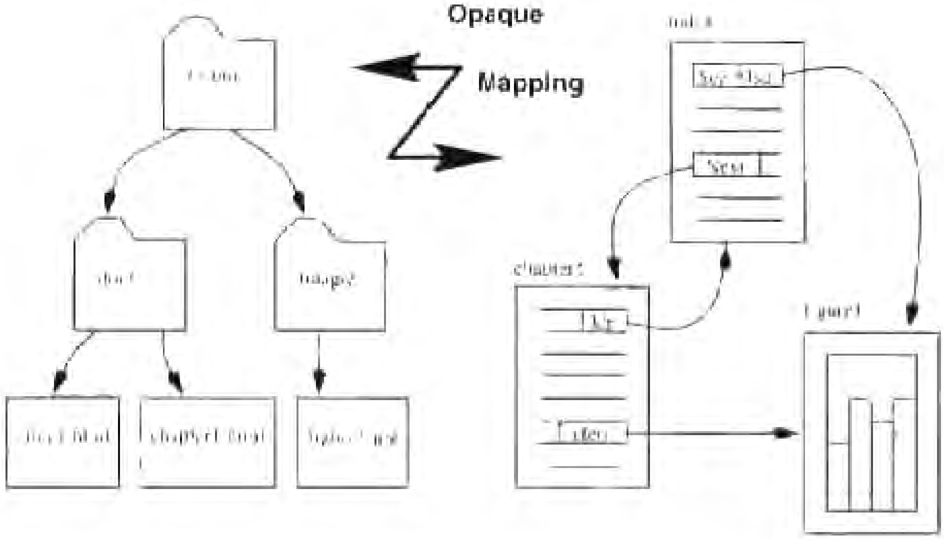

Figure 6.1: Opaque mapping between a file system and hypertext representation.

limitations of more conventional systems in an interesting manner. These limitations are explained in the following section.

6.1 Conventional Web Servers

A typical web server stores a hypertext entity (a member of a graph structure) as a conventional file (a member of a directory hierarchy). This form of storage does not provide a facility for representing the associations between hypertext entities, leaving it to the user to maintain hypertext link integrity.

Consider the example of three hyperlinked entities — an index document, a first chapter and an illustration (see Fig. 6.1) which might constitute part of a larger text. Stored on a file system they might be known to the author as the following three file locations:

```
/doc/index.html
/doc/chapter1.html
/image/figure1.jpg
```

The file storage system provides no information about the structural interrelationships. For example, it does not record that there exists a hyperlink named Next from the index to chapter 1. This makes tasks such as maintaining referential integrity (ensuring that there are no dangling or broken

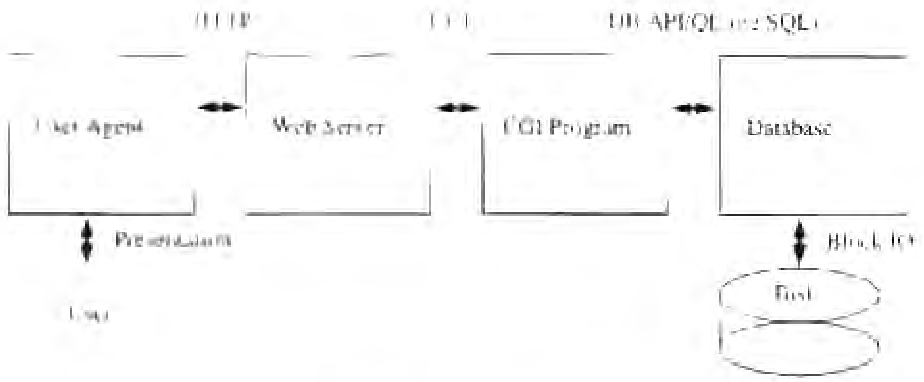

Figure 6.2: Overhead involved in requesting information from a conventional database through the web using the Common Gateway Interface (CGI)

links) difficult. Systems such as HyperWave [46] or Microcosm [36] attempt to address this issue of link management.

A further limitation of a file based storage system is that each hypertext document is stored as a single file. For purposes such as retrieval, a file is treated as an atomic entity, even though the stored hypertext document has a structure, and components of such a document (eg chapters of a text) may be of interest individually. This lack of resolution of file-based storage, which makes it difficult to query the internal structure of hypertext documents, has been identified in [32].

Moving the hypertext into a typical relational database does not necessarily improve matters. While one might have gained the stability and security of a database, additional effort is required to extract the information from the database (see Fig. 6.2). Not only is the extraction process inefficient (involving a communications overhead of passing a data stream through several processes), but there also exist significant mismatches. One is the classical impedance mismatch of embedding a query language (such as SQL) in a host language (such as C or PERL), while another is the mismatch between the structured relational model used by the database and the semi-structured network model used by the world wide web.

In order to achieve performance gains attempts have been made to reduce the above communications overhead — these include optimised gateway interfaces such as FastCGI [4] or web servers with built in interpreters such mod_perl or mod_php of Apache [23]. However, these systems do not attempt to address the host/query language and relational/web data model mismatches.

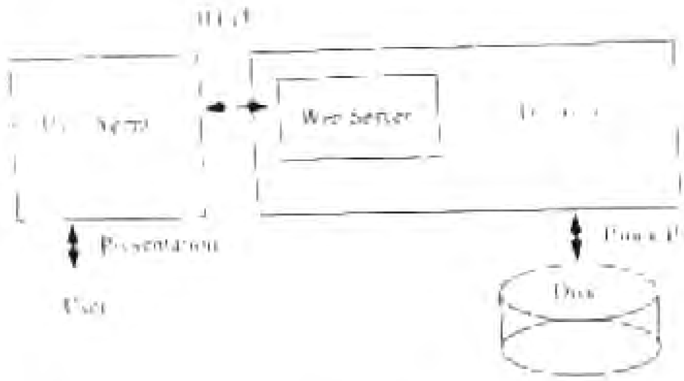

Figure 6.3: Reduction of communications overhead in the TENTACLE database with web server capabilities (compare with Fig. 6.2)

6.2 The TENTACLE Web Server

The TENTACLE database can be used to address the deficiencies identified in the above section. Its graph based data model can be used to provide a natural and reasonably direct representation of unstructured hypertext networks, where hypertext documents or fragments of documents are represented as nodes, while hyperlinks or other associations are represented as labeled arcs. Such a direct representation avoids the cost of translating from one data model to another.

The integrated query and scripting language of the TENTACLE system makes it is possible to fold the querying and post processing stages into a single phase, thereby avoiding most of the language impedance mismatch traditionally associated with a separate host and query languages.

Furthermore, because the database server is programmable, it is possible to eliminate the performance overhead of passing the data stream between several processes by implementing the web server inside the database process itself. The web server is a program written in the combined scripting and querying language and uploaded into the database server. This results in a direct interface between the database and the server within a single process (see Fig. 6.3 and compare with Fig. 6.2). Further performance gains should be achieved by making use of the TENTACLE storage subsystem, which has been designed to represent and access graph structures efficiently (see Chap. 7).

In order to explain this database/web server combination, it is probably useful to provide a brief overview of the Hypertext Transfer Protocol.[1] The

[1] The protocol version which will be explained is HTTP 1.0, defined in rfc1945 [illegible], the most widely used version at the time of writing.

protocol operates using a request/response mechanism where the request is initiated by the client. The request typically instructs the server to retrieve a named entity. The server responds by sending a status message and the requested entity.

Both the request and response message consist of a status line, additional header fields and an optional message body. Header and body are separated by an empty line. A typical session (where the client requests the entity known as `http://jade.cs.uct.ac.za/test/file.txt` from the host jade) might look as follows:

```
C_r  { GET /test/file.txt HTTP/1.0
     { Connection: Keep-Alive
C_h  { User-Agent: Mozilla/3.01 (X11; I; Linux 2.0.30 i686)
     { Host: jade.cs.uct.ac.za
     { Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
C_e  {
S_r  { HTTP/1.0 200 OK
     { Date: Tue, 19 May 1998 07:04:54 GMT
     { Server: Apache/1.1.3
S_h  { Content-type: text/plain
     { Content-length: 47
     { Last-modified: Mon, 09 Feb 1998 06:13:46 GMT
S_e  {
S_b  { This is the content of the test text file
```

In this example the client issues a GET request which consists of a request line (denoted by C_r), additional header information (C_h), the blank separating line (C_e) and a null body. The server responds with a status line (S_r) denoting success (OK [illegible] code 200), header fields (S_h) followed by the separating line (S_e) and the response body (S_b; here the content of the file `file.txt` [illegible] a single line of text). Note that the header fields (C_h, S_h) are optional.

Also observe that the path component of the URL (`/test/file.txt`) requested by the client is a file system path. However, the HTTP specifications do explicitly state that this component may not have to be a file system path, and the TENTACLE system uses paths through the database graph instead. This is transparent to the casual user — the appearance of the URL does not change.

Thus, in order to program the TENTACLE database to appear as an HTTP server, the following steps are necessary:

- The server has to accept and parse an incoming request.

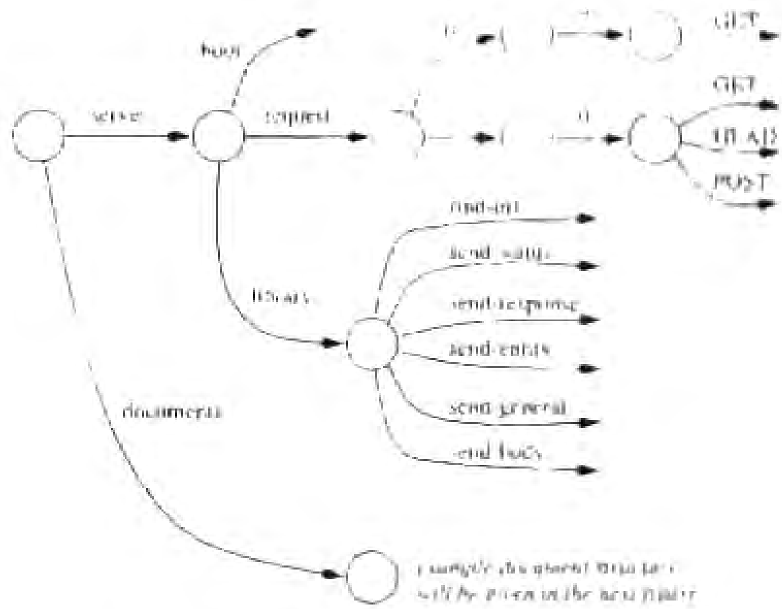

Figure 6.4: Web Server Script distributed over parts of the TENTACLE database graph. Note that only the structure is given, the content of the attribute values (ie the TENTACLE scripts) have been elided.

- use the path submitted by the user agent to traverse the database graph containing the hypertext documents,
- retrieve or generate the requested hypertext entity,
- and finally return a well-formed response.

In an attempt to explore the uses of a graph-based structure, the web server script itself has been distributed over a number of node attribute values in the database graph (see Figure 6.4). Each of these node attribute values may be viewed as the equivalent of a function or short script. The operation of these scripts will be explained in greater detail in the following sections.

6.2.1 Incoming Requests

In order to respond to HTTP requests, the database server is programmed to execute a designated script (`[server . boot]`) for each incoming connection. In other words, instead of the default operation of interpreting the command stream sent from the remote host (as has been the case in the examples of previous chapters), the server interprets a previously created command stream stored as part of the database graph to parse the incoming request.

The initial script is responsible for parsing the command line (C_l in the above example). The script attempts to retrieve the protocol method and major and minor protocol revision numbers. These are stored in the variables `method`, `major` and `minor` respectively. These variables are then used to dispatch the client request to the correct handling function.

But instead of using a `switch` or similar statement to perform the dispatch, a path expression is used — in particular the expression `[.server.request.major.minor.method.]` is used to retrieve the attribute containing the correct handling code for the given request. This illustrates how the query component of the language may be used as an interesting construct in the scripting language.

For example, given the request line `GET /logs/ HTTP/1.0`, the initial script will use the above path expression to invoke the handling script stored at the location `[.server.request.1.0.GET.]`, while the request line `POST /logs/ HTTP/1.0` would result in the invocation of the script stored at `[.server.request.1.0.POST.]`. An interesting aspect of this approach is that support for new operations (eg an HTTP/1.1 `PUT` handling script) could be added to the system at run-time — there would be no need to restart or suspend the web server.

The invoked handling script then processes the client request — this includes interpreting any additional header fields (C_h) and calling a script (stored as `[.server.library.find-url.]`) to traverse the database graph to return the requested hypertext entity. Scripts to perform other tasks common to several request types are also stored as attribute values of the `library` node.

6.2.2 Graph Traversal

The paths accepted from the client resemble file system paths in that segments are delimited by the / separator. However, instead of being used to traverse a file system, the paths are used to traverse the database graph[3]. The traversal is performed by treating every component of the supplied path (except for the trailing /) as a match for a node attribute key (ie an edge).

Consider the example given in Fig. 6.5. A requested path of the form `/status/` would be transformed by ignoring the trailing /, inserting a don't care node match between every path component and prepending the docu-

[3] The traversal starts at the node [.documents.] and not the real database root, in order to hide the server scripts.

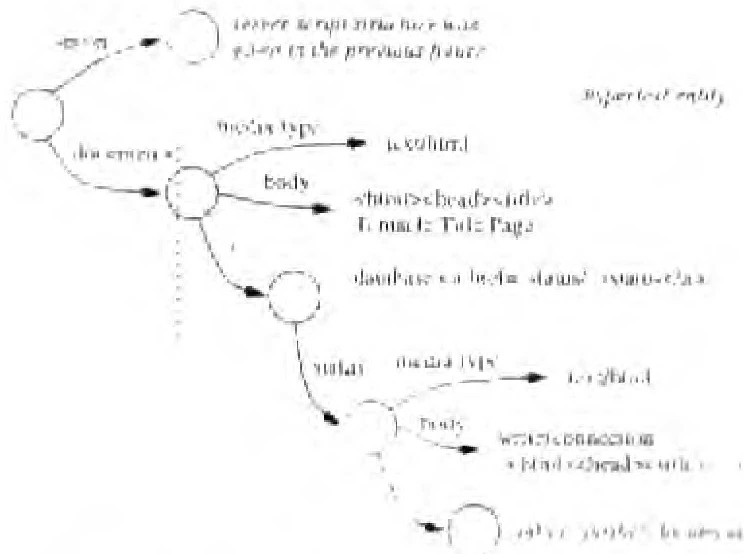

Figure 6.5: Two hypertext entities: / and /status/

ment root of the database graph. The final result of this transformation is the path expression [documents . '/' . status].

The attributes of the node retrieved by this path expression contain the data associated with the requested hypertext entity. The attribute with key body contains the entity itself (or a script to generate the entity), while the other attributes might contain header fields used in the HTTP response (such as the media type or an expiry date) and other data.

Of particular interest is the attribute with key /. By arranging that this attribute references a node which contains a list of all relative paths[1] emanating from the current hypertext document, it becomes possible to update relative links without having to modify the document body. For example, the root document (request path /, transformed path [documents]) contains a relative link to the status page (status/) in its body ([documents . body]). This link can be updated by modifying [documents . '/' . status] to reference another node without having to modify the document body.

6.2.3 Response Generation

As explained above, the response of the web server consists of a status line (S_l), header fields (S_h) and body (S_b). Both status line and header fields may be stored as attributes of the corresponding hypertext node, making

[1] Relative paths do not begin with a /. The user agent will transform such paths into absolute paths by prepending the current absolute path up to the last /.

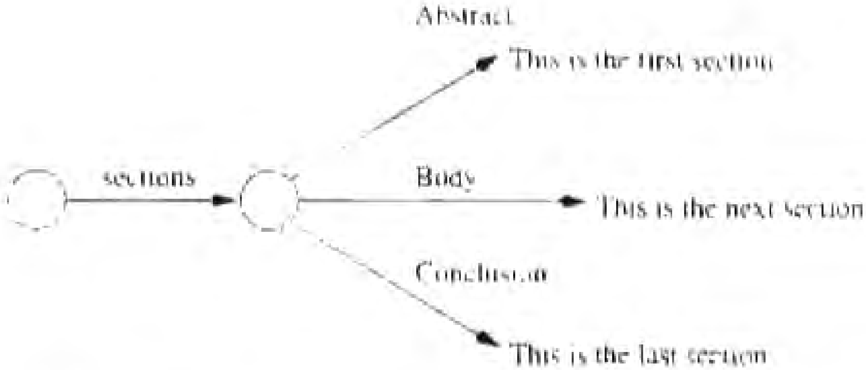

Figure 6.6: Document of three chapters stored as a node in the TENTACLE database

it possible to implement features such as expiration dates or redirects on a per-document basis.

The handling scripts may generate further fields (such as the `Server` header) or substitute defaults for a number of fields if they have not been specified explicitly.

The hypertext entity itself — ie the response body (S_b) — may either be retrieved directly from a node attribute value (static eg `/`) or be the output of a script (generated dynamically eg `/status/`).

The above explanation has shown how entities encountered on the World Wide Web can be mapped into the TENTACLE database, that this process is transparent, reasonably straightforward and that the mapping makes it possible to represent the links between hypertext documents explicitly.

The next section provides two examples which illustrate how the combined query and scripting language of the TENTACLE database system can be used to query the content of the database and that such queries can be hidden from the user, making it possible to generate document views automatically.

6.3 Document Views

Consider a document consisting of three sections: an abstract, a body and a conclusion. Such a document could be stored in the system as a node with three attributes as illustrated in Fig. 6.6, where the attribute names contain the chapter headings (`Abstract`, `Body`, `Conclusion`) and the attribute values contain the text of the chapters.

Assuming that the variable `doc` contains a reference to the document node, then the following query can be used to generate an HTML fragment which annotates the section titles with the appropriate header tags:

```
[doc 'sections'
 write(connection,'<h2>',here(),'</h2>',[here().1])]
```

This path expression consists of four top-level segments, where the last segment is a nested function call. The function call will be evaluated for all section titles, where `here()` returns the current title and `here().1` the text of the section. The output of this expression is given below[7]:

```
<h2>Abstract</h2>
This is the first section
<h2>Body</h2>
This is the next section
<h2>Conclusion</h2>
This is the last section
```

Note how the combination of query and scripting language allows one to add the generation of markup tags to the querying phase, removing the need for a postprocessing phase typically encountered in other systems. For example, using PHP3 [10] and a relational database to perform the equivalent task, would require the following script[8]:

```
$result_handle=query("SELECT heading,body FROM sections");
$row=fetch_row($result_handle);
while(is_array($row)){
  printf("<h2>%s</h2>%s",$row["heading"],$row["body"]);
  $row=fetch_row($result_handle);
}
```

The data given in Fig. 6.6 may be used by another query to present the document in another form. For example, the following query generates an index which allows the user to jump to the appropriate section in the document:

```
write(connection,
   '<h1>Linear Document with Table of Content</h1>',
   '<h2>Table of Content</h2><ul>')
[doc 'sections' write(connection,'<li><a href=\"#',
```

[7] Line breaks have been inserted into the output to improve readability.

[8] Database initialisation, connection and error checking code have been omitted, and generic database function names have been used in place of package specific ones.

```
 here(),'\">',here(),'</a>')]
write(connection,
   '</ul><h2>Document Content</h2>')
(doc.'sections'. write(connection,'<h2><a name=\"',
 here(),'\">',here(),'</a></h2>',[here()]))]
```

The output of this query is given below:

```
<h1>Linear Document with Table of Content</h1>
<h2>Table of Content</h2>
<ul>
<li><a href="#Abstract">Abstract</a>
<li><a href="#Body">Body</a>
<li><a href="#Conclusion">Conclusion</a>
</ul>
<h2>Document Content</h2>
<h2><a name="Abstract">Abstract</a></h2>
This is the first section
<h2><a name="Body">Body</a></h2>
This is the next section
<h2><a name="Conclusion">Conclusion</a></h2>
This is the last section
```

This output rendered by a web browser is given in Fig. 6.7.

By storing a script such as the above as the document body of a hypertext node, and storing the decomposed chapters as part of the metadata of the hypertext node, it is possible to maintain dynamically generated views of a document in a way which is transparent to the user.

6.4 Summary

This chapter has shown how the TENTACLE database may be used to build an application. The combined query and scripting language has been used in the construction of a web server, where the web server scripts have themselves been stored in the database, making it possible to use path expressions as programming constructs of the scripting language.

In addition, the combined querying and scripting language has been used to generate two different materialised views from the same data source. This was accomplished in a single phase — the query and presentation phases were folded, reducing the programming costs of accessing the database content.

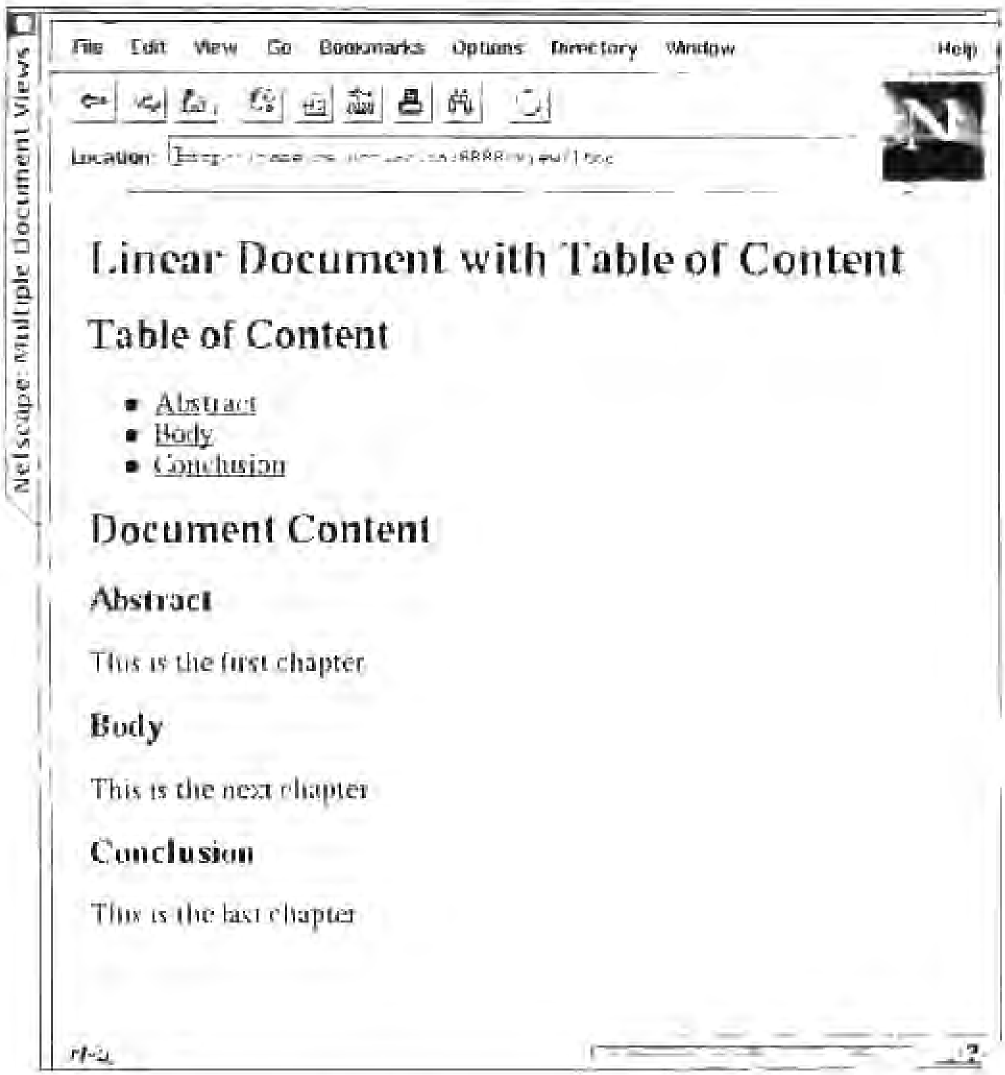

Figure 6.7 Generated HTML document as rendered by a web browser

Chapter 7

Conclusion

The TENTACLE database system is an attempt at constructing an atypical database system — it may be viewed as an exploration of the space of possible database system. Alternatives have been chosen in the design of several of the database components, including the data model, query language and implementation. These atypical components have been successfully integrated to implement a system which has been used to build a non-trivial example application.

The system has been developed around a simple, untyped, graph-based data model. This model was chosen because graphs are unconstrained, general structures[1], and thus make it possible to represent arbitrary aggregations and associations reasonably directly. Such unconstrained data models appear to be useful in semi-structured application domains — domains where it is not feasible to develop a conventional schema.

The data model defines the core of the system — the other components may be described in terms of this graph-based model:

- The query component uses path expressions to traverse and extract information from the database graph. The path expressions can be viewed as extensions of paths encountered in file systems or object-orientated languages and databases. The distinguishing feature of the query language is that path expressions are its sole querying construct — it lacks the more conventional query clauses encountered in languages such as SQL. Nevertheless, the path expressions are sufficiently powerful to express queries which are difficult to formulate in SQL. Examples of these include queries computing a closure (the language

[1]Graphs subsume trees, which in turn subsume lists, while [19] presents a trivial algorithm which may be used to encode the content of relational databases as a graph.

provides the * operator for this purpose) or queries which contain a typed cycle in their query graph (see Sect. 4.4 and [10]).

- The scripting module can be described as a procedural language which uses the database graph as a persistent composite data structure. The scripting component is tightly coupled to the query module. Conventional R-values (sometimes also referred to as expressions) occurring in the scripting language may be replaced by path expressions and vice versa — R-values of the scripting language may occur as components of path expressions. The integration of declarative query expressions and a procedural scripting language not only guarantees the computational completeness of the system, but also removes the so-called impedance mismatch encountered in loosely coupled two-language systems, where it is necessary to convert between the different data structures of the query and scripting language. A further advantage of the integration of scripting and query subsystem is that it is possible fold the query and postprocessing phases into a single stage — like the removal of the impedance mismatch, this reduces the effort required to access the content of the database.

In order to explore the possible applications of the above data model and combined querying and scripting language, an implementation of the system has been developed. The implementation possesses the following features:

- A native storage implementation which seeks to store graph structures efficiently. The storage module supports node identity using logical identifiers or surrogates. These make it possible to support strong node identities (see [48]) as well as future explorations of dynamic node caching strategies.

- A reentrant interpreter of the combined querying and scripting language. The interpreter uses a semi-naive evaluation strategy (as encountered in systems such as DATALOG) to perform a breadth-first evaluation of path expressions over the database graph.

- A TCP/IP network interface capable of servicing multiple requests from remote servers. The interface itself may be programmed using the scripting language, making it possible to support application level protocols in the database itself.

The implementation has made it possible to apply the system to a problem domain. The example domain chosen is the World Wide Web, where the TENTACLE system has been used to construct a web server.

In the example application the hypertext network which is the World Wide Web has been mapped to the graph-based model of the TENTACLE database system. This mapping is reasonably straightforward. The query language has been used to materialise several hypertext documents from the same data source, where the query and postprocessing phase have been folded into a single step. The scripting language has been used to construct a parser for the Hypertext Transfer Protocol (HTTP) which runs inside the database server process, which means that the database server appears as a web server to the outside world.

The creation of the TENTACLE system has been a large undertaking — it covered the design, implementation and example deployment of an alternative database system. In other words, the project has covered a breadth of topics, as opposed to being an in-depth study of a single one. This means that there exist numerous opportunities to explore aspects of the system further. A selection of these is given below:

- The TENTACLE data model might be extended to include facilities for encoding constraints or schema information, even if this information is not well specified. The LORE system attempts to encode such schema information using structures which the authors call *Data Guides*, and it might be interesting to equip the TENTACLE system with similar capabilities.

- The TENTACLE query language currently does not allow the user to specify in which order path expressions are matched against the database graph — at the moment the only supported (implicit) evaluation strategy is a breadth first traversal where the attributes of a node are examined in their sorted order. It would be interesting to add other evaluation strategies such as a depth first traversal, or even parameterized evaluation strategies which would allow the user to specify cost and sorting functions. Such an extension could even form the basis for addressing some of the limitations of network/graph based systems identified by [24] which motivated the introduction of relational databases.

- It might be interesting to employ path expressions to rewrite the database graph. The path expressions presented in this dissertation have been read-only in the sense that they have not modified the database graph — however, it should be possible to include functions which *do* modify the database graph as segments of path expressions. This would mean that the side effects of traversing the database graph would change the graph — the potential for making such a process recursive

In addition to improving and extending the existing system, there exist opportunities to apply the TENTACLE database system to a variety of other semi-structured domains. Of particular interest are structured texts such as XML or programming languages where path expressions may be a suitable tool to query the parse tree of a document or program. Similarly VRML scene graphs might be queried using TENTACLE path expressions.

Yet another potentially interesting topic would be a comparison of the TENTACLE database system with persistent programming languages (See [23] for a survey). Viewed from such a perspective the TENTACLE system is a persistent programming language which uses a graph as its bulk storage system, possesses builtin query facilities and runs inside a database.

Further application of TENTACLE could be as a graph storage system for other graph-based query languages. Examples include Hy^+ [25], which is implemented on a deductive database system, and Hyperlog [53], which used to be implemented on a functional database system.

Appendix A

Language Syntax

```
SCRIPT       → STATEMENTS '.'
STATEMENTS   → STATEMENT
             → STATEMENT STATEMENTS
STATEMENT    → ASSIGNMENT
             → LOOP
             → CONDITIONAL
             → DECLARATION
             → BLOCK
             → RVALUE
ASSIGNMENT   → LVALUE '=' RVALUE
LOOP         → while '(' RVALUE ')' STATEMENT
CONDITIONAL→ if '(' RVALUE ')' STATEMENT
             → if '(' RVALUE ')' STATEMENT else STATEMENT
DECLARATION→ var VARIABLES
VARIABLES    → variable
             → variable ',' VARIABLES
BLOCK        → '{' STATEMENTS '}'
LVALUE       → variable
RVALUE       → literal
             → variable
             → FUNCTION
             → '[' PATH ']'
FUNCTION     → literal '(' ')'
             → literal '(' ARGUMENTS ')'
ARGUMENTS    → RVALUE
             → RVALUE ',' ARGUMENTS
PATH         → RVALUE
             → RVALUE '.'  PATH
```

$\rightarrow$ '[' PATH ']' '*'
$\rightarrow$ '[' PATH ']'
$\rightarrow$ PATH '&' PATH
$\rightarrow$ PATH '|' PATH

LEGEND: Uppercase strings denote nonterminals, lowercase strings or singly quoted characters denote terminals.

Appendix B

Path Expression Semantics

The semantics of the TENTACLE path expressions will be explained by annotating the productions of the TENTACLE grammar with logic rules.

For this purpose it is useful to draw a distinction between the first segment of a path expression and other segments. This distinction is necessary since the initial segment has a different semantics — it specifies the set of graph components which serve as starting points, while subsequent segments are used in matching operations. This bears some resemblance to path expressions as encountered in object-orientated programming languages — for example the first segment of expression `ship.hold[2]` selects a starting point from the set of all available objects, while the second segment can be thought of as matching a neighbouring entity (selecting only from the immediate neighbours of the previous segment).

The TENTACLE grammar which draws the distinction between initial and subsequent segments is given below:

```
PATH → initial_segment
     → initial_segment '.'  TAIL
     → '[' PATH ']' '*'
     → '[' PATH ']'
     → PATH '&' PATH
     → PATH '|' PATH

TAIL → segment
     → segment '.'  TAIL
     → '[' TAIL ']' '*'
     → '[' TAIL ']'
     → TAIL '&' TAIL
     → TAIL '|' TAIL
```

This grammar may be annotated as follows: A predicate is associated with each production — as the production is matched against a query, the predicate is evaluated. The first argument ($$) of such a predicate contains the path expression which still remains to be parsed. The second argument (I) contains a set of graph components which serve as starting point for that path expression, while the third argument (R) contains the graph components at which the path expression terminates — in other words the result.

Note that the annotation uses a YACC (Yet Another Compiler Compiler) notation to indicate the relationship between the production and the predicate. Briefly $$ denotes the head of the production while $n where $n \in [1, 2, 3...]$ denotes the n'th token in the body of the production. Observe that the $$ and $n parameters serve as the equivalent of distinguishing subscripts.

```
Production:  PATH ::= initial_segment
Predicate:   path($$,I,I) :- initial($1,I)

Production:  PATH ::= initial_segment '.'  TAIL
Predicate:   path($$,I,R) :- initial($1,I), tail($3,I,R)

Production:  PATH ::= '[' PATH ']' '*'
Predicate:   path($$,I,R) :- path($2,I,X), tail($$,X,Y),
                             union(X,Y,R)

Production:  PATH ::= '[' PATH ']'
Predicate:   path($$,I,R) :- path($2,I,R)

Production:  PATH ::= PATH '&' PATH
Predicate:   path($$,I,R) :- path($1,I,X), path($3,I,Y),
                             intersection(X,Y,R)

Production:  PATH ::= PATH '|' PATH
Predicate:   path($$,I,R) :- path($1,I,X), path($3,I,Y),
                             union(X,Y,R)

Production:  TAIL ::= segment
Predicate:   tail($$,I,R) :- extend($1,I,R)

Production:  TAIL ::= segment '.'  TAIL
Predicate:   tail($$,I,R) :- extend($1,I,X), tail($3,X,R)
```

```
Production:  TAIL ::= '[' TAIL ']' '*'
Predicate:   tail($$,I,R) :- tail($2,I,R), union(I,R,I)
             tail($$,I,R) :- tail($2,I,X), tail($$,X,Y),
                             union(X,Y,Z), union(I,Z,R)
             tail($$,I,[])

Production:  TAIL ::= '[' TAIL ']'
Predicate:   tail($$,I,R) :- tail($2,I,R)

Production:  TAIL ::= TAIL '&' TAIL
Predicate:   tail($$,I,R) :- tail($1,I,X), tail($3,I,Y),
                             intersection(X,Y,R)

Production:  TAIL ::= TAIL '|' TAIL
Predicate:   tail($$,I,R) :- tail($1,I,X), tail($3,I,Y),
                             union(X,Y,R)
```

The above annotations make use of the following predicates:

```
union/3
difference/3
initial/2
extend/3
```

The predicates `union/3` and `intersection/3` have their conventional semantics, where the third argument is the union, respectively intersection, of the first two arguments. In this context the arguments to these two rules are sets of graph components.

`initial/2` is a predicate which, given an initial segment (first argument), computes the corresponding set of graph components (second argument). A segment is an R-value in the TENTACLE language (see Appendix A) — thus `initial/2` evaluates an R-value and returns its result as a set of graph components[1]. When compared to an object-orientated programming language, `initial/2` performs a task similar to the resolution of a variable label to an object address.

`extend/3` takes a set of graph components (second argument) and returns their immediate neighbours (third argument) which match a particular constraint encoded in the given segment (first argument).

[1] If the initial segment is the empty string, then `initial/2` returns the default entry point into the graph (the database root).

In order to supply a more detailed definition of the extend/3 predicate it is useful to encode the database graph as an existensional relation, and provide a suitable representation for sets of graph components.

For this purpose the database graph may be thought of as consisting of the following extensional relation[2]:

```
e(n,k,v)
```

where $n \in N$, $k \in M$ and $v \in N \cup M$ (refer to Chap. 3 for the definitions of N and M). In other words the relation e() contains a tuple for every attribute of every node, where n is the identifier of the node, k is the key of the attribute and v is the value of the attribute, either an atomic value ($k \in M$) or a reference to another node ($k \in N$). Note that the pair (n,k) is the primary key in this relation, since each n uniquely identifies a node, whilst k identifies an attribute within the scope of its node n[3].

A set of graph components may be represented as a list of sublists, where each sublist is a reference to a graph component. A reference to a node is denoted by single element sublist [n], a reference to an attribute key by a sublist of two elements [n,k] and a reference to an attribute value by the triple [n,k,v].

Given these two representations (of encoding a graph as the e() relation, and a graph component set as a list of sublists) it is possible to define extend/3 as follows:

```
extend(_,[],[])
extend(S,[HI|TI],TR) :- singlextend(S,HI,HR),
                        extend(S,TI,TR), member(HR,TR)
extend(S,[HI|TI],[HR|TR]) :- singlextend(S,HI,HR),
                             extend(S,TI,TR)

singlextend(S,[N],[N,K]) :- e(N,K,_), match(K,S)
singlextend(S,[N,K],[N,K,V]) :- e(N,K,V), match(V,S)
singlextend(S,[N,K,V],[V,L]) :- e(N,K,V), e(V,L,_),
                                match(L,S)
```

[2]Note that the translation does not take isolated nodes into consideration. Fortunately isolated nodes (nodes which possess no outgoing or incoming edges) are only of interest in trivial path expressions, namely those consisting of a single segment referring to the isolated node. Also note that the TENTACLE system maintains a sorted order on the keys of a node, thus e(n,k,v) should be sorted on (n,k), and the rule evaluation should be order preserving.

[3]This constraint has been introduced to simplify the semantics — the implementation itself permits duplicate node keys.

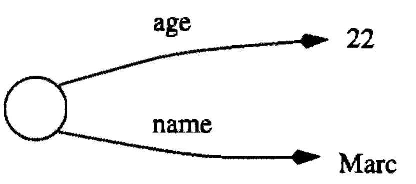

Figure B.1: Simple example database graph

Essentially the `extend/3` predicate determines the set of neighbours of each input set element and returns those which match the segment. Observe that the immediate neigbours of a node reference `[n]` are its attribute keys `[n,k]`, while the neighbour of an attribute key is its value `[n,k,v]`. If an attribute value is a node reference then its neighbours are the attribute keys of the referenced node.

`member/2` has its usual meaning of testing if the first argument is a member of the list given in second argument, while `match/2`, like `initial/2` provides an interface to the scripting component of the language — `match/2` succeeds if the graph component (first argument) matches the current segment (second argument), where the segment is an R-value of the language. A literal or variable is deemed to match if its value is the same as the graph component, while a function call matches if the function does not return false. If the R-value is the empty string, then `match/2` succeeds for any graph component.

The following small example illustrates how a TENTACLE query may be mapped to a logic program using the above method. Consider the simple path expression:

```
me.age.equal(here(),sum(14,8))
```

where `me` is a variable referring to the node in the database graph given in Fig. B.1. Assuming that this node has an identifier of `1`, the graph may be represented as the relation:

```
e(1,age,22)
e(1,name,Marc)
```

The evaluation of the above path expression starts when the first production `PATH ::= initial_segment '.' TAIL` is matched. The associated rule is:

```
path("me.age.equal(here(),sum(14,8))",I,R) :-
    initial("me",I), tail("age.equal(here(),sum(14,8))",I,R)
```

`initial/2` evaluates the R-value `"me"` which returns a set containing a reference to a single node `[[1]]`:

```
path("me.age.equal(here(),sum(14,8))",[[1]],R) :-
     initial("me",[[1]]),
     tail("age.equal(here(),sum(14,8))",[[1]],R)
```

The first argument to `tail/3` matches the production `TAIL ::= segment '.' TAIL` and its associated rule:

```
tail("age.equal(here(),sum(14,8))",[[1]],R) :-
     extend("age",[[1]],X), tail("equal(here(),sum(14,8))",X,R)
```

`extend/3` finds the immediate neighbour of `[1]` which matches the segment `"age"`. `"age"` is a literal, thus `match/2` compares its value against the keys of `[1]` and returns those which are the same:

```
extend("age",[[1]|[]],[HR|[]]) :-
       singlextend("age",[1],HR), extend("age",[],[])

singlextend("age",[1],[1,K]) :-
       e(1,K,_),match(K,"age")
```

The matching key value is returned to the callee:

```
tail("age.equal(here(),sum(14,8))",[[1]],R) :-
     extend("age",[[1]],[[1,age]]),
     tail("equal(here(),sum(14,8))",[[1,age]],R)
```

The production which matches the remainder of the expression is `TAIL ::= segment` and its associated rule is given below:

```
tail("equal(here(),sum(14,8))",[[1,age]],R) :-
     extend("equal(here(),sum(14,8))",[[1,age]],R)
```

`extend/3` finds the immediate neighbour of `[1,age]` which matches the segment `"equal(here(),sum(14,8))"`. `match/2` evaluates this segment and returns true (`here()` retrieves the value of the graph component to be matched, in this case 22):

```
extend("equal(here(),sum(14,8))",[[1,age]|[]],[HR|[]]) :-
      singlextend("equal(here(),sum(14,8))",[1,age],HR),
      extend("equal(here(),sum(14,8))",[],[])

singlextend("equal(here(),sum(14,8)",[1,age],[1,age,V]) :-
      e(1,age,V), match(V,"equal(here(),sum(14,8)")
```

Finally the nested calls unwind to the top level where the result set contains a single reference to the node attribute value [1,age,22]:

```
tail("equal(here(),sum(14,8))",[[1,age]],[[1,age,22]]) :-
     extend("equal(here(),sum(14,8)",[[1,age]],[[1,age,22]])

tail("age.equal(here(),sum(14,8))",[[1]],[[1,age,22]]) :-
     extend("age",[[1]],[[1,age]]),
     tail("equal(here(),sum(14,8))",[[1,age]],[[1,age,22]])

path("me.age.equal(here(),sum(14,8))",[[1]],[[1,age,22]]) :-
     initial("me",[[1]]),
     tail("age.equal(here(),sum(14,8))",[[1]],[[1,age,22]])
```

Appendix C

Elementary System Performance Test

This appendix presents the results of an elementary performance test of the database server. The test serves more as an example (of a number of robustness and performance tests undertaken during the implementation of the system) than a reliable benchmark. The inclusion of this example in the dissertation is intended to show that the implementation is more substantial than a toy prototype.

C.1 Description

The test consists of creating a single node as database root and adding a thousand attributes to this node. After this large node has been set up, the database server is stopped and restarted (in order to remove the effects of any caching performed by the server) and all the attributes are requested from the server using the following query:

```
[.write(connection,"\"",here(),"\":",[here().],"\n")]
```

This query returns a list containing elements of the form:

```
"attribute key":attribute value
```

The test input consists of a thousand words selected randomly from the system word list (`/usr/dict/words`), which on the test system consists of 45402 words. Each selected word serves as an attribute key, while the attribute value (of lesser importance in this test) simply stores the sample number of the selected word (ie a number in the range $1 - 1000$).

For example, given the random list of words `contingent`, `Britannica`, `contingencies`, `entire`, `disabler`, `liquid`, `hey`, `levers`, `fraternal`, `phenomenological`, the output of the above query would take the following form:

```
"Britannica":00002
"contingencies":00003
"contingent":00001
"disabler":00005
"entire":00004
"fraternal":00009
"hey":00007
"levers":00008
"liquid":00006
"phenomenological":00010
```

The TENTACLE storage subsystem arranges node attributes as a list of blocks where the attributes on each block are sorted on attribute key and packed contiguously. In addition a separate index structure (a modified Patricia Tree, see Chap. 5) is maintained for the node attribute keys. In other words the system contains provisions for both random and sequential access.

This test thus exercises the system component which re-arranges contiguous attributes packed into a block as well as the module which updates the index in response to such a re-arrangement. Other components exercised include the language parser, query evaluation module and network interface.

C.2 Platform

The test platform is a personal computer, dating from 1997 and having the following specifications:

- Cyrix Pentium clone (120 MHz)
- 512k secondary cache
- 64M RAM

The database server was compiled using the GNU C Compiler (2.7.2.1) with neither debugging nor optimisations enabled, the executable (unstripped

ELF format) was linked dynamically against libc.so.5.3.12 and was run under Linux (2.0.30). Both the server and client were run on the same (otherwise lightly-loaded) system, communicating via TCP/IP over the loopback network device. The database content was stored as a buffered file with asynchronous writes.

C.3 Data Recorded

The data which was recorded for each insertion and retrieval was the user and system times for the server process, as well as the elapsed time for the client process — the user and system times of the server may be viewed as the cost (ie a count of the number of instructions required) to perform the task[1], while the elapsed time recorded for the client is the metric of greatest subjective interest to the user (the wall time needed for the results to be returned). Data was collected for ten iterations of the test script given at the end of this appendix.

C.4 Results

	Average	Variance
User Time of Server (U)	1.732	0.0079
System Time of Server (S)	0.860	0.0318
Total CPU time of Server ($U+S$)	2.592	0.0214
Elapsed Time of Client (E)	2.581	0.0209

Figure C.1: Insertion times for 1000 attributes

	Average	Variance
User Time of Server (U)	0.486	0.0027
System Time of Server (S)	0.852	0.0014
Total CPU time of Server ($U+S$)	1.338	0.0007
Elapsed Time of Client (E)	1.321	0.0008

Figure C.2: Retrieval times for 1000 attributes

All results are in seconds. User time (U) is the time a processes spends runs as user privilege, system time (S) the time spent by the kernel servicing

[1]These measurements include the cost of starting and stopping the server.

the process and elapsed time (E) is the wall time which passed while the process had been running.

C.5 Discussion

The variance of the insertion test is larger than that of the retrieval test. This is to be expected: For insertions the input data sets are generated randomly — some input sets would be ordered in ways which require fewer re-arrangements than others. On the other hand retrievals operate on already sorted and indexed database content, thus the variance between retrievals is smaller.

The fact that the total CPU time of the server process exceeds the elapsed time recorded for the client may be attributed to the cost of starting and shutting down the server, as well as to the cost of deallocating resources set aside for a client after its completion. That this is visible at all confirms that client consumes only minimal system resources and that no other tasks were active while running the tests.

A counter-intuitive result is that the variance of the combined user and system times ($U + S$) is less than their individual variances (U, S). Interpreting this result as a negative correlation between user and system times is obviously not feasible; instead this result could possibly be attributed to a limited resolution of the system profiling utility (`time`) which might not be able to establish the exact time of transition between user and kernel execution modes.

C.6 Conclusion

The test shows that the system is capable of inserting $1000/2.592 = 385$ node attributes per second and retrieving $1000/1.338 = 747$ node attributes per second. While the test did not take the effects of multiuser accesses, disk bottlenecks or storage space fragmentation into account, it nevertheless showed that the system performs reasonably well, even on a modest platform.

C.7 Shell Script Test Harness

```
#!/bin/sh
# n: number of iterations
n=1000
# p: port number to use
```

```
p=9101
# m: number of words available
m=`wc -l < /usr/dict/words | tr -d \ `
# truncate files
> words-random.txt
> words-output.txt

# generate random word list
echo "Selecting $n random words from $m in system word list"
while [ "$n" -gt "0" ] ; do
  q=`printf "%05d" $n`
  sed -ne "$[RANDOM%m]s/\\(..*\\)/\"\1\":$q/p" \
    /usr/dict/words >> words-random.txt
  n=$[n-1]
done

# transform word list into insertion commands
echo -n "Preparing words for insertion... "
echo "var n n=newid() " > commands.txt
sed -ne 's/\(..*\):\(..*\)/link(n,\1,\2)/p' \
  < words-random.txt >> commands.txt
echo "root(global,n) write(connection,\"ok\n\")." \
  >> commands.txt
echo "ok"

# start the server: tentacle database daemon (tdbd)
echo "Starting database server on port $p... "
time -f "%U+%S" -a -o time-server-insert.txt \
  ../../bin/tdbd -N -p$p >& default.log &
# wait for the server to boot up
sleep 2

# send insertion requests to tentacle database
echo -n "Inserting words... "
time -f "%e" -a -o time-client-insert.txt \
  ../../bin/tdbsend -q -p$p commands.txt
if [ "$?" != "0" ] ; then
  echo "failed"
  echo "Consult default.log in `pwd` for diagnostics"
  exit 1
fi
```

```
# shut down server after insertion
echo -n "Shutting down database server... "
../../bin/tdbsend -q -p$p \
  "shutdown() write(connection,\"ok\\n\")."
if [ "$?" != "0" ] ; then
  echo "failed"
  echo "Consult default.log in `pwd` for diagnostics"
  exit 1
fi

# restart server for retrieval
echo "Starting database server on port $[p+1]... "
time -f "%U+%S" -a -o time-server-extract.txt \
  ../../bin/tdbd -p$[p+1] >& default.log &
# wait for the server to boot up
sleep 2

echo -n "Extracting words... "
time  -f "%e" -a -o time-client-extract.txt \
  ../../bin/tdbsend -q -p$[p+1] -o words-output.txt \
  "[.write(connection,\"\\\"\",here(),\"\\\":\",[here().],\"\n\")]."
if [ "$?" != "0" ] ; then
  echo "failed"
  echo "Consult default.log in `pwd` for diagnostics"
  exit 1
else
  echo "ok"
fi

echo -n "Shutting down database server... "
../../bin/tdbsend -q -p$[p+1] \
  "shutdown() write(connection,\"ok\\n\")."
if [ "$?" != "0" ] ; then
  echo "failed"
  echo "Consult default.log in `pwd` for diagnostics"
  exit 1
fi

# Final paranoia check
echo -n "Comparing before and after... "
```

```
sort words-random.txt > words-sorted.txt
diff -q words-output.txt words-sorted.txt
if [ "$?" = "0" ] ;  then
  echo "success - word lists are identical"
else
  echo "ouch - word lists not identical"
  exit 1
fi
```

Bibliography

[1] Altavista web search engine. http://www.altavista.digital.com/.

[2] Apache web server consortium. http://www.apache.org/.

[3] Common gateway interface (CGI) specifications. http://hoohoo.ncsa.uiuc.edu/cgi/.

[4] Fast common gateway interface web site. http://www.fastcgi.com/.

[5] IMS. http://www.software.ibm.com/data/ims/.

[6] Informix. http://www.informix.com/.

[7] Internet movie database. http://www.imdb.com/.

[8] Netcraft web server survey. http://www.netcraft.co.uk/survey/.

[9] Oracle. http://www.oracle.com/.

[10] Personal home pages group. http://www.php.net/.

[11] Yahoo web index. http://www.yahoo.com/.

[12] Forum on risks to the public in computers and related systems. http://catless.ncl.ac.uk/Risks/20.01.html, October 1998.

[13] S. Abiteboul. Querying semi-structured data. In *International Conference on Database Theory*, pages 1–18, January 1997.

[14] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The LOREL query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.

[15] P. Atzeni, G. Mecca, and P. Merialdo. Semistructured and structured data in the web: Going back and forth. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 144–153, May 1997.

[16] T. Berners-Lee and D. Connolly. Hypertext markup language specification version 2.0. http://ds.internic.net/rfc/rfc1866.txt, November 1995.

[17] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol HTTP/1.0. http://ds.internic.net/rfc/rfc1945.txt, May 1996.

[18] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform resource locators (url). http://ds.internic.net/rfc/rfc1738.txt, December 1994.

[19] P. Buneman. Semistructured data. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 117–121, May 1997.

[20] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *ACM SIGMOD International Conference on Management of Data*, pages 505–516, June 1996.

[21] R. G. G. Cattell, editor. *Communications of the ACM: Next Generation Database Systems*, pages 31–120. The Association for Computing Machinery, October 1991.

[22] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, 1994.

[23] S. M. Clamen. Data persistence in programming languages: A survey. Technical Report 155, School of Computer Science, Carnegie Mellon University, May 1991.

[24] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[25] M. P. Consens, F. C. Eigler, M. Z. Hasan, A. O. Mendelzon, E. G. Noik, A. G. Ryman, and D. Vista. Architecture and applications of the Hy^+ visualization system. *IBM Systems Journal*, 33(3):458–476, 1994.

[26] M. P. Consens and A. O. Mendelzon. GraphLog: a visual formalism for real life recursion. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 404–416, 1990.

[27] O. Deux et al. The story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, March 1990.

[28] R. Durbin and J. T. Mieg. A Caenorhabditis elegans database. ftp://ncbi.nlm.nih.gov/repository/acedb/ and http://probe.nalusda.gov:8000/acedocs/, 1991.

[29] G. C. Everest and M. S. Hanna. Survey of object-orientated database management systems. Technical report, Carlson School of Management University of Minnesota, January 1992.

[30] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for a web-site management system. *ACM SIGMOD Record*, 26(3):4–11, September 1997.

[31] T. Fiebig, J. Weiss, and G. Moerkotte. RAW: A relational algebra for the web. In *ACM SIGACT-SIGMOD-SIGART Workshop on Management of Semistructured Data*, May 1997.

[32] H.-W. Gellersen, R. Wicke, and G. Martin. WebComposition: An object-orientated support system for the web engineering lifecycle. *Computer Networks and ISDN Systems*, 29(8-13):1429–1437, September 1997.

[33] R. E. Griswold and M. T. Griswold. *The Icon Programming Language.* Peer-to-Peer Communications, 1996.

[34] R. H. Güting. GraphDB: A data model and query language for graphs in databases. Technical Report 155, FernUniversität Hagen, Feburary 1994.

[35] M. Gyssens, J. Paredaens, and D. Van Gucht. A graph-oriented object database model. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 417–424, 1990.

[36] W. Hall, H. Davis, and G. Hutchings. *Rethinking Hypermedia - The Microcosm Approach.* Kluwer Academic Publishers, 1996.

[37] B. W. Kernighan and D. M. Ritchie. *The C Programming Language: ANSI C.* Prentice Hall, 1988.

[38] S. N. Khoshafian and G. P. Copeland. Object identity. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 406–416, November 1986.

[39] N. Kiesel, A. Schürr, and B. Westfechtel. GRAS, a graph-oriented (software) engineering database system. *Information Systems*, 20(1):21–51, 1995.

[40] W. Kim. A model of queries for object-oriented databases. In *International Conference on Very Large Data Bases*, pages 423–432, August 1989.

[41] W. Kim, J. F. Garza, N. Ballou, and D. Woelk. Architecture of the ORION next-generation database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):109–124, March 1990.

[42] H. F. Korth and A. Silberschatz. *Database System Concepts*, chapter 3,4. McGraw-Hill, 1991.

[43] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. A declarative language for querying and restructuring the web. In *IEEE Workshop on Research Issues in Data Engineering*, pages 12–21, February 1996.

[44] G. Lausen and G. Vossen. *Models and Languages of Object-Oriented Databases*. Addison-Wesley, 1997.

[45] D. Maier and J. Stein. Development of an object-oriented DBMS. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 472–482. ACM SIGPLAN, September 1986.

[46] H. Maurer. *HyperWave: The Next Generation Web Solution*. Addison-Wesley Longman, 1996.

[47] H. Maurer, N. Scherbakov, and S. P. A new hypermedia data model. In *International Conference on Database and Expert Systems Applications*, pages 685–696, September 1993.

[48] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. LORE: A database management system for semistructured data. *ACM SIGMOD Record*, 26(3):54–66, September 1997.

[49] A. Mendelzon, G. Mihaila, and T. Milo. Querying the world wide web. *International Journal on Digital Libraries*, 1(1):54–67, April 1997.

[50] A. O. Mendelzon and C. G. Mendioroz. Graph clustering and caching. Technical report, Computer Systems Research Institute, University of Toronto.

[51] J. Postel. Internet protocol. http://ds.internic.net/rfc/rfc791.txt, September 1981.

[52] J. Postel. Transmission control protocol. http://ds.internic.net/rfc/rfc793.txt, September 1981.

[53] A. Poulovassilis and M. Levene. A nested-graph model for the representation and manipulation of complex objects. *ACM Transactions on Information Systems*, 12(1):35–68, January 1994.

[54] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. In *Deductive and Object-Oriented Databases*, pages 319–344, December 1995.

[55] A. Schürr. PROGRESS: A VHL-language based on graph grammers. In *International Workshop on Graph Grammars and Their Application to Computer Science*, March 1990.

[56] R. Sedgewick. *Algorithms in C++*, chapter 17, pages 245–257. Addison-Wesley, 1992.

[57] A. Silberschatz and S. Zdonik. Database systems — breaking out of the box. *ACM SIGMOD Record*, 26(3):36–50, September 1997.

[58] J. D. Ullman. *Principles of Database Systems*, chapter 5,6. Computer Science Press, 1983.

[59] M. Welz and P. T. Wood. Tentacle: A database system for the world wide web. In *International Conference on Database and Expert Systems Applications*, pages 658–667, August 1998.

[60] W. Yeong, T. Howes, and S. Kille. Lightweight directory access protocol. http://ds.internic.net/rfc/rfc1777.txt, March 1995.