Chapter 3. Research Method

In this chapter, we describe the research method and research model of this work. After related literature review in the previous section, we begin to develop the research model. The research structure is described in the next section. In this thesis, we adopt the prototyping method to illustrate the feasibility and validity of the research model.

3.1. Research Structure

In this research, the benchmark workload model is used to evaluate the performance of the heterogeneous information integration systems. The benchmark is run on several different systems, and the performance and price of each system is measured and recorded. The literatures studied in chapter 2 would help us to identify the important performance factors for XML and ontology processing. In chapter 3, we analyze the XML-specific and ontology-specific requirements in more details to justify the design of the benchmark. A generic benchmark model that is portable and scaleable is proposed in the thesis.

The research structure is shown in Figure 3.1. The benchmark study consists of two benchmark workload models, the XML benchmark workload model and the ontology benchmark workload model. Both of them consist of the data model and query model according to the generic constructs and constraints requirements. Next, the control model is created before the generic workload model to be generated and executed so as to measure and evaluate the systems.

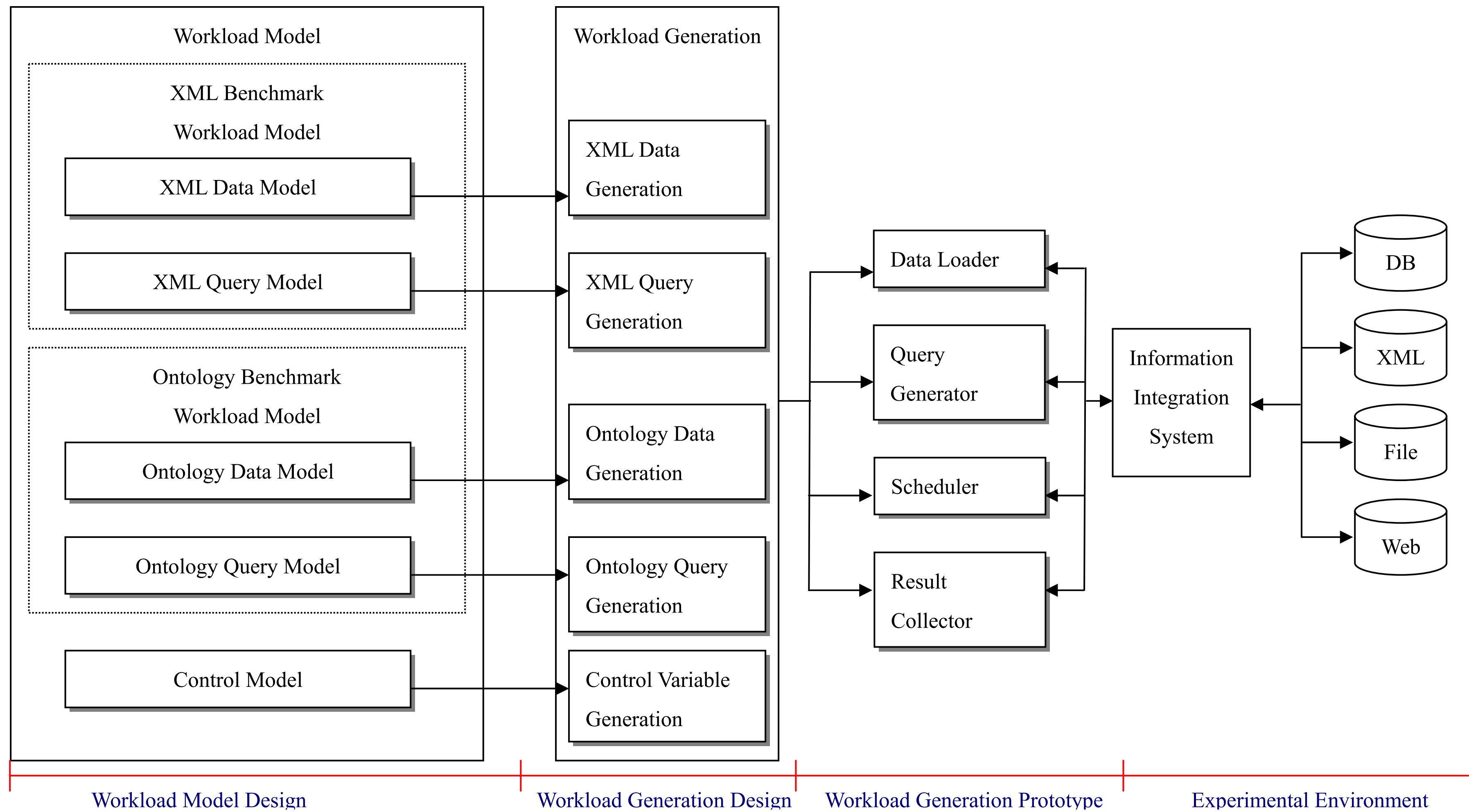

Figure 3.1: Research Structure

3.2. Research Model

In this research, we focus on heterogeneous information sources integrated in XML and ontology. The benchmark model we propose would capture most features of the released XML-based and ontology-based specifications. Designed as a generic benchmark model, it is easy to implement the benchmark on many different systems and architectures. To support scalability, this benchmark also provides different workloads.

Developing a benchmark requires the definition of the test workload model first. In this research, we provide a benchmark workload model that combines XML and ontology in heterogeneous information integration. In XML workload model, the data model describes a generic XML data model and the operation model defines a comprehensive set of test queries that covers the major aspects of XML query processing. In ontology workload model, the data model describes the major ontology component, and the operation model defines some important criteria to query the ontology. The control model defines the variables that used to set up the benchmark environment. The data model, operation model, and control model define the experimental factors of the benchmark. In addition, we should define the performance metrics to measure the benchmark results.

3.3. XML Data Model

XML is a hierarchical data format for information exchange on the Web. An XML document consists of nested elements that contain data or other elements. The boundaries of these elements are either delimited by start-tags and end-tags, or, for empty elements, by empty-element tags. The text between start-tags and end-tags is

the content of the element. Each element has a type, identified by name, sometimes called its "generic identifier" (GI), and may have a set of attribute specifications. Each attribute specification has a name and a value (Bray, Paoli, Sperberg-McQueen, & Maler, 2000). XML documents may comply with a Document Type Definition (DTD) or a XML Schema. DTD has traditionally been the most common method for describing the structure of XML document. But DTD lacks enough expressive power to properly describe highly structured data. XML Schemas are an XML language for describing and constraining the content of XML documents. It provides a richer and more powerful means for defining the data. Therefore, XML schema becomes the most common method for defining and validating highly structured XML documents rapidly.

We employ the XQuery 1.0 and XPath 2.0 Data Model published by the W3C to represent XML documents (Fernández, Malhotra, Marsh, Nagy, & Walsh, 2003). In the XQuery 1.0 and XPath 2.0 Data Model, XML documents are modeled as an ordered tree. The tree contains seven distinct kinds of nodes: document, element, attribute, text, namespace, processing instruction, and comment. In this research, for simplicity, we only consider document, element, attribute, and text nodes. The data model is a node-labeled, directed graph, in which each node has a unique identity. Document order is defined for all the nodes in the document and corresponds to the order in which the first character of each node occurs in the XML document. We briefly introduce the four nodes as follows:

- **Document nodes:** The document node is a virtual node pointing to the root element of an XML document. The document element in a XML document is a child of the document node.
- **Element nodes:** Every element in the document is an element node. Element nodes have zero or more children that can be element nodes or text nodes.

- **Attribute nodes:** Each element node has an associated set of attribute nodes. Note that the element node that owns this attribute is called its “parent” even though an attribute node is not a “child” of its parent element. An attribute node has an attribute name and an attribute value. Attribute nodes have no child nodes. If more than one attribute of an element node exists, the document order among the attributes is not distinguished. This is because there is no order among XML attributes.
- **Text nodes:** A text node must have only one parent and have no child nodes. A text node cannot contain an empty string as its content.

A graphical representation of the data model can be shown as Figure 3.2. Document order in this representation can be found by following the traditional in-order, left-to-right, depth-first traversal. The value D1 represents a document node; the values E1, E2, etc. represent element nodes; the values A1, A2, etc. represent attribute nodes; the values T1, T2, etc. represent text nodes. The IDREF attribute nodes are used for intra-document references (Fernández et al., 2003; YoshiKawa & Amagasa, 2001; Jiang, Lu, Wang, & Yu, 2002).

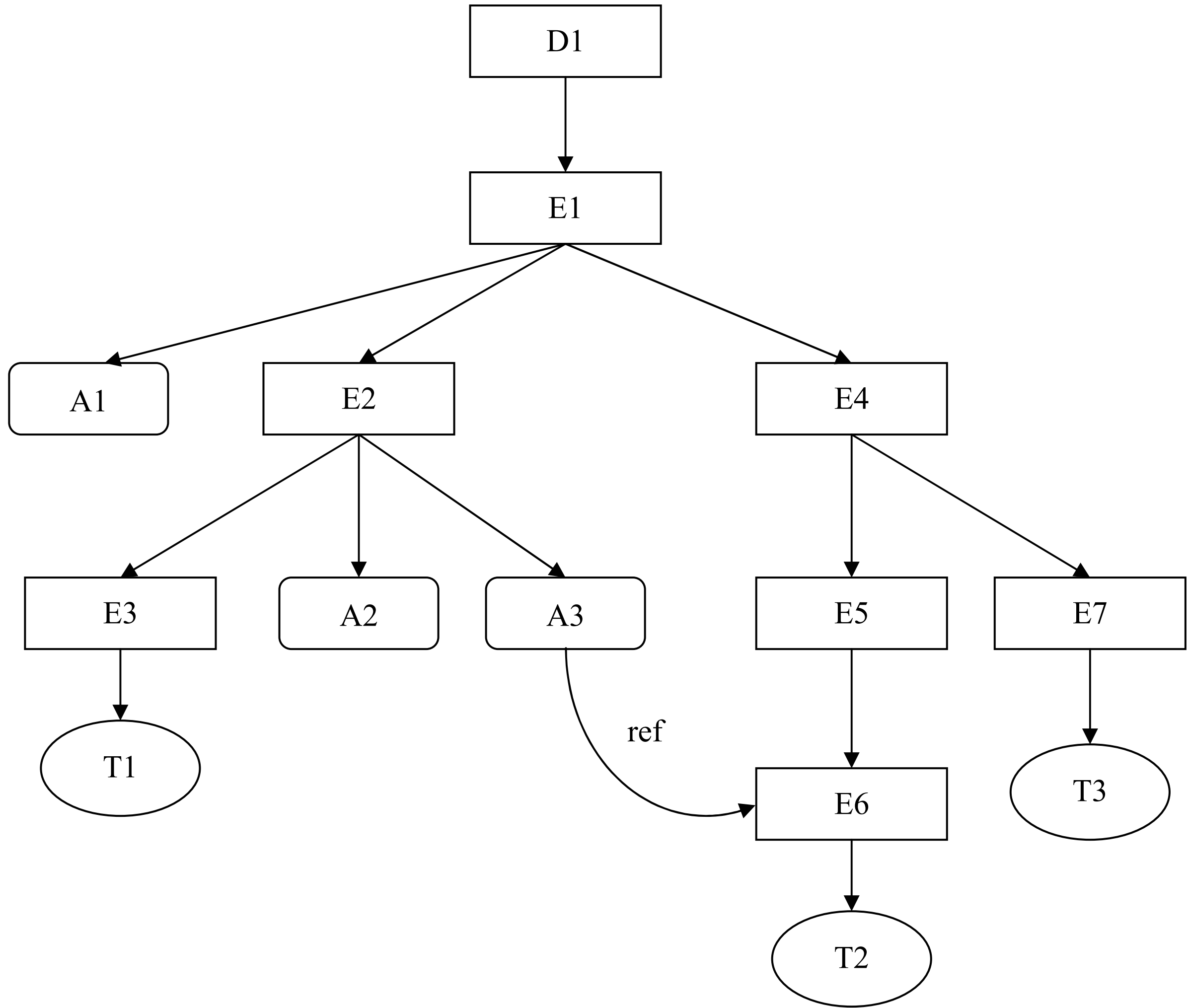

Figure 3.2: Graphic Representation of XML Data Model

3.4. XML Query Model

In this research, we attempt to propose a generic XML query model applicable to any scenario. We do not describe the queries based on a specific application domain such as an auction site or a library. The queries are specified in generic terms. It is easy for user to apply them in different scenarios. Besides, we further identify key factors that influence the complexity of each query. This would help users to evaluate performance of the system with increasing complex queries.

After analysis previous three XML benchmarks, we identify a comprehensive set of queries. The query model we defined can be classified into 10 categories, including 14 different queries. Each of them challenges different aspects of XML processing. Besides, users can specify queries according to their requirements, called “user-driven query”. Figure 3.3 shows the XML query model. The following will describe each category briefly, and express each query in generic terms. In each query, the generic term is written in italics. Then we illustrate them in XQuery. We use E1, E2 etc. to denote a certain element, and A1, A2 etc. to denote a certain attribute. The number of them does not indicate their order in a XML document, just for representing convenience. Finally, the complexity factors will be discussed.

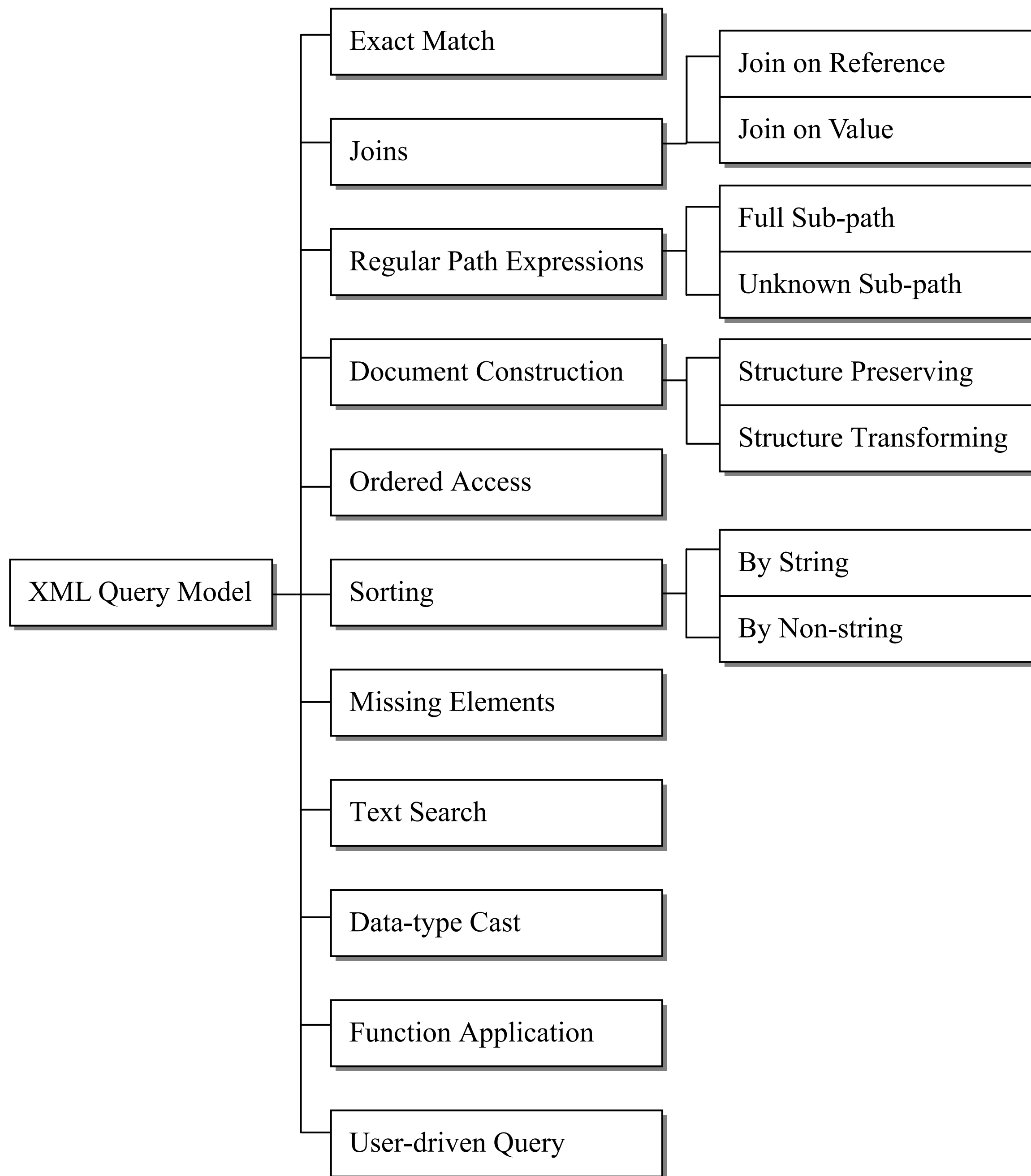

Figure 3.3: XML Query Model

3.4.1. Exact Match

This type of queries specifies a full path expression. One main concept of XQuery is the use of path expressions for selecting nodes. The length of the path

expression depends on the levels of predicates being queried in XML documents. This is the simplest query type. We can use this type of queries to establish a simple "metric" comparing performance of the following queries. It tests the database ability to handle simple string lookups with a fully specified path.

Generic terms:

Given a full path expression, find elements E1 that have an attribute A1 in a certain value X.

XQuery expression:

```
FOR     $a IN input()/SUBPATH/E1[@A1 = "X"]
RETURN  $a
```

The complexity of the query is influenced by the length of the path expression. Queries with different level of predicate would have different performance.

3.4.2. Joins

References are an integral part of XML identifying the relationship between related data. With using of reference, richer relationships can be represented than just hierarchical element structures. The system must be able to combine separate information together using joins. Horizontal traversals are defined in this type of queries. Joins can be on the basis of references and values. References are specified in the DTD and may be optimized with logical OIDs for example. The system should make use of the cardinalities of the sets to be joined. Joins based on values test the database's ability to handle large (intermediate) results.

- **Join on Reference**

Generic terms:

Find element E1 by the reference attribute A1 of E2. The reference attribute A1 of E2 refer to E1.

XQuery expression:

```
FOR      $a IN input()//E1
          $b IN input()//E2
WHERE   $a/@A2 = $b/@A1
RETURN  $a
```

- **Join on Value**

Generic terms:

This time reference is based on join of the data values. Find element E1 whose attribute A1 is equal to the attribute A2 of E2.

XQuery expression:

```
FOR      $a IN input()//E1
          $b IN input()//E2
WHERE   $a/@A1 = $b/@A2
RETURN  $a
```

The queries specified above are 2-way join. It is the simplest form. 3-way join, 4-way join, and N-way join would be generated with increasing complexity. In addition, the result size would affect the query efficiency too.

3.4.3. Regular Path Expressions

Regular path expressions are a basic building block of almost every XML language including XPath, XQuery, and XSLT. The system should be capable of optimizing path expressions and reducing traversals of irrelevant parts of the tree. We often use wildcards in regular path expressions and the system should realize that it is not necessary to traverse the complete document tree to execute such expressions. This type of queries tries to quantify the costs of long path traversals that do not include wildcards, and the costs of path traversals that include wildcards.

- **Full Sub-path**

Generic terms:

Find element E1 with a long path expression.

XQuery expression:

```
FOR     $a IN input()/SUBPATH/E1
RETURN  $a
```

- **Unknown Sub-path**

Generic terms:

Find element E1 with a regular path expression include wildcards.

XQuery expression:

```
FOR     $a IN input()//E1
RETURN  $a
```

The length of path expression would influence the complexity. In a path

expression, each step can apply one or more predicates to eliminate nodes that fail to satisfy a given condition. Therefore, numbers of element unknown in the sub-path would also affect the query complication.

3.4.4. Document Construction

Structure is very important to XML documents. But XML documents storing in relational DBMSs often need to be broken down. Reconstructing the original document is a big challenge to systems. We might retrieve fragments of original documents with original structures. But sometimes we may want to construct document fragments with new structures. These queries tests for the ability of the system to reconstruct portions of the original XML document.

- **Structure Preserving**

Generic terms:

Return a XML document constructed by element E1 and its sub-element E2. Retrieve E2 of E1 that has an attribute A1 equal to a certain value X.

XQuery expression:

```
FOR     $a IN input()//E1[@A1 = X]
RETURN  <$a> $a/E2 </$a>
```

- **Structure Transforming**

Generic terms:

Construct a new XML document. Find element E1 with an attribute A1 equal to a certain value X, and select several sub-element of E1 to construct a new XML

document.

XQuery expression:

```
FOR     $a IN input()//E1[@A1 = X]
RETURN  <output>
            {$a/E2/E3}
            {$a/E2/E4}
            {$a/E2/E3/E5}
            {$a/E6}
         </output>
```

The complication of the XML document structure would increase the difficulty of reconstruction. On the other hand, the structure of output document would also influence the query complexity.

3.4.5. Ordered Access

Order of elements is important in XML documents. Because documents will sometimes be fragmented when they are stored on disk, it is important that the order of these fragments in the original document is preserved. The system should be able to preserve these intrinsic orders. This type of queries attempts to test how efficient the system handle queries with order constraints.

Generic terms:

Find element E1 with attribute A1 in certain value X, and return the first sub-element E2 of E1.

XQuery expression:

```
FOR     $a IN input()//E1[@A1 = X]
RETURN  $a/E2[1]
```

The complexity depends on order constraints specified in the query. If there is an index build on the attribute, the query can take advantage of set-valued aggregates on the index attribute to accelerate the execution.

3.4.6. Sorting

The order by clause is the only facility provided by XQuery for specifying an order other than document order. In XML documents, the generic data type of element content is string, but users may cast the string type to other types. Therefore, the system should be able to sort values both in string and in non-string data types. This type of queries tests whether the system can do sorting efficiently.

- **By String**

Generic terms:

List sub-element E3 of element E1 sorted by sub-element E2.

XQuery expression:

```
FOR       $a IN input()//E1
ORDER BY  $a//E2
RETURN    $a/E3
```

- **By Non-string**

Generic terms:

List sub-element E3 of element E1 sorted by sub-element E2. This time E2 is a non-string value.

XQuery expression:

```
FOR       $a IN input()//E1
ORDER BY  $a//E2
RETURN   $a/E3
```

The number of tuples that are generated by the FOR and LET-clauses and satisfies the condition in the WHERE-clause would influence the complexity of the query. Also, if the ORDER BY-clause uses several options, the complexity would increase.

3.4.7. Missing Elements

In XML, schemas are more flexible and may have a number of irregularities. Queries in this type are to test how well the system knows to deal with the semi-structured aspect of XML data, especially elements that are declared optional in the schemas.

Generic terms:

Find element E1 whose sub-element E2 has NULL value.

XQuery expression:

```
FOR     $a IN input()//E1
```

```
WHERE   EMPTY($a/E2/text())
RETURN  $a
```

The complexity depends on the FOR and LET-clauses that generate the test tuples.

3.4.8. Text Search

Text search plays a very important part in XML document systems. This type of queries conducts a full-text search in the form of keyword search. They will challenge the textual nature of XML documents.

Generic terms:

Find element E1 whose sub-element E2 contains a specific text Y.

XQuery expression:

```
FOR     $a IN input()//E1
WHERE CONTAINS ($a/E2, "Y")
RETURN  $a
```

This query has to scan large part of the document. Therefore, the number of tuples that are generated by the FOR and LET-clauses would influence the query complexity. Also, if the query contains multiple texts, the difficulty would increase.

3.4.9. Data-type Cast

Strings are the generic data type in XML documents. But we often need to cast strings to another data type that carries more semantics. These queries challenge the system's ability to transform between data types.

Generic terms:

Find element E1 with a constraint that contain operations need to transform data value of sub-element E2 to other data-type. Retrieve element E1 whose sub-element E2 is bigger than a certain number X.

XQuery expression:

```
FOR     $a IN input()//E1
WHERE   $a/E2 > X
RETURN  $a
```

The number of tuples needs to be transformed affects the execution efficiency. If there are several casting conditions in a query, the complexity would increase.

3.4.10. Function Application

The following query challenges the system with aggregate functions such as count, avg, max, min and sum.

Generic terms:

Group element E1 by sub-element E2, and calculate the total number of elements for

each group.

XQuery expression:

```
FOR     $a IN DISTINT-VALUES (input()//E1/E2)
LET     $b := input()//E1[E2 = $a]
RETURN  count($b)
```

The complexity of this query is influenced by the number of tuples that are generated by the FOR and LET-clauses.

Table 3.1 summarizes the complexity factors of each query type.

Table 3.1: List of Complexity Factors

Complexity	Low	High
Exact Match	• Shallow Path Expression	• Deep Path Expression
Joins	• Two-way Join • Small Result Size	• N-way Join • Large Result Size
Regular Path Expressions	• Shallow Path Expression • Few Unknown Elements	• Deep Path Expression • Many Unknown Elements
Document Construction	• Simple Original Structure • Simple Output Structure	• Complex Original Structure • Complex Output Structure
Ordered Access	• No Index	• With Index
Sorting	• Few Qualified Tuples • Single Condition	• Many Qualified Tuples • Multiple Condition
Missing Elements	• Few Generated Tuples	• Many Generated Tuples

Text Search	• One Text • Few Generated Tuples	• Multiple Text • Many Generated Tuples
Data-type Cast	• Few Generated Tuples • Single Casting	• Many Generated Tuples • Multiple Casting
Function Application	• Few Generated Tuples	• Many Generated Tuples

3.5. Ontology Data Model

The term Ontology has been used in several disciplines. Recently, ontology becomes even common in computer science area. It can be used for many purposes, including enterprise integration, database design, information retrieval, and information interchange on the World Wide Web to overcome many traditional problems.

Gruber (1993) defines an ontology as “a formal, explicit specification of a shared conceptualization”. An ontology defines the terms used to describe and represent an area of knowledge. Ontologies are used by people, databases, and applications that need to share domain information. Ontologies include computer-usable definitions of basic concepts in the domain and the relationships among them. An ontology may take a variety of forms, but necessarily it will include a vocabulary of terms, and some specification of their meaning. This includes definitions and an indication of how concepts are inter-related which collectively impose a structure on the domain and constrain the possible interpretations of terms (Uschold, King, Moralee, & Zorgios, 1998).

Generally speaking, an ontology consists of the following main constructs (Stevens, Goble, & Bechhofer, 2000; Weißenberg & Gartmann, 2003).

- **Facts** represent explicit knowledge, consisting of:

1. **Classes or concepts** are generalizations of instances. Concepts are the focus of most ontologies. A concept is a representation for a conceptual grouping of similar terms. A concept can have subconcepts that represent concepts that are more specific than the superconcept. Concepts fall into two kinds:
   (a) Primitive concepts are those which only have necessary conditions (in terms of their properties) for membership of the class.
   (b) Defined concepts are those whose description is both necessary and sufficient for a thing to be a member of the class.
2. **Properties** can be subdivided into scalar attributes and non-scalar relations. The property can be defined to be a specialization (subproperty) of an existing property. An attribute is a property of a concept that refers to a datatype (integer, string, float, boolean etc.). An example of an attribute is "has-name" related to a string. A relation is a property of a concept that refers to another concept. Specialization / Generalization are one of the standard relations. For instance, "is a kind of" defines a relation that may be applied to the concepts "Enzyme" and "Protein".
3. **Instances** represent individual entities and are connected by type-of relation to at least one class; some authors only consider facts about instances as real facts. Strictly speaking, an ontology should not contain any instances, because it is supposed to be a conceptualization of the domain. The combination of an ontology with associated instances is what is known as a knowledge base. However, deciding whether something is a concept of an instance is difficult, and often depends on the application.

- **Axioms** are rules used to add semantics and to infer knowledge from facts. In contrast to facts, they represent implicit knowledge about concepts and relations,

e.g., whether a relation is transitive or symmetric.

3.6. Ontology Query Model

Initially, ontologies are introduced as an "explicit specification of a conceptualization". In an information integration system, ontologies can be used to establish common vocabularies and semantic interpretations of terms from information sources. With respect to the integration of data sources, they can be used for the identification and association of semantically corresponding information concepts. People can share and exchange information in a semantically consistent way.

Using ontology basic components described in the previous section, user can define their own ontology in any application domain. Then we conducted a series of tests to see how the system handles such ontologies. The operation model in the ontology benchmark workload model is a set of queries, and the answers are generated by inferring from the ontology. The queries we present here are representative for different application domains.

We conclude the reasoning tasks describing in chapter 2, and construct six basic reasoning queries for the ontology benchmark. Figure 3.4 shows the ontology query model in this research. We introduce these queries briefly and described them in generic terms as follows.

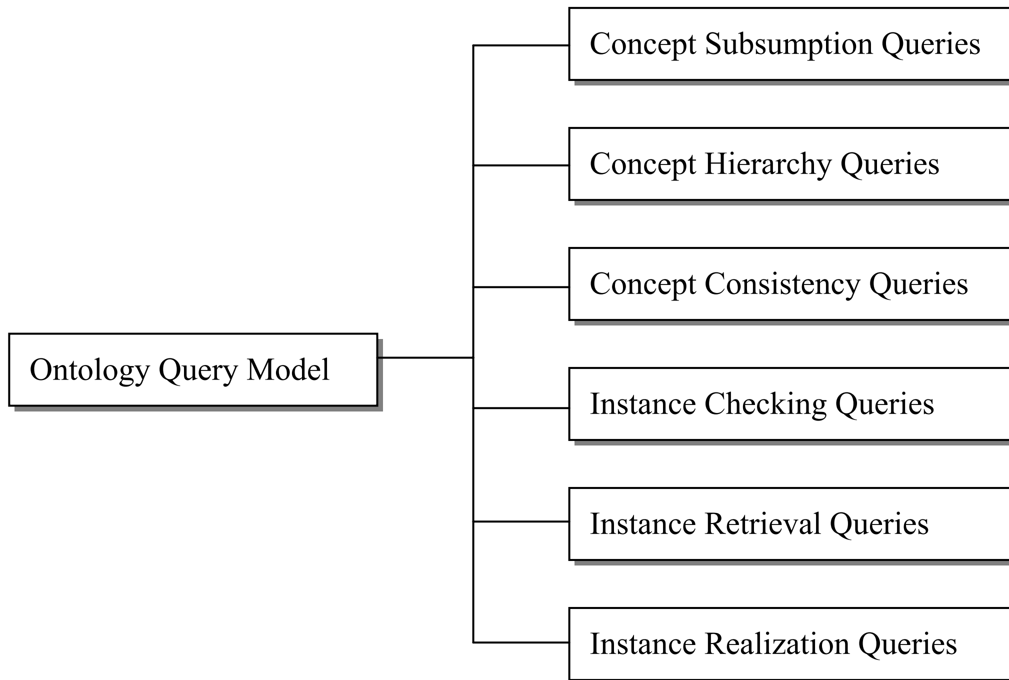

Figure 3.4: Ontology Query Model

1. **Concept Subsumption Queries:** checks if one concept is a subconcept of another.

   Generic terms:

   Given concepts C and D, determine if C is a subcconcept of D with respect to ontology O.

2. **Concept Hierarchy Queries:** determines the concepts that immediate subsume or are subsumed by a given concept.

   Generic terms:

   Given a concept C return all/most-specific superconcepts of C and/or all/most-general subconcepts of C.

3. **Concept Consistency Queries:** checks for (in)consistency of concept definitions.

   Generic terms:

   Given a concept C, determine if the definition of C is generally satisfiable (consistent).

4. **Instance Checking Queries:** given a partial description of an individual (instance) and a concept description, finds whether the concept describes the instance.

   Generic terms:

   Given a concept C, determine whether a given individual A is an instance of C.

5. **Instance Retrieval Queries:** finds all instances that are described by a given concept.

   Generic terms:

   Given a concept C, determine all the individuals in ontology O that are instances of C.

6. **Instance Realization Queries:** given a partial description of an instance, finds the most specific concepts that describe it.

   Generic terms:

   Given an individual A, determine all the concepts in ontology O that A is an instance of.

The queries mentioned above are simple and basic. When querying the heterogeneous information integration system, the reasoning service may not be so

straightforward. We need to evaluate the ontology with increasing complexity. When formulating the complex benchmark queries, several factors should be taken into account (Guo, Heflin, & Pan, 2003):

- **Input size:** This is measured as the proportion of the class instances involved in the query to the total class instances in the benchmark data.
- **Selectivity:** This is measured as the estimated proportion of the class instances involved in the query that satisfy the query criteria.
- **Complexity:** We use the number of classes and properties that are involved in the query as an indication of complexity.
- **Hierarchy information assumed:** This considers whether information of class hierarchy or property hierarchy is required to achieve the complete answer. Besides, the depth and width of class hierarchies should also been considered.

More complex queries may be formulated according to these factors mentioned above. It lets the system be evaluated under different level of workloads.

3.7. Test Database Generation

In order to evaluate the performance of the heterogeneous information integration system, we must define the workload. The workload consists of a test operation and a test database. The test database identifies what data must be loaded into the data sources, as well as the volume of the test data. Information integration system data sources are disparate and heterogeneous. Information comes from various sources (including structured, semi-structured and unstructured sources) and formats (such as database tables, XML files, PDF files, streaming media, internal documents, and Web pages). For this research, the data sources can be divided into three kinds: relational databases, object-oriented databases, and Web pages. For each data source,

we must analyze the actual data and extract statistical data. Data analysis characterizes data in terms of the size of the database, the number of records, the length of records, the types of fields, and the value distributions.

- **Determine data values**

A number of data types are supported in this research, including long integer number, double precision floating point number, decimal number, money, datetime, fixed-length and variable-length character strings. We must conduct extensive studies to characterize each data source with several distribution parameters. Frequency distributions are computed and standard probability distributions are fit to the data in order to generate the value of test data. Several standard benchmarks including the TPC-C, TPC-D, TPC-E, and AS^3AP all support uniform and non-uniform data distributions (e.g. exponential, normal, discrete, rotating, zipfian[2] or constant). Data values are created with these common data distributions.

- **Determine scaling factors**

After determining the value of the test data, we must define how much data should be generated, i.e. defining the database scaling factor. Generally speaking, the logical size for the test database used for the benchmark is at least equal to the logical size of physical memory on the host(s). For this research, we refer to the AS^3AP benchmark standard. In the AS^3AP benchmark, the test database consists of four generic relations. Each has the same number of fields and the same number of records. The database scales up by increasing tenfold number of records for each relation. The tuple length is 100 bytes on the average. The logical size of the AS^3AP database is defined as follows:

Logical size = (# tuples per relation) * (100 bytes/tuple) * (4 relations in database)

For this research, the tuple length is fixed at 100 bytes on the average as well. The size of the logical database can be scaled from 1 megabyte to 100 gigabytes by varying the number of tuples from 10,000 to one billion, as Table 3.2 shows.

Table 3.2: Scaling the Test Database

Number of Tuples (tuples)	Logical Database Size (bytes)
10,000	1Megabyte
100,000	10Megabytes
1,000,000	100Megabytes
10,000,000	1Gigabyte
100,000,000	10Gigabytes
1,000,000,000	100Gigabytes

- **Open data source**

We must determine the test data of the open data source, i.e. World Wild Web, but this is problematic. There is in excess of 9 billion pages on the Web, which include HTML files, text documents, PDF files, Microsoft Office documents and other similar data files. We cannot possibly download every page from the Web much less adequate sample size. Even the most comprehensive search engine currently indexes just a small fraction of the entire Web. Figure 3.5 shows the approximate number of text documents that have been indexed by major search engines.

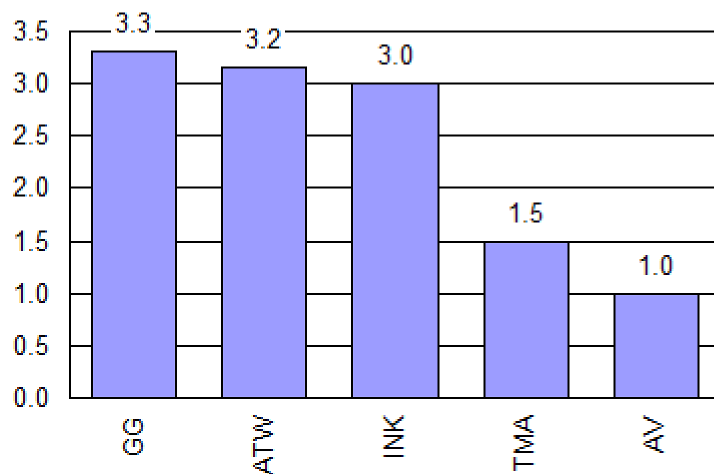

Figure 3.5: Billions of Textual Documents Indexed (Sullivan, 2003)

As such, it is important to carefully select the so-called "important" pages, so that the fraction of the Web that is visited becomes more meaningful. In order to select these important pages, we can use several metrics for prioritizing them. For any given Web page, we must define its importance using the following methods (Arasu, Cho, Garcia-Molina, Paepcke, & Raghavan, 2001):

1. **Interest-driven.** The goal is to obtain pages of interest to a particular user or set of users. Important pages are those that match user interest. One particular way to define this notion is through what we call a *driving* query. For any given query, the importance of a page is defined by the "textual similarity" between the page and the driving query. Assuming that query represents the user's interest, this metric shows how relevant the page is. Another interest-driven approach is based on a hierarchy of topics. Interest is defined by a topic, and we attempt to guess the page topics that will be visited by analyzing the link structure that leads to the candidate pages.
2. **Popularity-driven.** Page importance depends on how popular a page is. For instance, one way to define popularity is to use a page's backlink count. (We use

the term backlink for links that point to a given page.) Intuitively, a page that is linked to by many pages is more important than one that is seldom referenced.

3. **Location-driven.** The importance of a page is a function of its location, not its contents. For example, URLs ending with ".com" may be deemed more useful than URLs with other endings, or URLs containing the string "home" may be of more interest than other URLs. Another location metric that is sometimes used considers URLs with fewer slashes more useful than those with more slashes.

For this research, the data unit for the Web is a page. The size of a Web page is 1 kilobyte on average. We can use rules mentioned above to prioritize Web pages, and select the important pages to load into the test database.

3.8. Control Model

The control model defines environment variables to execute the benchmark. These variables are used to set up the execution environment.

- **Steady State**

The benchmark test must be executed in a steady state, in order to return true performance of the system.

- **Test Mode**

There are three kinds of test mode, cold mode, warm mode, and hot mode. In cold mode, there is no data in the cache. The system cannot retrieve data from the cache directly. Therefore, the performance in cold mode is usually slower than other two modes. In warm mode, the data is left in the cache from prior query. Because of that, the test response time decreases. In hot mode, a query is executed in cold mode first, and then be executed with cache data for several times. The average response time is computed.

- **Test Duration**

  Test duration means time intervals of the benchmark. Each interval must begin after the system has reached steady state and be long enough to generate reproducible throughput results. Each interval must extend uninterrupted for a period of time.

- **Test Sequence**

  Test sequence indicates the order of the queries execute.

- **Number of Repetitions**

  Number of repetitions means execution repeated times of an operation in a test.

3.9. Performance Metrics

Performance metrics are used to measure the execution result. Response time and throughput are two performance metrics often used in evaluation of computer systems.

- **Response time** means time interval between when a request is made and when the response is received by the requester.
- **Throughput** means the number of operations completed by the system per unit time.

In the ontology benchmark, system would generate answers that are entailed by the ontology. Notably it is not sufficient to consider response time and throughput as metrics. Two fundamental measures of the quality of information retrieval, recall and precision, can be used to evaluate the performance. Besides, error probability should be taken into account together (refer to Figure 3.5 and Table 3.2).

- **Recall** is the percentage of relevant data which has been retrieved. In this research, it can be used to measure whether the system can generate all answers that are entailed by the ontology. Therefore, it also can be called **completeness**.

Recall = A / (A + B)

- **Precision** is the percentage of retrieved data which is relevant. In this research, it defines the level of "noise" in the information presented to the user.

  Precision = A / (A + C)

- **Error probability.** If the answer is irrelevant, or the relevant answer is not retrieved, there is an error occur. The error probability should be calculated.

  Error Probability = (B + C) / (A + B + C + D)

Relevance is an abstract measure of how well the data satisfies the user's information need, i.e. what the user really wants to know. Ideally, your system should retrieve all of the relevant documents for you. Unfortunately, this is a subjective notion and difficult to quantify (Weiss, 1997). In this research, the ontology benchmark workload model is implemented on a small, human observable ontology. It is easy for users to identify the relevant information of each query.

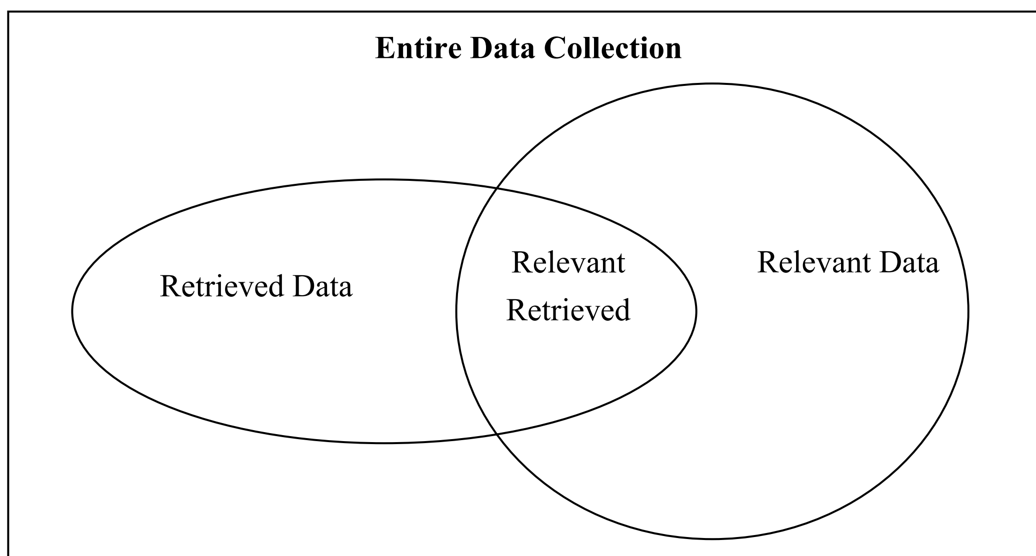

Figure 3.6: Retrieved and Relevant Data

Table 3.3: Retrieved and Relevant Data

	Relevant	**Irrelevant**	**Total**
Retrieved	A	C	A + C
Not Retrieved	B	D	B + D
Total	A + B	C + D	A + B + C +D