The Information Integration System K2

Val Tannen, Susan Davidson
Department of Computer and Information Science
University of Pennsylvania

Scott Harker
GlaxoSmithKline

In 1993, an invitational DOE (Department of Energy) Workshop on Genome Informatics published a report which claimed that until all sequence data is gathered in a standard relational database, none of the queries in the appendix to the report could be answered (see Figure 1 for a listing of the queries). While the motivation for the statement was largely political, the gauntlet had been laid in plain view for database researchers: The data to answer the queries in the appendix were (by and large) available, stored in a number of physically distributed databases. The databases represented their data in a variety of different formats using different query interfaces. The challenge was therefore one of integrating heterogeneous distributed databases and software programs in which the type of data was complex, extending well beyond the capabilities of relational technology.

As an example of the type of genomic data that is available online, consider the EMBL-format Swissprot entry shown in Figure 2 (see http://us.expasy.org/sprot). Each line begins with a two-character code, which indicates the type of data contained in the line. For example, each entry is identified by an accession number AC and is timestamped by up to three dates DT: the create date is mandatory, while the sequence update and annotation update dates only appear if the sequence or annotation has been modified since the entry was created. The sequence SQ (list of amino acids) appears at the end of the entry: the rest of the core data includes citation information (bibliographical references, lines beginning with R), taxonomic data (description of the biological source of the protein), and database references DR (explicit links to entries in other databases: EMBL (annotated nucleotide sequence database); HSSP (homology derived secondary structure of proteins); WORMPEP (predicted proteins from the Caenorhabditis elegans genome sequencing project); INTERPRO, PFAM, PRINTS, PROSITE (databases of protein families and domains) among other things). Annotation information, which is obtained by publications reporting new sequence data, review articles and external experts, is mainly found in the feature table FT, keyword lines KW, and comment lines CC (which do not appear in this example due to lack of space). Note that the bibliographical references are nested structures; there are two references, and the RP, RC, RA and RL fields are specific to each reference. Similarly, the feature table is a nested structure in which each line contains a start and end position (e.g. 14 to 21), a type of feature (e.g. NP\_BIND), and a description. The entry is designed to be easily read by a human being and structured enough to be machine parsed. However, several lines still contain a certain amount of structure that could be separated out during parsing. For example, the author list is a string, which
The following “unanswerable queries” were taken from Appendix 1 of the 1993 Invitational DOE Workshop on Genome Informatics report, available at http://www.ornl.gov/hgmis/publicat/miscpubs/bioinfo/contents.html. Rather than saying that all sequence databases must be relationalized, the wording of the report has now been modified to say “until a fully atomized sequence database is available (i.e., no data stored in ASCII text fields), none of the queries in this appendix can be answered.”

1. Return all sequences which map ‘close’ to marker M on human chromosome 19, are putative members of the olfactory receptor family, and have been mapped on a contig map of the region; return also the contig descriptions. (This is nominally a link between GenBank, GDB, and LLNL’s databases.)

2. Return all genomic sequences for which alu elements are located internal to a gene domain.

3. Return the map location, where known, of all alu elements having homology greater than “h” with the alu sequence “S”.

4. Return all human gene sequences, with annotation information, for which a putative functional homologue has been identified in a non-vertebrate organism; return also the GenBank accession number of the homologue sequence where available.

5. Return all mammalian gene sequences for proteins identified as being involved in intra-cellular signal transduction; return annotation information and literature citations.

6. Return any annotation added to my sequence number ##### since I last updated it.

7. Return the genes for zinc-finger proteins on chromosome 19 that have been sequenced. (Note that answering this requires either query by sequence similarity or uniformity of nomenclature.)

8. Return the number and a list of the distinct human genes that have been sequenced.

9. Return all the human contigs greater than 150 kb.

10. Return all sequences, for which at least two sequence variants are known, from regions of the genome within +/- one chromosome band of DS14###.

11. Return all publications from the last two years about my favorite gene, accession number #####.

12. Return all G1/S serine/threonine kinase genes (and their translated proteins) that are known (experimentally) or are thought (by similarity) also to exhibit tyrosine phosphorylation activity. Keep clear the distinction in the output.

Figure 1: DOE “Unanswerable” Queries
could be parsed into a list of strings so as to be able to index into the individual authors. Similarly, the taxonomic data is also a string and spread over several lines, and could again be parsed into a list.

As shown in this example, the type system for genomic data naturally goes beyond the “sets of records” of relational databases and include sequential data (lists), deeply nested record structures, and union types (variants). As an example of a union type, the format of the RL (reference location) line depends on the type of publication: an unpublished entry contains a brief comment; a journal citation includes the journal abbreviation, the volume number, the page range and the year; the format of a book citation includes the set of editor names, the name of the book, an optional volume, the page range, the publisher, city and year. The structure of this Swissprot entry can be described precisely in a data definition language with sufficiently rich types. We will show such a description later on in Figure 5.

The database group at the University of Pennsylvania responded to this challenge by developing a “view integration” (or “integration-on-the-fly”) environment. In such an environment, the schemas of a collection of underlying data sources are merged to form a global schema in some common model (e.g., relational, complex value, or object-oriented). Users query this global schema using a high level query language (such as SQL [MS83], OQL [CB00], or CPL [BLS+94]); the system then determines what portion of the global query can be answered by which underlying data source, ships local queries off to the underlying data sources, and then combines answers from the underlying data sources to produce an answer to the global query. The initial view integration environment developed by our group was called Kleisli, and was designed and implemented by Limsoon Wong. Limsoon Wong later re-designed and re-implemented the Kleisli system at Singapore’s Institute of Systems Science; this version of Kleisli is described elsewhere in the book. About the same time, other information integration projects were also developed (e.g., [CGMH+94, LSK95, HSK+01]) including one specifically aimed at bioinformatics, OPM* [CM95] also described elsewhere in this book.

K2 vs. Kleisli. K2 is a successor system to Kleisli that was designed and implemented at the University of Pennsylvania by Jonathan Crabtree, Scott Harker, and Val Tannen. Like Kleisli, K2 uses a complex value model of data and is based on the so-called “monad” approach (see section 7.4). However, the design of K2 also contains a number of new ideas and redirections: First, the model incorporates a notion of “dictionaries”, which allows a natural representation of object-oriented classes [PT99] as well as Web-based data. Second, the internal language features a new approach to aggregate and collection conversion operations [LT97]. Third, the syntax of the language follows a mainstream query language for object-oriented databases called OQL [CB00] rather than the elegant but less familiar comprehension-style syntax originally used in CPL (the Collection Programming Language) [BLS+94] (Kleisli now uses an adapted SQL syntax). Fourth, a separation is made between the mediator (global schema) level and the query level by introducing a mediator language, K2MDL. K2MDL combines an ODL (Object Definition Language) specification of the global schema with OQL (Object Query Language) statements that describe the data mapping. The ability to specify intermediate mediators allows a large integration environment to be created in layers and componentized. Finally, to improve portability, K2 is implemented in Java and makes use of several of the standard protocols and APIs that are part of the “Java Platform”\footnote{See \url{www.javasoft.com/j2se/}.}, including RMI\footnote{See \url{www.javasoft.com/products/rmi-iop/}.} and
Figure 2: Sample Swissprot entry
JDBC\textsuperscript{3}; Thus, K2 is not an extension of Kleisli, but rather a system implemented from scratch that shares with Kleisli some of its design principles, while featuring a number of distinct developments outlined above.

Overall, the goal of K2 is to provide a generic and flexible view integration environment appropriate for the complex data sources and software systems found throughout genomics, that is portable and appeals to common practices and standards.

7.1 Approach

It should be noted that a number of other techniques have also been developed over the past ten years in response to the "DOE Challenge", including link driven federations and warehouses. In a link driven federation, users start by extracting entries of interest at one data source and then hop to other related data sources via Web links that have been explicitly created by the developers of the system. SRS [EA98], LinkDB [FGM+98], and GenCards [RCCPL97] are examples of this approach. While the approach is easy to use, especially for novices, the approach does not scale well: When a new data source is added to the federation, connections between its entries and entries of all existing federation data sources must be added, commonly referred to as the "$N^2$" problem. Furthermore, if a user is interested in a join between two data sources in the federation, they must manually perform the join by clicking on each entry in the first data source and following all connections to the second data source.\textsuperscript{7} In contrast, a join can be expressed in a single high-level query in a view or warehouse integration strategy. In general, the query languages supporting view or warehouse integration approaches are much more powerful languages, and allow arbitrary restructuring of the retrieved data.

A warehouse strategy creates a central repository of information and annotations. One such example is the Genomics Unified Schema (GUS) [DCB+01], which integrates and adds value to data obtained from GenBank/EMBL/DDBJ, dbEST and Swissprot (and others), and contains annotated nucleotide (DNA, RNA) and amino acid (protein) sequences. Note that view integration systems can also be used to create warehouses, which are instantiations of the global schema. The advantage of a warehouse approach over a view integration is one of speed and reliability; since all data are local, delays and failures associated with networks can be avoided. Furthermore, there is greater control over the data. However, the problem is that the warehouse is not dynamic: Not only must it be kept up-to-date with respect to the underlying data sources, but including a new data source or algorithm is time-consuming. A more extended discussion of the problems and benefits of the link, warehouse and view integration approaches can be found in [DCB+01, DOTW96].

K2 is a system for generating mediators. Mediators are middleware components that integrate domain-specific data from multiple sources, reducing and restructuring data to an appropriate virtual view [Wie92]. A major benefit of mediation is the scalability and long-term maintenance of the integration systems structure.

As an example of how mediators can help the data integration task, consider the example in

\textsuperscript{3}See java.sun.com/products/jdbc/.
\textsuperscript{4}See srs.ebi.ac.uk.
\textsuperscript{5}See www.genome.ad.jp/dbget/.
\textsuperscript{6}See nciarray.nci.nih.gov/cards/.
\textsuperscript{7}A counterexample to this is SRS, in which a linking operator is provided to retrieve linked entries to a set of entries.
Figure 3. In this example, each of the boxes represents a machine on which a copy of K2 is used to provide a mediator for some local as well as external data sources and views. Mediator 1 resides behind a company firewall and was built to integrate data from a local database and local copies of SwissProt and Genbank, accessed through SRS, with the application program Blast. Mediator 2 was then built outside of the firewall to integrate data from some external data sources — PubMed and a patent database, which can only be accessed through a web interface. An external copy of PubMed was used due to its size and the fact that the most recent version was always needed. Mediator 3 was then built within the company firewall to integrate data from mediators 1 and 2, and Mediator 1 was enlarged to integrate data from Mediator 3.

K2 (together with Tsimmis [GMPQ+95]) distinguishes itself among approaches based on mediation in that it generates mediators starting from a concise high-level description. This makes K2 especially appropriate for configurations in which many mediators are needed or in which mediators must be frequently changed due to instability in the data sources or in the client needs. Some of the salient features of K2’s mediation environment are:

- K2 has a “universal” internal data model with an external data exchange format for interoperation with similar components.

- It has interfaces based on the ODMG (Object Data Management Group) standard [CB00] for both data definition and queries.

- It integrates nested data, while offering a Java-based interface (JDBC) to relational database systems and an ODMG interface to object-oriented database systems.

- It offers a new way to program integration/transformation/mediation in a very high-level declarative language (K2MDL) that extends ODMG.
- It has an extensible rule-based and cost-based optimizer.

- External or internal decision-support systems can easily be included.

- It is written entirely in Java, with corresponding consequences about portability.

The basic functionality of a K2-generated mediator is to implement a data transformation from one or more data sources to one data target. The component contains a high-level (in ODMG/ODL) description of the schemas (for sources and the target) and of the transformation (in K2MDL). From the target’s perspective, the mediator offers a view, which in turn can become a data source for another mediator.

An overview of the K2 architecture is given in Figure 4. In this diagram, clients can issue OQL queries or other commands against an integration schema constructed using K2MDL. The queries are then translated to the K2 internal language using the K2MDL and query translators. This internal language expression is then optimized and executed, using data drivers to ship sub-queries to external data sources and return results.

The remainder of this chapter walks through the architecture by describing the data model, illustrating K2MDL and OQL, and briefly discussing the internal language, data drivers, query optimization and user interfaces. We close with a discussion of scalability and impact.
7.2 Data Model and Languages

ODMG was founded by vendors of object-oriented database management systems and is affiliated with the OMG (Object Management Group – CORBA). The ODMG standard has two main components: The first is ODL (Object Definition Language), a data definition language which is used to define data elements. ODL is an extension of CORBA’s IDL (Interface Definition Language). The second is OQL (Object Query Language), an enhanced SQL92-like language which is used for querying. By building on these standards, K2 leverages the following features:

- rich modeling capabilities
- seamless interoperability with relational, object-oriented, information retrieval (dictionaries), and electronic data interchange (EDI) formats (e.g., ASN.1)
- compatibility with UML (the Universal Modeling Language)
- integration with XML (with a given DTD)
- official bindings to Java, C++ and Smalltalk
- industrial support from ODMG members (Ardent, Poet, Object Design)
- increasing use in building “ontologies”

K2 uses ODMG’s ODL to represent the data sources to be integrated. It turns out that many biological data sources can be described as dictionaries whose keys are simple strings and whose entries are complex values. We shall now explain these two concepts.

A dictionary is simply a finite function. It therefore has a domain which is a finite set and associates to each element of the domain (called a key) a value (called an entry). The type of a dictionary in ODL is denoted by \texttt{dictionary<T1,T2>} where T1 is the type of the keys and T2 is the type of the entries. In OQL, the entry in the dictionary \texttt{L} corresponding to the key \texttt{k} is denoted by \texttt{L[k]}. Since OQL has no syntax for the domain of a dictionary \texttt{L} we have added \texttt{dom(L)} for this purpose. Note that if \texttt{L} has type \texttt{dictionary<T1,T2>} then \texttt{dom(L)} has type \texttt{set<T1>}.

Complex value data is built by arbitrarily nesting records (tuples), collections – such as sets, bags (multisets) and lists – as well as variants. Variants are pieces of data representing tagged alternatives (also known as “tagged unions”).

To illustrate complex values (including variants), we present in Figure 5 an ODL declaration for a class whose objects correspond to (parts of) Swissprot entries. The attribute \texttt{Ref} returns complex values obtained by nesting sets, lists and records (\texttt{struct’s} in ODL) as well as variants, the latter identified by the \texttt{choice} keyword.

K2’s approach to data integration consists of two stages. In the first stage users specify data transformations between multiple sources and a single target. The target is virtual (unmaterialized), and is in effect a new view. The sources may consist of materialized data or virtual views that have been previously defined through similar data transformations.

In the second stage users formulate queries against the virtual views. OQL is an excellent vehicle for the second stage but since it does not construct new classes as output it is not expressive enough for the first stage. Hence, for defining sources-target transformations, K2 uses a new language,
class Entry

(extent Entries)

{
    attribute string ID;
    attribute string AC;
    attribute struct Dates { date Create; date SeqUpdate; date AnnotUpdate; } DT;
    ...
    attribute list<string> OC;
    attribute list<struct {
        string RP;
        list<string> RA;
        string RC;
        choice { string present; bool absent; } RT;
        choice {
            string Unpublished;
            struct { string JAbbrev;
                short Volume;
                struct { short from; short to; } Pages;
                short Year; } Journal;
            struct { set<string> Editors;
                string Title;
                short Volume;
                struct { short from; short to; } Pages;
                string Publisher;
                string City;
                short Year; } Book;
        ...
    } RL;
    ...
}> Ref;
...
attribute string KW;
attribute struct { string KeyName; long From; long To; string Desc; } FT;
attribute string SQ;
}

Figure 5: ODL description of a class of Swissprot entries (partial)
K2MDL, which combines the syntax of ODL and OQL. K2MDL stands for K2 Mediator Definition Language, because these transformations are high-level specifications of middleware components called mediators, as we explained earlier. Some examples of K2MDL syntax are in the next section. The rich type system used in K2 has allowed us to model a very large range of practical data sources in a transparent and friendly manner. We have yet to find a genomic data source that cannot be easily described in K2.

7.3 An Example

Let us now illustrate the K2 approach. We give an example in which the target data could be called an ontology, i.e., a schema agreed upon by a class of users, and then show how a mediator generated by K2 could implement it in terms of standard data sources. Consider the data description below, which is given in ODL syntax:

TARGET DATA DESCRIPTION:

```plaintext
class Protein
  (extent proteins)
  {
    attribute string SwissProtAccession;
    attribute string recommendedName;
    attribute set<String> alternateNames;
    attribute string sequence;
    attribute int seqLength;
    relationship set<Gene> hasSource
      inverse Gene::hasProduct;
  }

class Gene
  (extent genes)
  {
    struct Range
      {
        long from;
        long to;
      }
    attribute string name;
    attribute string organism;
    attribute Range location;
    relationship Protein hasProduct
      inverse Protein::hasSource;
  }
```

We record data about proteins using a Swissprot accession number, a recommended name, a set of alternate names, the protein sequence and its length, and a set of "references" to the genes that code for the protein. We also record data about genes in the form of the name of the gene, the name of the organism it comes from, the location of the gene in the genome, and a reference to the protein it codes for. While most of our attributes have simple values, strings and integers, alternateNames is a set of strings, and location is a record. The schema also specifies that hasSource in Protein and hasProduct in Gene are more than just attributes; they form a relationship between the extents of the two classes, and are inverses. This means that:

- Given a protein P, for each of the genes G in the set P.hasSource it is the case that G.hasProduct = P; and
- given a gene G, it is the case that G belongs to the set (G.hasProduct).hasSource.

Now assume that the data about proteins and genes resides in (for illustration purposes) four materialized data sources. Some protein data is in Swissprot, which can be accessed through an SRS driver that presents an object-oriented schema (i.e., a class). Assume also that organism and

---

8This is of course a very simplified model of proteins and genes, but the intent of this example is to demonstrate K2MDL, not to develop a scientifically viable model.
gene data are in two relations (in the same or in separate relational databases). We give here the SQL data description for these relations, but in fact K2 uses an equivalent description in ODL syntax, based on the observation that relations are simply sets of records. Finally, we have a web-based data source that contains protein name synonyms, which we model as a dictionary.

**SOURCES DATA DESCRIPTION**

```sql
CREATE TABLE Orgs
    (name string,
     orgid string
    )
;
CREATE TABLE Genes
    (name string,
     geneid string,
     orgid string,
     startpos long,
     length long
    )
;
```

ProteinSynonyms : dictionary< string, set<struct(syn string, lang string)>> >

Next we give the K2MDL description of the integration and transformation that is performed when the sources Swissprot, Orgs, Genes, and ProteinSynonyms are mapped into the ontology view. K2MDL descriptions look like the ODL definition in the ontology, enhanced with OQL expressions that compute the class extents, the attribute values, and the relationship connections (a related idea appears in [AB91])

Here is the definition of the class Protein as a K2MDL **classdef**. The definition starts with an OQL statement that shows how to compute the extent proteins of this class by collecting the accession numbers from Swissprot. The elements of the extent are used as **object identifiers** (oids) for the objects in the class. The rest of the definition shows how to compute the value of each attribute for a generic object identified by the oid (denoted by the keyword **self**). The OQL function `element(c)` extracts the unique element of the collection `c` when `c` is a singleton, and raises an exception otherwise.

**MEDIATOR DESCRIPTION I**
class Protein
    (extent proteins { select distinct s.Accession from swissprots s; })
{
    attribute string SwissprotAccession { self; };
    attribute string recommendedName {
        element(select s.Description from Swissprot s where s.Accession=self);
    };
    attribute set<string> alternateNames {
        select distinct ps.syn
        from swissprots s, ProteinSynonyms[s.Description] ps
        where s.Accession=self and ps.lang="eng";
    };
    attribute string sequence {
        element(select s.Sequence from swissprots s where s.Accession=self);
    };
    attribute int seqLength {
        element(select s.Sequence_Length from swissprots s where s.Accession=self);
    };
    relationship set<Gene> hasSource {
        select distinct gn
        from swissprots s, s.GeneNames gn
        where s.Accession=self;
    } inverse Gene::hasProduct;
}

Note, for example, the computation of the value of the attribute alternateNames. For an object identified by self, find the Swissprot entry s whose accession number is self and then use the description of s as a key in the dictionary ProteinSynonyms. The entry retrieved from the dictionary ProteinSynonyms[s.Description] is a set of records. Select from this set the records with names in English and collect those names into the answer. The value of the attribute is a set of strings. A further query posed against the class Protein may, for example, select objects whose alternateNames attribute contains a given synonym.

MEDIATOR DESCRIPTION II
classdef Gene
    (extent genes { select distinct g.geneid from Genes g; })
{
    struct Range
    {
        long from;
        long to;
    }
    attribute string name {
        element(select distinct g.name from Genes g where g.geneid=self);
    }
    attribute string organism {
        element(select o.name
            from Orgs o, Genes g
            where g.geneid=self and o.orgid=g.orgid);
    }
    attribute Range location {
        element(select struct(from: g.startpos, to: g.startpos+g.length-1)
            from Genes g
            where g.geneid=self);
    }
    relationship Protein hasProduct {
        element(select distinct s.Accession
            from swissprots s, s.GeneNames gn, Genes g
            where g.geneid=self and gn=g.name);
    } inverse Protein::hasSource;
}

What we hope this example illustrates is that relatively complex integrations and transformations can be expressed concisely and clearly, and can be easily modified.

7.4 Internal language

The key to making ODL, OQL, and K2MDL work well together is the expressiveness of the internal framework of K2, which is based on complex values and dictionaries. ODL classes with extents are represented internally as dictionaries with abstract keys (the object identities). This framework opens the door to interesting optimizations that make this approach feasible.

The K2 internal language is organized by its type structure. There are base types such as string, number, etc., record and variant types, collection types, namely sets, bags and lists, and dictionary types. For each type construction we have two classes of operations: constructors such as empty set and set union, and destructors such as record field selection. The operations for collection types are inspired from the theory of monads [Mac71], and are outlined in [BNTW95]. For details specific for aggregates and collection conversions see [LT97]; the operations on dictionaries are described in [PT99].

The internal language derives its expressiveness from its flexibility. The primitives are chosen according to the principle of orthogonality, which says that their meaning should not overlap and that we should not be able to simulate one primitive through a combination of the others. This produces a language with fewer, but better understood, primitives. As an example, consider the following basic query statement:

    select E(x)
    from x in R
    where P(x)
This translates internally into
\[
\text{set}(x \in R) \text{if } P(x) \text{ then } \text{ singleton}(e(x)) \text{ else empty set}
\]
where \( \text{set}(x \in S)T(x) \) is the set deconstructor suggested by the theory of monads and \( \text{singleton}(e) \) is a "singleton" set, i.e., the set with just one element, \( e \).

The semantics of the set deconstructor is that of the union of a family of sets. i.e., if \( S = \{a_1, \ldots, a_n\} \) then
\[
\text{set}(x \in S)T(x) = T(a_1) \cup \cdots \cup T(a_n)
\]

This approach increases the overall language expressiveness by allowing any type-correct combination of the primitives (the language is fully compositional). At the same time it provides a systematic approach to the identification of optimization transformations by yielding an equational theory for the internal language. The formulas of such a theory are equalities between equivalent parts of queries, and they are used for rewriting queries in several of the stages of the optimizer.

Finally, we exploit known efficient physical algorithms for operations such as joins by automatically identifying within queries the groups of primitives that compute these operations.

7.5 Data Sources

K2 maps data from external sources into its internal language, as described above. K2 also has a notion of functions, which are used to provide access both to standalone applications such as BLAST (a sequence similarity package), and to predefined and user-defined data conversion routines. This flexibility allows K2 to represent most data sources faithfully and usefully.

K2 accesses external information through data drivers. This is an intermediate layer between the K2 system proper and the actual data sources. There are two kinds of drivers in K2, those that are tightly integrated with the server, and those that are more loosely connected.

Integrated Data Drivers (IDDs) are created by extending the two abstract Java classes which form K2's driver API (application programming interface). IDD's export a set of entry points to K2, connect to the data source, send queries to it and receive results from it, and package the results for use in the rest of the K2 system. The tight coupling of IDDs with the K2 system minimizes the overhead associated with connecting to the data source, and allows for additional optimizations to be performed. K2 can also cache results of queries sent to IDDs to improve overall speed.

K2 comes with an IDD which can connect to any relational database system that implements Sun's JDBC API. A Sybase-specific IDD is also available, which takes advantage of some features of Sybase that are not available through JDBC. Oracle- and MySQL-specific versions are currently under development. To connect to a new relational database that supports JDBC, one merely needs to add the connection information to a configuration file, and K2 will automatically expose the underlying schema for querying. When the underlying schema changes, the K2 administrator must restart the IDD so it can rediscover the new schema. A procedure for automatically detecting and rediscovering schema changes is planned as a future enhancement.

Another IDD provided with K2 makes use of W4F, the World Wide Web Wrapper Factory, also developed at the University of Pennsylvania (http://db.cis.upenn.edu/Research/w4f.html). W4F is a toolkit for the generation of wrappers for web sources. New wrappers can easily be generated using W4F's interface, and can then be automatically converted to K2 drivers. Changes to the format of web sources are partly handled by W4F's declarative wrapper specification language.
Large format changes require human intervention to redefine and regenerate wrappers.

A very powerful feature of K2 is its ability to distribute query execution using its IDD for Java RMI. This IDD can make an RMI connection to a remote K2 server, and send it part of the local query for processing. Therefore, all that is required to connect to the remote K2 server and start distributing queries is a change to the local K2 configuration file.

Sometimes it is difficult to develop an IDD for a new type of data source. For example, to treat a group of flat files as a data source it is often easier to write a Perl script to handle the string manipulations involved rather than implementing them in Java as would be required in an IDD. In fact, some data sources cannot be accessed at all from Java, but only through APIs in other languages. To handle this, K2 has an IDD called the PipeDriver which does not connect to the data source directly, but to a Decoupled Data Driver (DDD).

A DDD is a simple standalone application, written in any language at all, which accepts queries through its standard input stream and writes results to its standard output. The PipeDriver takes care of sending queries to the DDD and converting the results into K2's internal representation. It can run multiple DDWs at once to take advantage of parallelism, and can make use of the caching mechanism built into all DDWs, all of which simplifies the job of the DDD writer.

It is the responsibility of the DDD to establish a connection to the data source (often nothing is required in this step), tell the PipeDriver it has made the connection, and wait for a query to come in. When the DDD receives a query it extracts the appropriate result from the data source and writes it out in a simple data exchange format. It then returns to waiting for the next query. This loop continues until the K2 server is brought down, at which point the PipeDriver tells the DDD to terminate.

DDDs have been written to connect with SRS, KEGG, and BLAST, as well as a number of web-based sources. The time it takes to create a new DDD depends greatly on the capabilities of the data source for which it is being written, and how much intelligence is to be built into the DDD. For example, it only takes an hour or two to write a Perl script to connect to a simple document storage system; the script must be written to receive an ID, retrieve the document, and print it out in K2’s exchange format, taking into account any error conditions that might occur.

However, the DDD writer has the flexibility to create special-purpose DDDs of any complexity. One DDD has been written that performs queries over a collection of documents that come from a remote website. This DDD maintains a local disk cache of the documents to speed access. It is responsible for downloading new versions of out-of-date documents, taking concurrency issues into account, and parsing and indexing documents on-the-fly. It also supports a complex language for querying the documents, and for retrieving structured subsets of their components. This DDD was written over the course of two weeks, and has been expanded periodically since.

7.6 Query Optimization

K2 has a flexible, extensible query optimizer which uses both rewrite rules and a cost model. The rewrite rules are used to transform queries into structurally minimal forms whose execution is always faster than the original query; this is independent of the nature of the data. On the other hand, a cost model uses information about the nature of the data such as the size of the data sets, selectivity of joins, as well as available bandwidth, and latency of the data sources. The cost information is used to choose between minimal queries that are incomparable with respect to the
rewrite rules.

After translating the query into an abstract syntax tree, which K2 uses to represent queries internally, it is manipulated by applying a series of rewrite rules. This is where the bulk of K2's optimization work is done.

First, a collection of rules is applied which simplifies the query by taking pieces expressed using certain kinds of tree nodes and replacing them with others. This reduces the number of types of nodes we need to deal with, thus reducing the number and complexity of the rewrite rules that follow.

Next the query is normalized. Normalization rules include things like taking a function that is applied to a conditional structure, and rewriting it so the function is applied to each of the expressions in the condition. Another normalization rule removes loops that range over collections that are known to be empty. Repeated application of the normalization rules, which currently number more than twenty, reduces the query to a minimal, or "normal" form.

A final set of rewrite rules are then applied to the normalized query. These rules do things like parallelizing the scanning of external data sources, and pushing selections, projections, and joins down to the drivers where possible.

Even after all of the rewrite rules have been applied, there may still be room for further optimization. In particular, a query may have a family of minimal forms rather than a single one. To choose between the minimal forms, the expected execution time of each version of the query is estimated using a cost model, and the fastest query form is chosen. The current cost model is still in the development stage. While it is functional and works well most of the time, it does not always choose the optimal form of the query.

7.7 User Interfaces

K2 has been developed using a client/server model. The K2 server listens for connections either through a socket, or through Java RMI. It is very easy to develop a client that can connect to K2 through one of these paths, issue queries, and receive results. Three basic clients come with K2: a text-based client, an RMI client, and one that runs as a servlet.

The interactive text-based client connects to a K2 server through a socket connection. It accepts a query in OQL through a command-line style interface, sends it to the server, gets the result back as formatted text and displays it, then waits for the next query to be entered. This simple client is generally used to test the socket connection to K2, and to issue simple queries during the development process. It is not intended to be an interface for end users.

The other type of user connection is through RMI. These connections are capable of executing multiple queries at once, and can halt execution of queries in progress. This is the connection method that K2 servers use to connect to other K2 servers to distribute the execution of a query.

There is a client which makes an RMI connection to a K2 server with administrator privileges. The server restricts these connections to certain usernames connecting from certain IP addresses, and requires a password. Currently, an administrator can examine the state of the server, add and remove connections to individual drivers, stop currently running queries, disconnect clients, and bring the server to a state where it can be stopped safely. More functionality is planned for administrators in the future.
A client that runs as a servlet is also included with K2. Using code developed at the Computational Biology and Informatics Laboratory (at the University of Pennsylvania), this servlet allows entry of ad-hoc K2 queries, and maintains the results for each username individually.

A major component of any user interface is the representation of the data to the user. As we have exemplified earlier, a user (perhaps one serving a larger group) can define in K2MDL a transformed/integrated schema for a class of users and applications and specify how the objects of this schema map to the underlying data sources. Users of this schema (called an ontology by some) can issue vastly simplified queries against it, without knowledge of the data sources themselves.

7.8 Scalability

In theory, the K2 system can be used to interconnect an arbitrarily large number of data sources. In practice, the system has been configured with up to thirty data sources and software packages, including GUS, PubMed, MacOstat, GenBank, SwissProt, BLAST, KEGG, and several relational databases maintained in Oracle, Sybase, and MySQL. Even when querying using this configuration, however, it has been rare to access more than five data sources and software packages in the same query.

The primary obstacles to scaling K2 to a larger system are 1) writing data drivers to connect to new data sources; and 2) peak memory consumption. The difficulty of writing data drivers to external data sources has to some extent been mitigated by the fact that they are type specific rather than instance specific. For example, once an Oracle driver is written it can be used for any Oracle data source. On the other hand, an ACeDB driver must take into account the schema of the ACeDB source and is therefore not generic. Drivers are also relatively simple in K2, since they merely perform data translation. Any complex, semantic transformations are performed by K2MDL code, which is high-level and therefore more maintainable.

While peak memory consumption has not as yet been an issue for the queries we have written, it could become a problem as applications become larger. K2 provides a means of limiting the number of queries it will run simultaneously, and the number of data source connections it will maintain. This allows an administrator to “tune” the system to the capabilities of the machine it is running on.

Since K2 is a view integration environment it does not store any data locally; however, it may need to store intermediate results for operations that cannot be streamed (i.e. processed on the fly). At present, intermediate results are stored in main memory.

Examples of operations that cannot be streamed include sorting, set difference, nesting and join. For example, when the difference of two data sets is taken, no output can be issued until both data sets are read; one of the data sets may have to be cached while the other is streamed and the difference is calculated. Similarly, although in a join output can be produced as soon as a match is found between elements of the two data sets, data cannot be discarded until it is known not to match any future input from the other data set (see [IFF+99, CDTW00] for discussions of implementations of operators in streaming environments). Thus, when data sets are large these operations may need to cache temporary results in persistent memory. Such techniques are not currently part of the K2 system and require further development.

Another difficulty of scaling to an arbitrarily large environment is the sheer complexity of understanding what information is available. To mitigate this, smaller mediated components can be
composed to form larger mediated components. Thus the user need not be aware of the numerous underlying data sources and software systems, but can interact with the system through a high-level interface representing the ontology of data.

7.9 Impact

As with Kleisli, a tremendous enhancement in productivity is gained by being able to express complicated integrations in a few lines of K2MDL code as opposed to much larger programs written in Perl or C++. What this means for the system integrator is the ability to build central client/server or mix-and-match components that inter-operate with other technologies. Among other things, it provides:

- Enhanced productivity (50 lines of K2MDL correspond to thousands of lines of C++).
- Maintainability, and easy transitions (eg. warehousing).
- Reusability (structural changes easy to make at the mediation language level).
- Compliance with ODMG standards.

K2 has been extensively used in applications within GlaxoSmithKline. Some of the major benefits of the system exploited in these applications have been the ease with which ontologies can be represented in K2MDL and the ability to conveniently compose small mediators into larger mediators.

K2 was also used recently to implement a distributed genomic-neuroanatomical database. The system combines databases and software developed at the Center for Bioinformatics at the University of Pennsylvania—including databases of genetic and physical maps, genomic sequences, transcribed sequences and gene expression data, all linked to external biology databases and internal project data (GUS [DCB+01])—with mouse brain atlas data and visualization packages developed at the Computer Vision Laboratory and Brain Mapping Center at Drexel University. The biological/medical value of the activity lay in the ability to correlate specific brain structure with molecular and physiological processes. The technological value of the activity was that K2 was ported to a MacOS X environment, visualization packages were tightly integrated into the environment, and both an ethernet and a gigabit network were used in the application. The work was significantly facilitated by the fact that K2 is implemented in Java. K2 now runs on Linux, Sun Solaris, MS Windows, and Apple MacOS platforms.

7.10 Summary

We have presented the K2 system for integrating heterogeneous data sources. K2 is general purpose, written in Java and includes JDBC interfaces to relational sources as well as interfaces for a variety of special-format bioinformatics sources. The K2 system has a “universal” internal data model that allows for the direct representation of relational, nested complex value, object-oriented, information retrieval (dictionaries) and a variety of electronic data interchange (EDI) formats, including XML-based sources. The internal language features a set of equivalence laws on which an extensible optimizer is based. The system has user interfaces based on the ODMG standard, including a novel ODL/OQL combined design for high-level specifications of mediators.
Acknowledgements: The authors would like to acknowledge the contributions of David Benton, Howard Bilofsky, Peter Buneman, Jonathan Crabtree, Carl Gustafson, Kazem Lellahi, Yoni Nissanov, G. Christian Overton, Lucian Popa, and Limsoon Wong.

References


