Problemseminar Bio-Datenbanken
WS 2002/2003

Anfragesprachen und Schnittstellenkonzepte für Bio-Datenbanken

Bearbeiter: Markus Rohrschneider

Betreuer: Timo Böhme

Prof. Dr. Erhard Rahm

Inhalt

1. Einleitung	3
2. Erweiterungen von SQL	5
2.1. GQL: Konzept der parametrisierten Sicht	5
2.2. Logik-Programmierung in Datenbanken	7
2.3. Beispiel	8
3. Ontologie-basierte Systeme für Biodatenbanken	11
3.1. TAMBIS Ontologie	12
3.2. GRAIL – Description Logic für TaO	13
3.3. Ein Anfragebeispiel	15
4. Formulargestützte Schnittstellen	16
4.1. Sequence Retrieval System (SRS)	16
5. Remote User Defined Functions	18
5.1. System-Architektur des LifeDB Query Interface	19
5.2. Beispiel einer RUDF Ausführung	20
6. Zusammenfassung	22
7. Glossar	23
8. Literatur und Bilderverzeichnis	24

1. Einleitung

Biologische Datenbanken sind äußerst heterogen zusammengesetzt und beinhalten häufig hoch komplexe Datensätze. Dies stellt besondere Anforderungen an die verwendeten Anfragesprachen. Problematisch ist hierbei, daß jedes Datenbanksystem eigene Systeme bzw. Sprachen für Datenanfragen bereitstellt. Ein einheitliches Konzept existiert nicht. Tools zur Datenanalyse werden häufig von Haus aus entwickelt und erfüllen meist projektsprezifische Aufgaben. Sie sind somit nicht nur systemabhängig, sondern operieren nur auf genau definierten, oft auch urheberrechtlich geschützten Domains. Die zugrundeliegenden Daten sind semistrukturiert, und es fehlen Standards. Das macht eine Zusammenarbeit zwischen mehreren Datenbanken sehr kompliziert und die Entwicklung von Anwendungen, die auf Biodatenbanken operieren extrem aufwändig, Daten aus verschiedenen DB-Systemen können häufig nicht integriert werden. Besonders in der heutigen, sich vor allem global entwickelnden Genforschung ist aber genau dieser Aspekt wichtig. Gleiches gilt für die Forschung in Biochemie, Biologie, Medizin u.a. Die komplexe Struktur der anfallenden Datensätze erfordert ein entsprechendes System zur Repräsentation. Die gegenwärtig weit verbreiteten relationalen Datenbanksysteme reichen dazu oftmals nicht aus.
Im Folgenden sollen vier Ansätze vorgeschlagen werden, die diese Anforderungen möglichst weitgehend erfüllen, die Zusammenarbeit heterogener Datenbanken ermöglichen und eine globale Zugängigkeit über das Internet unterstützen. Diese vier Ansätze stellen sehr unterschiedliche Vorgehensweisen dar mit dem Ziel, ein möglichst einheitliches System zu schaffen, unterschiedliche Datenbanken zu integrieren, dennoch flexibel und transparent zu halten. Besonders bei dieser Art von Daten möchte man in der Lage sein, es im Sinnes eines wissensbarsierten Systems zu behandeln, welches erlaubt, logisches Schließen und Folgern zu unterstützen. Dieser Aspekt wird vor allem im zweiten und dritten Teil der Ausarbeitung betrachtet. Formulargestützte Schnittstellen, das gebräuchlichste System stellt SRS dar, sollen im vierten Abschnitt eine Rolle spielen. Der fünfte Teil behandelt Remote User Defined Functions als Möglichkeit, auf möglichst flexible Weise eigene oder von einem entfernten Rechner bereitgestellte Analyse Tools in SQL-Anfragen einzubinden.

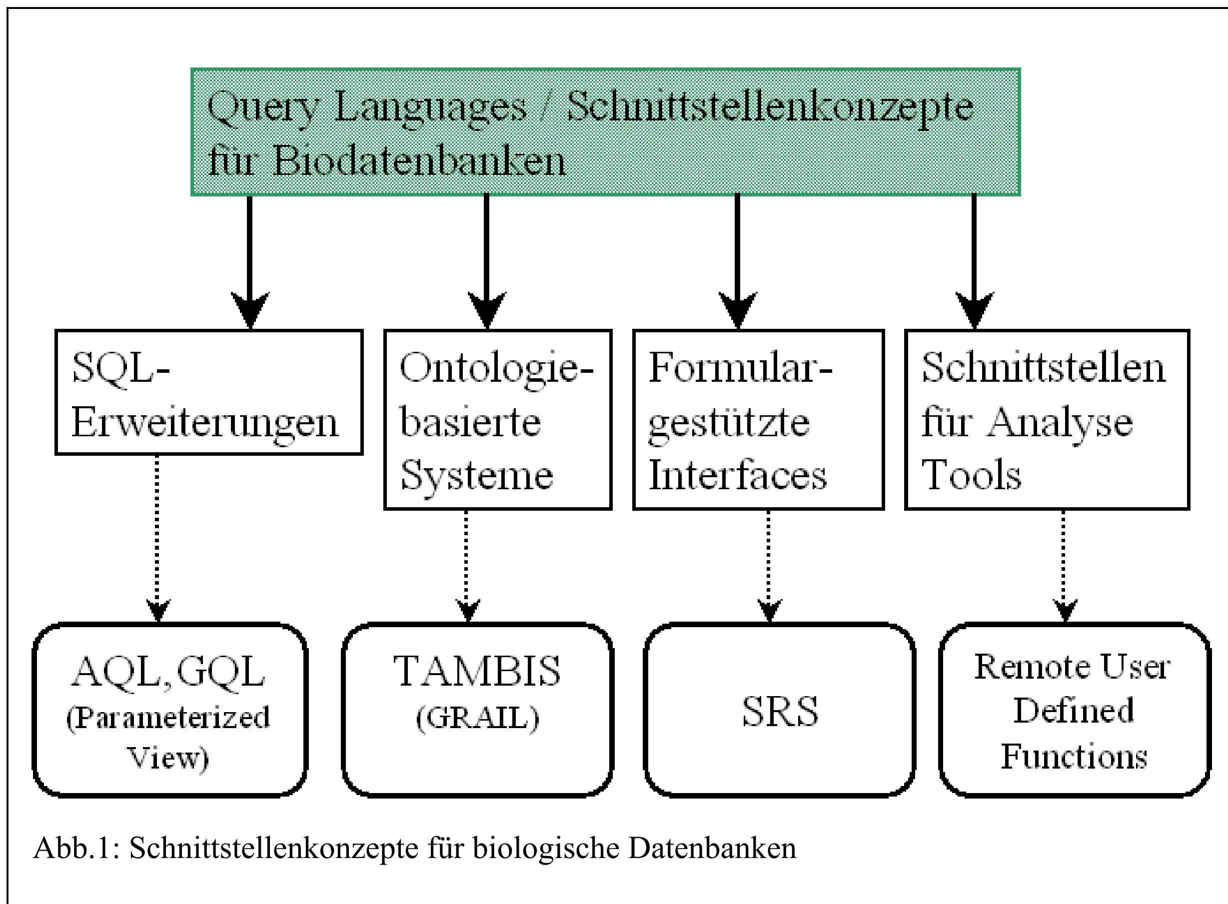


Abb.1: Schnittstellenkonzepte für biologische Datenbanken

Es gibt verschiedene Möglichkeiten, Informationen aus biologischen Datenbanken zu extrahieren. SQL ist die zur Zeit am häufigsten verwendete Anfragesprache für relationale und objektorientierte Datenbanken. Aufgrund der Heterogenität bzw. Komplexität der Daten in biologischen Datenbanken stößt SQL jedoch schnell an ihre Grenzen. Das erste vorgestellte Konzept soll sich daher mit bestimmten Erweiterungen für SQL befassen, die eine flexiblere Anwendung erlauben. Hier soll vor allem das Sicht-Konzept von SQL erweitert werden (parameterized views [JA00]), so daß die Idee des logischen Schlußfolgerns anhand einer gegebenen Wissensbasis mit einer bestimmten Regelmenge realisiert wird.
Eine weitere Möglichkeit für die Verwendung von Biodatenbanken ist die Repräsentation als Ontologie. Dadurch sollen in einer intuitiven Weise gespeichertes Wissen dargestellt und Zusammenhänge erschlossen werden. Das TAMBIS-Projekt (Transparent Access to Bioinformatics Information Sources, [BGP99]) stellt eine Entwicklung in dieser Richtung dar. Besonderer Bedeutung kommt hier der Anfragesprache GRAIL zu, die dieses Konzept unterstützt.
Die vermutlich aus Anwendersicht einfachste Methode stellen die formulargestützten Schnittstellen, wie zum Beispiel bei SRS (Sequence Retrieval System) verwendet, dar. Queries werden nicht mehr in textueller Form definiert, sondern durch eine graphische Nutzer-Schnittstelle, bei der logische Verknüpfungen zwischen verschiedenen Attributen möglich sind. Es gibt eine große Anzahl von Datenbanken, die dieses Konzept unterstützen.
Eine besondere Stellung nehmen die sogenannten Remote User Defined Functions (RUDF [CJ01]) ein. Durch eine spezielle Definition Language (DL) sollen Routinen zur Datenanalyse in SQL-Anweisungen eingebunden werden.

2. Erweiterungen von SQL

Die meisten Genom-Datenbanken basieren auf relationalen Datenbankschemata oder auf flat files im Textformat. Vor allem das Format von semistrukturierten flat files erschwert spezielle Datenanfragen. Relationale Datenbanken unterstützen solche Bedürfnisse nicht. Es gibt verschiedene Ansätze, basierend auf traditionellem SQL Anfragesprachen zu entwerfen, die in dieser Hinsicht erweiterte Funktionalität bieten. AQL (AceDB Query Language) ist ein Beispiel dafür. Sie stellt eine neue Anfragesprache für das AceDB Datenbanksystem dar. AQL nimmt neben dem bekannten `SELECT-FROM-WHERE`-Konstrukt von SQL weiterhin syntaktische Bestandteile von objektorientierten Anfragesprachen, wie OQL und ODMG, außerdem von Lorel, einer Sprache für Anfragen in semistrukturierten Datensätzen [Ace].

In der biotechnologischen Forschung sind logisches Folgern und Validieren von Hypothesen wichtige Hilfsmittel zur Wissenserweiterung [Ja00]. Im folgenden Abschnitt wird ein Konzept vorgestellt, wie SQL mit entsprechenden Erweiterungen in der Lage ist, logisches Schlußfolgern zu realisieren mit Hilfe einer entsprechenden Wissensbasis und einem Regelsystem. GQL (Genomic Query Language) ist ein solcher Ansatz, der das Sichtkonzept von SQL erweitert und dynamischer gestaltet. SQL kann in dieser Hinsicht als Teilmenge von GQL betrachtet werden.

2.1. GQL - Konzept der parametrisierten Sicht

Ziel dieses Ansatz ist es, die Datenbank um deduktive Mechanismen zu erweitern. Zwar ist es möglich, SQL durch Einbettung in eine Wirtssprache mit solchen engines zu koppeln, jedoch führt dies häufig zu dem oft diskutierten impedance mismatch [FJ89], d.h. zu einer „unsauberen" Kopplung zwischen Anfragesprache und Programmiersprache. Das Konzept der parametrisierten Sicht baut direkt auf SQLs Sichtkonzept für objekt-relationale Datenbanksysteme auf und umgeht somit dieses Problem. Basierend auf einem relationalen System werden damit deduktive Algorithmen in GQL (Genom Query Language) integriert. Das `CREATE VIEW` Statement von SQL wird üblicherweise dafür genutzt, um eine Teilmenge aus einer Menge von Datenbank-Relationen zu erzeugen, die besondere Bedingungen hinsichtlich der Attributparameter erfüllen (relational restrictions [Ja00]).

```
CREATE VIEW <vname> AS
SELECT t1.<attr1> t2.<attr2>
FROM <table1> AS t1, <table2> AS t2
WHERE t1.<attr3> = t2.<attr3> AND
      t1.<attr4> = <Restriction>
```

Abb.2 Create View Statement als Funktion, welche eine Join-Operation, Selektion bzg. des gemeinsamen Attributes attr3 und anschließend eine Projektion auf zwei Attribute durchführt

Diese Sichten können als Zwischenergebnis für weitere Operationen dienen, oder repräsentieren für bestimmte Nutzer nur die relevanten Daten. Allerdings sind diese views prinzipiell statisch und können nicht an besondere Bedürfnisse angepaßt werden. Speziell die Bedingungen in der `WHERE`-Klausel sind fixiert und können nicht geändert werden, ohne die gesamte `CREATE VIEW`-Anweisung neu zu definieren. Eine weitere Eigen-

schaft der views von SQL ist, daß mit Ausführen der `CREATE VIEW` – Anweisung eine entsprechende Tabelle erzeugt wird, dar, die mindestens für den Zeitraum der Nutzersitzung erhalten bleibt. Im Folgenden wird gezeigt, daß GQLs parametrisierte Views transient gehalten werden. Sie werden erst durch Aufruf (Invocation) mit den entsprechenden Parametern erzeugt und bleiben nur bis zum erneuten Aufruf erhalten.
Durch Einführung von formalen Parametern bei der `CREATE VIEW` – Anweisung können die views beim Aufruf an spezielle Voraussetzungen angepasst werden.

```
CREATE VIEW <vname>
WITH PARAMETER pname1,pname2,... AS
SELECT *
FROM <table1> AS t1, <table2> AS t2
WHERE t1.<attrX> = t2.<attrY> AND
      t1.<attrZ> = pname1 AND
      t2.<attrV> = pname2 AND ...
```

Abb.3 Create View Statement als Parameterisierte Sicht

Das in Abb.2 gezeigte SQL `CREATE VIEW`-Statement wird durch die `WITH PARAMETER`-Klausel in Abb.3 erweitert. Diese legt die maximale Menge der als Übergabe-Parameter erlaubten Attributwerte fest. Die `WITH PARAMETER`-Klausel stellt gewissermaßen das Interface zu der definierten Sicht dar. Diese Erweiterungen müssen bestimmten Prinzipien unterliegen. SQLs Unabhängigkeit der Aritäten bei der Definition von Queries soll unterstützt werden. D.h. es sollen Aufrufe mit unterschiedlicher Parameteranzahl möglich sein. Dies stellt in gewisser Weise ein Polymorphismus bezüglich des Überladens von Funktionen dar. Realisiert werden kann dies durch Verwendung von, wie aus der Logik-Programmierung bekannten „don't-care“-Variablen. Per definitionem werden solche "don't care“-Ausdrücke in select-Anweisungen als `null` und in where-Anweisungen als `true` behandelt. Beispielsweise sollten folgende Aufrufe der Sicht möglich sein:

```
CALL vname with (pname1/"Wert1", pname2/"Wert2", ...)
CALL vname with (pname2/"Wert2")
CALL vname with (pname1/"Wert1")
CALL vname
```

Die o.g. Definition entspricht der Intuition und unterstützt Deduktionsmechanismen der logischen Programmierung. (s.u.)
Desweiteren sollen diese Aufrufe an jeder beliebigen Stelle der Anfrage möglich sein, d.h. nicht wie bei SQLs built-in functions oder prozeduralen Konstrukten auf die `SELECT`-Klausel beschränkt sein müssen. Eine weitere Eigenschaft dieser in GQL verwendeten Views ist, daß sie nicht zur Definitionszeit ausgeführt werden. Bei Aufruf der so definierten „Prozeduren“ werden diese wie ad hoq Queries behandelt. Dadurch sind mehrere Aufrufe mit unterschiedlichen Parametern möglich. Dies hat zur Folge, daß die parametrisierten Sichten transienter Natur sind, welches durch Referenzierung realisiert wird. Mittels der `CALL`-Anweisung wird eine vorher definierte Sicht erzeugt und verhält sich genau wie eine Query-Ausführung im Gegensatz zu einer traditionellen `CREATE VIEW`-Anweisung. Der Unterschied zu der herkömmlichen Query-Ausführung ist, daß parameterized view - Definitionen ähnlich wie stored procedures als Teil des DB-Schemas abgespeichert werden [Ja00,p.6].
Wie dieser Ansatz deduktive Mechanismen unterstützt, sollen der folgende Abschnitt und das Beispiel in 2.3. zeigen.

2.2. Logik-Programmierung in Datenbanken

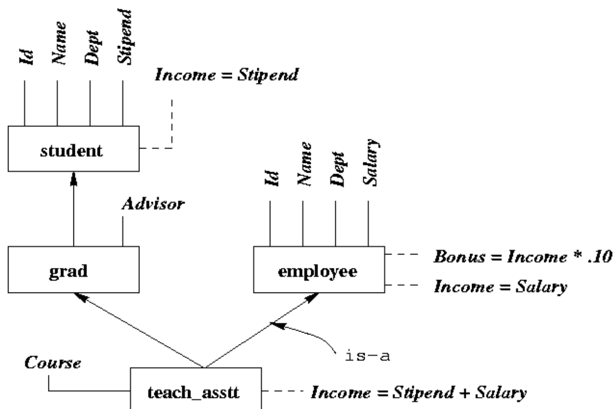
Logische Programmierung findet seine Anwendungen in Bereichen der Informatik, wo ausgehend von einer Wissensbasis (Axiome) und einer Menge von Regeln neue Informationen abgeleitet werden, v.a. in der KI und in wissensbasierten Systemen. Beispiele für Logische Programmiersprachen sind ProLog und Datalog. Letzteres orientiert sich mehr an Datenbanken. Relationale und objekt-relationale Datenbanken stellen im weiteren Sinne eine Wissensbasis dar und die definierten Relationen definieren die Regelmenge.
In Datalog läßt sich das `CREATE VIEW`-Statement aus Abb.1 folgendermaßen beschreiben:

```
R1: vname(attr1,attr2) <-   table1(attr1,_    ,X,...),
                            table2(_    ,attr2,X,...),
                            table1(_    ,_    ,_,Restriction,...)
```

Diese Regel liefert also alle Tupel bestehend aus dem ersten Attribut attr1 aus table1 und dem zweiten Attribut attr2 aus table2 mit der Bedingung, daß das dritte gemeinsame Attribut attr3 aus beiden Tabellen übereinstimmt und das vierte Attribut aus Tabelle 1 die konstante Bedingung „Restriction" erfüllt. R1 ist hier der Name der Regel, entsprechend dem Namen der Sicht vname. `vname(attr1,attr2)` heißt Kopf der Regel, entspricht hier dem Ergebnis der `CREATE VIEW`-Anweisung, und `table1(attr1,_,X,...), table2(_,attr2,X,...), table1(_,_,_,Restriction,...)` stellen den Rumpf bzw. die Prämisse der Regel R1 dar. Dies beschreibt die Bedingungen der `WHERE`-Klausel. Der Unterstrich steht hier für eine sogenannte "don't care"-Variable, d.h. die Werte dieser Attribute spielen bei der Selektion keine Rolle.
Auf diese Weise kann Wissen aus verschiedenen Datenquellen miteinander in Beziehung gebracht werden und Schlußfolgerungen gezogen werden. Im folgenden Kapitel soll die Beschränkung von SQL bei der Umsetzung solcher deduktiver Mechanismen und die Lösung durch GQLs parameterized views gezeigt werden.
Ein weiteres Beispiel für Anfragesprachen, das deduktive Mechanismen zur Ergebnissuche verwendet ist PQL (Pickle Query Language [MS02]), verwendet im GeneSeek genetic data integration project. Hier wird über ein mediated schema, ähnlich der Architektur von TAMBIS (s.u.) von mehreren Datenquellen abstrahiert und ein einheitlicher Zugriff über PQL geschaffen. Mit PQL lassen sich Anfragen formulieren, die bestimmte Bedingungen an die Suchpfade stellen. Anhand einer zugrundeliegenden Menge von Regeln wird im Sinne der logischen Programmierung das Ergebnis abgeleitet.

2.3. Beispiel

Es soll folgendes Datenbank Schema zugrunde gelegt werden (Abb.4 aus [JA00]):




```
create table student
 ( Id       integer,
   Name     char(10),
   Dept     char(10),
   Stipend  integer )

create table employee
 ( Id       integer,
   Name     char(10),
   Dept     char(10),
   Salary   integer )

create table grad
 ( Id       integer,
   Name     char(10),
   Dept     char(10),
   Stipend  integer,
   Advisor  char(10) )

create table teach_asstt
 ( Id       integer,
   Name     char(10),
   Dept     char(10),
   Stipend  integer,
   Advisor  char(10),
   Salary   integer
   Course   char(10) )
```

student			
Id	**Name**	**Dept**	**Stipend**
27	Oishii	CS	1000
45	Sharaar	EE	800

grad				
Id	**Name**	**Dept**	**Stipend**	**Advisor**
39	Susan	CS	1200	Carter
88	Tony	ME	2800	Miller

employee			
Id	**Name**	**Dept**	**Salary**
61	Brenda	CS	3600
22	Smith	PR	2800

teach_asstt						
Id	**Name**	**Dept**	**Stipend**	**Salary**	**Advisor**	**Course**
12	Joe	PH	1200	1600	Pots	PH327
82	Marni	CS	1300	1800	Renee	EE114

Tabelle 1 aus [JA00]: Student database schema

Aufgrund der Vererbungshierarchie im ER-Schema erbt die Klasse teach_asstt die virtuellen Methoden Income von student und employee und überschreibt diese durch die Methode Stipend+Salary. Bei der Umsetzung dieses ER-Schemas werden nun folgende Sichten erzeugt:

CREATE VIEW *ta_info* **AS**
SELECT *Id, Name, Dept, Stipend, Advisor, Salary, Course, Income, Bonus*
FROM *teach_asstt* **AS** *t, ta_income_view* **AS** *i, ta_bonus_view* **AS** *b*
WHERE *t.Id = i.Id* **AND** *i.Id = b.Id*

Datalog-Äquivalent:
R1: ta_info(Id,Name,Dept,Stipend,Advisor,Salary,Course,Income,Bonus) ←
teach_asstt(Id,Name,Dept,Stipend,Salary,Course,Advisor),
ta_income_view(Id,Income,Dept),
ta_bonus_view(Id,Bonus)

CREATE VIEW *ta_income_view* **AS**
SELECT *Id, Income=Stipend+Salary, Dept*
FROM *teach_asstt*
WHERE *Dept = “CS”*

Datalog-Äquivalent:
R2: ta_income_view(Id, Income, Dept) ←
teach_asstt(Id,Name,Dept,Stipend,Salary,Course,Advisor),
Income=Salary+Stipend

```
CREATE VIEW ta_bonus_view AS
SELECT Id, Bonus=Income * 0.1
FROM ta_income_view
```

Datalog-Äquivalent:
R3: ta_bonus_view(Id,Bonus) ← ta_income_view(Id,Income), Bonus=Income * 0.1

ta_income_view und ta_bonus_view erzeugen Tabellen, die die an teach_asstt vererbten Attribute bonus und income berechnen. Die folgenden beiden Anfragen heben den Unterschied zwischen traditionellem SQL und Datalog hervor:

q1: Gib alle teacher assistants der Informatik mit deren Bonus und Gehältern aus.
q2: Gib alle teacher assistants der Physik mit deren Bonus aus.
q3: Gib alle teacher assistants mit Bonus und Gehalt aus mit einem Bonus > 250 aus.

Die entsprechenden Anfragen in SQL und Datalog haben die folgende Form:

```
q1   SELECT Name, Bonus, Salary     ? ta_info(_,X,"CS",_,_,Y,_,_,Z)
     FROM ta_info
     WHERE Dept="CS"
q2   SELECT Name, Bonus             ? ta_info(_,X,"PH",_,_,_,_,_,Y)
     FROM ta_info
     WHERE Dept="PH"
q3   SELECT Name, Bonus             ? ta_info(_X,_,_,_,Y,_,_,Z), Z>250
     FROM ta_info
     WHERE Bonus>250
```

Während alle Datalog-Anfragen das gewünschte Ergebnis liefern, resultiert die zweite SQL-Anfrage in einer leeren, die dritte in einer unvollständigen Tabelle, da ta_income_view ausschließlich alle teacher assistants der Informatik erfasst. SQLs View-Konzept stößt aufgrund seiner Statik hier an seine Grenzen.

<table>
<tr><td rowspan="2">q1</td><td colspan="2">Name</td><td colspan="2">Bonus</td><td colspan="2">Salary</td><td colspan="2">Name</td><td colspan="2">Bonus</td><td colspan="2">Salary</td></tr>
<tr><td colspan="2">Marni</td><td colspan="2">310</td><td colspan="2">1800</td><td colspan="2">Marni</td><td colspan="2">310</td><td colspan="2">1800</td></tr>
<tr><td rowspan="2">q2</td><td colspan="3" rowspan="2">Name</td><td colspan="3" rowspan="2">Bonus</td><td colspan="3">Name</td><td colspan="3">Bonus</td></tr>
<tr><td colspan="3">Joe</td><td colspan="3">280</td></tr>
<tr><td rowspan="3">q3</td><td colspan="3">Name</td><td colspan="3">Bonus</td><td colspan="3">Name</td><td colspan="3">Bonus</td></tr>
<tr><td colspan="3" rowspan="2">Marni</td><td colspan="3" rowspan="2">310</td><td colspan="3">Marni</td><td colspan="3">310</td></tr>
<tr><td colspan="3">Joe</td><td colspan="3">280</td></tr>
</table>

Tabelle 2: Ergebnis der Anfragen mittels SQL (li.) und Datalog (re.)

Die Verwendung der parametrisierten Sicht schafft hier Abhilfe:

```
CREATE VIEW ta_info
WITH PARAMETER D,B AS
SELECT Id, Name, Dept, Stipend, Advisor, Salary, Course, Income, Bonus
FROM teach_asstt AS t, CALL ta_income_view WITH (D'/D) AS i, ta_bonus_view AS b
WHERE t.Id = i.Id AND i.Id = b.Id AND Dept = D AND Bonus > B
```

```
CREATE VIEW ta_income_view
WITH PARAMETER D' AS
SELECT Id, Income=Stipend+Salary, Dept
FROM teach_asstt
WHERE Dept = D'
```

In ta_info repräsentiert D, in ta_income_view D' den formalen Parameter. Bei der View-Definition von ta_info wird D als aktueller Parameter an D' in ta_income_view übergeben. Die obigen SQL-Anfragen werden nun durch eine CALL-Anweisung erweitert:

```
SELECT Name, Bonus, Salary FROM CALL ta_info WITH (D/"CS")
SELECT Name, Bonus          FROM CALL ta_info WITH (D/"PH")
SELECT Name, Bonus, Salary FROM CALL ta_info WITH (B/"300")
```

und liefern das gleiche Ergebnis wie die Datalog-Anfragen.

3. Ontologie-basierte Systeme für Biodatenbanken

Eine Ontologie ist ein System, welches Konzepte und deren Beziehungen zueinander beschreibt [BG99]. Sie besteht aus mehreren Komponenten: Konzepte, Relationen, Instanzen oder Objekte und Axiome. Ein Konzept repräsentiert eine Menge von Objekten innerhalb eines Grundbereichs mit bestimmten gemeinsamen Merkmalen. Relationen beschreiben die Beziehungen zwischen ihnen. Instanzen sind Objekte eines Konzepts, und Axiome repräsentieren allgemeine, der Ontologie zugrunde liegenden Regeln [Sk01]. Die Ontologie kann auf verschiedene Arten repräsentiert werden. Abb. 5 gibt einen Überblick.

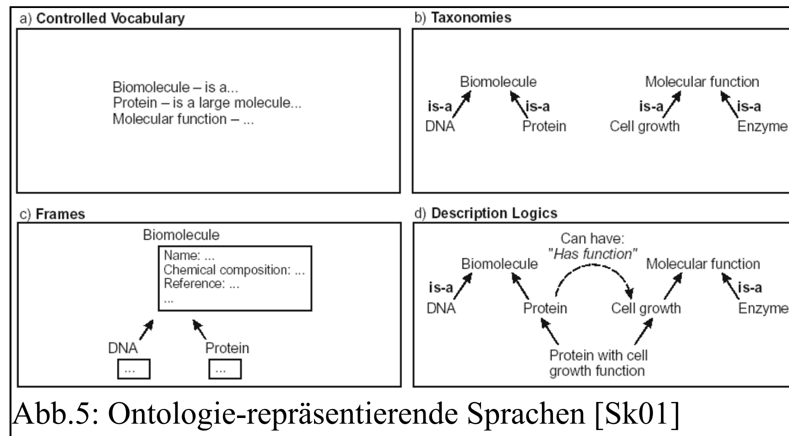


Abb.5: Ontologie-repräsentierende Sprachen [Sk01]

Zur Realisierung einer Ontologie als computergestütztes System ist eine formale Sprache notwendig, die diese Konzepte und Beziehungen beschreibt. Dafür wird eine "beschrei-bende Logik“ (Description Logics, DL) entwickelt. DLs sind äußerst flexibel und mächtig genug, um biologische Konzepte zu erfassen und zu klassifizieren [BG99]. Sie sind weiterhin in der Lage, anhand der vorhandenen Wissensbasis durch deduktives Folgern logische Schlüsse zu ziehen [Bo95], und damit neues Wissen abzuleiten . Die Idee der Ontologie wurde ursprüng-lich im Rahmen der KI entwickelt. Wissensbasierte Systeme können als Ontologie dargestellt werden. Besonders interessant ist dieser Aspekt für die biologische Forschung, da sich das Wissen in diesem Bereich in den letzten Jahren rapide erweitert hat. Dieses Wissen ist weltweit in zahlreichen Datenbanken gespeichert. Sehr schwierig gestaltet sich eine Anfrage an mehrere Quellen, da den Datenbanken häufig proprietäre Formate und Datenbereiche zugrunde liegen. Queries müssen für jedes System neu formuliert und ausgeführt werden. Ziel ist es daher, ein System zu schaffen, welches dem Nutzer maximale Transparenz bietet und einheitlichen Zugang zu vielen Quellen weltweit verschafft. Das TAMBIS (Transparent Access to Bioinformatics Information Sources) ist ein Projekt mit dieser Zielstellung. Im Folgenden soll auf die TAMBIS Ontologie (TaO) und deren Description Logic GRAIL näher eingegangen werden. Weitere Beispiele sind das Gene Ontology Project des Gene Ontology Consortium [GO], EcoCyc für Pathways von Escherichia Coli [EC] und Ontology for Molecular Biology [MBO].

3.1. TAMBIS Ontologie

TAMBIS entstand 1995 aus der Zusammenarbeit der Informatik- und Biologie-Abteilung der Universität Manchester. Es beinhaltet derzeitig ca. 1800 verschiedene Konzepte und vereinigt verschiedene biologische Datenbanken, einschließlich Swiss-Prot, CATH, Prosite, Enzyme and BLAST DBs auf virtueller Ebene mittels einer gemeinsamen graphischen Benutzerschnittstelle. Es wird die Illusion einer einzigen Datenquelle, Anfragesprache und Datenmodell geschaffen [Sk01]. Hinter den eben genannten Datenbanken verbergen sich jedoch individuelle Schemata mit jeweils eigenen Anfragesprachen. Die Integration dieser Datenbanken in ein einziges System geschieht durch die spezielle Architektur von TAMBIS.

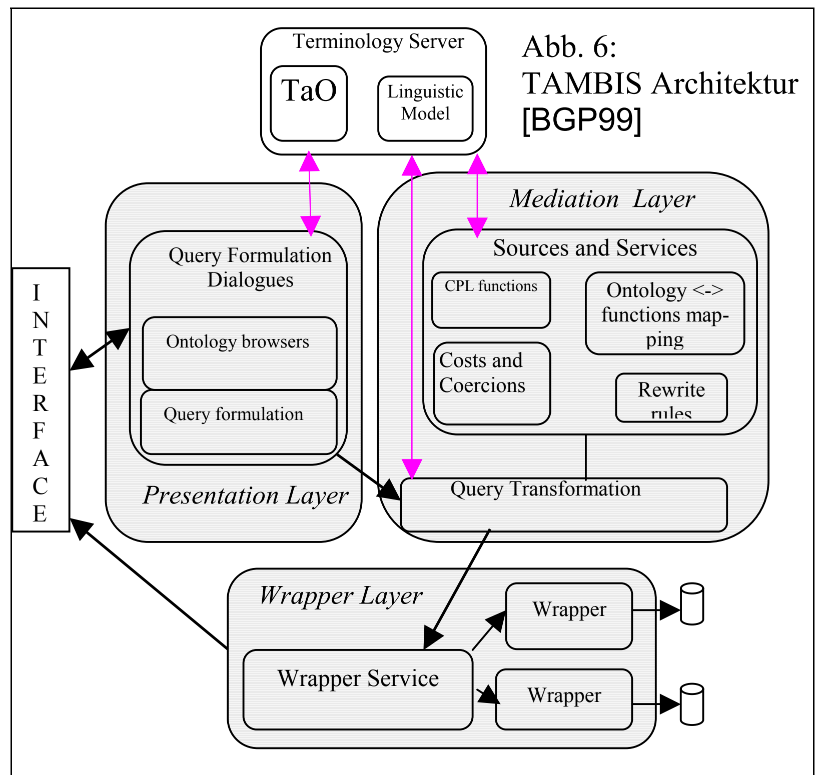


Abb. 6: TAMBIS Architektur [BGP99]

TAMBIS besteht aus fünf Hauptkomponenten, die dreischichtig angeordnet sind. Der Terminology Server beinhaltet die Ontologie selbst, definiert sich also über die Logik des Systems. Das Graphical User Interface ist die Schnittstelle, über die der Nutzer Anfragen an das System stellen kann. Diese ist Teil des Presentation Layer. Hier werden die (quell-unabhängigen) Queries auf graphische Weise zusammengesetzt. Dabei wird bereits die Terminologie und Logik des Systems verwendet, um semantisch korrekte Queries zu produzieren. Die Mediation Schicht identifiziert die der Anfrage entsprechenden Zieldatenbanken und formuliert sie entsprechend um. Dabei werden die betreffenden Konzepte der Ontologie mit den jeweiligen Gegenstücken der Quelldatenbank in Verbindung gebracht. Die Wrapper Schicht verwaltet die externen Datenquellen. Hier wird ein einheitliches Interface erzeugt, welches Transparenz innerhalb des Systems schafft. [BGP99]

TAMBIS ist eine dynamische Ontologie. Durch die Verwendung der Description Logic ist es möglich, neue biologische Konzept hinzuzufügen solange sie konform mit den zugrundeliegenden Axiomen der Ontologie sind. Derzeitig beinhaltet die TaO Konzepte wie Protein einschließlich ihrer Komponenten, Motiv (Bindungsstellen, aktive Zentren), Proteinstruktur, Enzymfunktion, metabolic pathway, Nukleinsäure, Genfunktion, Genexpression usw. und schließen Basisrelationen zwischen diesen Konzepten wie „is-component-of“, „is-homologous-to“, „has-name“, „has-function“ ein. Beim Formulieren von Anfragen wird das Basiswissen des Wissenschaftlers bezüglich Konzept-Angehörigkeit und Beziehungen genutzt, um erstens sinnvolle Queries zu konstruieren und zweitens die entsprechende Datenbank abzurufen. Die zugrundeliegende DL schließt Nonsense-Anfragen aus – eine Leistung, die andere graphische Query Formulators, wie z.B. SRS, nicht erbringen.

Im Abschnitt 3.3 soll kurz beschrieben werden, wie eine Anfrage an TAMBIS gestellt und intern verarbeitet wird.

3.2. GRAIL – Description Logic für TaO

Wie im vorhergehenden Abschnitt beschrieben, basiert die TaO auf einer Description Logic (DL) namens GRAIL. Als konzeptuelle Modellierungssprache wird GRAIL (Gene Recognition and Assembly Internet Link) verwendet, um biologische Konzepte zu beschreiben [BG99]. Abb.7 zeigt die höheren Ebenen der TAMBIS Ontologie, die durch GRAIL modelliert werden. Die DL beschreibt den Anwendungsbereich des Systems in Form von Konzepten, Rollen (i.e. Beziehungen, Relationen) und Instanzen. Ähnlich wie in UML läßt sich so die reale Welt als System interagierender Objekte beschreiben. Als Beschreibungssprache für TaO hat GRAIL besondere Eigenschaften, die GRAIL von anderen DLs unterscheidet. Beispielsweise sind in GRAIL sämtliche Rollen bidirektional. Das bedeutet, daß jede Relation von einem Konzept zum anderen eine gleichartige Relation in der Rückrichtung hat. Es gibt also sowohl die Relation hasComponent von dem Konzept Protein zum Konzept Motiv, als auch die Relation isComponentOf von Motiv zu Protein. Das ermöglicht Anfragen sowohl in der einen als auch der anderen Richtung. In GRAIL gibt es zwei Arten von Relationen: Kompositionen im Sinne von horizontalen Relationen (z.B. `isComponentOf`) und Klassifikation im Sinne von Vererbung, bzw. vertikalen Relationen (z.B. `isA` oder `isKindOf`). Im GRAIL Modell gibt es drei wichtige Bestandteile [BG99]:

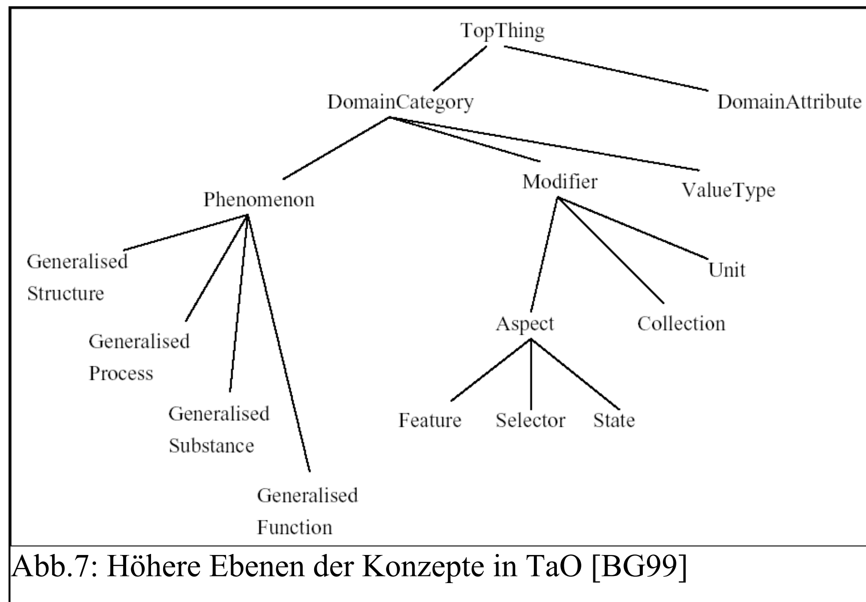


Abb.7: Höhere Ebenen der Konzepte in TaO [BG99]

- Grundsätzliche Konzept-Definitionen
- Konzeptbildende Operationen und Deduktionsmechanismen
- Sanktionen.

Grundsätzliche Konzept-Definitionen beschreiben einfache, atomare Konzepte, die sich nicht weiter zerlegen lassen können. Hier werden auch deren assoziierten Rollen beschrieben.

Beim zweiten Aspekt ist die Vererbung ein zentraler Gesichtspunkt für Deduktion oder logisches Schlußfolgern in der Ontologie. Man kann beispielsweise ausdrücken, aus Konzept A kann Konzept B abgeleitet werden, genau dann wenn sämtliche Instanzen von B ebenfalls Instanzen von A sind. GRAIL unterstützt Mehrfachvererbung, was einen weiteren Unterschied zu anderen DLs darstellt, wo Konzepte in einer strengen Baum-Repräsentation dargestellt werden. Dadurch erhält GRAIL eine höhere Flexibilität in Bezug auf die Sichtweise von Konzepten. Ein Konzept kann also unter verschiedenen Gesichtspunkten betrachtet werden.

Durch die Angabe von Sanktionen bei der Konstruktion von Konzepten und deren Rollen werden Regeln definiert, die semantisch sinnlose oder falsche Beziehungen verhindern. Dadurch ist es in TAMBIS auch nicht möglich, nonsense-Anfragen zu stellen. Beim Aufbau von Queries (s.u.) werden diese Regeln beachtet. Es gibt zwei Arten von Beschränkungen: auf abstrakter Ebene (*grammatical sanctions*) und auf Instanzebene

(*sensible sanctions*). Hier wird das Konzept der Vererbung ausgenutzt. Folgende Abbildung zeigt ein Beispiel für Sanktionierung:

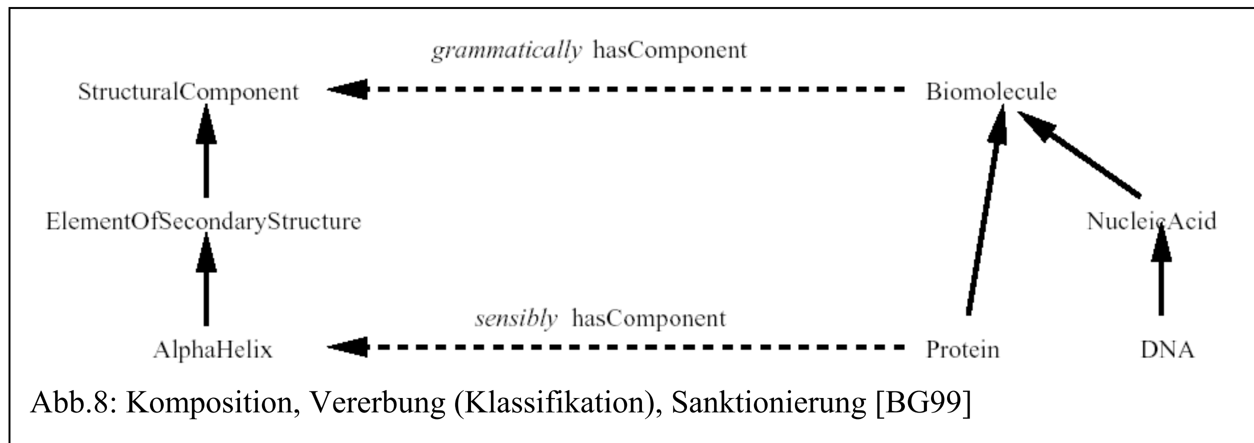


Abb.8: Komposition, Vererbung (Klassifikation), Sanktionierung [BG99]

`Biomolecule` und `StructuralComponent` sind abstrakte Konzepte, können also nicht instanziert werden. Die Kompositionsbeziehung `hasComponent` wurde auf grammatikalischer Ebene sanktioniert, denn Biomoleküle können, aber müssen nicht strukturelle Komponenten besitzen. Sanktionierung stellt also die Möglichkeit, aber nicht zwingende Notwendigkeit einer Komposition dar. Dagegen wäre eine *sensible sanction* zwischen `Biomolecule` und `StructuralComponent` nicht sinnvoll, da dies auch zuließe, daß DNA in alphahelikaler Form vorkommen könnte. Beim Erstellen neuer Konzepte muß also sehr genau darauf geachtet werden, auf welcher Ebene der Vererbungshierarchie *sensible* bzw. *grammatical sanctions* angebracht sind.

3.3. TAMBIS - Ein Anfragebeispiel

Dem Nutzer steht ein Query-Builder zur Verfügung, anhand dessen Schritt für Schritt das gesuchte Ergebnis spezifiziert wird. Dieses wird durch kontextsensitive Submenüs der GUI in jedem Konzept realisiert. Die zwischen den Konzepten möglichen Relationen werden präsentiert, zwischen denen der Nutzer die passende Beziehung auswählt. So wird ein baumartiger Pfad erstellt, der intern durch die DL der TaO (hier GRAIL, s.u.) repräsentiert wird. Dies geschieht im Presentation Layer. Der zu Abb.7 zugehörige Query-Ausdruck hat die Form:

```
Motif
which is ComponentOf
Protein
which hasOrganismSource
PoeciliaReticulata
```

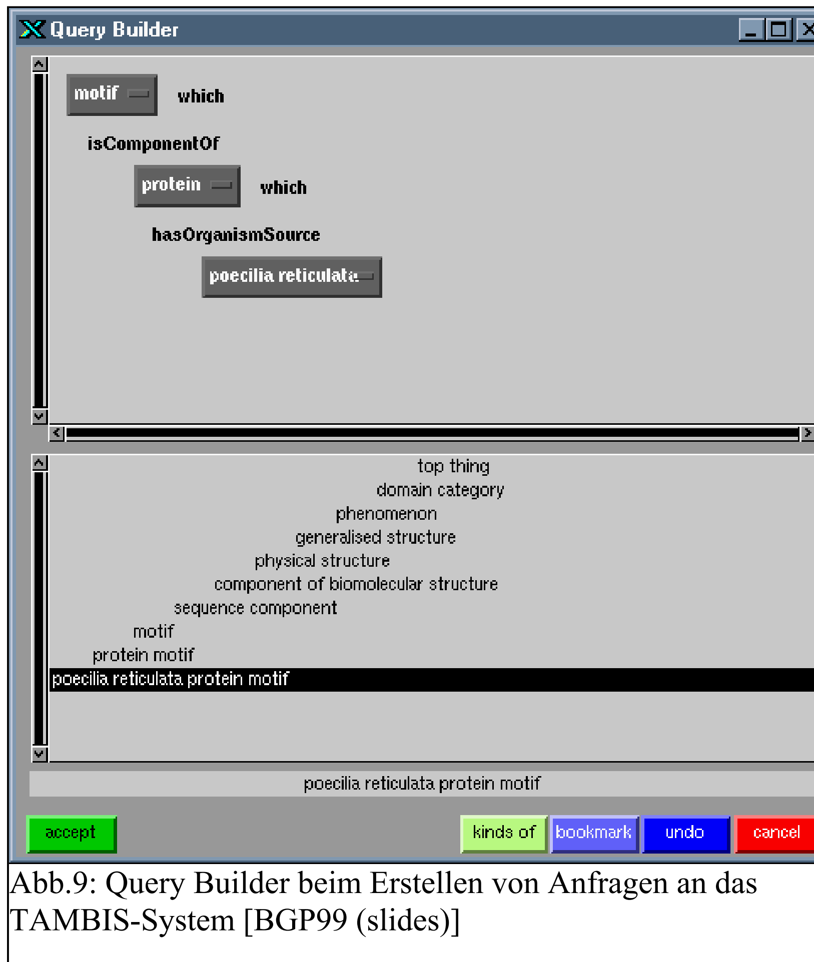


Abb.9: Query Builder beim Erstellen von Anfragen an das TAMBIS-System [BGP99 (slides)]

Gesucht sind also alle Motive von Proteinen (beispielsweise homologe Strukturen, wie Bindungsmotive etc.) des Organismus Poecilia Reticulata. Anhand der GRAIL-Formulierung wird nun im Mediation Layer ein Query Plan in CPL (Collection Programming Language [PS99]) erstellt. In diesem CPL-Ausdruck sind bereits die abzufragenden Datenbanken enthalten:

```
{motif1 |
   \protein1 <- get-sp-entry-by-os("POECILIA+RETICULATA"),
   \motif1 <- do-prosite-scan-by-entry-rec(protein1)
}
```

Diese Anfrage enthält zwei Teilanfragen an unterschiedliche Datenbanken. Die beiden auf das Symbol "|" folgenden Funktionen werden als Generators bezeichnet. Sie liefern eine Menge von Werten, die aus den entsprechenden Datenbanken extrahiert wurden. Der "|"-Operator erzeugt eine Projektion ausschließlich auf die Motive `motif1`. Das Konzept `protein` wird in der SWISS-Prot Datenbank nach dem Organismusnamen "Poecilia Reticulata" aufgesucht. Sämtliche Resultate werden in `protein1` gespeichert. Diese Variable wird in CPL als Liste oder Menge interpretiert. Schließlich wird für jede Komponente aus `protein1` eine Suche in der Prosite Datenbank durchgeführt und in `motif1` abgespeichert. Dies ist wiederum eine Liste, die das Anfrageergebnis enthält. Schließlich wird das Ergebnis als HTML-Seite im Browser ausgegeben.

Eine ausführlichere Beschreibung der CPL Funktionen, insbesondere die Umsetzung von Filtern in Queries läßt sich in [PS99] finden.

4. Formulargestützte Schnittstellen

Wie im vorangegangenen Kapitel gezeigt, müssen Anfragen an Biodatenbanken keineswegs in textueller Form, d.h. in einer speziellen Anfragesprache formuliert werden. Da aufgrund der Heterogenität von biologischen Datenbanken keine einheitliche DB-Anfragesprache existiert wie z.B. SQL für relationale DBs, würde sich die Suche nach Informationen in biologischen Datenbanken für den Benutzer allgemein als äußerst schwierig gestalten. Es ist daher kaum verwunderlich, daß sich besonders im Bereich der biologischen Datenbanken graphische Benutzerschnittstellen zur Formulierung von Anfragen weit verbreitet haben. Dabei muß man unterscheiden zwischen graphischen Schnittstellen, die für eine einzige Datenbank entworfen wurden und Schnittstellen, hinter denen sich eine ganze Reihe von zum Teil unterschiedlichsten Datenbanksystemen verbergen. Das Anfragesystem für GPCRDB (G-Protein coupled receptor Database [CC99]) spricht demzufolge nur diese eine Datenbank an, während das im Folgenden beschriebene SRS (Sequence Retrieval System) mehrere hundert wissenschaftliche Datenbanken umfasst.

4.1. Sequence Retrieval System (SRS)

SRS ist ein sehr weitverbreitetes Werkzeug für Anfragen an biologische Datenbanken, welches ermöglicht, gleichzeitig in einer großen Anzahl von weltweit verteilten Datenbanken nach Informationen zu suchen [HE99]. Es wird eine globale Sicht auf das gespeicherte Wissen gegeben. Die SRS-Technologie wurde speziell dafür entwickelt, eine Vielzahl von heterogenen biologischen Datenquellen in ein einziges Query-System zu integrieren. Es wurde eine webbasierte, graphische Benutzerschnittstelle entwickelt, mit deren Hilfe man derzeitig über 400 verschiedene wissenschaftliche Datenbanken nach Informationen durchsuchen kann. Diese Schnittstelle bietet allerdings nur lesenden Zugriff auf die Daten und hat nicht die Zielstellung, die Daten-inhalte der darunterliegenden Quellen zu integrieren [Sk01].

Abb.10 zeigt die Startseite von SRS. Hier werden die Daten-banken ausgewählt, die man bei der Anfrage mit einbeziehen will. [LION]

Bei der Suche nach Informationen mittels SRS steht auf der Query Page ein Formular zu Verfügung, womit die Suche nach bestimmten Gesichtspunkten eingeschränkt werden kann. Die einzelnen Eingabefelder sowie angegebene Schlüsselwörter lassen sich mit boolschen Operatoren verknüpfen. Es können reguläre Ausdrücke formuliert und Vergleichsoperatoren verwendet werden. Dabei können Attributnamen (Typen) angegeben werden. Da die meisten mit SRS verknüpften Datenbanken jedoch auf flat files im Textformat basieren, läuft die Suche ohne Angabe von Attributnamen auf Textsuche innerhalb der Dokumente hinaus. Neben den Eingabefeldern gibt es allerdings zusätzlich die Möglichkeit, Anfragen in textueller Form zu definieren.
SRS bietet darüber hinaus noch viele weitere Möglichkeiten, die Ergebnisse auszuwerten. So lassen sich die einzelnen Anfragen einer Session abspeichern, miteinander verknüpfen, um benutzerdefinierte Sichten (Views, s.o.) zu erzeugen. Durch Verlinken der einzelnen Ergebnisse ist eine sehr einfache Navigation möglich. Beziehungen zwischen einzelnen Datensätzen werden ebenfalls durch Hyperlinks dargestellt. Man hat mit SRS die Möglichkeit, Ergebnisse mit Standardmethoden der Bioinformatik zu analysieren. Programme wie ClustalW und BLAST können innerhalb der SRS-Umgebung aufgerufen werden mit deren Hilfe man beispielsweise Sequence Alignments berechnen oder Clusteranalysen durchführen kann.

Abb.11: SRS – Einbinden von Analyseprogrammen zur weiteren Datenauswertung

5. Remote User Defined Functions (RUDF)

Aufgrund der sich rasant entwickelnden Genforschung entstehen gewaltige Mengen an Sequenzdaten, die sich in Datenbanken weltweit verteilt befinden. In gleichem Maße wächst das Angebot von Analyseprogrammen für die Verarbeitung dieser Daten. CLUSTALX, FASTA, BLAST oder Sacc3D sind nur einige Beispiele. Diese Programme sind überwiegend zugänglich über das Internet, open source, können entweder zur Installation auf dem lokalen Rechner installiert werden oder sind über ein Web-Interface nutzbar. Dadurch hat jeder Wissenschafter die Möglichkeit, solche Werkzeuge an die eigene Problemstellung anzupassen und danach wieder öffentlich zugängig zu machen. Problematisch hierbei gestaltet sich jedoch das Fehlen von Standards, wodurch die Zusammenarbeit der Datenbanken untereinander sowie zwischen Analyse Tools und Datenbanken äußerst erschwert wird. Eine Integration solcher Programme direkt in die jeweiligen Datenbanken ist nicht praktikabel, da das Problem der fehlenden Interoperabilität der Datenbanken nicht gelöst ist.
Derzeitig gibt es verschiedene Bemühungen, ein einheitliches System mit einer dynamisch steuerbaren Sprache zu schaffen, welches verschiedenste biologische Daten integriert [CJ01]. Einige Beispiele sind das Gene Ontology project, BioKleisli und das oben vorgestellte TAMBIS ontology project. Die meisten dieser Projekte nutzen Informationen von entfernten Datenbanken, um sie mit lokalen Daten zu verarbeiten. Dabei werden eher lokale als entfernte Analysewerkzeuge verwendet. Prinzipiell hat ein Wissenschaftler zwei Möglichkeiten, lokale Daten zu verarbeiten. Entweder die zu untersuchenden Sequenzen werden via Web-Interface eines Datenanalyseprogramms, wie z.B. BLAST, nacheinander einem entfernten Host übermittelt und mit den Sequenzen der entsprechenden Datenbank verarbeitet. Die Ergebnisse können dann per e-mail zugesandt oder online abgerufen werden. Dieses Verfahren ist jedoch extrem langsam und kann nicht automatisiert werden, da eine Eingabe der Daten von Hand erforderlich ist. Alternativ gibt es die Möglichkeit, die gesamte entfernte Datenbank einschließlich der erforderlichen Analyseprogramme auf den lokalen Rechner zu übertragen, das Programm zu compilieren und als eigene User Defined Function in SQL-Anweisungen einzubinden. Allerdings muß dann die übertragene Datenbank ständig auf Aktualisierungen überprüft werden, um auf dem neuesten Stand zu sein. Code Migration stellt ein weiteres Problem der Plattformabhängigkeit dar. Oft werden Analyseprogramme für hoch spezialisierte Hardwarekonfigurationen entworfen. Oft operieren sie auf einem eng definierten Bereich von Daten. Ziel sollte es sein, eine Lösung zu finden, die sowohl Code Migration als auch Datenmigration vermeidet.
Das Konzept der Remote User Defined Functions stellt eine solche Lösung dar. Im Folgenden sollen Online Analyse Tools, wie BLAST oder CLUSTALX, als Internet Functions [CJ01] bezeichnet werden, da sie wie Funktionen im herkömmlichen Sinne ein bestimmtes Ergebnis für eine definierte Eingabe liefern. Um solche Funktionen in SQL-Anfragen einbinden zu können, muß die DDL (Data Definition Language) von SQL zur Internet Function Definition Language (IFDL) erweitert werden. Diese Definition stellt gewissermaßen das Protokoll zwischen SQL Query Engine und Internet Function dar, welches das Ein-und Ausgabeverhalten der Internet Function definiert. Mit Hilfe der IFDL sollen solche Funktionen in SQL-Anfragen im Sinne von stored procedures eingebunden werden können mit dem Unterschied, daß sie nicht als Teil des Datenbanksystems vorliegen, sondern von einem remote host aufgerufen werden.

Eine IFDL Funktionsdefinition beinhaltet folgende Informationen:

1. Name der verwendeten Internet Function mit dem die Funktion in die spätere SQL-Anfrage eingebunden wird.
2. URL der Internet Function
3. Name und Typ des Eingabeparameters für die Internet Function
4. Name und Typ des Rückgabewertes
5. HTQL-Wert (s.u.)

Weiterhin ist eine Hypertext Query Language (HTQL) notwendig, die die Schnittstelle zwischen Datenbanksystem und Internet Function realisiert. HTQL stellt eine spezialisierte Form von Query Languages dar, die verwendet werden, um Web Dokumente zu durchsuchen. HTQL orientiert sich an den in Hypertext Dokumenten verwendeten Tags und nutzt diese zur Navigation. Dies ist eine sehr effiziente Möglichkeit, Daten aus un- oder semistrukturierten Dokumenten zu extrahieren. Der HTQL-Wert in der IFDL Definition zeigt die Position des von der Internet Function gelieferten Rückgabewertes im HTML- oder XML-Dokument an. Mittels HTQL wird das Ergebnis der Internet Function aus dem Web-Dokument extrahiert und als Eingabewert zur Ausführung der SQL-Anfrage verwendet.

5.1. System-Architektur des LifeDB Query Interface

LifeDB ist ein Datenbanksystem, welches ihr Anwendungsgebiet in der biologischen Wissenschaft findet. LifeDB unterstützt das hier vorgestellte Konzept der Remote User Defined Functions.

Das LifeDB Interface ist im Aufbau sehr einfach gehalten. Es besteht aus drei Teilen: einem Textfeld links oben für die Eingabe der IFDL Function Definitions und SQL-Anfragen, der Query-Ergebnisseite rechts und einem Feld zur Auflistung bereits getätigter Anfragen. Sämtliche Ergebnisse sind hot-linked zur einfachen Zugänglichkeit vorheriger Anfragen. Jede Anfrage erzeugt je eine materialisierte Sicht, die für die Dauer der Session erhalten bleibt, so können Ergebnisse von vorherigen Anfragen leicht abgerufen werden.

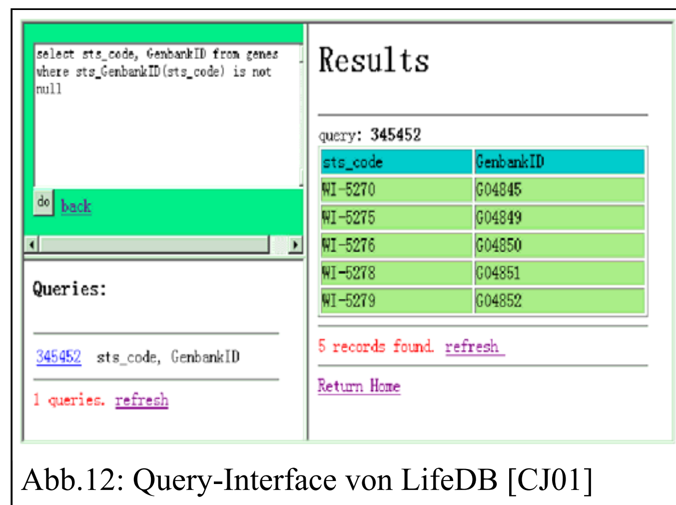


Abb.12: Query-Interface von LifeDB [CJ01]

Wie in Abb. 13 dargestellt, besteht die System-Architektur des LifeDB Interface aus mehreren Komponenten. Ziel dieser Architektur ist wie oben bereits erwähnt, daß die eingebundenen Funktionen weder in die Datenbank noch in das System des lokalen Clients eindringen.

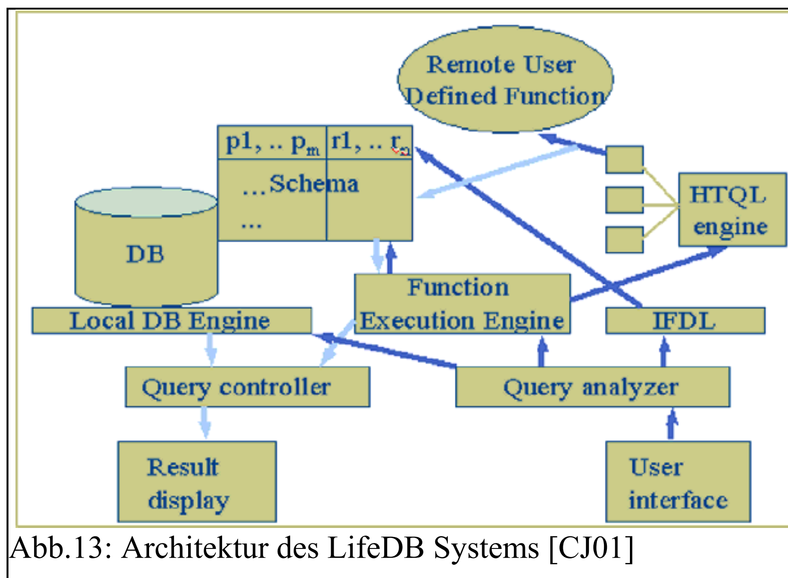


Abb.13: Architektur des LifeDB Systems [CJ01]

Sämtliche Eingaben in das Query-Eingabefeld, also sowohl Function Definitions als auch SQL-Anfragen mit oder ohne eingebundene Internet Funktionen durchlaufen zuerst den Query Analyzer. Hier erfolgt ein Parsen der Eingaben und Weiterleiten der Daten an die zuständigen Instanzen. Funktionsdefinitionen verfaßt in IFDL werden an das IFDL-Modul weitergeleitet, traditionelle SQL-Anfragen direkt an die Lokale DB Engine und SQL-Anfragen mit eingebetteten Internet Funktionen an die Function Execution Engine. Das IFDL Modul erzeugt anhand der Funktionsdefinition eine Metadatentabelle in der lokalen Datenbank. Diese wird von der Function Execution Engine benötigt, um die RUDF über die HTQL-Engine einzubinden und auszuführen. Die HTQL-Engine nimmt hier eine Schlüsselposition ein. Sie nutzt das weit verbreitete http zur Übermittlung der Eingabedaten an die Internet Funktion. Diese wird ausgeführt und legt das Ergebnis in einer HTML-Seite ab. Die HTQL-Engine extrahiert das Ergebnis anhand des HTQL-Wertes aus der vom IFDL erzeugten Metadaten-Tabelle im Datenbanksystem. Das Ergebnis wird in einer weiteren Tabelle in der Datenbank abgelegt. Der Query Controller führt die SQL-Anfrage aus und verwendet dabei die Ergebnistabelle aus der Ausführung der Internet Function. Schließlich wird das Ergebnis dem Nutzer präsentiert.
Dieser scheinbar komplexe Ablauf soll an einem Beispiel verdeutlicht werden.

5.2. Beispiel einer RUDF Ausführung

Man stelle sich folgendes Szenario vor: Es sollen Gensequenzen mit allen bereits bekannten Sequenzen der Kenianischen Fruchtfliege verglichen werden. Es sind nur solche Sequenzen von Interesse, die eine Ähnlichkeit von mindestens 98 % zu den unbekannten Sequenz aufweisen. Um diese Editierdistanz, bzw. das Maß der Ähnlichkeit zwischen zwei Gensequenzen zu berechnen, wird das über eine Internet Schnittstelle zugängliche Programm BLAST (http://ncbi.nlm.nih.gov/blast/blast.cgi) verwendet. Die entsprechende SQL-Anfrage sieht folgendermaßen aus [CJ01]:

```
SELECT b.sequence
FROM (SELECT get_seq( blast( a.sequence ))
        FROM local as a) as b
WHERE b.organism = “Drosophila” AND b.source(country) = “Kenia”
        AND b.e-value >= 0.98
```

get_seq und *blast* sind zwei Remote User Defined Functions. Die SQL-Anfrage führt nun folgende Operationen aus: Führe für alle Sequenzen der DB-Tabelle local (hier genannt a) die BLAST-Suche in GenBank durch. Speichere die Ergebnisse in die Tabelle b ab und gib alle Sequenzen aus, die zur Kenianischen Fruchtfliege gehören und einen Ähnlichkeits-Score von mehr als 0.98 haben. Die blast-Funktion verwendet als

Eingabe eine Sequenz der Tabelle local und gibt einen "requestID"-Wert zurück. Dieser wird von *get_seq* verwendet, um in GenBank die zugehörige Sequenz zu ermitteln. Das Ergebnis von *get_seq* ist eine Tabelle, die die ermittelten Ähnlichkeitsscores, Herkunft und Art enthält. Über diese Tabelle wird entsprechend der Bedingungen in der `WHERE`-Klausel selektiert und auf das Attribut *sequence* abgebildet.
Die beiden Internetfunktionen get_seq und blast müssen nun durch die Internet Function Definition Language definiert werden:

define function *blast*
href "http://www.ncbi.nlm.nih.gov/blast/Blast.cgi"
parameters *query* varchar(10000)
results *request_id* varchar(40)
htql value: <form>.<input>;

und

define function *get_seq*
href "http://www.ncbi.nlm.nih.gov/blast/Blast.cgi"
parameters rid varchar(40)
results *sequence* varchar(10000)
htql value: <pre>.<pre>;

Das Ergebnis der Funktion blast befindet sich nach Ausführung in der von Blast.cgi erzeugten HTML-Datei im Eingabefeld des ersten form-Tags. Dieser Wert wird extrahiert und dient als Parameter für *get_seq*, dessen Ergebnis nach dem zweiten pre-Tag der erzeugten HTML-Datei steht.
Der erste Schritt der Anfrage ist die IFDL Beschreibung von *blast* und *get_seq*, welche in das Textfeld des LifeDB-Interface eingegeben wird. Da es sich um IFDL Definitionen und nicht um Queries handelt, werden sie vom Query Analyzer zum IFDL Modul gesendet. Dort wird für jede Definition eine Tabelle mit den beschriebenen Metadaten erzeugt und in der Datenbank abgelegt. Der Nutzer erhält über die Ergebnisseite feedback über die erfolgreiche Bearbeitung der Funktionsdefinitionen. Nun kann die oben formulierte SQL-Anfrage eingegeben werden. Der Query Analyzer entscheidet sich nun für die Weiterleitung an die Function Execution Engine. Jetzt werden die Metadaten der beiden erzeugten Tabellen für *blast* und *get_seq* abgerufen, BLAST gestartet und die Ergebnisse mittels der HTQL Engine extrahiert und Ergebnistabellen in der Datenbank angelegt. Mit diesen Tabellen werden nun die üblichen SQL-Operationen Selektion nach den Bedingungen der `WHERE`-Klausel und anschließende Projektion auf das Attribut sequence durchgeführt.

6. Zusammenfassung und Diskussion

Die beschriebenen Ansätze sind äußerst unterschiedlicher Natur. Ziel ist es, heterogene Datenquellen bei Anfragen zu integrieren. Das traditionelle relationale DB Schema ist aufgrund der hohen Komplexität von biologischen Daten oft nicht anwendbar. Insofern sind auch in den Anfragesprachen, die auf SQL basieren, entsprechende Erweiterungen notwendig. Im ersten Abschnitt wurde gezeigt, daß mit gewissen Erweiterungen sogar Deduktion und Herleiten von Wissen mit Hilfe der Datenbasis möglich ist.

Formulargestützte Schnittstellen bieten gegenüber Anfragesprachen den großen Vorteil, daß auch Wissenschaftler ohne tiefere informatischen Kenntnisse auf sehr einfache Weise in der Lage sind, Informationen aus biologischen Datenbanken zu extrahieren. Hier muß jedoch unterschieden werden, ob die graphische Schnittstelle für eine einzige Datenbank entworfen wurde wie bei GPCRDB oder eine große Anzahl von Datenbanken abfragt. Beide Varianten haben sicher ihre Vor- und Nachteile. Bei SRS hat der Forscher die Möglichkeit, sehr globale Anfragen zu stellen und weltweit gespeichertes und aktuelles Wissen abzurufen. Allerdings lassen sich die Anfragen auch oft nicht präzise genug formulieren, um sehr spezielle Ergebnisse zu erhalten. Da liegt die Stärke bei GPCRDB, die in ihrem Bereich sehr eingeschränkte aber hochspezialisierte Forschungsdaten enthält, nämlich nur die der G-Protein gekoppelten Rezeptoren, und demzufolge mit einer formulargestützten Schnittstelle ausgestattet ist, die genau auf diese Domäne zugeschnitten ist.

SRS bietet ein Browser Interface und eine rudimentäre Anfragesprache für eine große Reihe von Datenquellen, jedoch werden im Gegensatz zu TAMBIS die Formate und Konventionen der zugrunde liegenden Datenbanken nicht verdeckt. Dies erschwert die Transparenz des Systems. TAMBIS dagegen stellt mit der verwendeten Description Logics ein semantisch korrektes und konsistentes System bei maximaler Transparenz für den Nutzer dar. Von den Datenquellen selbst wird abstrahiert und der Nutzer hat keinen Einfluß auf die ausgewählten Datenbanken, die die Anfrage betreffen.

Obwohl derzeit das Konzept der Remote User Defined Functions noch keine weit verbreitete Anwendung findet, ist es doch ein sehr vielversprechender Ansatz. Es gewährt hohe Flexibilität in bezug auf Einbindung von Analyse Instrumenten. Prinzipiell ist es möglich, jede beliebige Funktion einzusetzen, die über ein Web Interface zugänglich ist. Durch Verwendung des vom Internet genutzten http sind hierdurch keine Einschränkungen gegeben. Solche Funktionen können in ähnlicher Weise wie stored procedures des Datenbanksystems in SQL-Anweisungen eingebettet werden. Nachteilig ist aber, daß bei deren Verwendung ein Verändern dieser Routinen immer mit einem Neucompilieren verbunden ist. CGI-Scripte sind hier wesentlich flexibler. Weiterhin wird durch diesen Ansatz gewährleistet, daß die strenge Trennung zwischen Datenbank und Programmiersprache erhalten bleibt und so das oft diskutierte Impedance Mismatch umgangen wird.

Glossar

BLAST	In der Bioinformatik weit verbreitetes Analyseprogramm zum Vergleich zweier oder mehrerer Sequenzen (einschließlich RNA-, DNA-, Aminosäure-Sequenzen)
CPL	Collection Programming Language, funktionale Programmiersprache mit LISP-ähnlicher Syntax, mit deren Hilfe Anfragen innerhalb verschiedener Systeme (TAMBIS, BioKleisli [DO96]) erstellt und ausgewertet werden. CPL stellt besondere Datentypen zur satz- bzw. mengenorientierten Abarbeitung von Anfrageergebnissen zur Verfügung (List, Array, Set, ...)
DL	Description Logics. Sprache, die eine Ontologie beschreibt. Sie definiert in der Ontologie vorkommende Konzepte, Objekte und deren (semantisch sinnvollen) Rollen und Beziehungen. → GRAIL
GRAIL	DL der TAMBIS Ontologie
HTQL	Hypertext Query Language; Sprache, mit der man eine Suche in Web-Dokumenten realisieren kann. Die Navigation erfolgt hierbei durch Angabe von Tag-Pfaden.
IF	Internet Functions; Programme mit Web-Interface, die interaktive Datenanalyse bzw. Verarbeitung durchführen können
IFDL	Internet Function Definition Language; Sprache, die zur Definition von →RUDFs verwendet wird. Diese Definition enthält Daten über Art der Ein- und Ausgabe sowie dir URL des Programmes.
RUDF	Remote User Defined Functions; Funktionen, die über das Internet aufgerufen und in SQL-Anfragen eingebunden werden können.
View / Sicht	Eine durch SQL erzeugte Tabelle, die eine gewisse Sicht der DB-Daten darstellt.

Literatur

[Ace] AceDB Query Language: http://www.acedb.org/Software/whelp/AQL/

[BG99] Baker, P. G., Goble, C. A. et al: An Ontology For Bioinformatics Applications. Bioinformatics, Vol. 15 No. 6, 510-520, 1999

[BGP99] Brass, A., Goble, C., Paton, N.: TAMBIS: Transparent Access to Bioinformatics Information Sources. http://www.cs.man.ac.uk/~tambis

[Bo95] Borgida, A.: Description Logics in Data Management. IEEE Trans Knowledge and Data Engeneering, 7: 671-782, 1995

[CC99] Che, D., Chen, Y., Aberer, K.: A Query System in a Biological Database. Proceedings 11th International Conference on Scientific and Statistical Management 1999: 158-167

[CJ01] Chen, L.; Jamil, H. M.: Supporting Remote User Defined Functions in Heterogeneous Biological Databases. Proceedings IEEE International Conference on BIBE 2001: 144-152, 2001

[DO96] Davidson, S., Overton, C. et al.: BioKleisli: A Digital Library for Biomedical Researchers. Dept. of Computer and Information Science, University of Pennsylvania, 1996.

[EC] EcoCyc Database. http://www.ecocyc.org/

[FJ89] Freire, Juliana: Practical problems in coupling deductive engines with relational databases. Proceedings of the 5th KRDB Workshop: 11-1…11-7, 1998

[GO] Gene OntologyTM Consortium. http://www.geneontology.org/

[HE99] Hansen, D. ,Etzold, T. et al: Cross Database Visualization. 1999

[Ja00] Jamil, H. M.: PQL : A Reasonable Complex SQL for Genomic Databases. Proceedings 1st IEEE International Symposium on Bioinformatics and Biomedical Engineering (BIBE’ 2000): 50-59

[LION] LION Bioscience SRS 6.1 Documentation. LION Bioscience AG. 2001

[MBO] MBO Java ontology browser: http://igd.rz-berlin.mpg.de/~www/oe/mbo.html

[MS02] Mork, P., Shaker, R. et al.: {PQL}: A Declarative Query Language over Dynamic Biological Data. Proceedings American Medical Informatics Association (AMIA) Anual Symposium, 2002.

[PS99] Paton, N., Stevens, R. et al. :Query Processing in the TAMBIS Bioinformatics Source Integration System.1999. http://www.cs.man.ac.uk/~tambis

[Sk01] Skylar, N.: Survey of existing Bio-Ontologies. IfI, Universität Leipzig, 2001

[SM99] Stokes, A.; Matsuda, H.; Hashimoto, A.:Making High-level Queries on Diverse Genome Data: A Structured Genome Document Database System Based on GXML and GQL.1999 http://www.genome.ad.jp/manuscripts/GIW99/Oral/GIW99F18.pdf.

Abbildungsverzeichnis

Abb. 4 ER-Modell und student Datenbank-Schema. [Ja00].

Abb. 5 Ontologie-repräsentierende Sprachen. [Sk01]

Abb. 6 TAMBIS Architektur. [BGP99]

Abb. 7 Höhere Ebenden der Konzepte in TaO. [BG99]

Abb. 8 Komposition, Vererbung, Sanktionierung. [BG99]

Abb. 9 Query Builder des TAMBIS Systems. [BGP99]

Abb. 10 SRS, Auswahl der Datenbanken. [LION, manual]

Abb. 11 SRS, Einbinden von Analyseprogrammen

Abb. 12 Query-Interface von LifeDB. [CJ01]

Abb. 13 Architektur des LifeDB-Systems. [CJ01]