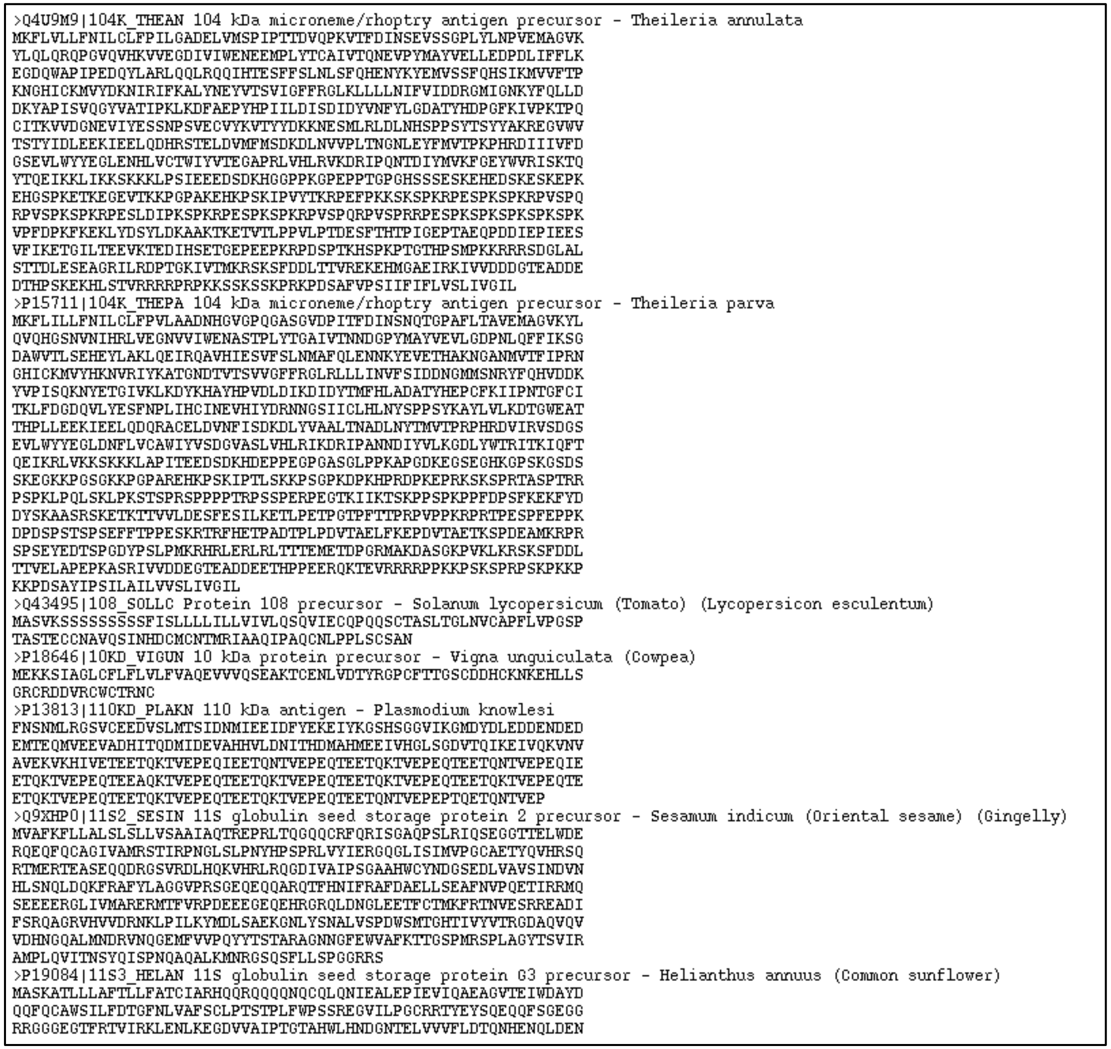
Introduction to Database Systems, Data Modeling and SQL

Leandro Hermida
EMBnet
August 27, 2007

- Structured vs. Unstructured Data
  - Flat file = unstructured data
  - Database = structured data
- The Problem with Unstructured Data
  - High maintenance costs
  - Data Redundancy: the same data will be represented multiple times in the file
  - Data dependence: if you change things about the file format then there will be expensive changes to programs that use it
  - Ensuring data consistency and controlling access to the data is difficult (i.e. you cannot finely control multi-user access to the file)

• A Simple Flat File Example: FASTA Sequence Files

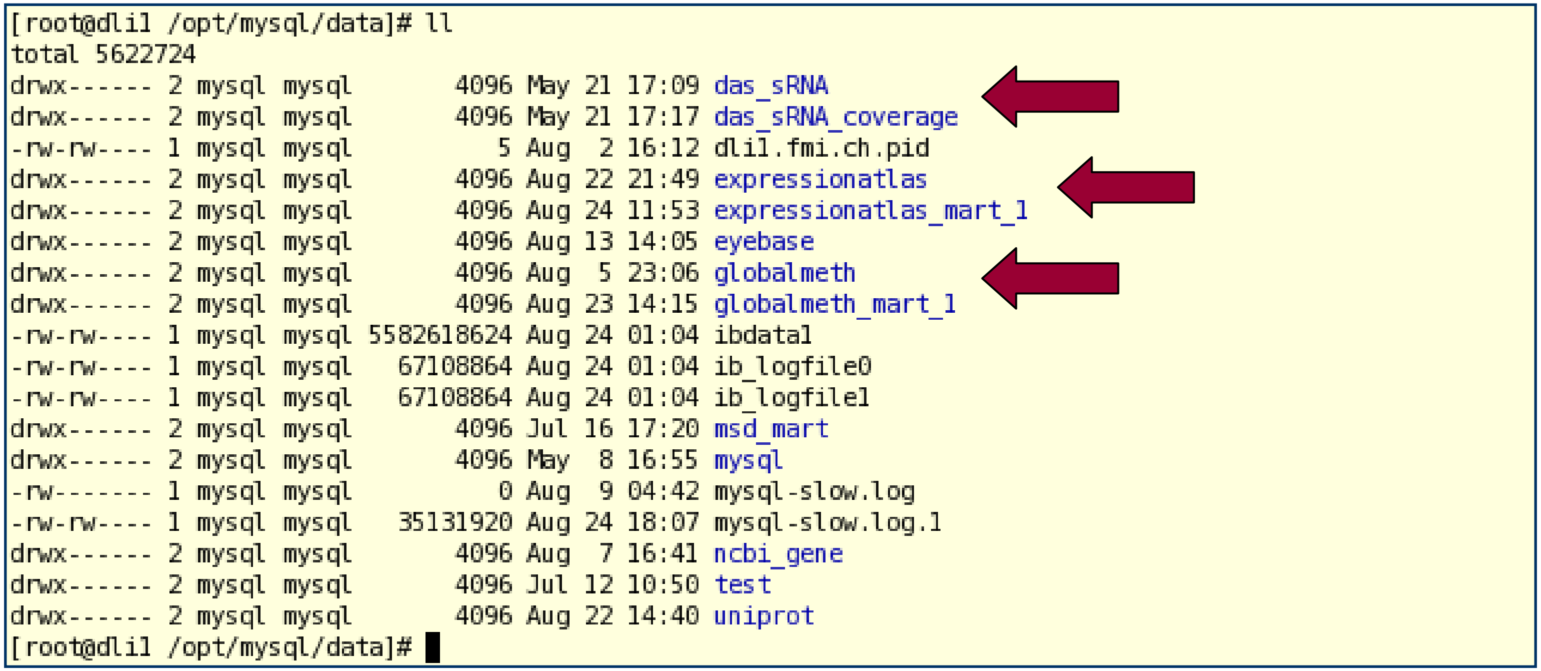


```
>Q4U9M9|104K_THEAN 104 kDa microneme/rhoptry antigen precursor - Theileria annulata
MKFLVLLFNILCLFPILGADELVMSPIPTTDVQPKVTFDINSEVSSGPLYLNPVEMAGVK
YLQLQRQPGVQVHKVVEGDIVIWENEEMPLYTCAIVTQNEVPYMAYVELLEDPDLIFFLK
EGDQWAPIPEDQYLARLQQLRQQIHTESFFSLNLSFQHENYKYEMVSSFQHSIKMVVFTP
KNGHICKMVYDKNIRIFKALYNEYVTSVIGFFRGLKLLLLNIFVIDDRGMIGNKYFQLLD
DKYAPISVQGYVATIPKLKDFAEPYHPIILDISDIDYVNFYLGDATYHDPGFKIVPKTPQ
CITKVVDGNEVIYESSNPSVECVYKVTYYDKKNESMLRLDLNHSPPSYTSYYAKREGVWV
TSTYIDLEEKIEELQDHRSTELDVMFMSDKDLNVVPLTNGNLEYFMVTPKPHRDIIIVFD
GSEVLWYYEGLENHLVCTWIYVTEGAPRLVHLRVKDRIPQNTDIYMVKFGEYWVRISKTQ
YTQEIKKLIKKSKKKLPSIEEEDSDKHGGPPKGPEPPTGPGHSSSESKEHEDSKESKEPK
EHGSPKETKEGEVTKKPGPAKEHKPSKIPVYTKRPEFPKKSKSPKRPESPKSPKRPVSPQ
RPVSPKSPKRPESLDIPKSPKRPESPKSPKRPVSPQRPVSPRRPESPKSPKSPKSPKSPK
VPFDPKFKEKLYDSYLDKAAKTKETVTLPPVLPTDESFTHTPIGEPTAEQPDDIEPIEES
VFIKETGILTEEVKTEDIHSETGEPEEPKRPDSPTKHSPKPTGTHPSMPKKRRRSDGLAL
STTDLESEAGRILRDPTGKIVTMKRSKSFDDLTTVREKEHMGAEIRKIVVDDDGTEADDE
DTHPSKEKHLSTVRRRRPRPKKSSKSSKPRKPDSAFVPSIIFIFLVSLIVGIL
>P15711|104K_THEPA 104 kDa microneme/rhoptry antigen precursor - Theileria parva
MKFLILLFNILCLFPVLAADNHGVGPQGASGVDPITFDINSNQTGPAFLTAVEMAGVKYL
QVQHGSNVNIHRLVEGNVVIWENASTPLYTGAIVTNNDGPYMAYVEVLGDPNLQFFIKSG
DAWVTLSEHEYLAKLQEIRQAVHIESVFSLNMAFQLENNKYEVETHAKNGANMVTFIPRN
GHICKMVYHKNVRIYKATGNDTVTSVVGFFRGLRLLLINVFSIDDNGMMSNRYFQHVDDK
YVPISQKNYETGIVKLKDYKHAYHPVDLDIKDIDYTMFHLADATYHEPCFKIIPNTGFCI
TKLFDGDQVLYESFNPLIHCINEVHIYDRNNGSIICLHLNYSPPSYKAYLVLKDTGWEAT
THPLLEEKIEELQDQRACELDVNFISDKDLYVAALTNADLNYTMVTPRPHRDVIRVSDGS
EVLWYYEGLDNFLVCAWIYVSDGVASLVHLRIKDRIPANNDIYVLKGDLYWTRITKIQFT
QEIKRLVKKSKKKLAPITEEDSDKHDEPPEGPGASGLPPKAPGDKEGSEGHKGPSKGSDS
SKEGKKPGSGKKPGPAREHKPSKIPTLSKKPSGPKDPKHPRDPKEPRKSKSPRTASPTRR
PSPKLPQLSKLPKSTSPRSPPPPTRPSSPERPEGTKIIKTSKPPSPKPPFDPSFKEKFYD
DYSKAASRSKETKTTVVLDESFESILKETLPETPGTPFTTPRPVPPKRPRTPESPFEPPK
DPDSPSTSPSEFFTPPESKRTRFHETPADTPLPDVTAELFKEPDVTAETKSPDEAMKRPR
SPSEYEDTSPGDYPSLPMKRHRLERLRLTTTEMETDPGRMAKDASGKPVKLKRSKSFDDL
TTVELAPEPKASRIVVDDEGTEADDEETHPPEERQKTEVRRRRPPKKPSKSPRPSKPKKP
KKPDSAYIPSILAILVVSLIVGIL
>Q43495|108_SOLLC Protein 108 precursor - Solanum lycopersicum (Tomato) (Lycopersicon esculentum)
MASVKSSSSSSSSSFISLLLLILLVIVLQSQVIECQPQQSCTASLTGLNVCAPFLVPGSP
TASTECCNAVQSINHDCMCNTMRIAAQIPAQCNLPPLSCSAN
>P18646|10KD_VIGUN 10 kDa protein precursor - Vigna unguiculata (Cowpea)
MEKKSIAGLCFLFLVLFVAQEVVVQSEAKTCENLVDTYRGPCFTTGSCDDHCKNKEHLLS
GRCRDDVRCWCTRNC
>P13813|110KD_PLAKN 110 kDa antigen - Plasmodium knowlesi
FNSNMLRGSVCEEDVSLMTSIDNMIEEIDFYEKEIYKGSHSGGVIKGMDYDLEDDENDED
EMTEQMVEEVADHITQDMIDEVAHHVLDNITHDMAHMEEIVHGLSGDVTQIKEIVQKVNV
AVEKVKHIVETEETQKTVEPEQIEETQNTVEPEQTEETQKTVEPEQTEETQNTVEPEQIE
ETQKTVEPEQTEEAQKTVEPEQTEETQKTVEPEQTEETQKTVEPEQTEETQKTVEPEQTE
ETQKTVEPEQTEETQKTVEPEQTEETQKTVEPEQTEETQNTVEPEPTQETQNTVEP
>Q9XHP0|11S2_SESIN 11S globulin seed storage protein 2 precursor - Sesamum indicum (Oriental sesame) (Gingelly)
MVAFKFLLALSLSLLVSAAIAQTREPRLTQGQQCRFQRISGAQPSLRIQSEGGTTELWDE
RQEQFQCAGIVAMRSTIRPNGLSLPNYHPSPRLVYIERGQGLISIMVPGCAETYQVHRSQ
RTMERTEASEQQDRGSVRDLHQKVHRLRQGDIVAIPSGAAHWCYNDGSEDLVAVSINDVN
HLSNQLDQKFRAFYLAGGVPRSGEQEQQARQTFHNIFRAFDAELLSEAFNVPQETIRRMQ
SEEEERGLIVMARERMTFVRPDEEEGEQEHRGRQLDNGLEETFCTMKFRTNVESRREADI
FSRQAGRVHVVDRNKLPILKYMDLSAEKGNLYSNALVSPDWSMTGHTIVYVTRGDAQVQV
VDHNGQALMNDRVNQGEMFVVPQYYTSTARAGNNGFEWVAFKTTGSPMRSPLAGYTSVIR
AMPLQVITNSYQISPNQAQALKMNRGSQSFLLSPGGRRS
>P19084|11S3_HELAN 11S globulin seed storage protein G3 precursor - Helianthus annuus (Common sunflower)
MASKATLLLAFTLLFATCIARHQQRQQQQNQCQLQNIEALEPIEVIQAEAGVTEIWDAYD
QQFQCAWSILFDTGFNLVAFSCLPTSTPLFWPSSREGVILPGCRRTYEYSQEQQFSGEGG
RRGGGEGTFRTVIRKLENLKEGDVVAIPTGTAHWLHNDGNTELVVVFLDTQNHENQLDEN
```

- Why Databases?
  - Data constitute an organizational asset: We want integrated control
    - Reduction of redundancy
    - Avoidance of inconsistency
    - Sharability
    - Standards
    - Improved security
    - Data integrity
  - Programmer productivity: More data independence
  - Flat files should be used for *data exchange* between databases.

- A Simple Database Structure: ISAM (Index Sequential Access Method)
  - As in a flat file, data records are stored sequentially
  - One data file for each “table” of data
  - Data records are composed of fixed length fields
  - Hash table files are the indexes containing pointers into the data files which allow individual records to be retrieved without having to search the entire file
  - Excellent read access performance
  - Still many limitations:
    - No support for transactions
    - No referential integrity constraints
    - Not very good concurrency (i.e. entire table must be locked when a user seeks to change a record)

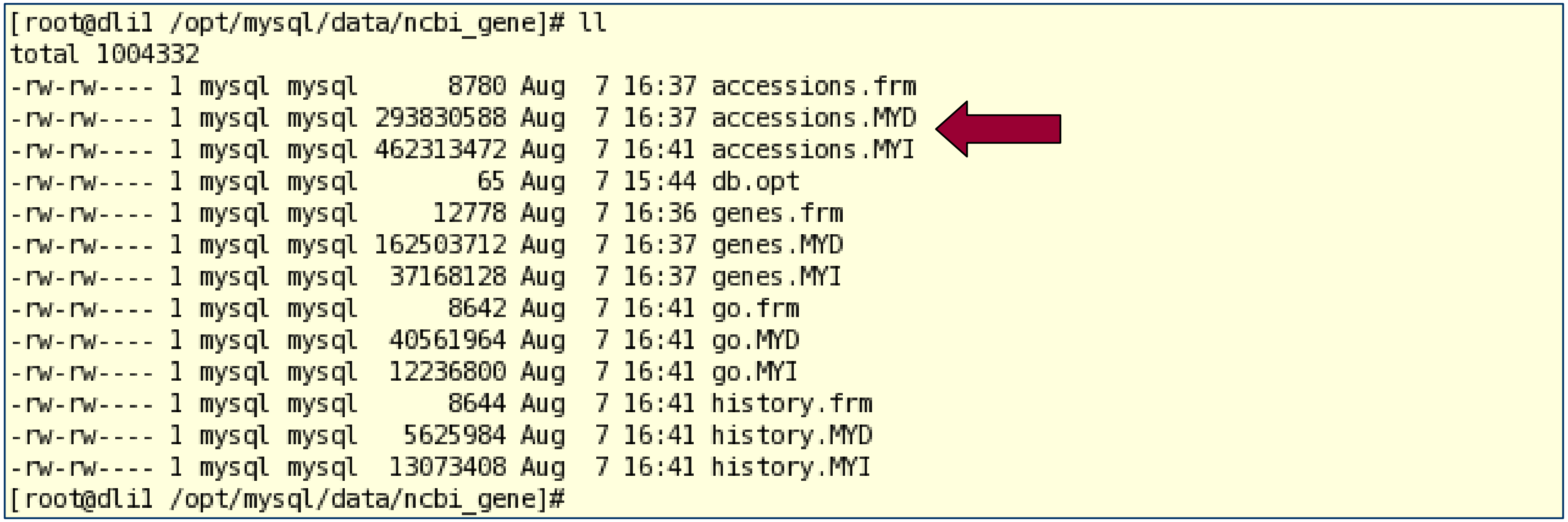
- ## Data Storage Example: MySQL
  - Default storage engine of MySQL is MyISAM, an extended form of ISAM
  - On a MySQL server in the data directory you will see a folder for each database:



```
[root@dli1 /opt/mysql/data]# ll
total 5622724
drwx------ 2 mysql mysql       4096 May 21 17:09 das_sRNA
drwx------ 2 mysql mysql       4096 May 21 17:17 das_sRNA_coverage
-rw-rw---- 1 mysql mysql          5 Aug  2 16:12 dli1.fmi.ch.pid
drwx------ 2 mysql mysql       4096 Aug 22 21:49 expressionatlas
drwx------ 2 mysql mysql       4096 Aug 24 11:53 expressionatlas_mart_1
drwx------ 2 mysql mysql       4096 Aug 13 14:05 eyebase
drwx------ 2 mysql mysql       4096 Aug  5 23:06 globalmeth
drwx------ 2 mysql mysql       4096 Aug 23 14:15 globalmeth_mart_1
-rw-rw---- 1 mysql mysql 5582618624 Aug 24 01:04 ibdata1
-rw-rw---- 1 mysql mysql   67108864 Aug 24 01:04 ib_logfile0
-rw-rw---- 1 mysql mysql   67108864 Aug 24 01:04 ib_logfile1
drwx------ 2 mysql mysql       4096 Jul 16 17:20 msd_mart
drwx------ 2 mysql mysql       4096 May  8 16:55 mysql
-rw------- 1 mysql mysql          0 Aug  9 04:42 mysql-slow.log
-rw-rw---- 1 mysql mysql   35131920 Aug 24 18:07 mysql-slow.log.1
drwx------ 2 mysql mysql       4096 Aug  7 16:41 ncbi_gene
drwx------ 2 mysql mysql       4096 Jul 12 10:50 test
drwx------ 2 mysql mysql       4096 Aug 22 14:40 uniprot
[root@dli1 /opt/mysql/data]# █
```

Data Storage Example: MySQL

- Inside each database folder you will see the a set of the following file types per table:
  - *.MYD are the data files
  - *.MYI are the hash table index files



```
[root@dli1 /opt/mysql/data/ncbi_gene]# ll
total 1004332
-rw-rw---- 1 mysql mysql      8780 Aug  7 16:37 accessions.frm
-rw-rw---- 1 mysql mysql 293830588 Aug  7 16:37 accessions.MYD
-rw-rw---- 1 mysql mysql 462313472 Aug  7 16:41 accessions.MYI
-rw-rw---- 1 mysql mysql        65 Aug  7 15:44 db.opt
-rw-rw---- 1 mysql mysql     12778 Aug  7 16:36 genes.frm
-rw-rw---- 1 mysql mysql 162503712 Aug  7 16:37 genes.MYD
-rw-rw---- 1 mysql mysql  37168128 Aug  7 16:37 genes.MYI
-rw-rw---- 1 mysql mysql      8642 Aug  7 16:41 go.frm
-rw-rw---- 1 mysql mysql  40561964 Aug  7 16:41 go.MYD
-rw-rw---- 1 mysql mysql  12236800 Aug  7 16:41 go.MYI
-rw-rw---- 1 mysql mysql      8644 Aug  7 16:41 history.frm
-rw-rw---- 1 mysql mysql   5625984 Aug  7 16:41 history.MYD
-rw-rw---- 1 mysql mysql  13073408 Aug  7 16:41 history.MYI
[root@dli1 /opt/mysql/data/ncbi_gene]#
```

- **What is data modeling?**
  - An information system typically consists of a database (contained stored data) together with programs that capture, store, manipulate, and retrieve the data.
    - Process Model => the programs
    - Data Model => the database
  - Definition (from Wikipedia.org):
    - Data modeling is the analysis and design of the information in the system, concentrating on logical entities and the logical dependencies between these entities. Data modeling is an abstraction activity in that the details of the values of individual data observations are ignored in favor of overall structure, relationships, names and formats of the data of interest. The data model should not only define the data structure, but also what the data actually means (semantics).

- Database Design

  1. Planning and Analysis
  2. Conceptual Design
  3. Logical Design
  4. Physical Design
  5. Implementation

- Why is the data model so important?
  - We generally settle for a design that “does the job” even though we recognize that with more time and effort we might be able to develop a more elegant solution.
  - Reasons:
    - LEVERAGE
      - A small change in the data model may have a major impact on the system as a whole.
      - Programs and applications using the database are heavily influenced by the database design (i.e. the data model).
      - A well-designed data model can make the development of programs and applications simpler and easier.
      - Poor data organization can be very expensive to fix.

- Why is the data model so important?
  - Reasons (cont’d):
    - CONCISENESS
      - A data model is a very powerful tool for expressing the heart of the information systems requirements and capabilities.
    - DATA QUALITY
      - Data held in a database is almost always a valuable asset that is built up over time.
      - Poor data quality (inaccurate data) reduces the asset value and can be expensive or impossible to correct.
      - The data model is important in achieving good data quality because it defines how the data is to be organized and interpreted.

- What makes a good data model?
  - How does this model support the sound overall system design and meets the business requirements?
  - Quality Criteria:
    - COMPLETENESS
      - Does the model support all the necessary data?
    - NONREDUNDANCY
      - Does the model specify a database in which the same fact could be recorded more than once?
    - ENFORCEMENT OF BUSINESS RULES
      - How accurately does the model reflect and enforce the rules that need to apply to the data?

- What makes a good data model?
  - Quality Criteria (cont'd):
    - DATA REUSABILITY
      - Will the data stored in the database be reusable for purposes beyond those anticipated in the present process model?
      - If data has been organized with one particular application in mind, it is often difficult to use for other purposes.
    - STABILITY AND FLEXIBILITY
      - How well will the model cope with possible changes to the business requirements?
      - A data model is stable when, in the face of change to requirements, we do not need to modify it.
      - A data model is flexible when it can be readily extended to accommodate new requirements with minimal impact on the existing structure.

- What makes a good data model?
  - Quality Criteria (cont'd):
    - ELEGANCE
      - Does the data model provide a reasonably neat and simple classification of the data?
    - COMMUNICATION
      - Does the data model represent the concepts that the users who are familiar with it and can easily verify?
      - Will programmers interpret the model correctly?
    - INTEGRATION
      - How will the database fit into the organization's existing databases? How many other databases hold similar data?

- Performance
  - Why is performance not in the list of quality criteria?
    - Performance differs from our other criteria because it depends heavily on the software and hardware platforms on which the database will run.
    - Exploiting the capabilities of the hardware and software is a technical task quite different from the modeling activity.
    - The usual and recommended procedure is to develop the data model without considering performance, then attempt to implement it with the available hardware and software.

- Database Design Stages
  - Conceptual Data Model
    - A technology-independent specification of the data to be held in the database. Usually represented as a straightforward diagram with supporting documentation.
  - Logical Data Model
    - Translation of the conceptual model into structures that can be implemented using a DBMS. This usually means the models specifies tables and columns.
  - Physical Data Model
    - The logical model incorporating any changes necessary to achieve adequate performance and is also presented in terms of tables and columns, together with a specification of the physical storage.

- Where do data models fit in?
  - Process-Driven Approaches
    - Identifying all of the processes and the data the each requires.
    - Designing the data model to support this fairly precise set of data requirements.
  - Data-Driven Approaches
    - Emphasis on developing the data model *BEFORE* the detailed process model to achieve:
      - Reusability of data
      - Establishment of a set of consistent names and definitions for data
      - Automatic generation of a significant part of the process model

- Where do data models fit in? (cont'd)
  - Parallel (Blended) Approaches
    - In practice, you usually don't encounter a purely data-driven or process driven approach - you do both.
  - Object-Oriented Approaches
    - Many information systems (including bioinformatics ones) remain intrinsically data-centric.
    - True object-oriented DBMSs are not widely used, but there is ORM (object relational mapping) which provides the bridge.
    - Data associated with object-oriented applications are typically stored in conventional or extended relational databases.

- Summary
  - Data and databases are central to information systems and bioinformatics.
  - The data model is a crucial determinant of the design of the associated applications and systems which use it.
  - Data modeling is not optional -- no database was ever built without a model.
  - Data modeling is a *design* process -- there can be more than one candidate model that is “correct” based on analysis, past experience, creativity.
  - Quality criteria must be assessed and there are often trade-offs involved in order to satisfy these criteria.
  - Data and process modeling usually happen in parallel.
  - Object-oriented development still requires the same data modeling principles when dealing with structured data held in a database.

- Basics of Sound Structure
  - Normalization
- Conceptual Design
  - Entity-Relationship (E-R) Approach
  - A Diagrammatic Representation
  - Modeling Using a Top-Down Approach
    - Entity Classes
    - Relationships
    - Attributes
    - Keys
    - Generalization Hierarchies (Subtypes and Supertypes)
    - Integrity Rules

- Logical Design
  - Mapping the E-R Model to the Relational Model
  - Normalization
- Physical Design
  - Data, relationship and constraint definition using SQL DDL

- Basics of Sound Structure: Normalization
  - Relational basis: two dimensional tables of rows and columns linked together through relationships
  - A definition of database normalization
    - Normalization is a set of rules for allocating data into tables in such a way as to eliminate certain types of redundancy and incompleteness.
- An Example: NCBI *Gene*

Gene ID Gene Symbol Gene Name Chromosome ID Chromosome Name Chromosome Location	**26972** **SPO11** **sporulation protein, meiosis-specific, SPO11 homolog (S. cerevisiae)** **NC_000068** **2** **2 H4**
Related Sequence ID	Type
AF443910	**genomic**
AF126400	**mRNA**
AF149309	**mRNA**
AF163053	**mRNA**

- ## Basics of Sound Structure: Normalization
  - Step 1: Put data into two-dimensional tabular form
    - One fact per column
    - Hidden data
      - make sure we haven’t lost any data in the translation to tabular form
    - Derivable data
    - Name table
    - Determine the *primary key*
      - A minimal set of columns whose value uniquely identify ONE row of the table

Gene Table

<u>Gene ID</u>	Gene Symbol	Gene Name	Chromosome ID	Chromosome Name	Chromosome Location	Related Sequence ID 1	Related Sequence Type 1	Related Sequence ID 2	Related Sequence Type 2
26972	SPO11	sporulation protein, meiosis-specific, SPO11 homolog (S. cerevisiae)	NC_000068	2	2 H4	AF443910	Genomic	AF126400	mRNA
140579	ELMO2	engulfment and cell motility 2, ced-12 homolog (C. elegans)	NC_000068	2	2 H3	AL591430	Genomic	AK035710	mRNA
24088	TLR2	toll-like receptor 2	NC_000069	3	3 F1	AF124741	mRNA	AF165189	mRNA
23856	DIDO1	death inducer-obliterator 1	NC_000068	2	2 H4	AK032843	mRNA	AK044919	mRNA

- ## Basics of Sound Structure: Normalization

  – Step 2: Put tabular data into First Normal Form (1NF)

  - Identify and move repeating groups to new tables

    1. Remove each separate set of repeating group columns to a new table (one new table for each set) so that each occurrence of the group becomes a row in its new table.
    2. Include the primary key of the original table in each new table as a cross reference (*foreign key*).
    3. Name each new table.
    4. Identify the *primary key* of each new table.

Gene Table

Gene ID	Gene Symbol	Gene Name	Chromosome ID	Chromosome Name	Chromosome Location
26972	SPO11	sporulation protein, meiosis-specific, SPO11 homolog (S. cerevisiae)	NC_000068	2	2 H4
140579	ELMO2	engulfment and cell motility 2, ced-12 homolog (C. elegans)	NC_000068	2	2 H3
24088	TLR2	toll-like receptor 2	NC_000069	3	3 F1
23856	DIDO1	death inducer-obliterator 1	NC_000068	2	2 H4

Related Sequence Table

Related Sequence ID	Related Sequence Type	Gene ID
AF443910	Genomic	26972
AF126400	mRNA	26972
AL591430	Genomic	140579
AK035710	mRNA	140579
AF124741	mRNA	24088
AF165189	mRNA	24088
AK032843	mRNA	23856
AK044919	mRNA	23856

- ## Basics of Sound Structure: Normalization
  - Step 2: Put tabular data into Third Normal Form (3NF) (via 2NF)
    - Eliminate redundancy by identifying determinants moving determined sets to new tables
      1. Identify any determinants other than the (entire) primary key and the columns they determine.
      2. Establish a separate table for each determinant and the columns it determines. The determinant becomes the key of the new table.
      3. Name the new tables.
      4. Remove only the determined columns for the original table. Leave the determinant columns as the cross reference between the original and new tables (*foreign key*).

Gene Table

Gene ID	Gene Symbol	Gene Name	Chromosome Location	Chromosome ID
26972	SPO11	sporulation protein, meiosis-specific, SPO11 homolog (S. cerevisiae)	2 H4	NC_000068
140579	ELMO2	engulfment and cell motility 2, ced-12 homolog (C. elegans)	2 H3	NC_000068
24088	TLR2	toll-like receptor 2	3 F1	NC_000069
23856	DIDO1	death inducer-obliterator 1	2 H4	NC_000068

Chromosome Table

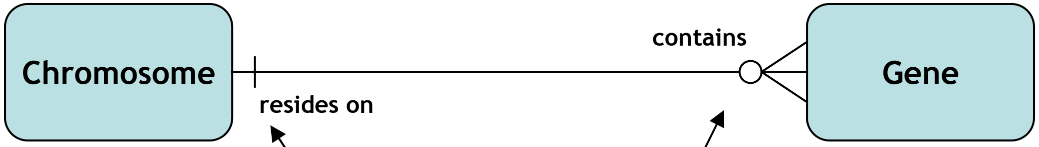
Chromosome ID	Chromosome Name
NC_000068	2
NC_000069	3

Related Sequence Table

Related Sequence ID	Related Sequence Type	Gene ID
AF443910	Genomic	26972
AF126400	mRNA	26972
AL591430	Genomic	140579
AK035710	mRNA	140579
AF124741	mRNA	24088
AF165189	mRNA	24088
AK032843	mRNA	23856
AK044919	mRNA	23856

- Conceptual Design
  - Entity-Relationship (E-R) Approach
  - A Diagrammatic Representation
  - Modeling Using a Top-Down Approach
    - Entity Classes
    - Relationships
    - Attributes
    - Keys
    - Generalization Hierarchies (Subtypes and Supertypes)
    - Integrity Rules

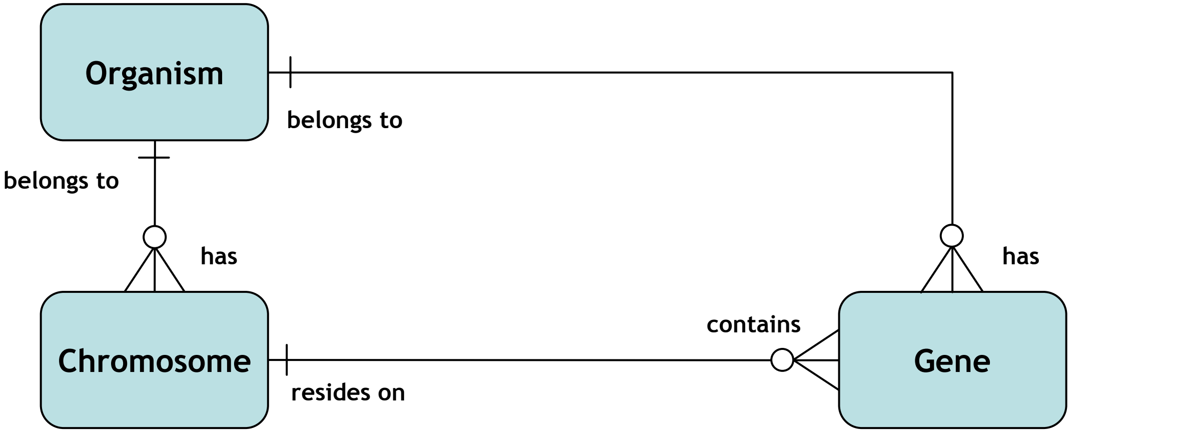
- Entity-Relationship (E-R) Model
  - Proposed by Peter Chen from MIT in 1976 in the paper, “The entity-relationship model—towards a unified view of data”.
- E-R Diagrammatic Representation
  - Box = Table
  - Line = Foreign-To-Primary Key Connection
  - “Crow’s Foot” = Foreign Key Side of Connection
  - Optionality
    - Pipe “|” means relationship is mandatory
    - Circle “O” means relationship is optional



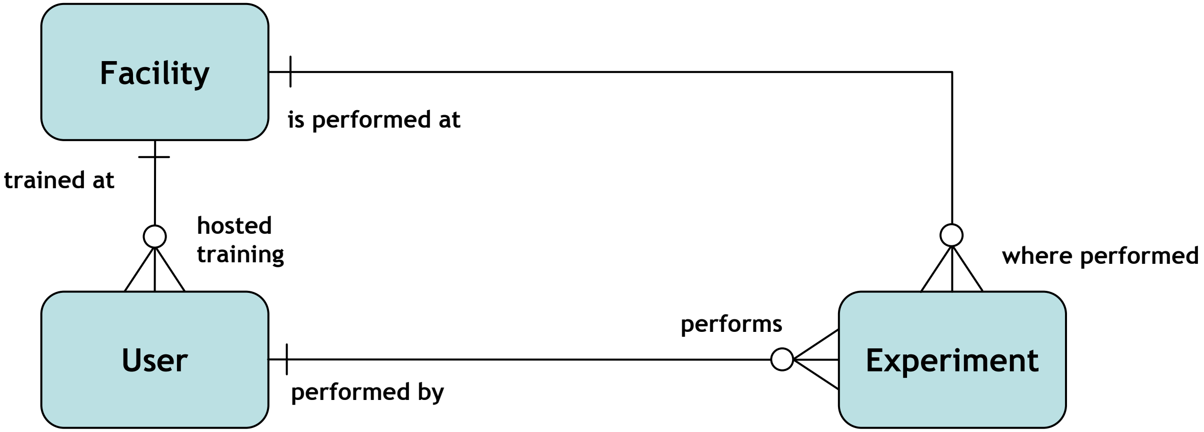
Each gene *must* reside on a chromosome.

Each chromosome *may* contain genes.

- E-R Diagrammatic Representation
  - Redundant Relationships



NOT REDUNDANT because relationship describes two different things

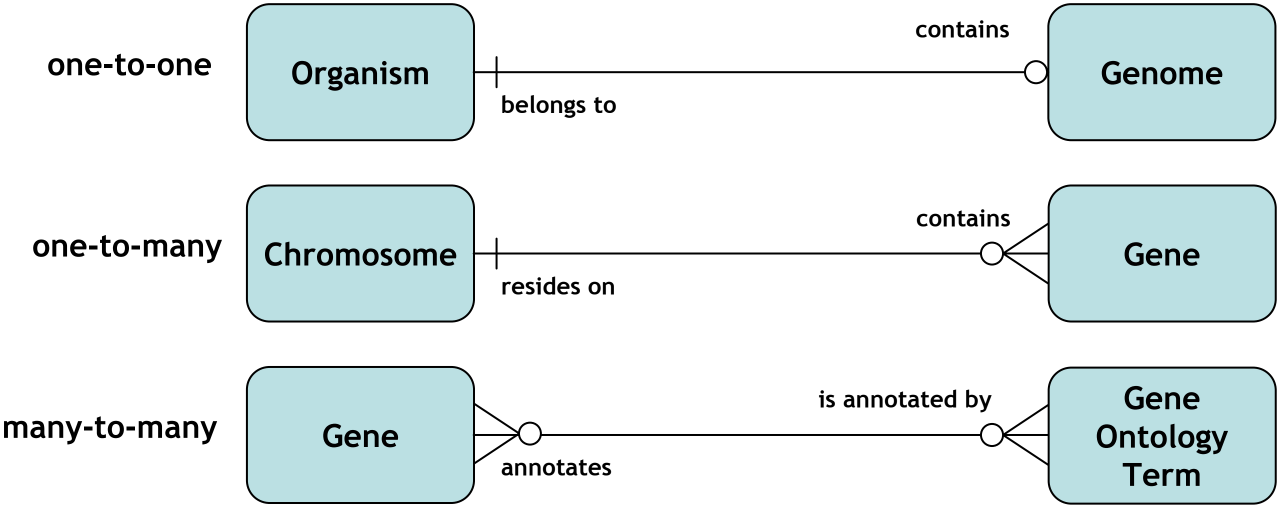


- E-R Modeling Using a Top-Down Approach
  - Entity Classes
    - Categories of things of interest to the business, represented by boxes on the diagram and generally implemented as tables in the logical modeling phase.
  - Relationships
    - How entities are related to each other, represented by lines with “crow’s feet” in the conceptual model diagram and generally implemented through the use of foreign keys in the logical model.
  - Attributes
    - What we want to know about the entity classes; not usually shown in the conceptual model diagram and generally implemented as columns in tables in the logical modeling phase.

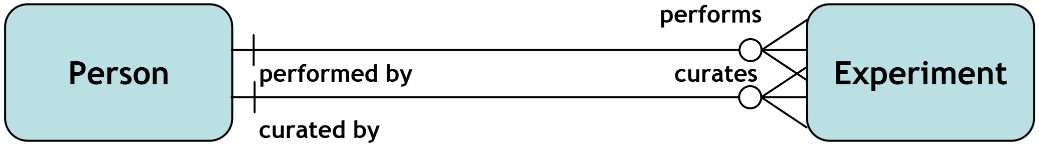
- Entity Classes
  - A class of things we want to keep information about.
  - Naming
    - The name must be in the singular and refer to a single instance of that class
      - *Sample* or *Gene* rather than Samples, Genes, Sample Record, or Sample Table
    - Do not name the entity class after the “most important” attribute
      - Use *Gene* instead of Gene Symbol
    - Avoid naming the entity class a name that reflects only a subset of what it really is
      - Use *Transcript* instead of mRNA

- Entity Classes
  - Naming (cont’d)
    - Avoid naming the entity class by adding a prefix to the name of another
      - External Experiment where we already have *Experiment*
    - Avoid abbreviating unnecessarily
      - Abbreviations tend to be inconsistent (same abbreviation could be used for different words)
      - Remember the “Communication” quality criteria! We need to prevent confusion.
  - Definitions
    - Provide guidance on the correct use of the resulting database
    - Prevent incorrect assumptions about what the tables contain

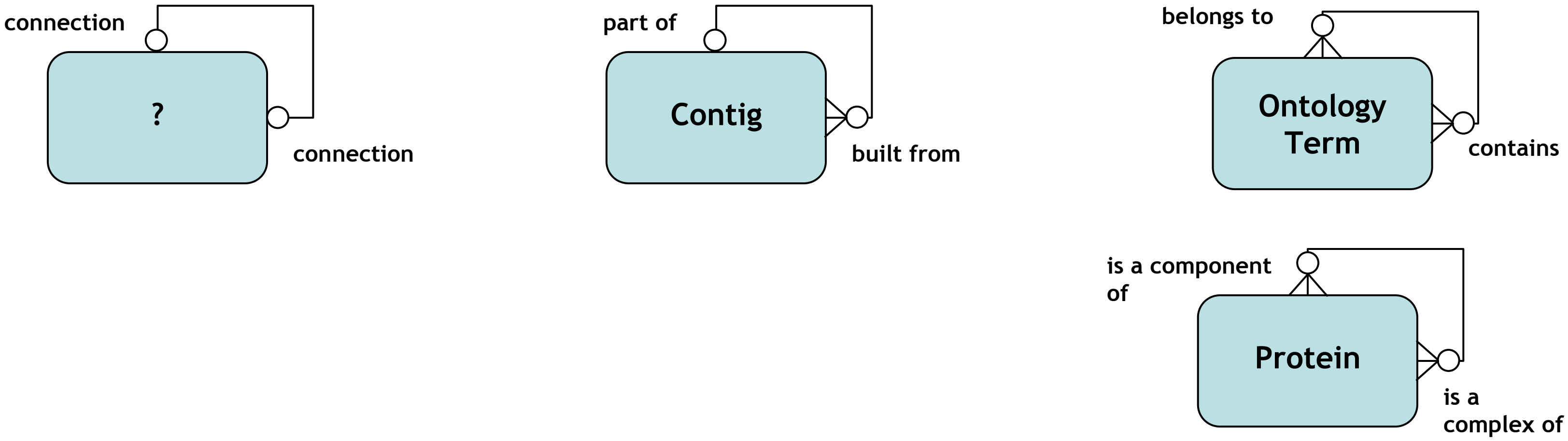
- **Relationships**
  - Meaning (relationship names)
    - A *Gene* lies on a *Chromosome*
  - Cardinality (the “crow’s foot”)
  - Optionality (the pipe “|” or circle “O”)
  - Naming
    - Relationship names that fit a sentence structure, trying to use the same verb in both directions
    - Name the relationship in both directions even though it adds little meaning
  - Types



- **Relationships**
  - More than one relationship between two entity classes



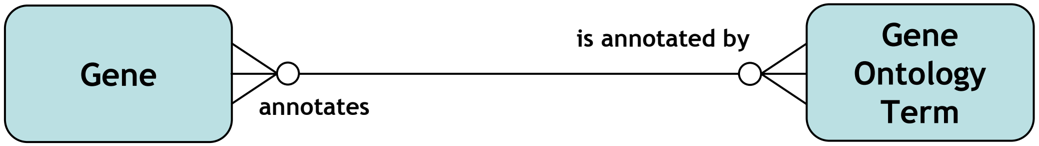
  - Self-referencing one-to-one (chains), one-to-many (hierarchies), many-to-many (networks)



- ## Relationships

  - ### Resolving Many-to-Many Relationships via Normalization

many-to-many



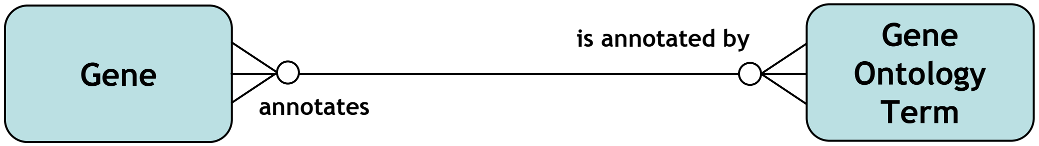
Unnormalized:

Gene

Gene ID	Gene Symbol	Gene Name	GO Term ID 1	Go Term Name 1	GO Term Type 1	GO Term ID 2	GO Term Name 2	GO Term Type 2	GO Term ID 3	GO Term Name 3	GO Term Type 3
24088	TLR2	toll-like receptor 2	0007166	cell surface receptor linked to signal transduction	Biological Process	0016021	integral to membrane	Cellular Component	0005866	plasma membrane	Cellular Component
20619	SNAP23	Synaptosomal-associated protein 23	0015031	protein transport	Biological Process	0005866	plasma membrane	Cellular Component	0006903	vesicle targeting	Biological Process

- Relationships
  - Resolving Many-to-Many Relationships via Normalization

many-to-many



First Normal Form (1NF)

Gene

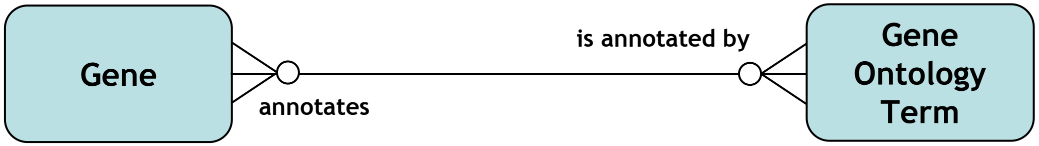
Gene ID	Gene Symbol	Gene Name
24088	TLR2	toll-like receptor 2
20619	SNAP23	Synaptosomal-associated protein 23

Gene Ontology Term

GO Term ID	Go Term Name	GO Term Type	Gene ID
0007166	cell surface receptor linked to signal transduction	Biological Process	24088
0016021	integral to membrane	Cellular Component	24088
0005866	plasma membrane	Cellular Component	24088
0015031	protein transport	Biological Process	20619
0005866	plasma membrane	Cellular Component	20619
0006903	vesicle targeting	Biological Process	20619

- **Relationships**
  - Resolving Many-to-Many Relationships via Normalization

many-to-many



Third Normal Form (3NF) via 2NF

Gene

Gene ID	Gene Symbol	Gene Name
24088	TLR2	toll-like receptor 2
20619	SNAP23	Synaptosomal-associated protein 23

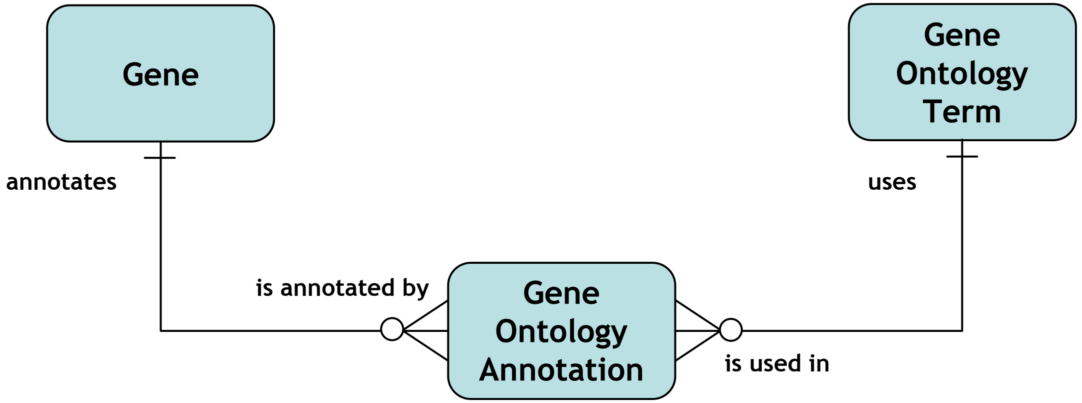
Gene-Gene Ontology Term Relationship

GO Term ID	Gene ID
0007166	24088
0016021	24088
0005866	24088
0015031	20619
0005866	20619
0006903	20619

Gene Ontology Term

GO Term ID	Go Term Name	GO Term Type
0007166	cell surface receptor linked to signal transduction	Biological Process
0016021	integral to membrane	Cellular Component
0005866	plasma membrane	Cellular Component
0015031	protein transport	Biological Process
0006903	vesicle targeting	Biological Process

- Relationships
  - Many-to-Many Relationships => An Intersection Entity Class and Two One-to-Many Relationships
    - The “one” ends of the new relationships will always be mandatory
    - The “many” ends of the new relationships will have the same optionality as the original relationship
    - The intersection entity class probably represents an entity class in itself



- **Relationships**
  - Independent Entity Classes
    - One whose instances can have an independent existence (no parent relationships)
  - Dependent Entity Classes
    - One whose instances can only exist in conjunction with instances of another entity class, and cannot be transferred between instances of that other entity class.

Gene

Gene Annotation

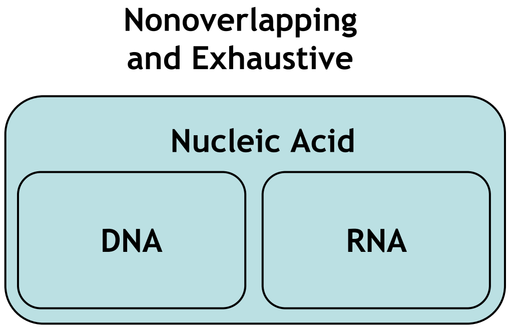
Independent

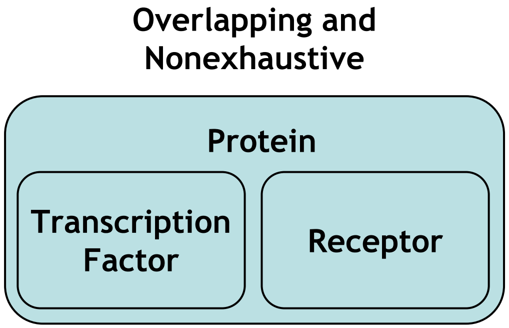
Dependent

- Attributes
  - Attribute disaggregation: “one fact per attribute” (promotes reusability, simplicity)
  - What fact about the entity instance are you providing information about?
  - What value should you enter to state that fact?
  - Violations
    - Simple Aggregation (e.g. storing the unit with the measurement, names)
    - Conflated Codes (embedding multiple pieces of information)
    - Inappropriate Generalization

- Attributes
  - Attribute Taxonomy
    - Identifiers
      - System generated, externally defined
    - Categories
      - Flags, enumerations, types
    - Quantifiers
      - Count, dimension, amounts, factor, intervals, time points
  - Naming
    - Use standard names whenever possible
    - Building off the entity class name

- Generalization Hierarchies (Subtypes and Supertypes)
  - A structured grouping of entities that share similar attributes
  - Generalization promotes "Stability" of the model by being flexible if the business rules are evolving.
  - Overlapping/Nonoverlapping
    - Overlapping is when a specific entity instance could be both of the subtypes.
    - Nonoverlapping (or disjoint) is when all entity instances have to be one of the subtypes and no more.
  - Exhaustive
    - All of the subtypes possible are represented





- Primary Keys
  - Basic Technical Criteria
    - Applicability
    - Uniqueness
    - Minimality
    - Stability
  - Surrogate or Natural?
  - Structured (Composite) Primary Keys
    - Usually exist in the context of a dependent entity class
    - Try to stay away from composite primary keys unless it is in an intersection entity

- Logical Design
  - Mapping the E-R Model to the Relational Model
  - Normalization

- Logical Design
  - Relational Model
    - Model proposed by E.F. Codd in 1970 based on rigorous mathematical principles.
    - The basic principle of the relational model is the Information Principle: all information is represented by data values in relations.
  - Mapping
    - Entity Classes => Tables
      - Exceptions
        - » Entity Class excluded from the database
        - » Generalization hierarchies

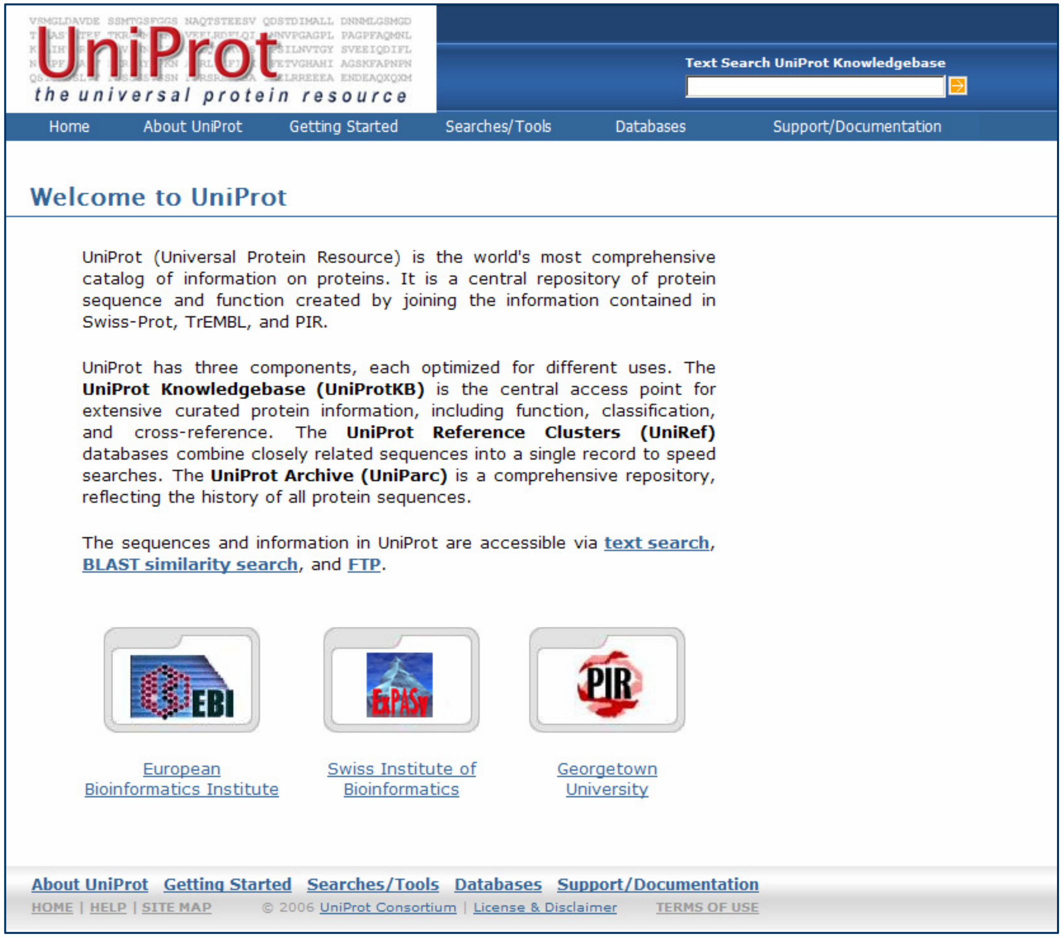
- Logical Design
  - Mapping
    - Relationships
      - One-to-One and One-to-Many
        - Done through primary and foreign keys
      - Many-to-Many
        - Intersection entity class becomes a table
    - Attributes => Columns
    - Generalization hierarchies
      - Option 1: Only represent the supertype as a table
      - Option 2: Only represent the subtypes as separate tables
      - Option 3: Supertype as a table with shared attributes and subtype tables with individual attribute linked with keys

- Physical Design
  - Data, relationship and constraint definition using SQL DDL
  - We’ll do this later!

- Database Design and Database Systems
  - Now we are going to use an example bioinformatics project to illustrate the data modeling concepts we learned earlier and to go through the complete process of database design. Through this process we will introduce database systems and their use.
  - The UniProt System
    - The UniProt Archive (UniParc) provides a stable, comprehensive sequence collection without redundant sequences by storing the complete body of publicly available protein sequence data.
    - The UniProt Knowledgebase (UniProtKB) is the central database of protein sequences with accurate, consistent, and rich sequence and functional annotation.
    - The UniProt Reference Clusters (UniRef) databases provide non-redundant reference clusters based on the UniProt knowledgebase (and selected UniParc records) in order to obtain complete coverage of sequence space at several resolutions.
    - Web Site: http://www.uniprot.org/
    - FTP Site: ftp://ftp.expasy.org/databases/uniprot/

- Database Design
  - Step 1: Planning and Requirements Analysis
    - What is the general structure of the UniProt System?

- **Planning and Requirements Analysis**
  - UniProt Web Site



Welcome to UniProt

UniProt (Universal Protein Resource) is the world's most comprehensive catalog of information on proteins. It is a central repository of protein sequence and function created by joining the information contained in Swiss-Prot, TrEMBL, and PIR.

UniProt has three components, each optimized for different uses. The **UniProt Knowledgebase (UniProtKB)** is the central access point for extensive curated protein information, including function, classification, and cross-reference. The **UniProt Reference Clusters (UniRef)** databases combine closely related sequences into a single record to speed searches. The **UniProt Archive (UniParc)** is a comprehensive repository, reflecting the history of all protein sequences.

The sequences and information in UniProt are accessible via text search, BLAST similarity search, and FTP.

European Bioinformatics Institute

Swiss Institute of Bioinformatics

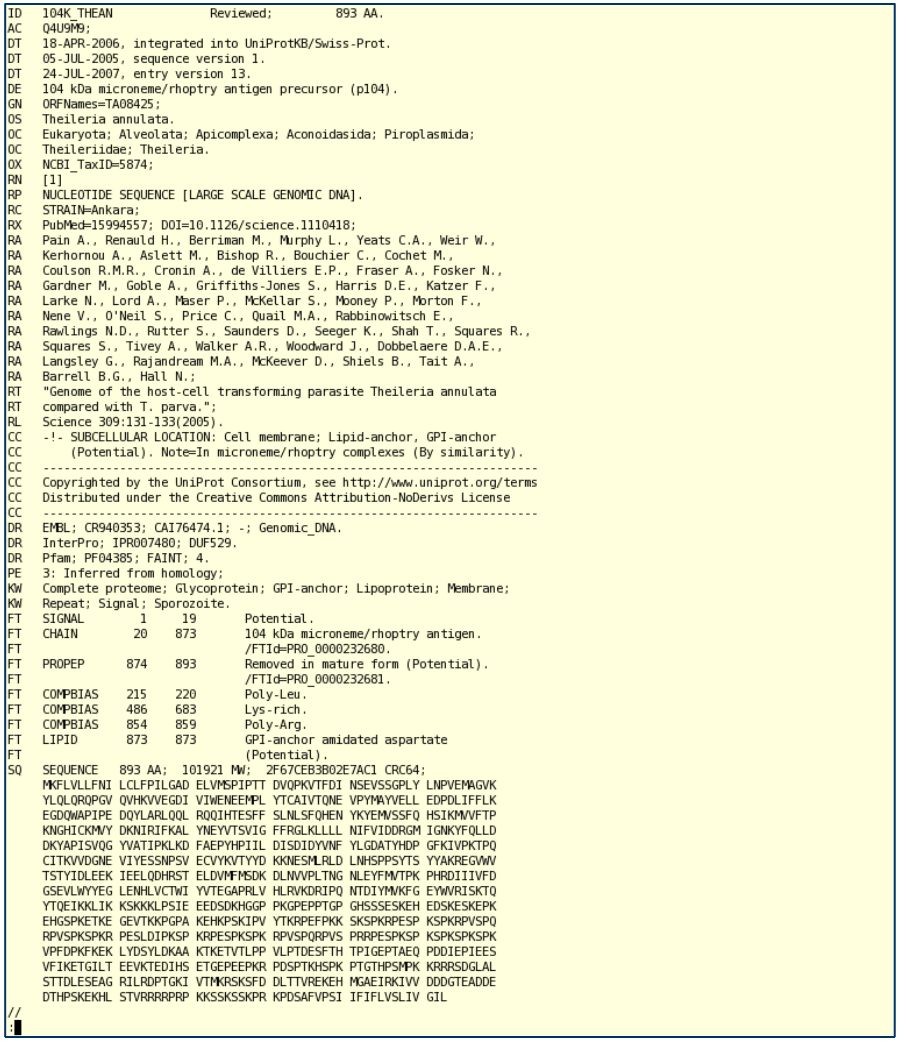
Georgetown University

EBI
ExPASy
PIR
About UniProt Getting Started Searches/Tools Databases Support/Documentation
HOME | HELP | SITE MAP
© 2006 UniProt Consortium | License & Disclaimer
TERMS OF USE

- Database Design
  - Step 1: Planning and Requirements Analysis
    - What is the general structure of the UniProt System?
      - It is organized by sequence.
    - What data do we need and what is the detailed structure of the data?

- Planning and Requirements Analysis
  - UniProt FTP Site
    - We have various types of data files explained in README file:
      - uniprot_sprot.dat.gz
      - uniprot_trembl.dat.gz
      - uniprot_sprot.xml.gz
      - uniprot_trembl.xml.gz
      - uniprot.xsd.gz
      - uniprot_sprot.fasta.gz
      - uniprot_trembl.fasta.gz

- **Planning and Requirements Analysis**
  - **uniprot_sprot.dat.gz file**

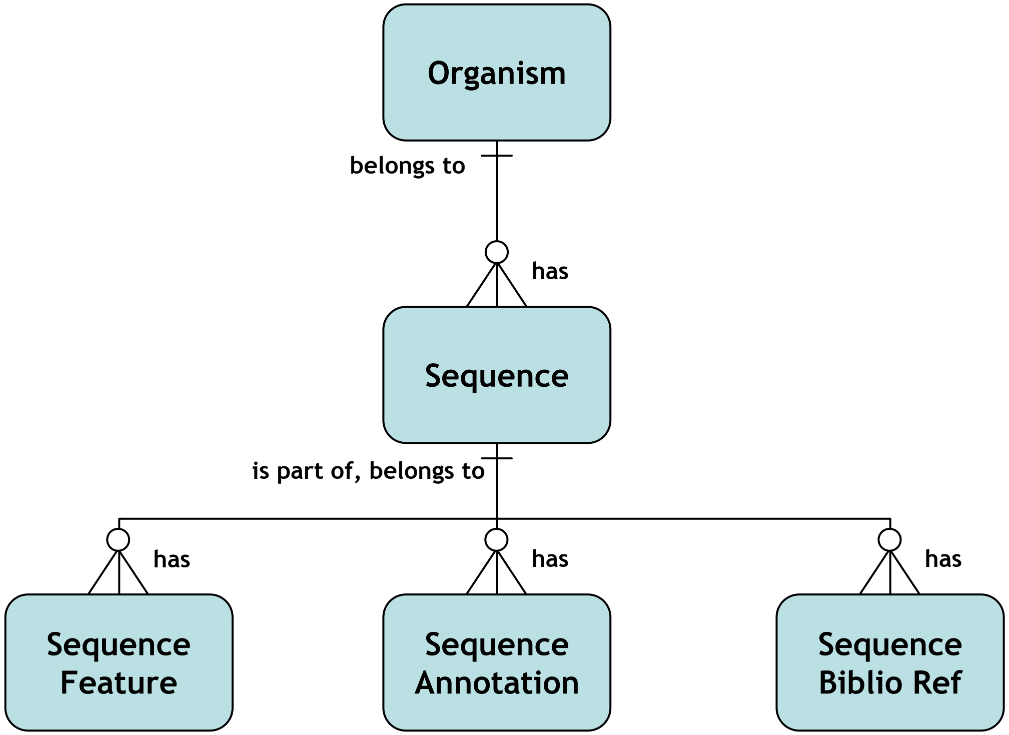


```
ID   104K_THEAN              Reviewed;         893 AA.
AC   Q4U9M9;
DT   18-APR-2006, integrated into UniProtKB/Swiss-Prot.
DT   05-JUL-2005, sequence version 1.
DT   24-JUL-2007, entry version 13.
DE   104 kDa microneme/rhoptry antigen precursor (p104).
GN   ORFNames=TA08425;
OS   Theileria annulata.
OC   Eukaryota; Alveolata; Apicomplexa; Aconoidasida; Piroplasmida;
OC   Theileriidae; Theileria.
OX   NCBI_TaxID=5874;
RN   [1]
RP   NUCLEOTIDE SEQUENCE [LARGE SCALE GENOMIC DNA].
RC   STRAIN=Ankara;
RX   PubMed=15994557; DOI=10.1126/science.1110418;
RA   Pain A., Renauld H., Berriman M., Murphy L., Yeats C.A., Weir W.,
RA   Kerhornou A., Aslett M., Bishop R., Bouchier C., Cochet M.,
RA   Coulson R.M.R., Cronin A., de Villiers E.P., Fraser A., Fosker N.,
RA   Gardner M., Goble A., Griffiths-Jones S., Harris D.E., Katzer F.,
RA   Larke N., Lord A., Maser P., McKellar S., Mooney P., Morton F.,
RA   Nene V., O'Neil S., Price C., Quail M.A., Rabbinowitsch E.,
RA   Rawlings N.D., Rutter S., Saunders D., Seeger K., Shah T., Squares R.,
RA   Squares S., Tivey A., Walker A.R., Woodward J., Dobbelaere D.A.E.,
RA   Langsley G., Rajandream M.A., McKeever D., Shiels B., Tait A.,
RA   Barrell B.G., Hall N.;
RT   "Genome of the host-cell transforming parasite Theileria annulata
RT   compared with T. parva.";
RL   Science 309:131-133(2005).
CC   -!- SUBCELLULAR LOCATION: Cell membrane; Lipid-anchor, GPI-anchor
CC       (Potential). Note=In microneme/rhoptry complexes (By similarity).
CC   -----------------------------------------------------------------------
CC   Copyrighted by the UniProt Consortium, see http://www.uniprot.org/terms
CC   Distributed under the Creative Commons Attribution-NoDerivs License
CC   -----------------------------------------------------------------------
DR   EMBL; CR940353; CAI76474.1; -; Genomic_DNA.
DR   InterPro; IPR007480; DUF529.
DR   Pfam; PF04385; FAINT; 4.
PE   3: Inferred from homology;
KW   Complete proteome; Glycoprotein; GPI-anchor; Lipoprotein; Membrane;
KW   Repeat; Signal; Sporozoite.
FT   SIGNAL        1     19       Potential.
FT   CHAIN        20    873       104 kDa microneme/rhoptry antigen.
FT                                /FTId=PRO_0000232680.
FT   PROPEP      874    893       Removed in mature form (Potential).
FT                                /FTId=PRO_0000232681.
FT   COMPBIAS    215    220       Poly-Leu.
FT   COMPBIAS    486    683       Lys-rich.
FT   COMPBIAS    854    859       Poly-Arg.
FT   LIPID       873    873       GPI-anchor amidated aspartate
FT                                (Potential).
SQ   SEQUENCE   893 AA;  101921 MW;  2F67CEB3B02E7AC1 CRC64;
     MKFLVLLFNI LCLFPILGAD ELVMSPIPTT DVQPKVTFDI NSEVSSGPLY LNPVEMAGVK
     YLQLQRQPGV QVHKVVEGDI VIWENEEMPL YTCAIVTQNE VPYMAYVELL EDPDLIFFLK
     EGDQWAPIPE DQYLARLQQL RQQIHTESFF SLNLSFQHEN YKYEMVSSFQ HSIKMVVFTP
     KNGHICKMVY DKNIRIFKAL YNEYVTSVIG FFRGLKLLLL NIFVIDDRGM IGNKYFQLLD
     DKYAPISVQG YVATIPKLKD FAEPYHPIIL DISDIDYVNF YLGDATYHDP GFKIVPKTPQ
     CITKVVDGNE VIYESSNPSV ECVYKVTYYD KKNESMLRLD LNHSPPSYTS YYAKREGVWV
     TSTYIDLEEK IEELQDHRST ELDVMFMSDK DLNVVPLTNG NLEYFMVTPK PHRDIIIVFD
     GSEVLWYYEG LENHLVCTWI YVTEGAPRLV HLRVKDRIPQ NTDIYMVKFG EYWVRISKTQ
     YTQEIKKLIK KSKKKLPSIE EEDSDKHGGP PKGPEPPTGP GHSSSESKEH EDSKESKEPK
     EHGSPKETKE GEVTKKPGPA KEHKPSKIPV YTKRPEFPKK SKSPKRPESP KSPKRPVSPQ
     RPVSPKSPKR PESLDIPKSP KRPESPKSPK RPVSPQRPVS PRRPESPKSP KSPKSPKSPK
     VPFDPKFKEK LYDSYLDKAA KTKETVTLPP VLPTDESFTH TPIGEPTAEQ PDDIEPIEES
     VFIKETGILT EEVKTEDIHS ETGEPEEPKR PDSPTKHSPK PTGTHPSMPK KRRRSDGLAL
     STTDLESEAG RILRDPTGKI VTMKRSKSFD DLTTVREKEH MGAEIRKIVV DDDGTEADDE
     DTHPSKEKHL STVRRRRPRP KKSSKSSKPR KPDSAFVPSI IFIFLVSLIV GIL
//
:
```

- Database Design
  - Step 1: Planning and Requirements Analysis
    - What is the general structure of the UniProt System?
      - It is organized by sequence.
    - What data do we need and what is the detailed structure of the data?
      - We need the actual sequence data.
      - We need the organism information the sequence comes from.
      - We need the sequence annotations and features.
      - We need any bibliographic information about the sequence.

- Database Design
  - Step 2: Building the Conceptual Model Using the E-R Approach
    - Entities
      - Sequence
      - Organism
      - Sequence Annotation
      - Sequence Feature
      - Sequence Bibliographic Reference
    - Relationships
      - Via the E-R Diagram

- **Database Design**
  - Step 2: Building the Conceptual Model Using the E-R Approach



- Database Design
  - Step 2: Building the Conceptual Model Using the E-R Approach
    - Attributes
      - Organism

» ID	*Primary Key (Surrogate)*
» Name	*Required, Unique*
» Common Name	
» Classification	

      - Sequence

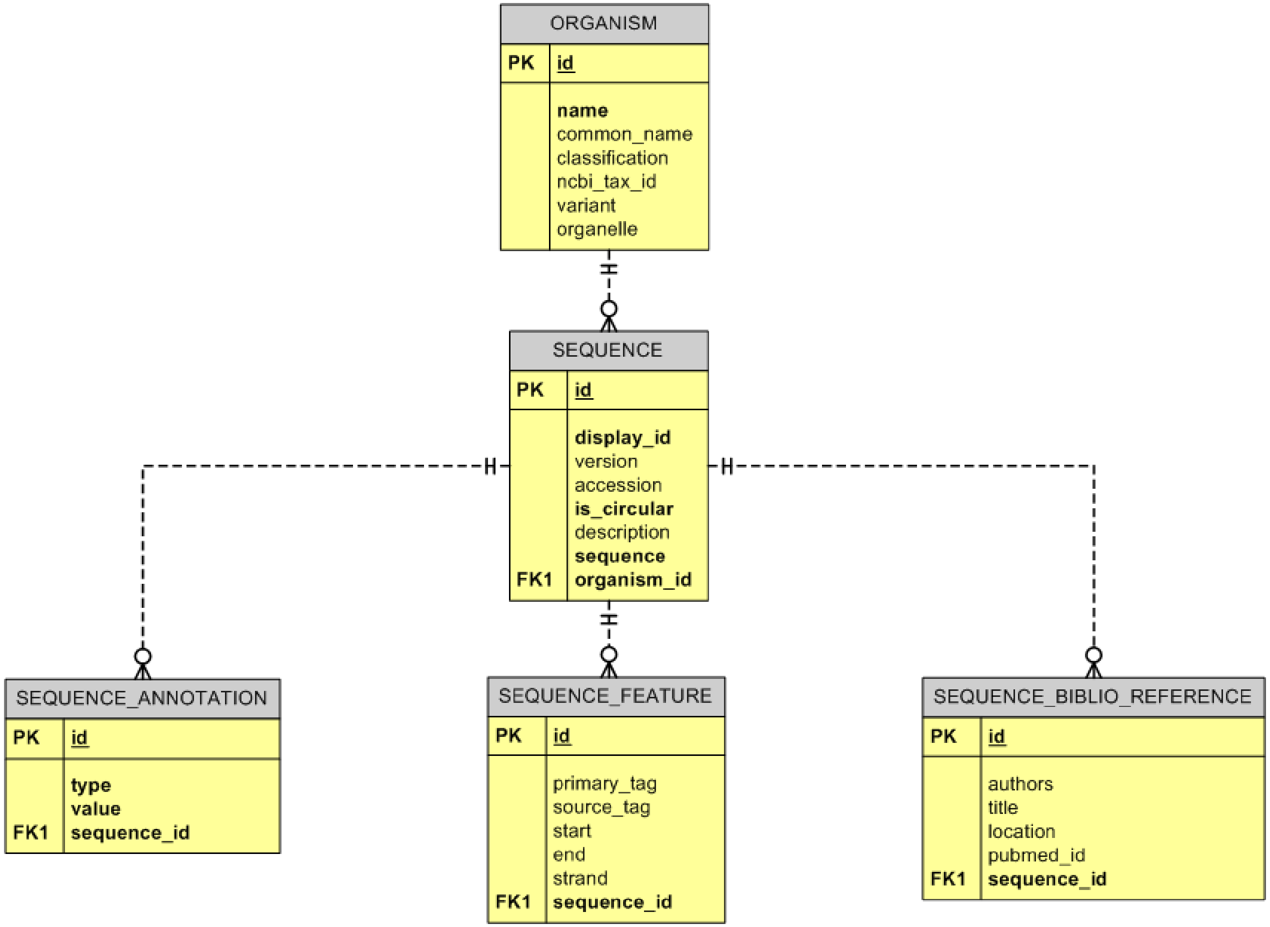
» ID	*Primary Key (Surrogate)*
» Display ID	*Required, Unique*
» Version	
» Accession	*Required, Unique*
» Is Circular	*Required*
» Description	
» Sequence	*Required*
» Organism ID	*Foreign Key, Required*

- Database Design
  - Step 2: Building the Conceptual Model Using the E-R Approach
    - Attributes
      - Sequence Annotation
        - ID *Primary Key (Surrogate)*
        - Type *Required*
        - Value *Required*
        - Sequence ID *Foreign Key, Required*
      - Sequence Feature
        - ID *Primary Key (Surrogate)*
        - Primary Tag
        - Source Tag
        - Start
        - End
        - Strand
        - Sequence ID *Foreign Key, Required*

- Database Design
  - Step 2: Building the Conceptual Model Using the E-R Approach
    - Attributes
      - Sequence Bibliographic Reference
        - ID *Primary Key (Surrogate)*
        - Authors
        - Title
        - Location
        - Pubmed ID
        - Sequence ID *Foreign Key*, *Required*

- ## Database Design

  - Step 3: Creating the Logical Model From the Conceptual Model Using Relational Constructs



- Physical Database Design and Database Systems
  - SQL (Structured Query Language)
    - Is a set-based, declarative computer language
    - Originally called "Structured English Query Language" or *SEQUEL*
    - Is designed for a specific, limited purpose - to query data in a database
    - Violates true relational completeness
    - "SQL is neither structured, nor is it limited to queries, nor is it a language"
    - Is a functionally simple language with a simple command set
      - Con: makes answering certain database questions difficult
      - Pro: generating SQL automatically using other languages is fairly easy

- Physical Database Design and Database Systems
  - SQL (Structured Query Language)
    - Data Connection Language
      - USE, CONNECT
    - Data Manipulation Language (DML)
      - SELECT, INSERT, UPDATE, DELETE
    - Data Definition Language (DDL)
      - CREATE, DROP, ALTER
    - Data Control Language (DCL)
      - GRANT, REVOKE
    - Data Transaction Language
      - BEGIN WORK (or START TRANSACTION), COMMIT, ROLLBACK

- Physical Database Design and Database Systems
  - SQL: Data Definition Language (DDL)
    - MySQL: http://dev.mysql.com/doc/refman/5.0/en/
    - CREATE (Physical Database Design)
      - Used to create database objects (users, tables, indexes, sequences, etc.)
      - Example CREATE SEQUENCE syntax:

```
CREATE SEQUENCE sequence_name;
```

      - Example CREATE TABLE syntax:

```
CREATE TABLE table_name (
    col_name       col_type    [ NOT NULL ] [ DEFAULT default_value],
    col_definition…
    more…
);
```

- Physical Database Design and Database Systems
  - SQL: Data Definition Language (DDL)
    - Column Data Types
      - Numeric Types
        - Exact Types: INTEGER, NUMBER (or NUMERIC), DECIMAL
        - Approximate Types: FLOAT, REAL, DOUBLE_PRECISION
      - Date Types
        - DATE, TIMESTAMP
      - String Types
        - CHAR, VARCHAR, TEXT
      - Binary Types
        - BLOB, CLOB

- Physical Database Design and Database Systems
  - SQL: Data Definition Language (DDL)
    - CREATE (Physical Database Design)
      - Used to create database objects (users, tables, indexes, sequences, etc.)
      - Example CREATE INDEX syntax:

```
CREATE INDEX index_name  ON  table_name (col_name, [col_name, ] );
```

- ## Physical Database Design and Database Systems
  - ### SQL: Data Definition Language (DDL)
    - ALTER (Physical Database Design)
      - Used to alter existing database objects
      - Example Alter Table Syntax:

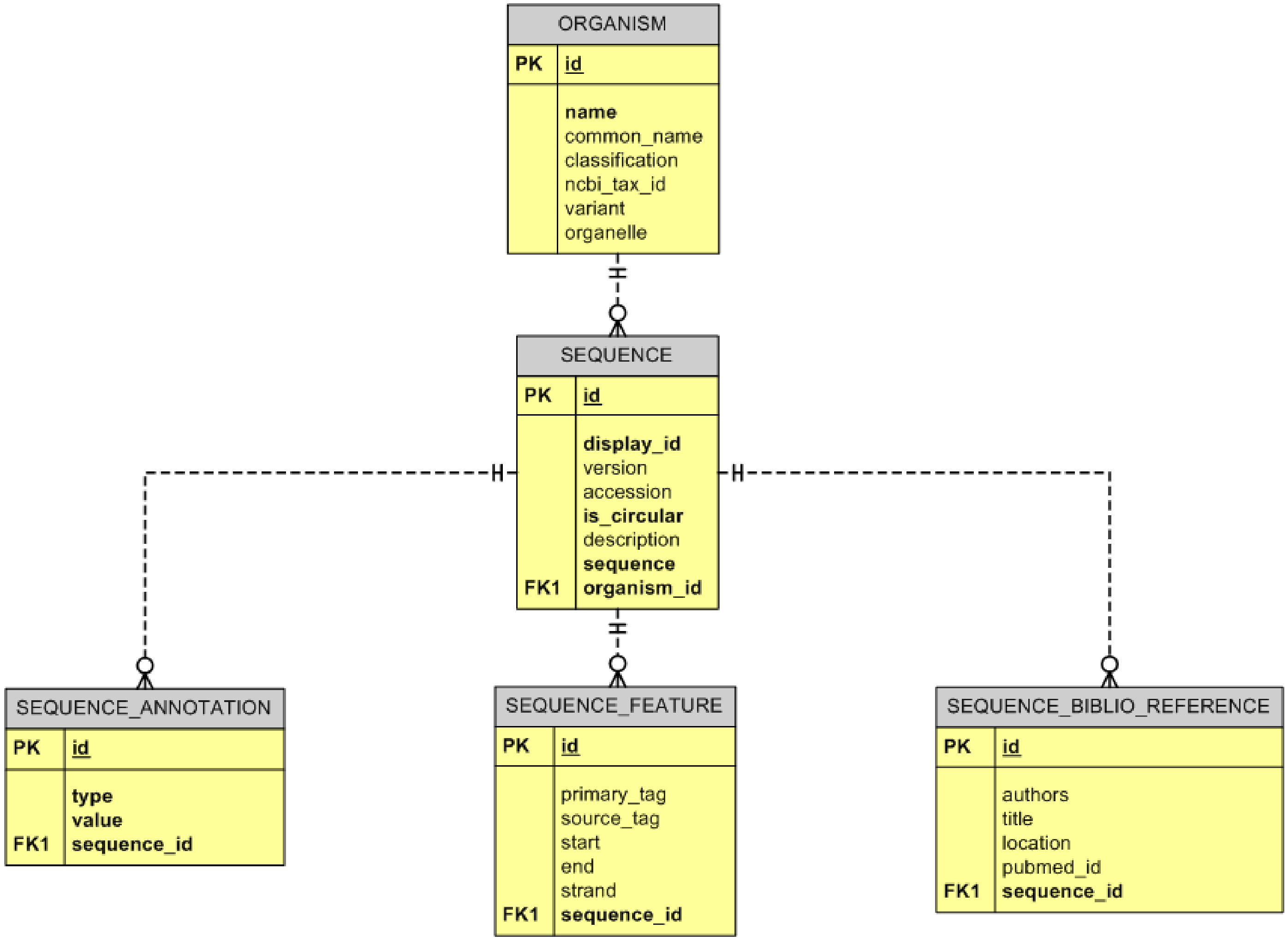
```
ALTER TABLE table_name
    ADD CONSTRAINT   constraint_name      PRIMARY KEY (col_name, [col_name, ] ),
    ADD CONSTRAINT   constraint_name      UNIQUE       (col_name, [col_name, ] ),
    ADD CONSTRAINT   constraint_name      FOREIGN KEY (col_name, [col_name, ] )
                                          REFERENCES table_name (col_name,
                                          [col_name, ] ) [ ON DELETE CASCADE | SET
                                          NULL],
    ADD CONSTRAINT   constraint_name      CHECK (where clause expression*),
    ADD COLUMN       col_name   col_type  [ NOT NULL ] [ DEFAULT default_value],
    more…
;
ALTER TABLE table_name
        ADD INDEX index_name (col_name, [col_name, ] )
;
```

- Physical Database Design and Database Systems
  - Step 4: Create Physical Database SQL Script and Integrity Rules
    - Sequences (DBMS specific)
    - Tables
    - Constraints (Integrity Rules)
    - Indexes
      - Index columns that are important to process model that are not already represented by another index (or implicit index via a primary key/unique constraint).
      - Index all foreign keys that are not already represented by another index (or implicit index via a primary key/unique constraint).
  - Step 5: SQL client environment for particular DBMS
    - Oracle: *sqlplus*
    - PostegreSQL: *psql*
    - MySQL: *mysql*

- Database Management Systems (DBMS)
  - Definition (wikipedia.org):
    - A complex suite of software programs and services (system) running on a computer/server which are designed to manage multiple databases and run operations on the data requested by numerous users.
    - The DBMS controls the organization, storage, and retrieval of the structured data in a database.
  - DBMS Basic Functionalities:
    - Provides a *modeling language* to defined how data is generally organized and described (the schema) – the most common is the relational model (or, to be exact, the *pseudorelational* model)
    - Provides a *database interface* and *query language* to allow for creation and manipulation of databases, interrogation and changing their data, and data security.
    - Provides *data storage structures* optimized to deal with large amounts of data recorded to a permanent storage device
    - Provides a *transaction mechanism*, that ideally would guarantee ACID properties, in order to ensure data integrity despite concurrent user access and/or faults.

- SQL Data Manipulation Language (DML)
  - `SELECT`
    - Used to retrieve zero or more rows from one or more tables in the database.
  - `INSERT`
    - Used to add zero or more rows (formally called *tuples*) to an existing table.
  - `UPDATE`
    - Used to modify the values of a set of existing table rows.
  - `DELETE`
    - Used to removed zero or more rows from an existing table.
  - `TRUNCATE`
    - Used to remove all of the existing rows from a table.

- # Example Database



- SELECT (Data Retrieval)

  - The user essentially describes a description of the result set without describing what physical operations must be executed to produce that result set (which is the DBMS's job).

  - Example:

```
SELECT   *
FROM     organism
WHERE    id = 5;
```

  - The Main Clauses (keywords):

    - FROM is used to indicate which tables the data is to be taken from, as well as how to JOIN the tables together.

    - WHERE is used to identify which rows are to be retrieved or applied to GROUP BY.

    - GROUP BY is used to combine rows with related values into elements of a smaller set of rows.

    - HAVING is used to identify which rows, following a GROUP BY, are to be retrieved.

    - ORDER BY is used to identify which columns are to be used to sort the retrieved data.

- `SELECT` (Data Retrieval)
  - Clause Processing Order (very important):
    1. `FROM`
    2. `WHERE`
    3. `GROUP BY`
    4. `HAVING`
    5. `SELECT` (the columns you want)
    6. `ORDER BY`

- `SELECT` (Data Retrieval)
  - `FROM` clause
    - When we are constructing a `SELECT` statement the first thing we do is look at our physical database schema (*tables*, *columns*, *relationships*, *etc*.) and determine what *data fields* we want (i.e. *columns*) and which table(s) they are in. These table name(s) need to go into the `FROM` clause.
    - EXAMPLE: we want all of the organism data (with column order in the order they were listed when the table was created with the `CREATE TABLE` statement)

```
SELECT   *
FROM     organism;
```

    - EXAMPLE: we want all the sequence display IDs, accessions and sequence

```
SELECT   display_id, version, sequence
FROM     sequence;
```

    - If all of the columns you want are in the same table then we simply need to put that table name.

- **SELECT (Data Retrieval)**
  - **FROM clause**
    - Now lets say we want columns from *two* different tables. We need to put the tables together using their *primary-foreign key relationship*.
    - EXAMPLE: we want all the sequence display IDs, versions, sequence *and their corresponding organism names*

```
SELECT    sequence.display_id, sequence.version, sequence.sequence, organism.name
FROM      sequence, organism
WHERE     sequence.organism_id = organism.id;
```

Explicitly (and better):

```
SELECT    sequence.display_id, sequence.version, sequence.sequence, organism.name
FROM      sequence INNER JOIN organism ON sequence.organism_id = organism.id;
```

Using aliases:

```
SELECT    S.display_id, S.accession, S.version, S.sequence, O.name
FROM      sequence S, organism O
WHERE     S.organism_id = O.id;
```

```
SELECT    S.display_id, S.accession, S.version, S.version, O.name
FROM      sequence S INNER JOIN organism O ON S.organism_id = O.id;
```

- **SELECT (Data Retrieval)**
  - **FROM, WHERE clauses**
    - Now that we have our *data fields* (i.e. columns) and the tables they are in, we need to decide which *tuples* (i.e. rows) we want (unless want all of them). We do this by setting *condition(s)* on the rows to be returned using any appropriate column(s) in our database schema. These *condition(s)* go in the **WHERE** clause. The table(s) where these column(s) are located also need to go in the **FROM** clause.
    - EXAMPLE: we want all sequence display IDs, versions and sequences *where the version is ‘2’*

```
SELECT     display_id, version, sequence
FROM       sequence
WHERE      version = '2';
```

-
  -
    - EXAMPLE: we want the sequence display IDs, versions and sequences *where the version is ‘2’ and the sequence is circular*

```
SELECT     display_id, version, sequence
FROM       sequence
WHERE      version = '2'
           AND
           is_circular = 1;
```

- SELECT (Data Retrieval)
  - FROM, WHERE clauses
    - EXAMPLE: we want all sequence display IDs, versions and sequences *for the organism named ’Homo sapiens’*

```
SELECT    S.display_id, S.version, S.sequence
FROM      sequence S INNER JOIN organism O ON S.organism_id = O.id
WHERE     O.name = ’Homo sapiens’;
```

- EXAMPLE: we want all sequence display IDs, versions and sequences *for the organism named ‘Homo sapiens’ which are lipoproteins*

```
SELECT    S.display_id, S.version, A.sequence
FROM      sequence S INNER JOIN organism            O ON S.organism_id = O.id
                     INNER JOIN sequence_annotation F ON F.sequence_id = S.id
WHERE     O.name = ’Homo sapiens’
          AND
          F.keyword = ’Lipoprotein’;
```

- SELECT (Data Retrieval)
  - ORDER BY clause
    - Now that we have the tuples (i.e. rows) of data we want and the data fields (i.e. columns) describing them we might want to retrieve the data in a certain order. We do this using the ORDER BY clause and one or more appropriate columns in our database schema. The table(s) where these columns are located also need to go into the FROM clause. By default rows are order in ascending alphabetical or numerical order.
    - EXAMPLE: we want all of the organism data *ordered by organism name in ascending alphabetical order*

```
SELECT    *
FROM      organism
ORDER BY  name ASC;
```

- EXAMPLE: we want the organism names, sequence display IDs, versions and sequence for each organism *ordered by organism name in descending alphabetical order first*, *then by display ID in ascending alphabetical order*

```
SELECT    S.display_id, S.accession, S.version, S.version, O.name
FROM      sequence S INNER JOIN organism O ON S.organism_d = O.id
ORDER BY  O.name DESC, S.display_id;
```

- SELECT (Data Retrieval)
  - GROUP BY clause
    - Now that we have the tuples (i.e. rows) of data we want and the data fields (i.e. columns) describing them we might want to *aggregate* rows using certain appropriate column(s) in our database schema. We do this by putting these column(s) into the GROUP BY clause. After grouping, only columns in the GROUP BY clause or SQL *aggregate functions* can be in the SELECT line.
    - EXAMPLE: we want the *number* of sequences for each organism

```
SELECT    O.name, COUNT(*) AS num_sequences
FROM      organism O INNER JOIN sequence S ON O.id = S.organism_id
GROUP BY  O.name;
```

- EXAMPLE: we want all the sequence display IDs *for the organism named 'Homo sapiens'* and *number* of annotations in *each sequence*

```
SELECT    S.display_id, COUNT(*) AS num_annotations
FROM      organism O INNER JOIN sequence S              ON O.id = S.organism_id
                     INNER JOIN sequence_annotation A ON A.sequence_id = S.id
WHERE     O.name = 'Homo sapiens'
GROUP BY  S.display_id ;
```

- **SELECT (Data Retrieval)**
  - **HAVING clause**
    - If we have a statement where we have already used a GROUP BY clause to aggregate rows, the HAVING clause works in the same manner on the grouped rows as the WHERE clause works on individual rows by filtering rows based one or more *conditions*.
    - EXAMPLE: we want the *number* of sequences for each organism *only for organisms with 100 or more sequences*

```
SELECT    O.name, COUNT(*) AS num_sequences
FROM      organism O INNER JOIN sequence S ON O.id = S.organism_id
GROUP BY  O.name
HAVING    COUNT(*) >= 100;
```

    - EXAMPLE: we want all the sequence display IDs for the organism name ‘Homo sapiens’ *which have at least 10 or more HELIX domain sequence features*

```
SELECT    S.display_id
FROM      organism O INNER JOIN sequence S          ON O.id = S.organism_id
                     INNER JOIN sequence_feature F ON S.id = F.sequence_id
WHERE     O.name = 'Homo sapiens'
          AND
          F.primary_tag = 'HELIX';
GROUP BY  S.display_id
HAVING    COUNT(*) >= 10;
```

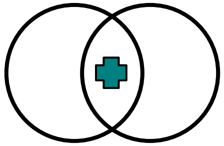
- SELECT (Data Retrieval)
  - DISTINCT keyword
    - We would use this to suppress identical rows in the result columns we are retrieving.
    - EXAMPLE: we want to return *one copy* of each of the sequence annotation types

```
SELECT     DISTINCT type
FROM       sequence_annotation;
```

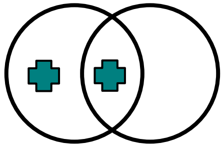
- EXAMPLE: we want to return *one copy* of each of the chromosome names for each organism

```
SELECT     DISTINCT S.display_id, A.type
FROM       sequence S INNER JOIN sequence_annotation A ON S.id = A.sequence_id;
```

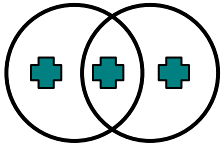
- SELECT (Data Retrieval)
  - JOIN types



- INNER JOIN
  - When we want to perform a query returning joining only those rows from both tables that satisfy the JOIN condition (after the ON keyword).



- LEFT | RIGHT [OUTER] JOIN
  - When we want to perform a query returning joining the rows from both table that statisfy the JOIN condition plus all of the rest of the rows of table to the left of the JOIN keyword (in the case of a LEFT JOIN). For those rows of the table to the left of the JOIN keyword that have no matching rows with those in the right table, NULLs are populated.



- FULL [OUTER] JOIN
  - When we want to perform a query returning all of the rows from both tables joined together, extended with NULLs if they do not satisfy the JOIN condition.

- EXAMPLE: we want *all* of the sequence display IDs and sequence data and the corresponding feature inforation for the sequence *if it is available*

```
SELECT    S.display_id, S.sequence, F.primary_tag, F.start, F.end, F.strand
FROM      sequence S LEFT JOIN sequence_feature F ON S.id = F.sequence_id;
```

- **SELECT (Data Retrieval)**
  - Operators in WHERE clauses
    - Comparison:

      `= != > < BETWEEN IN LIKE IS NULL`
    - Logical:

      `NOT AND OR`
    - Functions:

      `LENGTH() SUBSTR() UPPER() LOWER() MAX() MIN() SUM()`
    - Set:

      `UNION INTERSECT MINUS`

- SELECT (Data Retrieval)
  - Subqueries
    - The FROM and WHERE clauses can have embedded queries which return data used as a table (in the case of where the subquery is in the FROM clause) or in a condition (in the case of where the subquery is in the WHERE clause)
    - EXAMPLE: we want all of the sequences *that have a corresponding pubmed ID of 4556, 7889, or 12990 in the sequence bibliographic reference table*

```
SELECT    *
FROM      sequence
WHERE     sequence_id IN (SELECT sequence_id
                          FROM   sequence_biblio_reference
                          WHERE  pubmed_id IN (4556, 7889, 12990));
```

- **INSERT (Data Manipulation)**

  - Structure:

```
INSERT INTO table_name [ (col_name1, col_name2, col_name3, … ) ]
VALUES                   (value1,    value2,    value3, … );
```

```
INSERT INTO table_name_1 [ (col_name1, col_name2, col_name3, … ) ]
SELECT                      col_name1, col_name2, col_name3, …
FROM                        table_name_2
[ WHERE                     condition  ];
```

  - Example:

```
INSERT INTO organism (id, name, common_name)
VALUES               (value1,  value2,   value3);
```

- **UPDATE** (Data Manipulation)

  - Structure:

```
UPDATE     table_name
SET        col_name1 = value1, col_name2 = value2, col_name3 = value3, …
[ WHERE    condition  ];
```

  - Example:

```
UPDATE     sequence
SET        description = ’This is a circular sequence’
WHERE      is_circular = 1;
```

- **DELETE** (Data Manipulation)
  - Structure:

```
DELETE
FROM     table_name
[ WHERE    condition   ];
```

- **TRUNCATE** (Data Manipulation)
  - Structure:

```
TRUNCATE     table_name;
```

Logically equivalent to (except you cannot undo (rollback) a TRUNCATE statement!)

```
DELETE
FROM     table_name;
```

More SQL Data Definition Language (DDL)

- DROP

```
DROP OBJECT_TYPE object_name [ CASCADE | CASCADE CONSTRAINTS ];
```

- Examples:

```
DROP USER  hermida CASCADE;
```

```
DROP TABLE sequence CASCADE CONSTRAINTS;
```

```
DROP INDEX sequence_id_idx;
```

```
DROP CONSTRAINT organism_id_fk;
```

- DESCRIBE

```
DESCRIBE table_name;
```

- Example Database Management Systems (DBMS)
  - *Oracle*
    - A proprietary, scalable, enterprise-class relational and object-relational DBMS
    - Very powerful and has unique capabilities
  - *MySQL*
    - An easy-to-use, easy-to-install, scalable and open-source relational DBMS
    - Different storage engines
  - *PostgreSQL*
    - An easy-to-use, easy-to-install, scalable and open-source relational and object-relational DBMS
    - Good object-relational capabilities

- DBMS Specifics
  - SQL language extensions and individual syntaxes
  - Proprietary native interface languages
    - Example: PL/SQL
  - Different physical data storage requirements, capabilities and features

- ## DBMS Client Programs

  - Oracle: *sqlplus*

```
[user@computer ~]$ sqlplus ”username/password@database” [ @file.sql ]
```

  - MySQL: *mysql*

```
[user@computer ~]$ mysql -u username -h host -ppassword -D database [ < file.sql ]
```

  - PostgreSQL: *psql*

```
[user@computer ~]$ psql -U username -h host -d database [  -f file.sql ]
```

- ## SQL Data Connection Language

  - **USE**

```
USE database_name;
```

  - **CONNECT**

```
CONNECT database_name;
```

- ## MySQL Specifics

  - **SHOW**

```
SHOW DATABASES;
SHOW TABLES;
SHOW CREATE TABLE table_name ( … );
```

  - **LIMIT**

```
SELECT   *
FROM     organism
LIMIT     10;
```

  - Information Schema Database
    - *information_schema*
    - Gives global server information on tables, columns, constraints, privileges, views, triggers, etc.

- ACID (Transactions)
  - Atomicity
    - Refers to the ability of the DBMS to guarantee that either all of the tasks of a transaction are performed or none of them are.
  - Consistency
    - Refers to the database being in a legal state when the transaction begins and when it ends. This means that a transaction can't break the rules, or *integrity constraints*, of the database.
  - Isolation
    - Refers to the ability of the DBMS to make operations in a transaction appear isolated from all other operations. This means that no operation outside the transaction can ever see the data in an intermediate state. More formally, isolation means the transaction history (or schedule) is serializable. For performance reasons, this ability is the most often relaxed constraint.
  - Durability
    - Refers to the guarantee that once the user has been notified of success, the transaction will persist, and not be undone. This means it will survive system failure, and that the database system has checked the integrity constraints and won't need to abort the transaction. Typically, all transactions are written into a log that can be played back to recreate the system to its state right before the failure. A transaction can only be deemed committed after it is safely in the log.

- SQL Data Transaction Language

  - COMMIT

    - Tells the DBMS to make permanent all of the previous data manipulation statements (i.e. INSERT, UPDATE, DELETE) since the last COMMIT or ROLLBACK.

```
COMMIT;
```

  - ROLLBACK

    - Tells the DBMS to completely undo all of the previous data manipulation statements (i.e. INSERT, UPDATE, DELETE) since the last COMMIT or ROLLBACK.

```
ROLLBACK;
```

  - Remember: Data Definition Language (DDL) statements (i.e. CREATE, DROP, ALTER) DO NOT apply, are immediately commited once executed, and CANNOT be rolled back!

- ## Database Indexs

  - ### B-TREE

    - The default index type used by almost all DBMSs when you do a CREATE INDEX statement. Like the index of a book - the indexed column data is stored separately from the rest of the table data for fast lookup and these indexes have pointers to the corresponding table data so we do not have to do a table scan.

  - ### BITMAP

    - Available in Oracle

    - Saves a lot of space (when the columns have low cardinality) and solves the dilemma of not having the right index for the columns you want to search by.

    - Used in data warehousing and not transaction databases

  - ### FUNCTION-BASED

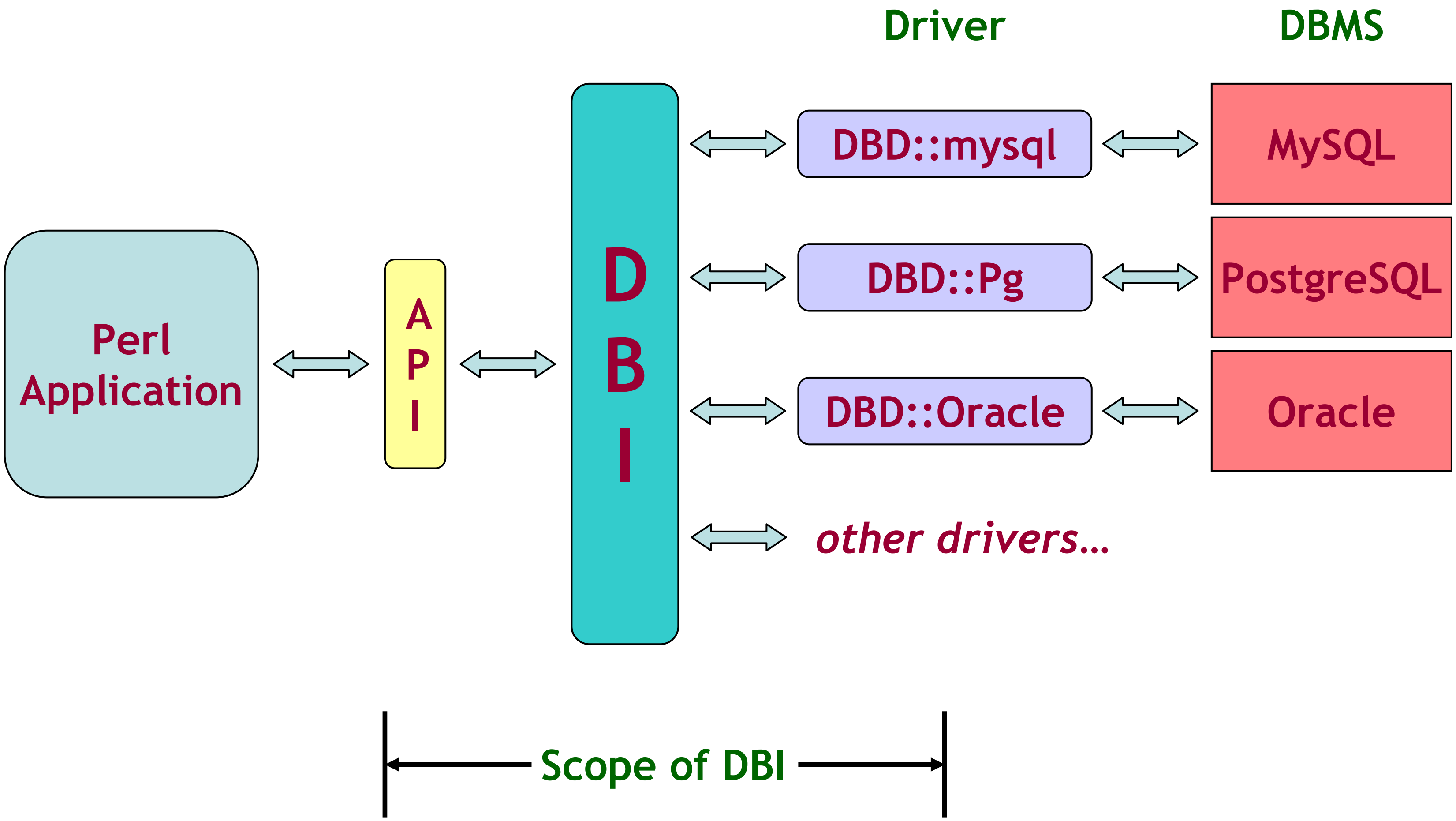
    - Available in Oracle and PostgreSQL

```
CREATE INDEX organism_name_idx_ci ON organism (LOWER(name));
```

- External Database Interfaces
  - SQL is a very limited and focused language so we need to provide interfaces from more powerful and complete imperative (procedural) languages like Java, C++, Perl, Python, PHP, etc.
  - All of the interfaces for each language have the same construct types and methods.
  - All of the interfaces define a set of methods, variables, and conventions that provide a (fairly) consistent database interface, independent of the actual database being used.
  - We will take Perl's *DBI* as an example http://search.cpan.org/~timb/DBI-1.59/DBI.pm

- Perl DBI *in Depth*
  - DBI (Database Independent Interface for Perl) is *the* database/DBMS access module for Perl.
  - Other programming languages almost always have their own database interfaces, some of them provide an database/DBMS independent and thus universal API, others have specific APIs depending on the DBMS.
  - For example, look at:
    - MySQL: http://dev.mysql.com/downloads/
    - PostgreSQL: http://www.postgresql.org/download/
  - The power of having a database/DBMS independent interface is that you can change the DBMS back-end without having to alter (much) of the application code that works with the database. Data access is *abstracted*, meaning you use the same functions, methods, attributes (the API) regardless of what DBMS you are working with.

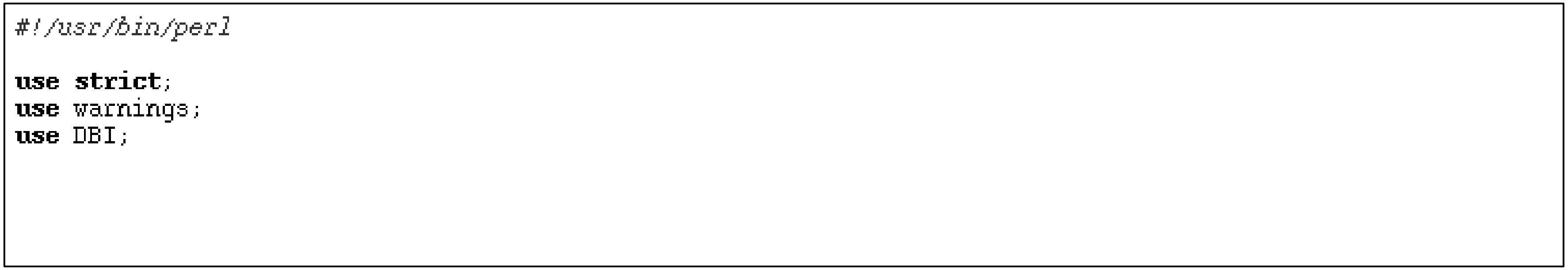
- DBI Architecture



- DBI Architecture
  - The DBI API defines the methods, attributes and variables for Perl applications to use. This API is implemented by the DBI package.
  - The DBI dispatches the method calls to the appropriate DBD::* driver for execution. DBI is also responsible for dynamic loading of drivers, error checking and handling, providing default implementations for methods (if the DBD::* driver doesn't have its own implementation) and other non-database specific duties.
  - Each DBD::* driver contains implementations of the DBI methods using the private interface functions of the corresponding DBMS.
  - Each DBD::* driver can implement their own specific methods and attributes to extend DBI's capabilities with respect to the particular DBMS.

- ## Starting a DBI Perl Script

Load the DBI module in typical Perl fashion:



```
#!/usr/bin/perl

use strict;
use warnings;
use DBI;
```

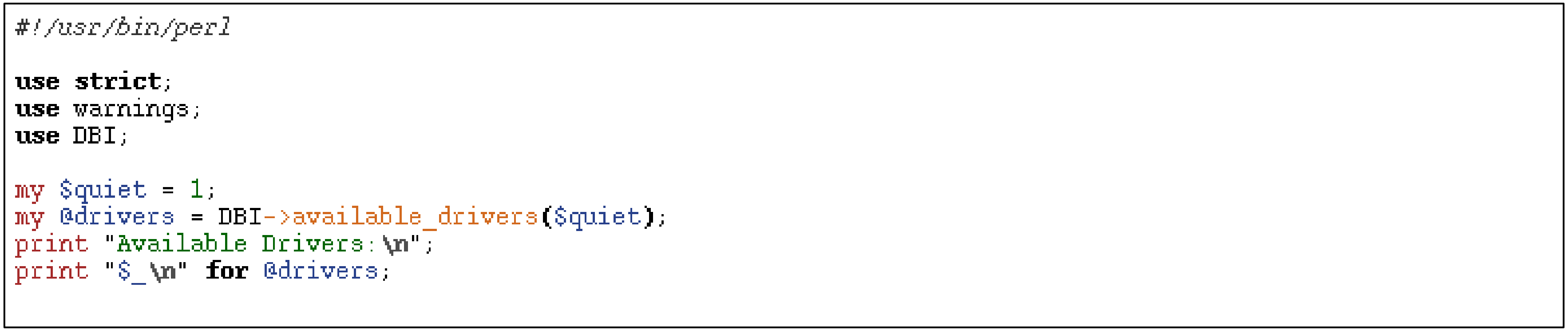
- ## The DBI Class

  - ### Important class methods

    - `DBI->connect($dsn, $username, $password, [ \%attrs ])`
      - Establishes a database connection, or session, to the requested data source specified in `$dsn`. Returns a database handle object if the connection succeeds.

    - `DBI->available_drivers([ $quiet ])`
      - Returns a list of drivers available on the system.

Example:



```
#!/usr/bin/perl

use strict;
use warnings;
use DBI;

my $quiet = 1;
my @drivers = DBI->available_drivers($quiet);
print "Available Drivers:\n";
print "$_\n" for @drivers;
```

- ## DSN (Data Source Name)

  - The DSN is a single string that has the following general format (but there is not standard and you must read DBD::* driver documentation for exact details and options):

```
DBI:DriverName:database_name
DBI:DriverName:database=$db_name;host=$host;port=$port
```

  - Examples:

```
DBI:Oracle:uniprot
DBI:Oracle:host=dbserver.unibas.ch;sid=uniprot
DBI:Oracle:host=dbserver.unibas.ch;sid=uniprot;port=1521
```

```
DBI:mysql:uniprot
DBI:mysql:database=uniprot;host=dbserver.unibas.ch
DBI:mysql:database=uniprot;host=dbserver.unibas.ch;port=3306
```

```
DBI:Pg:uniprot
DBI:Pg:database=uniprot;host=dbserver.unibas.ch
DBI:Pg:database=uniprot;host=dbserver.unibas.ch;port=5432
```

- The Database Connection Attributes Hash (`%attrs`)
  - Used to pass database handle object attributes in order to alter the default database behavior during creation of the object.
  - Important attributes:
    - `PrintError` (boolean, default = 1)
      - Turn on when you want DBI errors to issue warnings in addition to returning error codes in the normal way
    - `RaiseError` (boolean, default = 0)
      - Turn on when you want DBI errors to turn into exceptions rather than just returning error codes the normal way. If you turn `RaiseError` on then your would normally turn `PrintError` off.
    - `HandleError` (code ref, no default)
      - Used when you want to provide an alternative behavior during DBI errors. If set to a subroutine reference, then this subroutine will be called when an error is detected. The subroutine is called with three parameters: 1) the error message that `PrintError` and `RaiseError` use, 2) the DBI handle being used and 3) the first value returned by the method that failed.

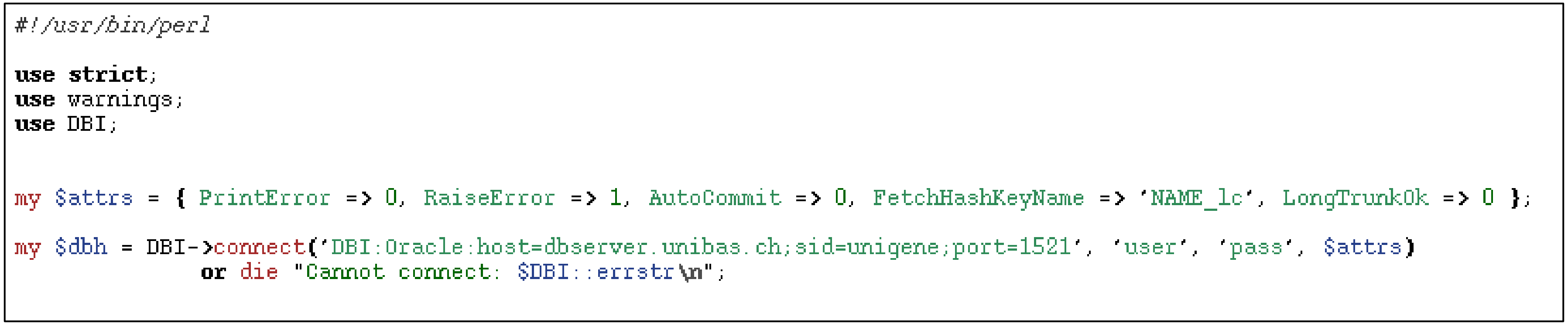
The Database Connection Attributes Hash (%attrs)

Important Attributes:

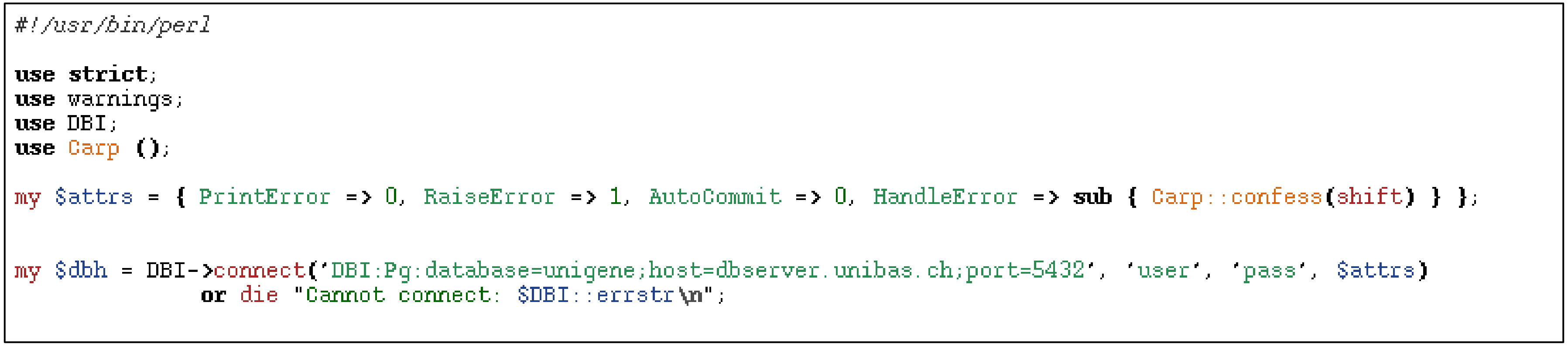
- FetchKeyHashName (string, default = 'NAME')
  - Used to specify whether the *_hashref methods should perform case conversion on the field names used for the hash keys. Set to 'NAME_lc' to convert to lower case or 'NAME_uc' to convert to upper case.
- AutoCommit (boolean, default = 1)
  - If on, database changes are commited right away and CANNOT be rolled back. If off, then all database changes occur in a transaction and must be explicitly committed or rolled back by the database handle object.
- LongReadLen (unsigned integer, default = 0 but DBD::* driver specific)
  - Used to control the maximum length (in bytes) of 'long' type fields (BLOB, CLOB, LONG, TEXT, etc.) during fetching of the data from these columns by the driver. Does not affect inserts, updates or deletes. For drivers which use this attribute the value has a direct effect on memory used by the statement handle and thus your Perl script so don't be too generous. If a field value is longer than LongReadLen it will be truncated it that length.
  - Implemented by DBD::Oracle and NOT implemented by DBD::mysql, DBD::Pg.
- LongTruncOk (boolean, default = 0)
  - Used to control whether the effect of fetching a 'long' field value beyond the length specified by LongReadLen should raise an error.
  - Implemented by DBD::Oracle and NOT implemented by DBD::mysql, DBD::Pg.

- **Establishing a DBMS/Database Connection and Creating a Database Handle Object (`$dbh`)**

Examples:



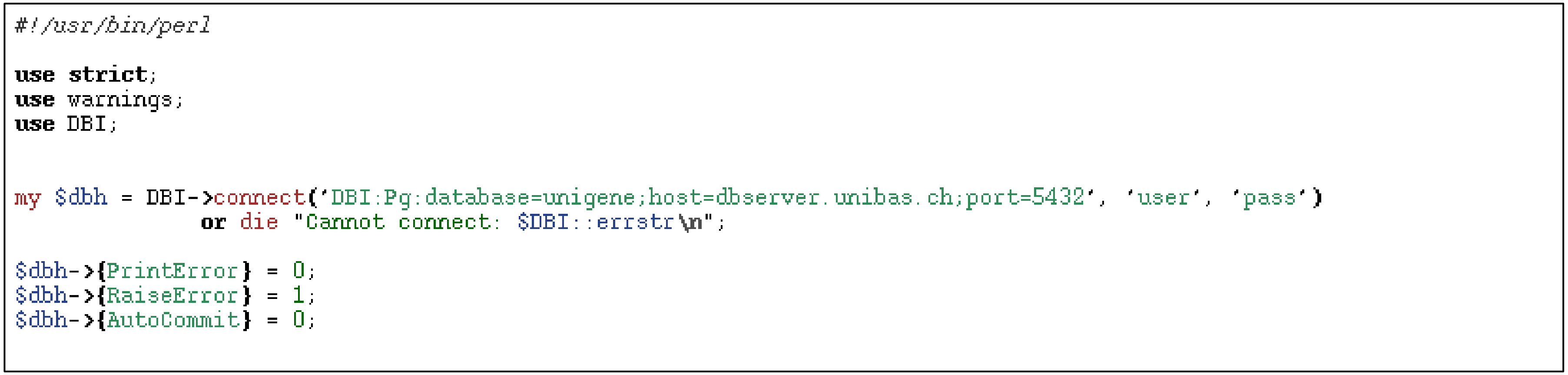
```
#!/usr/bin/perl

use strict;
use warnings;
use DBI;


my $attrs = { PrintError => 0, RaiseError => 1, AutoCommit => 0, FetchHashKeyName => 'NAME_lc', LongTrunkOk => 0 };

my $dbh = DBI->connect('DBI:Oracle:host=dbserver.unibas.ch;sid=unigene;port=1521', 'user', 'pass', $attrs)
              or die "Cannot connect: $DBI::errstr\n";
```



```
#!/usr/bin/perl

use strict;
use warnings;
use DBI;
use Carp ();

my $attrs = { PrintError => 0, RaiseError => 1, AutoCommit => 0, HandleError => sub { Carp::confess(shift) } };


my $dbh = DBI->connect('DBI:Pg:database=unigene;host=dbserver.unibas.ch;port=5432', 'user', 'pass', $attrs)
              or die "Cannot connect: $DBI::errstr\n";
```

• The Database Handle Object ($dbh)

Database handle object attributes can also be set after the connection:



```
#!/usr/bin/perl

use strict;
use warnings;
use DBI;


my $dbh = DBI->connect('DBI:Pg:database=unigene;host=dbserver.unibas.ch;port=5432', 'user', 'pass')
                 or die "Cannot connect: $DBI::errstr\n";

$dbh->{PrintError} = 0;
$dbh->{RaiseError} = 1;
$dbh->{AutoCommit} = 0;
```

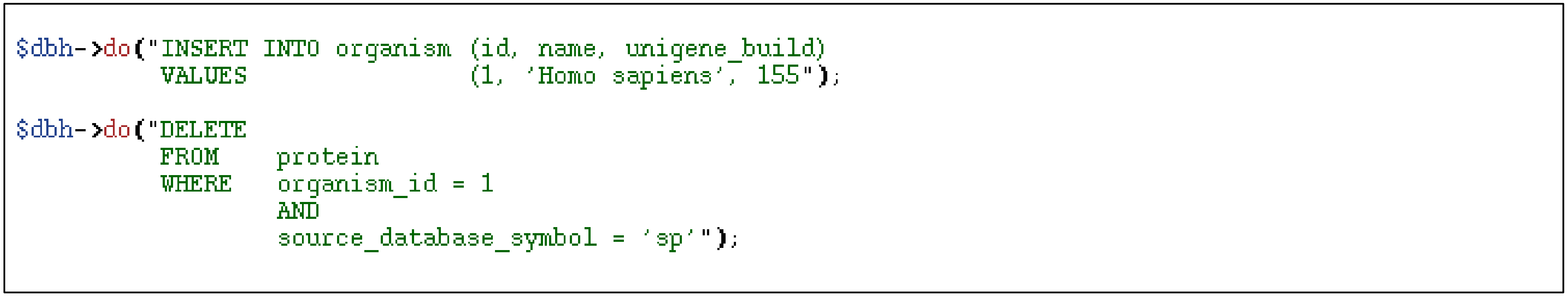
– You can have multiple database connections (database handles) in the same script and they do not affect each other. They are separate, distinct, sessions to the database (or different databases or DBMSs).

- ## The Database Handle Object (`$dbh`)

  - ### Some important methods:

    - `$dbh->data_sources([ \%attrs ])`
      - Returns a list of data sources (databases) available for the connection credentials used in DBI->connect.
    - `$dbh->do($sql_statement, [ \%attrs, @bind_values ])`
      - Used to prepare and execute a single SQL statement. Returns the number of rows affected or `undef` on error. Used only for non-`SELECT` statements (`INSERT`, `UPDATE` and `DELETE`) since it does not fetch any data and when you don’t want to execute the statement repeatedly. Use this for DDL.

Examples:

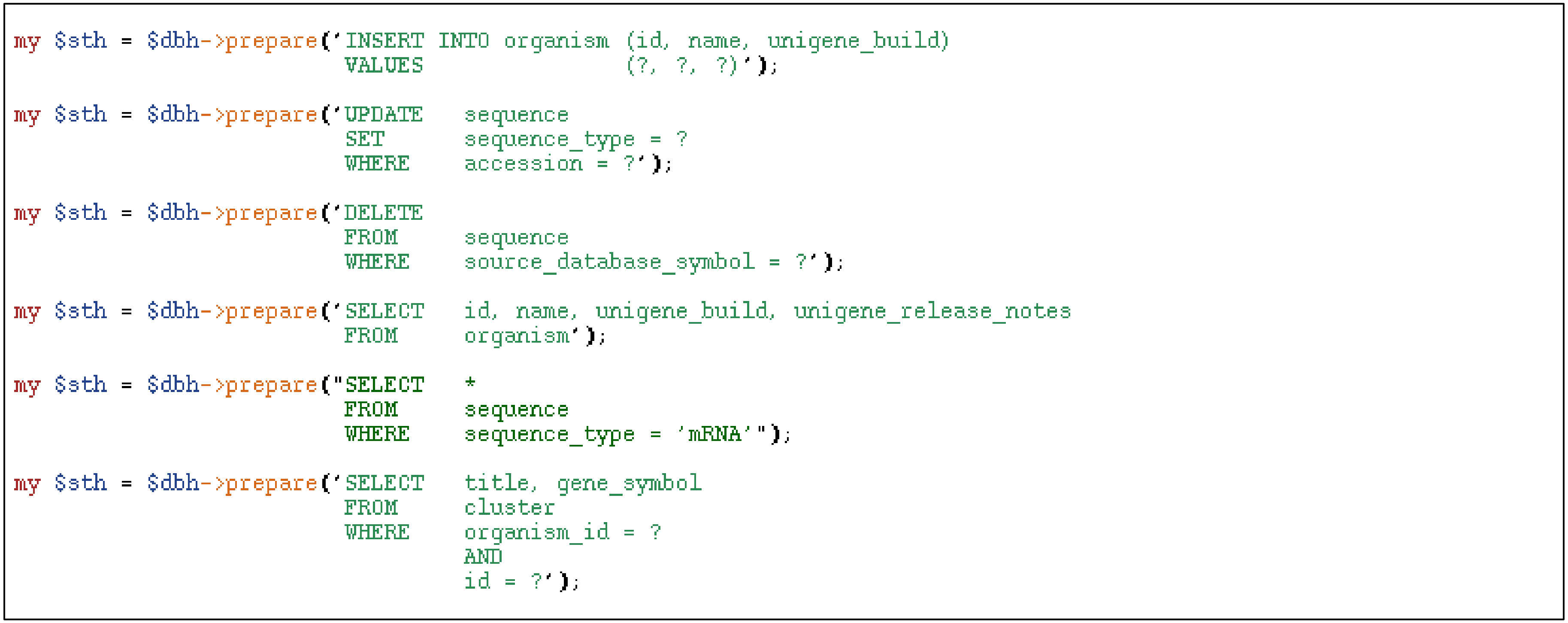


```
$dbh->do("INSERT INTO organism (id, name, unigene_build)
         VALUES               (1, ‘Homo sapiens’, 155");

$dbh->do("DELETE
         FROM    protein
         WHERE   organism_id = 1
                 AND
                 source_database_symbol = ‘sp’");
```

- The Database Handle Object (`$dbh`)
  - Preparing SQL statements:
    - `$dbh->prepare($sql_statement, [ \%attrs ])`
      - Prepares a SQL statement for later execution by the DBMS and returns a statement handle object. Typically the SQL statement is sent directly to the DBMS to be parsed and prepared. DBD::* drivers which have no concept of preparing a statement will typically just store the statement for later use. Use prepare for all `SELECT` statements and for `INSERT`, `UPDATE` and `DELETE` statements that will be executed repeatedly.

- ## The Database Handle Object ($dbh)

$dbh->prepare Examples:

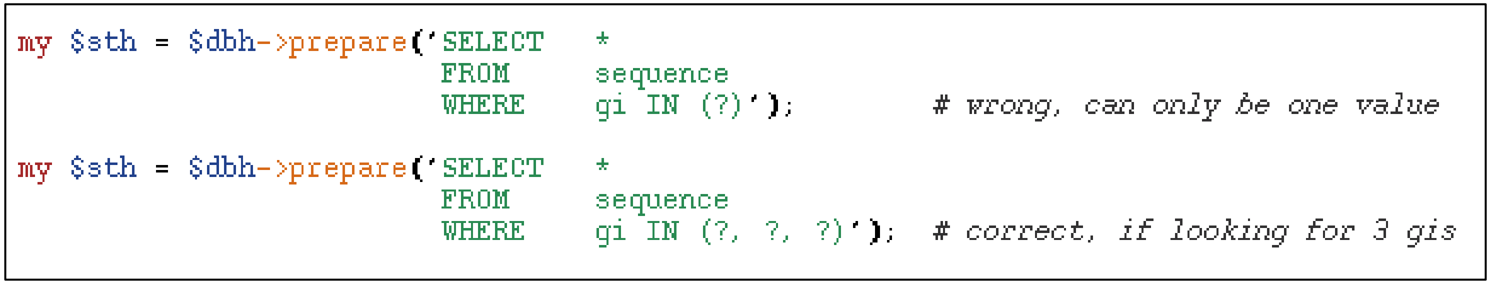


```
my $sth = $dbh->prepare('INSERT INTO organism (id, name, unigene_build)
                         VALUES               (?, ?, ?)');

my $sth = $dbh->prepare('UPDATE   sequence
                         SET      sequence_type = ?
                         WHERE    accession = ?');

my $sth = $dbh->prepare('DELETE
                         FROM     sequence
                         WHERE    source_database_symbol = ?');

my $sth = $dbh->prepare('SELECT   id, name, unigene_build, unigene_release_notes
                         FROM     organism');

my $sth = $dbh->prepare("SELECT   *
                         FROM     sequence
                         WHERE    sequence_type = 'mRNA'");

my $sth = $dbh->prepare('SELECT   title, gene_symbol
                         FROM     cluster
                         WHERE    organism_id = ?
                                  AND
                                  id = ?');
```

- ## The Database Handle Object (`$dbh`)

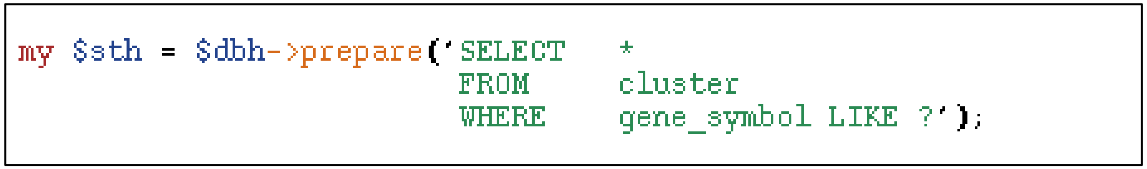
  - ### Placeholders (`?`) used in `$dbh->prepare`

    - *Placeholders*, or parameter markers, are used to indicate values in a SQL statement that will be supplied before the SQL statement is executed.
    - The association of actual values with the placeholders is called *binding*, and the values themselves are called *bind values*.
    - Note that ? is NOT enclosed in quotation marks, even when the placeholder represents a string.
    - With most drivers, placeholders CANNOT be used for any element of a SQL statement that would prevent the DBMS from parsing and validating the SQL statement and creating a query execution plan for it (done during the `prepare`). Placeholders CANNOT be used for things such as table names and column names.
    - Placeholders can only represent single scalar values.



```
my $sth = $dbh->prepare('SELECT   *
                         FROM     sequence
                         WHERE    gi IN (?)');        # wrong, can only be one value

my $sth = $dbh->prepare('SELECT   *
                         FROM     sequence
                         WHERE    gi IN (?, ?, ?)'); # correct, if looking for 3 gis
```

- The Database Handle Object ($dbh)
  - Placeholders (?) used in $dbh->prepare
    - When using placeholders with the SQL LIKE qualifier, remember that the placeholder substitutes for the entire LIKE string. You should use a single ? and include any LIKE wildcard characters in the bind value.



```
my $sth = $dbh->prepare(‘SELECT  *
                         FROM    cluster
                         WHERE   gene_symbol LIKE ?’);
```

    - In DBI, undef represents NULL values. Remember in SQL that to return rows where a field is NULL you DO NOT put in the WHERE clause ”gene_symbol = NULL” but instead ”gene_symbol IS NULL”. Therefore do not use placeholders and bind undef to it to return rows where a field is NULL.
    - Performance: without placeholders, SQL statements executed repeatedly (e.g. INSERT statements to load data into a table) would have to contain the literal values in the SQL statement and be re-prepared each and every time. prepare calls can be expensive and need to be done only once with placeholders.
    - Using placeholders allows you to not have to worry about escaping quotes in your SQL statements because this is taken care of for you by DBI during binding.

- # The Database Handle Object (`$dbh`)

  - ## Preparing SQL statements:

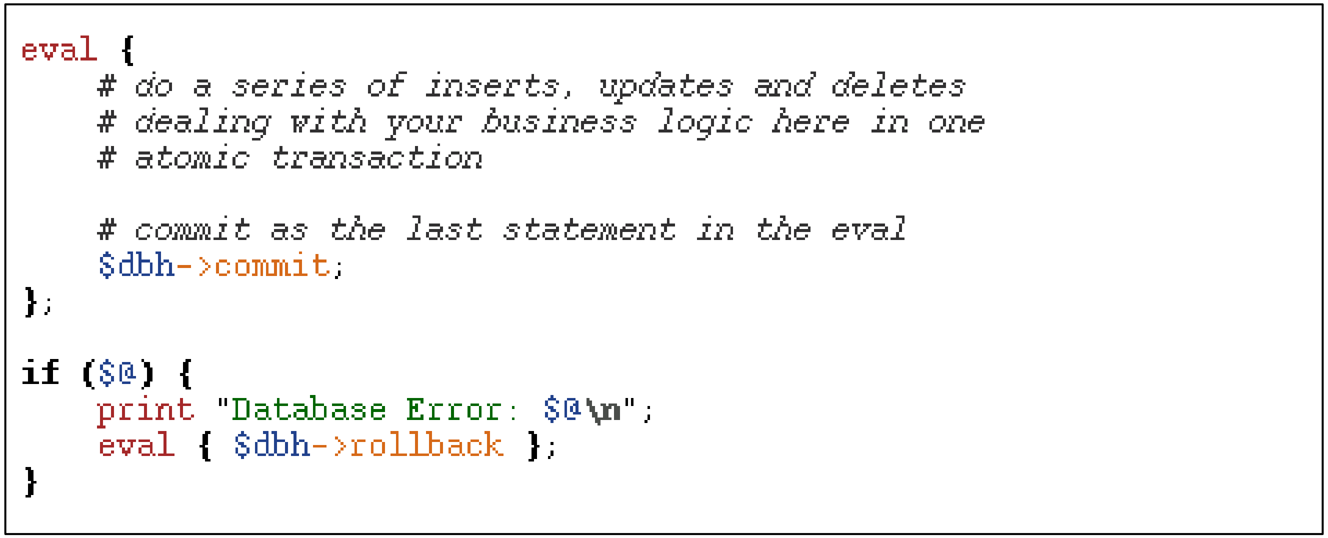
    - `$dbh->prepare_cached($sql_statement, [ \%attrs, $if_active ])`

      - Just like `prepare` except that the statement handle return will be stored away in a hash associated with the database handle (`$dbh`). If another call is made to `prepare_cached` with the same `$sql_statement` and `%attrs`, the cached statement handle will be returned without contacting the DBMS.
      - Usually used to improve performance in larger Perl applications like web applications which have persistent database handles that do not get destroyed when scripts in the web application exit.
      - You need to be careful with `prepare_cached` because you can get yourself into trouble if you have nested `prepare_cached` calls with the same parameters. DBI will return the cached statement handle which will still be active and will not be where you want. The `$if_active` parameter, which is by default set to 0, will issue a warning and reset the statement handle before returning it to you. You can vary this behavior by setting `$if_active` to other values (see DBI documentation).

- ## The Database Handle Object ($dbh)

  - ### Transactions

    - $dbh->commit
      - Commit and make permanent any database changes (INSERT, UPDATE and DELETE) since the last commit or rollback if the DBMS supports transactions and AutoCommit is OFF.
    - $dbh->rollback
      - Rollback and undo any database changes (INSERT, UPDATE and DELETE) since the last commit or rollback if the DBMS supports transactions and AutoCommit is OFF.



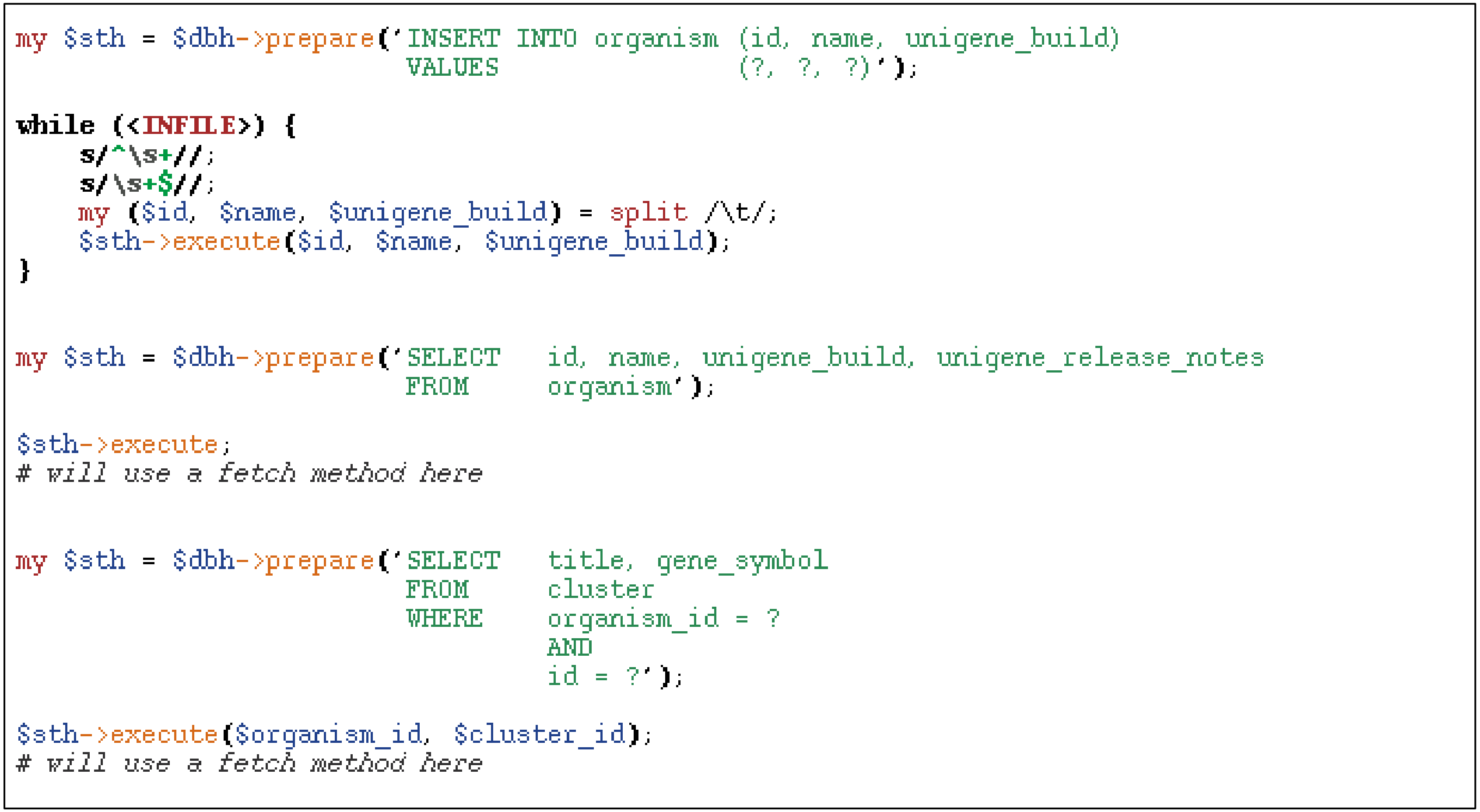
```
eval {
    # do a series of inserts, updates and deletes
    # dealing with your business logic here in one
    # atomic transaction

    # commit as the last statement in the eval
    $dbh->commit;
};

if ($@) {
    print "Database Error: $@\n";
    eval { $dbh->rollback };
}
```

- ## The Database Handle Object (`$dbh`)
  - ### Disconnecting from the database/DBMS:
    - `$dbh->disconnect`
      - Disconnects the database from the database handle. This is done just before the Perl program exits.
      - The transaction behavior of `disconnect` varies from DBD::* driver to driver. To avoid any problems and warnings and to make sure DBI is doing exactly what you want, call `commit` or `rollback` just before `disconnect`. It is good style to call `rollback` because it make sense to undo any incomplete transactions that your code has not had a chance to `commit`.
      - Also, if you `disconnect` from the database while you still have active statement handles you will get a warning. Make sure to call use the `$sth->finish` method on any active statement handles.

- The Statement Handle Object (`$sth`)
  - Executing a prepared SQL statement:
    - `$sth->execute([ @bind_values ])`
      - Processes and executes the prepared SQL statement. Returns `undef` if an error occurs. A successful `execute` always returns true regardless of the number of rows affected (even if its zero are affected because `execute` returns "0E0").
      - For a non-`SELECT` SQL statement, `execute` returns the number of rows affected, if known.
      - For a `SELECT` SQL statement, `execute` "starts" the query within the DBMS. You need to use one of the fetch methods to retrieve data after calling execute. The `execute` method DOES NOT return the number of rows affected because most DBMSs cannot tell you this in advance, it simply returns a true value.
      - If `@bind_values` are given, then `execute` will effectively bind the to the placeholders for you using `$sth->bind_param` for each value before executing the statement.

- ## The Statement Handle Object ($sth)

$sth->execute examples:

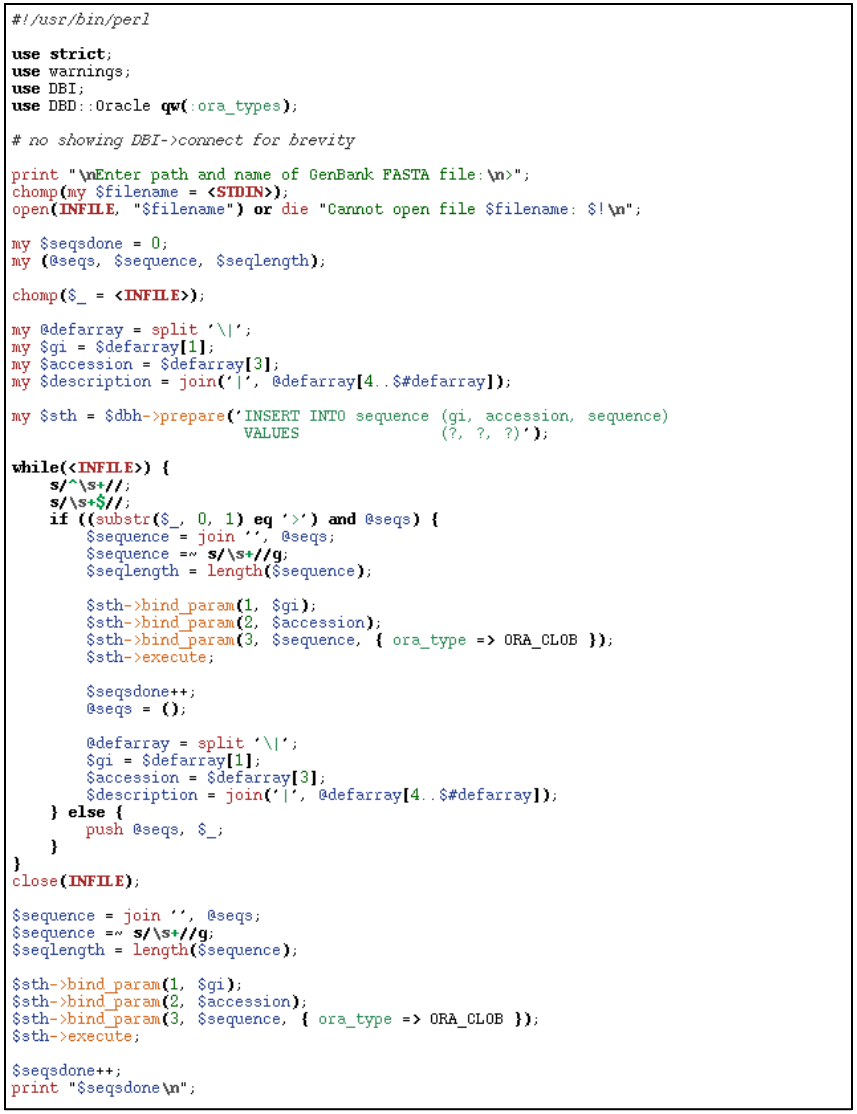


```
my $sth = $dbh->prepare('INSERT INTO organism (id, name, unigene_build)
                         VALUES               (?,  ?,  ?)');

while (<INFILE>) {
    s/^\s+//;
    s/\s+$//;
    my ($id, $name, $unigene_build) = split /\t/;
    $sth->execute($id, $name, $unigene_build);
}


my $sth = $dbh->prepare('SELECT   id, name, unigene_build, unigene_release_notes
                         FROM     organism');

$sth->execute;
# will use a fetch method here


my $sth = $dbh->prepare('SELECT   title, gene_symbol
                         FROM     cluster
                         WHERE    organism_id = ?
                                  AND
                                  id = ?');

$sth->execute($organism_id, $cluster_id);
# will use a fetch method here
```

- The Statement Handle Object (`$sth`)
  - Additional utility methods for non-`SELECT` statments:
    - `$sth->bind_param($placeholder_num, $bind_value, \%attr)`
      - Binds a value to a non-SELECT SQL statement previously prepared by `$dbh->prepare`. Placeholder numbers start from 1. The `\%attr` parameter can be used to hint at the datatype the placeholder should have.

- ## The Statement Handle Object ($sth)

$sth->bind_param example:

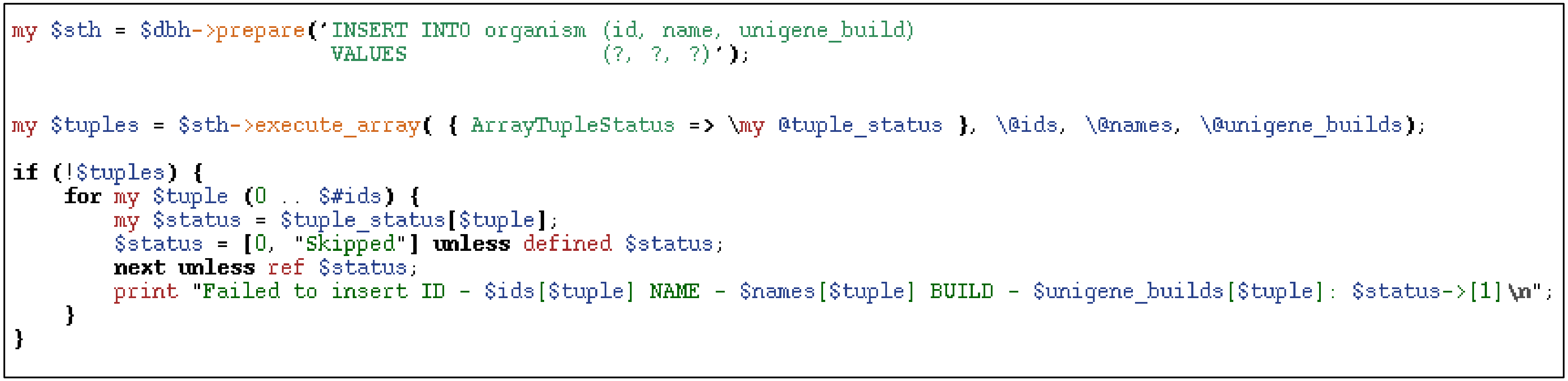


```
#!/usr/bin/perl

use strict;
use warnings;
use DBI;
use DBD::Oracle qw(:ora_types);

# no showing DBI->connect for brevity

print "\nEnter path and name of GenBank FASTA file:\n>";
chomp(my $filename = <STDIN>);
open(INFILE, "$filename") or die "Cannot open file $filename: $!\n";

my $seqsdone = 0;
my (@seqs, $sequence, $seqlength);

chomp($_ = <INFILE>);

my @defarray = split '\|';
my $gi = $defarray[1];
my $accession = $defarray[3];
my $description = join('|', @defarray[4..$#defarray]);

my $sth = $dbh->prepare('INSERT INTO sequence (gi, accession, sequence)
                         VALUES                (?, ?, ?)');

while(<INFILE>) {
    s/^\s+//;
    s/\s+$//;
    if ((substr($_, 0, 1) eq '>') and @seqs) {
        $sequence = join '', @seqs;
        $sequence =~ s/\s+//g;
        $seqlength = length($sequence);

        $sth->bind_param(1, $gi);
        $sth->bind_param(2, $accession);
        $sth->bind_param(3, $sequence, { ora_type => ORA_CLOB });
        $sth->execute;

        $seqsdone++;
        @seqs = ();

        @defarray = split '\|';
        $gi = $defarray[1];
        $accession = $defarray[3];
        $description = join('|', @defarray[4..$#defarray]);
    } else {
        push @seqs, $_;
    }
}
close(INFILE);

$sequence = join '', @seqs;
$sequence =~ s/\s+//g;
$seqlength = length($sequence);

$sth->bind_param(1, $gi);
$sth->bind_param(2, $accession);
$sth->bind_param(3, $sequence, { ora_type => ORA_CLOB });
$sth->execute;

$seqsdone++;
print "$seqsdone\n";
```

- The Statement Handle Object (`$sth`)
  - Executing a prepared SQL statement:
    - `$sth->execute_array(\%attrs, [ @bind_values ])`
      - Executes a prepared SQL statement once for each *tuple* (group of values) provided either in @bind_values or via a reference passed in \%attrs.
      - With the present implementation this is only used for non-`SELECT` SQL statements and `execute_array` returns the number of tuples successfully executed.
      - If any tuple execution returns an error, then `execute_array` will return `undef`.
      - A successful `execute_array` always returns true regardless of the number of tuples successfully executed (even if its zero are affected because `execute_array` returns ”0E0”).
      - The mandatory attribute `ArrayTupleStatus` in `%attrs` is used to specify a reference to an array which will receive the execute status of each executed parameter tuple.
      - The attribute `ArrayTupleFetch` can be used to specify a reference to a subroutine that will be called to provide the bind values for each tuple execution. The subroutine should return an reference to an array which contains the appropriate number of bind values, or return an `undef` if there is no more data to execute.

- # The Statement Handle Object ($sth)

$sth->execute_array example:



```
my $sth = $dbh->prepare(‘INSERT INTO organism (id, name, unigene_build)
                         VALUES                (?, ?, ?)’);


my $tuples = $sth->execute_array( { ArrayTupleStatus => \my @tuple_status }, \@ids, \@names, \@unigene_builds);

if (!$tuples) {
    for my $tuple (0 .. $#ids) {
        my $status = $tuple_status[$tuple];
        $status = [0, "Skipped"] unless defined $status;
        next unless ref $status;
        print "Failed to insert ID - $ids[$tuple] NAME - $names[$tuple] BUILD - $unigene_builds[$tuple]: $status->[1]\n";
    }
}
```

- The Statement Handle Object (`$sth`)
  - Fetching data methods (after `$sth->execute`):
    - `$sth->fetchrow_arrayref` (alias: `$sth->fetch`)
      - Fetches the next row of data and returns a reference to an array holding the field values. `NULL` fields are returned as `undef` values in the array. This is the fastest way to fetch data, particularly if used with `$sth->bind_columns`.
      - Returns `undef` when there is no more data left to fetch.
    - `$sth->fetchrow_array`
      - An alternative to `$sth->fetchrow_arrayref`. Returns an empty list when there is no more data left to fetch.
    - `$sth->fetchrow_hashref`
      - An alternative to `$sth->fetchrow_arrayref`. Returns a reference to a hash containing field name and value pairs (remember `FetchKeyHashName`).

- ## The Statement Handle Object (`$sth`)

Fetching data methods examples:

```
my $sth = $dbh->prepare("SELECT   gi, accession, sequence, is_representative_sequence
                         FROM     sequence
                         WHERE    sequence_type = ’mRNA’");

$sth->execute;
while (my $row = $sth->fetchrow_arrayref) {
    print "GI: $row->[0]\nACCESSION: $row->[1]\nREPRESENTATIVE: $row->[3]\n\n";
}


my $sth = $dbh->prepare(’SELECT   A.title, A.tissue_type, A.developmental_stage
                         FROM     library A INNER JOIN organism B ON A.organism_id = B.id
                         WHERE    B.name = ?’);

$sth->execute(’Homo sapiens’);
while (my $row = $sth->fetchrow_hashref) {
    print "TITLE: $row->{title}\nTISSUE TYPE: $row->{tissue_type}\nDEV STAGE: $row->{developmental_stage}\n\n";
}


my $sth = $dbh->prepare(’SELECT   A.gi, A.source_database_symbol, A.source_database_id
                         FROM     protein A INNER JOIN organism B ON A.organism_id = B.id
                         WHERE    B.name = ?’);

$sth->execute(’Homo sapiens’);
while (my ($gi, $source_db_symbol, $source_db_id) = $sth->fetchrow_array) {
    print "GI: $gi\nSOURCE DB SYMBOL: $source_db_symbol\nSOURCE ID: $source_db_id\n\n";
}
```

- ## The Statement Handle Object ($sth)

  - Additional utility methods used after $sth->execute and before fetching:

    - `$sth->bind_col($column_num, \$var_to_bind, \%attr)`

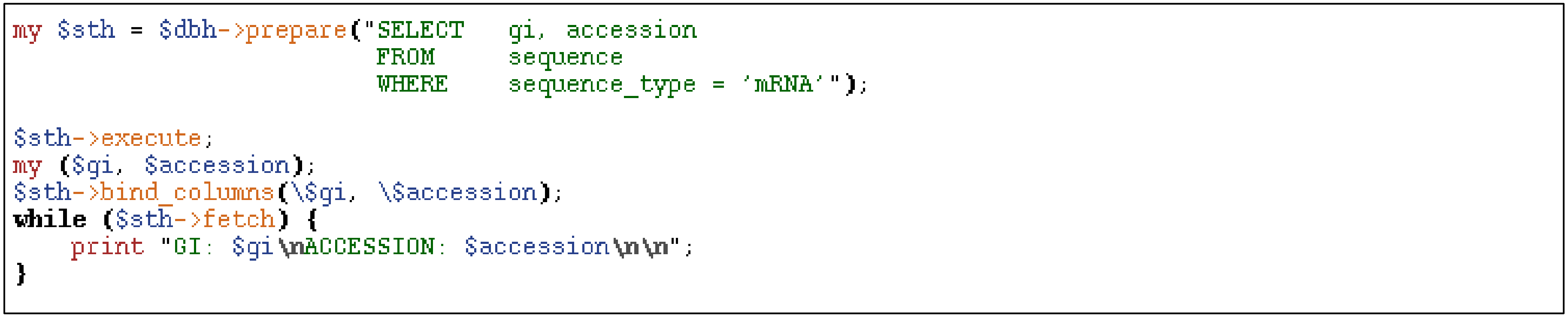
      - Binds a Perl variable to an output column of the `SELECT` SQL statement previously executed by `$sth->execute`. Column numbers start from 1. The `\%attr` parameter can be used to hint at the datatype the column should have so as to return the data in a certain format versus the DBMS native formatting (e.g. DATE and TIMESTAMP datatypes). Use `$sth->fetch` after doing `$sth->bind_col`.

    - `$sth->bind_columns (@list_of_refs_of_vars_to_bind)`

      - Shortcut method which calls `$sth->bind_col` for each column of the `SELECT` SQL statement and binds each column to each variable in `@list_of_refs_of_vars_to_bind`, respectively. Fails with a `die` if the number of variables to bind does not equal the number of columns in the `SELECT` statement. Use `$sth->fetch` after doing `$sth->bind_columns`.

- ## The Statement Handle Object ($sth)

$sth->bind_columns example:



```
my $sth = $dbh->prepare("SELECT   gi, accession
                         FROM     sequence
                         WHERE    sequence_type = ’mRNA’");

$sth->execute;
my ($gi, $accession);
$sth->bind_columns(\$gi, \$accession);
while ($sth->fetch) {
    print "GI: $gi\nACCESSION: $accession\n\n";
}
```

The Database Handle Object ($dbh)

All-in-one data retrieval methods:

- `$dbh->selectrow_arrayref($sql_statement, [ \%attr, @bind_values ])`
  - Utility method combines `prepare`, `execute` and `fetchrow_arrayref` into a single call. Returns an array reference to the one row of data (or first row of data returned). Should only be used if your SQL statement is returning ONE row. The `$sql_statement` parameter can also be a previously prepared statement handle, in which case `prepare` is skipped.
- `$dbh->selectrow_array($sql_statement, [ \%attr, @bind_values ])`
  - Utility method combines `prepare`, `execute` and `fetchrow_array` into a single call. Returns an array containing one row of data (or first row of data returned). Should only be used if your SQL statement is returning ONE row. The `$sql_statement` parameter can also be a previously prepared statement handle, in which case `prepare` is skipped.
- `$dbh->selectrow_hashref($sql_statement, [ \%attr, @bind_values ])`
  - Utility method combines `prepare`, `execute` and `fetchrow_hashref` into a single call. Returns an hash reference containing one row of data (or first row of data returned). Should only be used if your SQL statement is returning ONE row. The `$sql_statement` parameter can also be a previously prepared statement handle, in which case `prepare` is skipped.

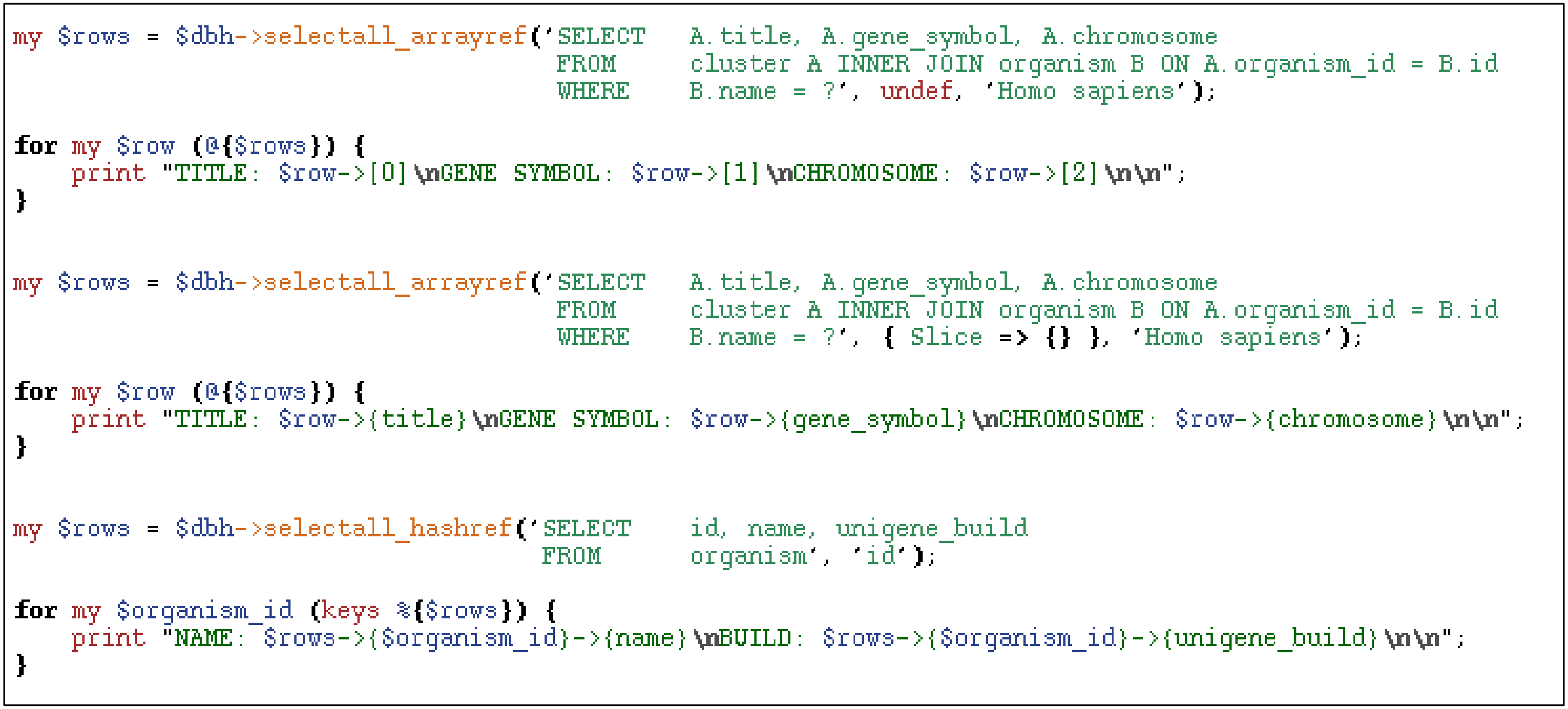
- # The Database Handle Object ($dbh)

  - ## All-in-one data retrieval methods:

    - `$dbh->selectall_arrayref($sql_statement, [ \%attrs, @bind_values ])`
      - Utility method combines prepare, execute and fetchall_arrayref into a single call. Returns an reference to an array which contains in each element an array reference to each row of data. The $sql_statement parameter can also be a previously prepared statement handle, in which case prepare is skipped. The %attrs parameter can have the Slice, Columns, and MaxRows attributes. Slice causes $dbh->selectall_arrayref to return hash references for each row (or to only return certain columns as hash references). Columns causes $dbh->selectall_arrayref to return only certain column numbers. MaxRows specifies the maximum number of rows you want returned.
    - `$dbh->selectall_hashref($sql_statement, [  $key_field, \%attrs, @bind_values ])`
      - Utility method combines prepare, execute and fetchall_hashref into a single call. Returns a reference to a hash keyed by $key_field and the values which are hash references for each row of data. The $sql_statement parameter can also be a previously prepared statement handle, in which case prepare is skipped.
    - `$dbh->selectcol_arrayref($sql_statement, [ \%attrs, @bind_values ])`
      - Utility method combines prepare, execute and fetching ONE column for all the rows into a single call. Returns an array reference containing containing the values of that column. The $sql_statement parameter can also be a previously prepared statement handle, in which case prepare is skipped.

- ## The Statement Handle Object ($sth)

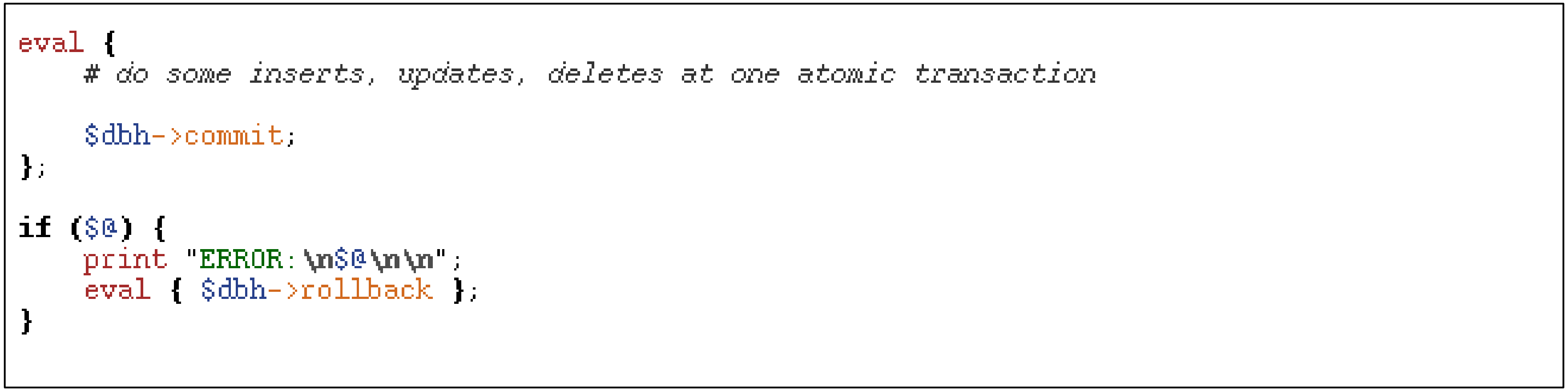
All-in-one examples:



```
my $rows = $dbh->selectall_arrayref('SELECT   A.title, A.gene_symbol, A.chromosome
                                     FROM     cluster A INNER JOIN organism B ON A.organism_id = B.id
                                     WHERE    B.name = ?', undef, 'Homo sapiens');

for my $row (@{$rows}) {
    print "TITLE: $row->[0]\nGENE SYMBOL: $row->[1]\nCHROMOSOME: $row->[2]\n\n";
}


my $rows = $dbh->selectall_arrayref('SELECT   A.title, A.gene_symbol, A.chromosome
                                     FROM     cluster A INNER JOIN organism B ON A.organism_id = B.id
                                     WHERE    B.name = ?', { Slice => {} }, 'Homo sapiens');

for my $row (@{$rows}) {
    print "TITLE: $row->{title}\nGENE SYMBOL: $row->{gene_symbol}\nCHROMOSOME: $row->{chromosome}\n\n";
}


my $rows = $dbh->selectall_hashref('SELECT    id, name, unigene_build
                                    FROM      organism', 'id');

for my $organism_id (keys %{$rows}) {
    print "NAME: $rows->{$organism_id}->{name}\nBUILD: $rows->{$organism_id}->{unigene_build}\n\n";
}
```

- The Statement Handle Object (`$sth`)
  - To tidy up a statement handle `$sth`
    - `$sth->finish`
      - Indicates to DBI that no more data will be fetched from the statement handle before it is executed again (thus starting over) or destroyed. When all the data has been retrieved from a statement handle, DBD::* drivers should automatically call the `finish` method for you. It is useful in larger Perl applications when you want to make sure not to get a warning when accidentally disconnecting from the database with `$dbh->disconnect` while still having unfinished statement handles open.

- **Perl DBI**
  - Doing a transaction



```
eval {
    # do some inserts, updates, deletes at one atomic transaction

    $dbh->commit;
};

if ($@) {
    print "ERROR:\n$@\n\n";
    eval { $dbh->rollback };
}
```

- DBD::* driver specifics
  - MySQL
    - `$sth->mysql_insertid`
    - `$sth->mysql_store_result`, `$sth->mysql_use_result`
  - Oracle
    - `use DBD::Oracle qw(:ora_types);`
    - Oracle environment variables need to be set in the shell (particularly ORACLE_HOME)
    - Need to use `$dbh->{LongReadLen}`, `$dbh->{LongTruncOk}`
    - Need to use `$sth->bind_param` with datatype hint during `INSERT` or `UPDATE` of BLOB, CLOB, LONG columns.