Development of a Data Management Architecture for the Support of Collaborative Computational Biology

Lance W. Feagan

Submitted to the Department of Electrical Engineering & Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Defended: March 31st, 2008

Thesis Committee:

Dr. Terry Clark: Chairperson

Dr. Victor Frost

Dr. David Andrews

The Thesis Committee for Lance Warren Feagan certifies that this is the approved version of the following thesis:

Development of a Data Management Architecture for the Support of Collaborative Computational Biology

Approved: March 31st, 2008

Thesis Committee:

Dr. Terry Clark: Chairperson

Dr. Victor Frost

Dr. David Andrews

Abstract

The collection, management, and dissemination of provenance is vital to all researchers and is particularly important for bioinformaticians because of the exponential growth of data available for analysis. While robust and featureful provenance management systems exist in areas other than bioinformatics, none of the systems that support computational bioinformatics are capable of managing all provenance areas. Lastly, few systems feature a flexible security model that encourages collaboration while still ensuring that provenance information is retained. The BCJ-DB addresses these problems through a user-extensible type hierarchy, a data abstraction model that supports powerful data and meta-data search while preserving computational performance, a security model that supports fine-grained controls on data sharing, and an extension to the RDF 3-tuple that defines relationships between the subject and object that are "essential" and will be enforced by the security model to prevent loss of important provenance.

Table of Contents

Chapter I: Introduction

Bioinformatics Challenges

The volume of information present in the bioinformatics field is increasing at an exponential rate. Consequently, researchers must be able to rapidly discover, assimilate, utilize and organize information while maintaining vital provenance about the data being used in their experiments, the analysis performed, and the results of those analyses. In a more traditional biology setting, where a wet-lab is the primary experimentation facility, researchers have not typically been forced to deal with daily data updates from a variety of sources. In this environment, the traditional paper laboratory journal was sufficient. Copying and pasting in the pages output from a mass spectrometer or a similar analysis device was typically the most interaction necessary with any electronic device. In the modern world of bioinformatics, however, traditional paper notebook methods fail to provide the capabilities researchers need to manage their workflow. No longer can an experimenter hope to print out and subsequently manage the volume of pages that would be required to describe the work being done.

In the 2nd International Workshop on Data Integration in the Life Sciences, Shankar Subramaniam, Professor of Bioengineering and Chemistry gave a keynote speech titled "Challenges in Biological Data Integration in the Post-Genome Sequence Era" wherein he stated "the standard paradigm in biology that deals with '*hypothesis to experimentation (low throughput data) to models*' is being gradually replaced by *'data to hypothesis to models'* and *'experimentation to more data and models'*," [Gupt06]. From this statement is is clear why bioinformatics is fundamentally different from more traditional biology research. Instead of beginning with hypothesis formation, bioinformatics researchers start with large volumes of data to analyze in the hopes of arriving at some conclusion. As an interesting corollary, high-energy physics has also changed in a similar way. The Large Hadron Collider being built near Geneva, Switzerland will produce data at a rate of around 1 Terabyte per second at the front-end. Within this data lie patterns that physicists will try to pry information out of [Rose06]. To address this paradigm shift Subramaniam suggests "robust data repositories that allow interoperable navigation, query and analysis across diverse data, a plug-and-play environment that will facilitate seamless interplay of tools and data and versatile biologist-friendly user interfaces," [Gupt06].

The concerns that need to be addressed in bioinformatics primarily relate to rigorous provenance collection, tracking, and usage. Prior work developing provenance management systems (PMS) failed to provide a comprehensive, integrated solution. In the second keynote of the DILS workshop, Peter Buneman listed key challenges in curated databases as being "annotation, provenance, archiving, publishing and security," [Gupt06]. To implement a system that will advance the efficency with which computational biologists work requires a PMS that integrates functionality providing rigorous provenance maintenance, data archiving, publishing, data security, and collaboration capabilities.

Why is Provenance Important?

Provenance plays a vital role in assisting humans in understanding the context work was done in and the specifications used to derive results. As described by Jim Gray, provenance should make it possible for:

> "scientific data [to] remain available forever so that other scientists can reproduce the results and do new science with the data…To understand the data, those later users need the *metadata*: (1) how the instruments were designed and built; (2) when, where, and how the data was gathered; and (3) a careful description of the processing steps that led to the derived data products that are typically used for scientific data analysis." [Gray02]

In a traditional setting, this is accomplished through careful note taking in laboratory notebooks. In an electronic setting, however, capturing the data related to the three categories of metadata information and maintaining it in a readily accessible format with proper correspondence to results can be challenging. This is because, as discussed by Gupta, the standard paradigm of moving from a well-researched hypothesis to a carefully-constructed experiment to test the hypothesis to a model has been reversed. This reversal places data at the start of the process and follows with hypothesis and models that then produce further experimentation resulting in yet more data and models. Thus, fundamentally, the paradigm has shifted from a low-throughput model where information collected will often be related to the work at hand to one where many failed experimental attempts may occur. In a high-throughput scenario, the task of careful note-taking suffers from being useful in only a small percentage of the test cases. This diminished return on time invested encourages bioinformatics researchers to abandon their old habits and adopt new habits. These new habits rarely include careful recording of experimental steps

as the experiment proceeds. Rather, they encourage seeking out results of import first and later establishing the procedure used to obtain those results.

In "A Survey of Data Provenance Techniques," Yogesh L. Simmhan et al. provided the provenance researcher with a carefully developed taxonomy that can be used to characterize the capabilities and focus of a provenance tracking system [Simm05a]. The diagram, replicated below, uses square-corner boxes to represent an attribute that a provenance system may contain and round-corner boxes to represent the possible values the attribute may take on. Where the attribute may take on only one of the possible values, a circle is used to denote that the values are an 'OR' type relationship. If no circle is present, the attribute may take on zero or more of the possible values, including all of the values (0...N). For instance, the subject of provenance may be either process or data oriented. Provenance dissemination may be through a visual graph and/or a query and/or a service API.

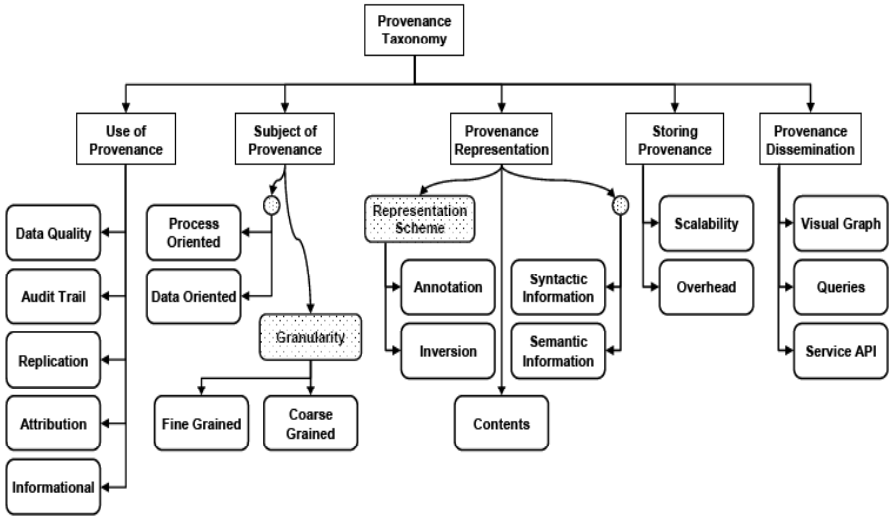


Figure 1: Provenance Taxonomy [Simm05b]

Of particular importance in the taxonomy are the five uses of provenance. *Data quality* provenance is used to evaluate the fitness of a data set for a particular application. Typically lineage is used to estimate this property based on the source of the data and the transformations that have been applied to the data. *Audit trail* provenance traces the process used to produce data. This provides information about resource usage and may help detect errors in data generation. *Replication recipes* provenance establishes the steps used

to derive a particular data set and should make it possible to reproduce results with the original data or maintain the currency of data by running the same steps on new input data. *Attribution* pedigree establishes the ownership of the source data used to generate a set of results. *Informational* provenance is generic information used to aid in finding data of interest. A common synonym for informational provenance is an annotation. Informational provenance can be particularly useful in helping other researchers place results in the proper context.

It should be noted that other researchers have developed slightly different perspectives on the provenance problem based on their goals and the terminology they have used to decompose the problem. For example, at the 2006 ACM SIGMOD Conference in the paper, "Provenance Management in Curated Databases," Peter Buneman described the two types of provenance as workflow and data flow. Workflow provenance, also called coarse-grained provenance, is information describing how derived data has been calculated from raw observations. Dataflow provenance, also known as fine-grained provenance, describes how data has moved through one or more databases. These two definitions suggest that to have fine-grained provenance in a set of curated databases implies a specific set of functionality related to managing provenance information between databases. Nevertheless, taken in the larger context, Buneman clearly is describing a situation where multiple databases that have data movement between them must be capable of differentiating the data that has moved. In a single database system, perhaps one that focuses on data warehousing and archiving, tracking the movement of data within the database is essential and provides an equivalent description of granularity. Ultimately, however, the taxonomy developed by Simmhan et al. provides a more detailed, and therefore satisfying, metric for the evaluation of provenance management systems.

Chapter II: Related Work

Introduction

In the early 90's many computer researchers touted that the 'paperless' era had arrived. The concept of removing the need for paper was to be applied to all areas of life from the home to the workplace and from top to bottom in all businesses. Going 'paperless' promised to reduce wasted space, time, and money associated with creating, moving, and managing vast amounts of paper that were then required to run a business. Similarly, in the laboratory environment an electronic notebook was touted as being capable of replacing the traditional paper notebook. The advantages were to include rapid searching and easy collaboration between researchers.

In a 1995 interview, Raymond E. Dessy, emeritius professor of chemsitry at Virginia Polytechnic Institute, mentions a couple of useful bits of information to be considered when reading about related systems. With regards to the use of computers vs paper, he said, "To me, technology is a small part of the problem - 75% is people and getting them to change work habits, interact with other individuals in new ways, accept new technologies," [Anon95]. The ability to gain acceptance among the large community of biologists and chemists has always been one of the stumbling blocks with electronic notebook systems. Physicists, on the other hand, have been more apt to move to an electronic medium. The primary differentiator between the two areas is the emphasis physics places on computation. Although chemsistry often makes use of computers, the computations performed are typically done to supplement laboratory experiments, not in lieu of them. In pure biology, computation is almost entirely in the realm of statistics on field observations. Dessy goes on to discuss the potential gains from collaboration by stating:

> "Now the beauty of this is that not only can you do that, but other people in the laboratory can share those notebooks by taking results from more than one worker and linking them because they're related. Examining the results, they might perhaps see things that neither original worker did and share those insights with the original workers and other responsible individuals. [Anon95]

To this point, this all sounds great. However, he continues, "Here we start to get into the problem that people are going to resist sharing notebooks like this because they view them as private diaries," [Anon95]. To summarize, sharing of notebooks of information can be very powerful. Systems that only allow sharing an entire notebook, however, will fundamentally fail at collaboration attempts because of an ill-defined security model using objects that are too coarse-grained.

In the 2005 Nature article "Electronic Notebooks: A New Leaf", many improvements in the capabilities present in electronic laboratory notebooks are detailed, including: sharing between researchers, archiving, encryption of data, and verifiability. However, these systems typically lack the flexibility needed by academic researchers. Gwen Jacobs, head of the open-source NeuroSysm electronic notebook project at Montana State University says "Academic e-notebooks need to include user-friendly tools allowing researchers to customize them without demanding computing skills… Any system must be flexible enough to handle any type of data, and any experimental protocol," [Butl05]. Unfortunately, the concepts of customization provided by most systems are simply allowing users to make custom templates and user interface elements. Most systems cannot support extending the available types and then using those types with new tools that are added to the system.

Once the electronic notebook established its initial marque with projects such as NoteCards in mid 80s, the 90s largely continued in the same tradition. Electronic journals are typically based upon familiar concepts such as pages where items can be placed. More modern projects, such as Electronic Laboratory Notebook (ELN) and Virtual Notebook Environment (ViNE), often continue to use this interface paradigm.

NoteCards

One of the earliest significant attempts at developing a comprehensive electronic notebook system was done at the Xerox Palo Alto Research Center (PARC) with NoteCards in 1987 [Hala87a]. The similarities between NoteCards and the subject of this paper, the Bioinformatics Computational Journal Database (BCJ-DB), are striking and a thorough discussion providing insight into the design strengths and weaknesses is important. In the words of the Frank G. Halasz, one of the designers, NoteCards is "an extensible environment designed to help people formulate, structure, compare, and manage ideas," [Hala87a]. To accomplish this task, a semantic network composed of two primary objects and two supporting objects is created by the user. The two primary objects are:

1. *Notecards*, which "store an arbitrary amount of some editable 'substance'," and
2. Typed *links*, which are "binary connections between cards," and are the edges between nodes.

The first support object, a *browser*, is "a notecard that contains a structural diagram of a network of notecards." Lastly, a *FileBox* is a "specialized card that can be used to organize or categorize large collections of notecards."

NoteCards biggest strength was that it was developed using LISP on a Xerox workstation. This environment provided a multitude of features in libraries for manipulating the graphics area and data structures with ease. Its biggest weakness was that it only ran on the Xerox D workstation. This limited its ability to become widespread and the lifetime of NoteCards was effectively limited to that of the popularity of the Xerox D. This was, unfortunately, near the end of the era where LISP machines were in common use.

NoteCards used the concept of a semantic network that related notes. This network presented an tree-like graphical browser and manipulation tool which was very advanced and is rarely seen even today. Most provenance systems that feature display of a provenance graph do not allow interaction with the graph to manipulate the provenance network that connects the notes. This strength, however, was also a weakness. The user was in fact forced to manage the proper creation and maintenance of the network. This was tedious work and would prove unacceptable quickly for a biological researcher trying to get practical work accomplished. The network itself was only meant to handle collections of related ideas as the typed links. The browser assumed that links were related ideas. This limits links to only being useful for managing ideas, not provenance tracking between users of ideas/information.

The search capabilities within NoteCards were also limited. Specifically, only very localized searching inside a NoteFile for a particular bit of text in the title or the body of a text NoteCard was supported. Lastly, collaboration was greatly hindered by the NoteFile, which could only be accessed by one user at a time. While NoteCards was a wonderful system for managing ideas, it failed to be helpful while researchers were performing their work.

Virtual Notebook Environment (ViNE)
Website: http://www.csi.uoregon.edu/nacse/vine/

This project sought to develop a "platform-independent, web-based interface designed to support a range of scientific activities across distributed, heterogeneous computing platforms," [Skid98]. The design is based on the traditional paper notebook with extensions to support collaboration through data sharing, security through authentication and coarse-grained access control, and support for computational tool access and management. Some of the more notable features of this project include the ease with which new

computational tools can be integrated into the system, the visual experiment creation tool, and the access to distributed computational resources that may not be available on all machines participating in the ViNE environment.

Scientific Annotation Middleware (SAM)

Website: http://collaboratory.emsl.pnl.gov/docs/collab/sam/

The Scientific Annotation Middleware project, lead by Jim Myers and developed by the team that did the DOE2000 electronic notebook project, has been one of the most successful open-source electronic notebook projects. Development work began in 2001 at Pacific Northwest National Laboratory. SAM was originally developed as a component of the PNNL Electronic Laboratory Notebook (ELN) but has continued on with other projects, such and the Collaboratory for Multi-scale Chemical Science (CMCS) at Sandia National Laboratory (Website: http://cmcs.ca.sandia.gov). References to ELN are effectively a synonym for SAM. SAM is designed to support a secure, collaborative, web-based environment for unifying data and meta-data. Core capabilities are provided through three sets of servies:

1. Metadata Management Services: base mechanisms for generating, federating, and translating annotation and relationship data,
2. Semantic Services: support for advanced searching, relationship browsing, and pattern recognition, and
3. Notebook Services: mechanisms related to records management — collections, digital signatures and timestamps, pagination, annotation display mechanisms, etc [Myer01a].

SAM attempts to:

> "use emerging technologies to separate the initial capture and storage of data and metadata from its subsequent presentation to others and to shift the focus from up-front standardization to on-demand mapping of the original data and metadata into schemas of interest. SAM's key concept is the idea of a "schemaless" data store tha can accept arbitrary input and dynamically registered translators that map data and metadata into the formats and schema expected by applications and underlying data repositories. [Myer03a]

SAM supports the extension of types that can be used. SAM is one of the few systems that features automatic annotation via methods in the database that process incoming data. This feature, however, requires the creation of database-resident code that can identify and process any new data type. [Myer01b]

SAM uses the concept of submitting an entry to publish data. An entry may be edited up until the time it is submitted. After being submitted, the entry may no longer be altered. An entry cannot be viewed by others until it is submitted. ELN uses a notebook object (NOb) as the central data management concept. The data types stored within a notebook include chapters, pages, notes, sub-notes, and other notebook objects. ELN supports both internally stored data and unmanaged, externally stored data via URL reference.

Biological Collaborative Research Environment (BioCoRE)

Website: http://www.ks.uiuc.edu/Research/biocore/

Developed at the University of Illinois at Urbana-Champaign, BioCoRE supports a collaborative work environment for biomedical research and learning. BioCoRE's primary design was to connect three tools, also developed at UIUC, together into a coherent whole. The other three tools are:

1. Visual Molecular Dynamics (VMD),
2. Not (just) Another Molecular Dynamics package (NAMD), and
3. MDTools.

VMD is an OpenGL application used to visualize molecular models and manipulate them in real time. NAMD is the back-end that processes molecular dynamics simulations and feeds VMD data to display. MDTools helps connect VMD on the front-end with NAMD on the back using a collection of programs, scripts, and utilities that make many visualization and simulation tasks easier to accomplish. In addition to linking these components together, BioCoRE adds an overarching structure to organize work using projects. This project structure provides services to users such as project reports, forums/chats, and access to information from other members of the same projects. Lastly, BioCoRE provides functionality for running parallel NAMD jobs and monitoring their progress along with more basic capabilities for non-NAMD jobs.

Core Features

In 2003 BioCoRE surveyed its users. In this survey the ten key features that BioCoRE provided were identified. This list is presented below along with a description of each of the 10 areas.

1) BioFS—BioCoRE's file system used to organize projects and share information between users. Built on MySQL.
2) Message Board—A forum for collaboration between members of a project.

3) Control Panel—A location for chatting (IRC) with other users. Logging allows users to see messages sent before they logged in. [Link: http://biocore.ks.uiuc.edu/biocore/docs/controlpanel.html]
4) Use with VMD—BioCoRE enables users to share views of molecules they have created with VMD. [Link: http://biocore.ks.uiuc.edu/biocore/docs/pubsynch.html]
5) Submit & Monitor NAMD jobs
6) Lab Book Use—Allows users to take notes and share the information they have obtained. It features sharing bookmarks, images, etc. and is conceptually similar to a blog. The Website Library also plays in here.
7) Website Link Library—Stores links to resources referenced in the lab book.
8) NAMD-CFG Use—Tools for easing creation of NAMD configuration files.
9) Java Molecular Viewer (JMV) Use—Enables viewing of molecular structures stored in BioFS.
10) Submit & Monitor non-NAMD jobs.

Strengths

BioCoRE is tightly integrated with the three key applications that run on it: JMV, VMD, and NAMD. Everything is built around servicing those applications. By limiting the scope, they are able to attain a very simple and intuitive interface for using those three applications. Some of the other strengths are the collaborative mechanisms built into it which interact in a coherent fashion with the project-oriented nature of BioCoRE. WebDAV allows users to mount the BioFS file-system on their local machine. BioCoRE has a number of paper prototypes of features to be added to their system. For instance, the inclusion of a useful job management system to control when and where jobs run on the collection of supercomputers and clusters available at UIUC.

Weaknesses

The most notable external weaknesses are the inability to take advantage of grid-computing, discover remote services (web services), and discover remote data sources and as well as the inability to interact with these services as though they were local. Upon examining the internals of BioCoRE, it becomes obvious that it is not a coherent system with clearly defined metadata repositories and management facilities as is the case with myGrid. Though at the surface BioCoRE appears to be one neatly kept collection of tools, below the surface it seems to be connected together by a large collection of shell scripts and small programs which keep things in order (MDTools).

Other weaknesses include:

1) No data provenance or other metadata management.
2) No concept of a workflow that others can view, import, or alter.
3) The collaborative features fail to support easy inclusion of results. For example, the chat, forum, and library features are essentially stand-alone applications appended onto BioCoRE.
4) BioFS is really the only collaborative feature which is of note, and it is weak in its implementation. It only allows users to share views of a molecule (essentially a JPEG). It does not give the users viewing the shared data any access to modifications which may have been made to the stock molecular model, such as special colorings, clippings, iso-surfaces, or other such features.
5) Though BioFS does support users getting data from a common repository, it does so in a rather obtuse way—by using WebDAV.
6) The interface can become extremely cumbersome if a researcher is involved in 20 projects as the interface puts up tabs for all the projects the researcher is on. The result is clutter with no mechanism in place to properly scope the view presented by the interface.

Conclusions

BioCoRE, while an acceptable package for users who are only interested in NAMD and VMD, is very weak in terms of its underlying infrastructure. From looking at BioCoRE, what to do and what not to do with various collaboration tools, data sharing methods, and user interfaces becomes obvious. Collaborative tools need to include chat, audio, and video with a seamless integration to data sources, workflows, and results present in the system. Data sharing methods need to emphasize a convenient way to give "packages" to others that include not just a static image but that include everything that led up to that result so that it can be recreated by others and explored by altering various attributes of the original experiment. Lastly, the user interfaces in BioCoRE, despite the small set of programs supported, feels somewhat disjointed. An environment where programs feel connected in a seamless fashion needs to be implemented.

Finding Out More

BioCoRE Tutorial http://www.ks.uiuc.edu/Research/biocore/tour/tour_intro.html
Job Monitoring Prototype http://www.ks.uiuc.edu/Research/biocore/eval/prototypes/job_monitor_prototype_3.pdf
Job Submission Tool http://www.ks.uiuc.edu/Research/biocore/eval/prototypes/jobsubprot/biocorejobsubprot.htm
Presentation Notebook http://www.ks.uiuc.edu/Research/biocore/eval/prototypes/BioCoRE_Notebook_Prototype.pdf

More information about VMD, NAMD, and MDTools can be found at their respective websites:

- NAMD, http://www.ks.uiuc.edu/Research/namd/
- VMD, http://www.ks.uiuc.edu/Research/vmd/
- MDTools, http://www.ks.uiuc.edu/Development/MDTools/

myGrid/Taverna

Website: http://www.mygrid.org.uk/

myGrid is a middleware system supporting *in silico* biology experiments. These experiments are modeled as workflows and are executed on a grid environment [Stev03]. myGrid has advanced support for resource discovery, workflow execution, and provenance management in the semantically complex bioinformatics environment. Provenance information about workflow enactment describing the services invoked, parameters used for the various workflow components, timing information, output data, and ontology descriptions is automatically logged when executed.

Analysis

Table 1 lists the key feature areas and compares eight systems on those metrics. The three sets of tables each have two header rows. The top header indicates a general category. The second row of the header indicates a specific attribute. The 'Y' and 'N' indicate if a system does or does not have the attribute specified in the column heading.

The first set of columns evaluates the data abstraction capabilities of the various systems. The ability to extend the type hierarchy to support new types of data is key to a developing field. The "Plug-In Architecture" column evaluates the ability to expand the system through user-developed plug-ins. The "Content Storage" category evaluates how the system stores content. Some systems support only unmanaged content, where the content is not secured by a server. With unmanaged content storage, the provenance management system typically has pointers to external storage locations. Because the content is unmanaged, the system cannot prevent loss of information. In managed content storge, however, the PMS stores the data in internal locations where control over the creation, modification and deletion of content can be enforced. The potential drawback of man-

		Data Abstraction		Content Storage	
		Extensible Type Hierarchy	Plug-In Architecture	Unmanaged	Managed
System	NoteCards	Y	Y	N	Y
	ViNE	N	N	Y	N
	SAM	Y	Y	Y	Y
	myGrid	N	N	Y	N
	BioCoRE	N	N	N	Y
	KEPLER	Y	Y	Y	Y
	PASOA	Y	N	Y	N
	BCJ	**Y**	**Y**	**N**	**Y**

		Provenance				
		Attribution	Audit Trail	Data Quality*	Informational	Replication
System	NoteCards	Y	N	Y	Y	N
	ViNE	N	N	N	Y	Y
	SAM	Y	N	Y	Y	Y
	myGrid	N	N	N	Y	Y
	BioCoRE	Y	N	Y	N	N
	KEPLER	Y	Y	Y	Y	Y
	PASOA	Y	Y	Y	Y	Y
	BCJ	**Y**	**Y**	**Y**	**Y**	**Y**

* = Data quality for bioinformatics type data is often implied by attribution

		Security Granularity		Search		
		Fine	Coarse	Content	Metadata	Complex Joins
System	NoteCards	N	Y	Y	Y	N
	SAM	N	Y	N	N	N
	myGrid	N	N	Y	Y	N
	BioCoRE	N	N	N	N	N
	KEPLER	N	Y	N	N	N
	PASOA	N	N	N	Y	N
	BCJ	**Y**	**N**	**Y**	**Y**	**Y**

Table 1: Comparison of Related Work Systems

aged content storage is a large volume of information degrading performance. In the second table, the support for storing the five types of provenance is evaluated.

Finally, the last set of columns evaluates the granularity of the security controls implemented. In systems with no security controls, both the fine and coarse columns will be marked with 'N'. Many systems choose to take the paper notebook paradigm to its extreme and implement access controls on a per-notebook level. This type of coarse-grained access control limits the ability to selectively share information and may result in a researcher revealing information accidentally for any number of reasons. For instance, if a new user is added to a group that was granted access to a notebook, that user would suddenly have access. Also, if a researcher placed new information in a notebook that was shared with others but failed to remember all with whom it was shared, he could potentially reveal information unwittingly simply through casual usage. In a fine-grained security model, atomic objects that are very compact and typically only include one type of information are the objects used by the security model. This allows great discretion in the information shared with others. By enabling this fine level of control, a researcher may feel more comfortable sharing with others because the reseracher is not forced to share an entire journal simply to give another a workflow or data set to use.

Lastly, the search capabilities present in the system evaluate the types of searches that can be accomplished. Powerful but quick search decreases the time necessary to complete a task. Some systems only allow search over the content, while others only allow searches over the metadata. The best systems allow searches over content, metadata, and joins between all content and metadata to form complex queries that make possible a more rich and friendly environment for the user of a system. As can be seen, the BCJ does quite favorably against its competitors. The next section provides an introduction to the BCJ-DB.

Introduction to the BCJ-DB

The research of similar systems' strengths and weaknesses clearly showed a pattern where one or two novel concepts were involved in a particular project but often times the work suffered from not being complete enough for practical application to a variety of areas. The related works review identified five key areas as being important to aiding biologists in completing their work. The key areas are:

1. Type Hierarchy & Data Abstraction
2. Data Storage
3. Provenance Management
4. Data Security
5. Data & Provenance Search

In the following paragraphs these five areas and the features necessary to implement them is discussed. The goal in this section is not to be extensive. Rather, it is to give an overview of the high-level objectives of the project. The following chapters will discus in detail the five key areas. If a concept seems vague in definition at this time, be assured that clarification will be forthcoming.

The BCJ seeks to be flexible and extensible to the problems generally encountered during computational tasks. To reach the BCJ project goals, an appropriate set of abstraction techniques combined with a solid type hierarchy and a data driven architecture that uses the type hierarchy to process the abstraction techniques was deemed appropriate. Research into the methods used by traditional wet-lab biologists heavily influenced the development of a data abstraction model that would support an electronic journal with pages representing atomic units of information. The pages of a traditional paper notebook would be represented as entries in a journal. An entry is an atomic building block upon which nearly all information in the BCJ is stored. An entry has a singular piece of type information associated with it. This makes processing of an entry simpler because there are no nested types within the content of a single entry that must be processed. The type hierarchy itself is in fact stored as a number of entries. An entry can only have one writer, the creator of the entry. This eases the determination of attribution.

Just as the entry is the fundamental unit of information, the journal is the fundamental organizational tool for collections of entries. An entry is a member of one and only one journal. Journals may be used to represent a hierarchy with a parent pointer. In this sense journals are similar to a folder. A key difference from the paper research model pertains to

who may place entries in a journal. Clearly it would be difficult to have multiple writers in a single paper journal. In the BCJ, however, this possibility is very real and important to the completion of collaborative work.

The type hierarchy should be designed to be extensible, flexible, and robust. Extensibility should include the ability to add new types to the system. Both users and administrators should be able to add new types. Flexibility builds upon extensibility by including the notion that the type hierarchy may need to change. When these changes inevitably take place, the changes should occur without negatively impacting the programs that make use of the BCJ. Robustness should be present to ensure that actions that would have fatal consequences to the stability of the system are prevented. The use of journals as namespaces may help to isolate changes and aid robustness. Extensibility and flexibility should not come at the cost of robustness, even though they are highly desired. By having these three attributes, the type hierarchy forms the basis of a solid abstraction model and thus encourages experimentation.

The data storage system needs to support a number of challenging requirements that are somewhat contradictory. For starters, the system needs to support access from clients for inputting data, creating workflows and experiments, annotating data, viewing results, and other daily tasks. These accesses need to have a low-latency and a short time to present the client with information. Opposing this is the need for the computational cluster, which executing an experiment, to have low-latency, high-bandwidth access to read input data, read experiment definitions, and write experiment output. To address these challenges and others related to search (discussed shortly) a hybrid approach using three content storage types was developed. The storage type is used to specify if the data is stored within the database or external to the database and if the data is character-based or binary. The goals of partitioning the storage hierarchy were to achieve better performance with reduced database overhead.

Provenance, as discussed in the introduction, has become more onerous to maintain because the value of careful documentation has been reduced as the number of experimental iterations required to obtain meaningful results has increased proportionally with increases in the volume of data to be analyzed. As Gupta discussed, the revised paradigm for research places data first in the experimental process. This placement drives the rapid increase in the number of iterations used in searching for results of significance. Clearly

the new paradigm places a heavy burden on the researcher. To counter this, systems that document the process as it proceeds are needed. The BCJ provides four of the five provenance uses discussed by Yogesh L. Simmhan et al. These four types are:

1. Audit Trail—complete traces of the execution can be stored,
2. Replication Recipes—the steps, parameters, and other transformations are recorded in experiments,
3. Attribution—authentication and database control over ownership metadata ensures proper pedigree can be established, and
4. Informational—annotations of varying types along with indexing of data.

The fifth type, data quality, is not part of the BCJ system or any of the other systems designed to support scientific computational research. Although initially this shortcoming might seem troubling, it should be noted that typical scientific researchers evaluate the quality of the data they are using not by the simplistic heuristics used in systems that attempt to evaluate data quality. Rather, researchers typically use the data source as the most reliable metric of data quality. Using this methodology makes data attribution equivalent to data quality, and as such any system that addressed attribution thoroughly would also be able to assess data quality.

Data security is rarely addressed in any significant fashion. Of the systems that have a data security model, the model typically only addresses a broad scope through which to assert permissions. The scope is typically for an entire project or journal of information. The entry concept utilized in the BCJ allows a fine-grained scope in which to assert permissions. The actors in the model are users and groups. Users are a member of one or more groups. For write access, only the user participates. The model states that write access is only granted to the owner (creator) of an entry. For read access, both the user and the group or groups a user is a member of participate. Every entry may have zero or more groups associated with it. If the current user is the owner of the entry, access is granted. If the user is not owner but the intersection between the groups attached to the entry and the groups the current user is a member of is non-null, access is granted. Although simple, because of its fine-grained nature this model is quite capable of allowing useful collaboration while preventing leakage of data unnecessarily.

In addition to the standard data security model, two additional concepts make the BCJ-DB much more unique. The first concept, commit, specifies if an entry has been locked from further changes. Before an entry is committed, the owner is free to make

changes to the content. After an entry is committed, only metadata fields not affecting provenance may be altered. This concept of committal is the first step in ensuring provenance is maintained. The second unique concept is an extension of the typical Resource Description Framework (RDF) 3-tuple of <subject, predicate, object>. This 3-tuple is used in nearly all of the frameworks discussed. The weakness of this 3-tuple is that it does not specify the importance of the subject or predicate to maintenance of provenance information. All five types of provenance have the potential to be negatively impacted if a subject or predicate is removed. By extending the 3-tuple by adding two fields describing if the subject has an essential dependency on the object or vice-a-versa, the BCJ-DB can prevent the loss of important provenance. This capability encourages users to use information provided by others in their research, even if not copied, because they can be guaranteed that the information cannot be removed.

Lastly, with the vast increases in the information generated, data search capabilities, typically an after-thought with other systems, are a primary concern in the BCJ-DB. The BCJ-DB addresses three search areas: content search, metadata search, and complex joins between content and metadata. Content search simply means indexing the content stored in the database and allowing searches against it. Metadata search expands the field by allowing searches for provenance related information or other categorizations the entry may be associated with, such as content type or being committed.

Paper Structure

The five key areas of focus for the BCJ-DB system were decomposed into the following system. While the nomenclature used in the diagram specified below may be unclear, it is only intended at this point to provide a high-level overview of the connections between the systems described in the next five chapters. However, a brief description of the diagram is still in order. This diagram shows the interface to the BCJ-DB on the left and the implementation features that provide the implementation of that interface in the middle. The unidirectional arrows from left-to-right show this mapping. The bidirectional arrows between items on the right show that there is an association between the items at the ends. The chapter that describes the implementation of that portion of the interface brackets the appropriate items on the right.

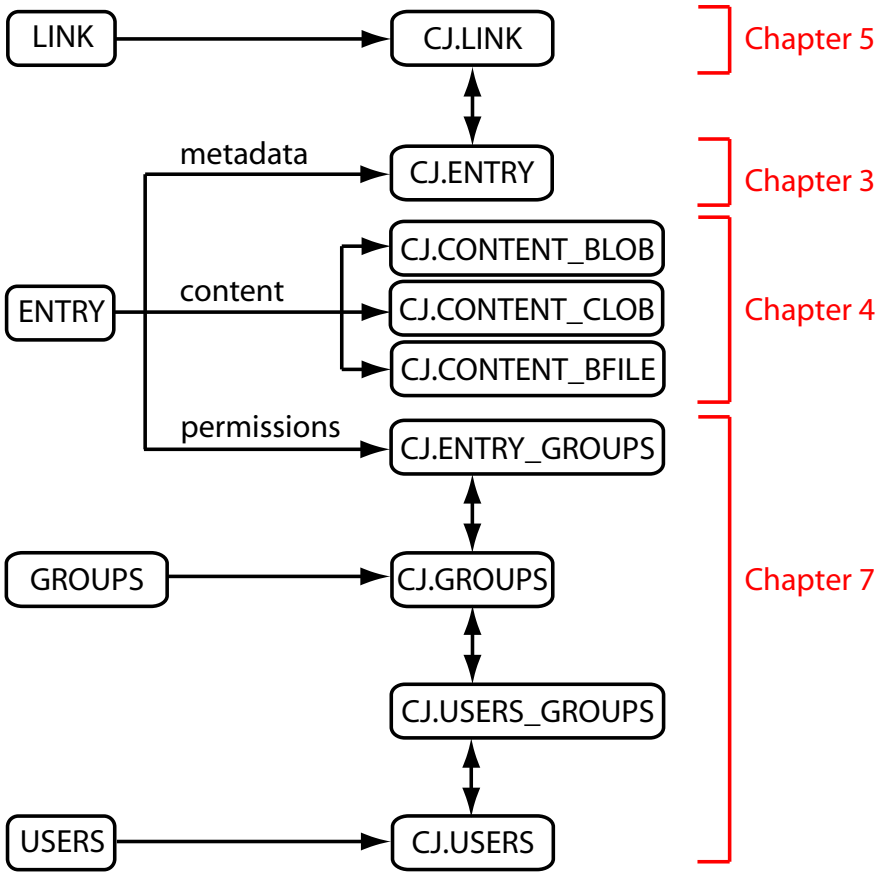


Figure 2: Chapter Overview

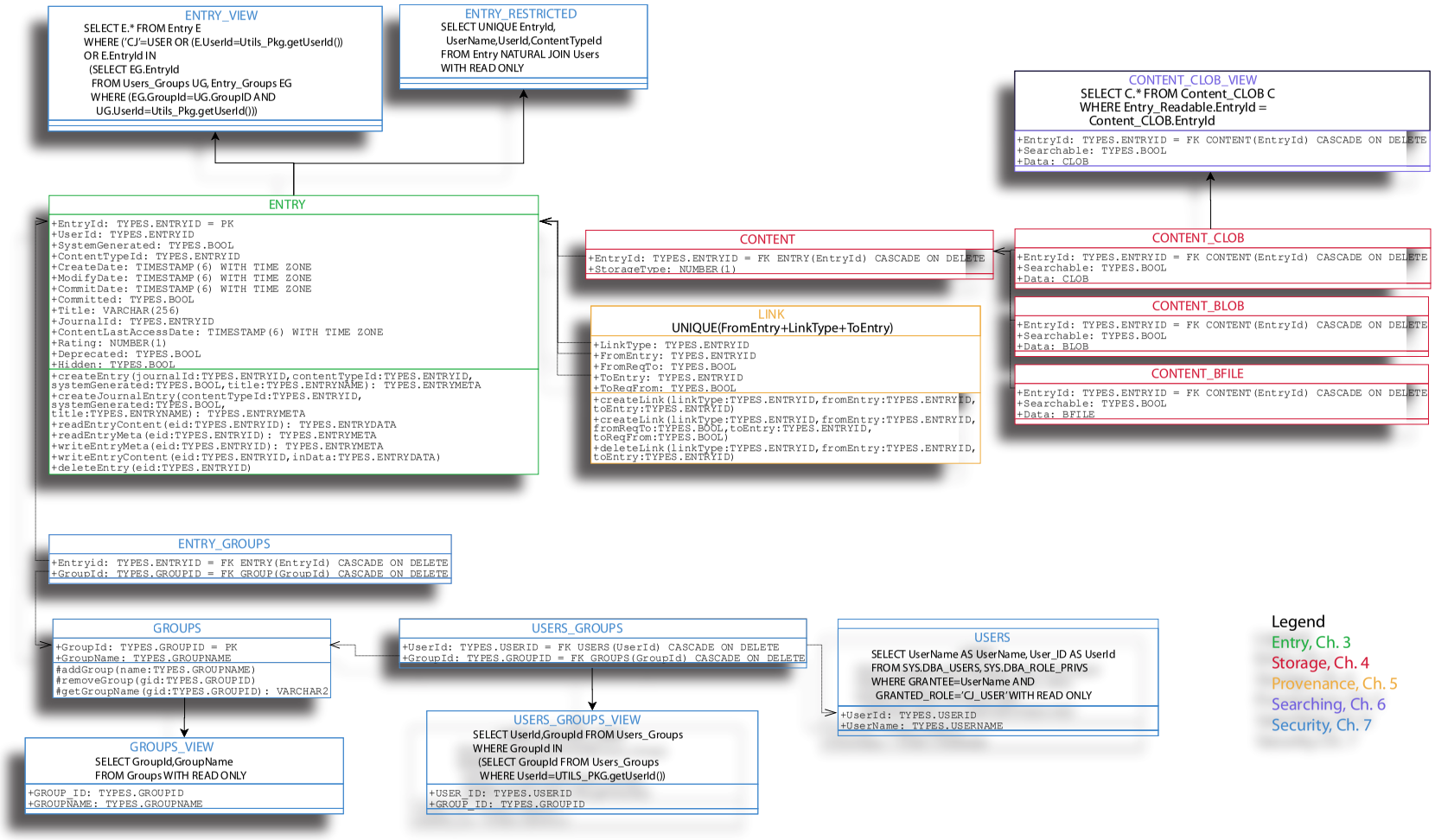
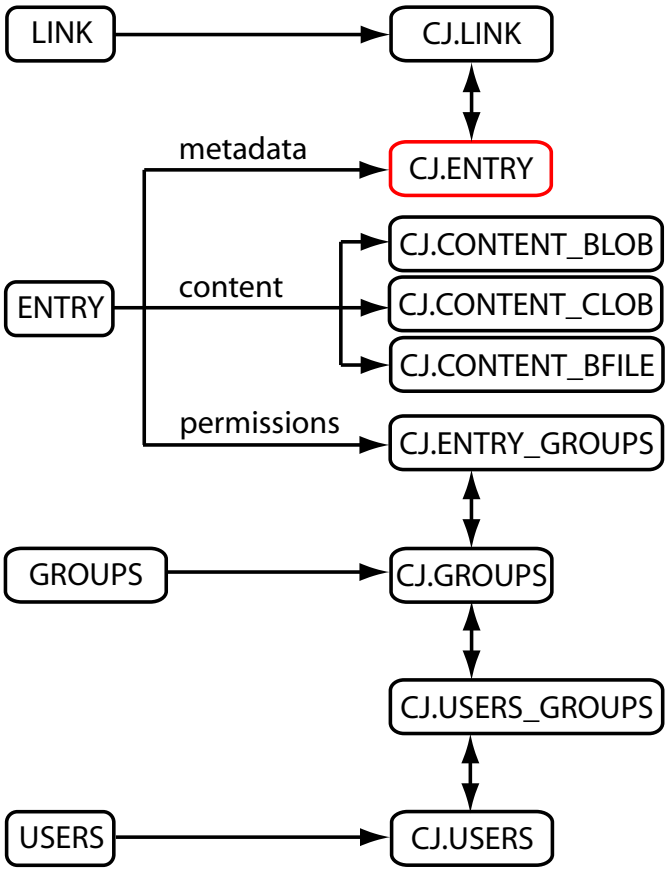


Figure 3: BCJ-DB Oracle diagram indicating relationships and color coding the chapter covering that system



Chapter III: Type Hierarchy & Data Abstraction

Introduction

The core of the BCJ system is its type hierarchy and data abstraction model. This hierarchy and its rules shape many of the defining attributes of the system. The design goals for the type hierarchy were to provide a system that:

1. Provided core type definitions that could be used pervasively,
2. Could be extended by users as well as administrators, and
3. Would allow grouping of types to facilitate a rudimentary form of namespaces.

Given these three goals, a simple system utilizing a core of 3 "master" types was developed.

The data abstraction and storage capabilities present in the BCJ system allow interaction with data in a generic manner while enabling key search and computation with better performance and lower latency than a more traditional approach would have. The fundamental abstraction present in the BCJ system is the concept of an entry as the

atomic unit of information. A single entry should, ideally, contain information necessary to characterize what is stored without the client being concerned with how it is stored. This is accomplished through use of various content types and content storage types. Additionally, an entry should contain the important metadata necessary for attribution. The content storage types will be discussed in the data storage chapter, and attribution will be discussed in the chapter on provenance.

Entry Basics

Note: For simplification other fields present in the metadata of an entry that are not relevant to the present discussion have been omitted from discussion at this time but will be discussed as they become pertinent.

An entry is composed of many fields. Entries are represented in the database as a single row in the ENTRY table. The five most important fields in the definition of an entry for the type hierarchy are:

1. *EntryId* (INTEGER): This is the primary key for the *Entry* table. All entries created after initialization of the database shall have a positive value. Only core database entries may have a negative value. As the primary design philosophy is one of recursion, the value of 0 is reserved to prevent infinite recursion in the definition of the master types.
2. *UserId* (INTEGER): This field references the creator of the entry. Used in combination with other tables this defines the security permissions on an entry and aids a user in locating resources.
3. *ContentTypeId* (INTEGER): References by *EntryId* the type of information stored in this entry.
4. *Title* (VARCHAR2): Used to assign a title to an entry. *Title*s need not be unique.
5. *JournalId* (INTEGER): References by *EntryId* another entry, which must be of content type journal, in which this entry is located. All journals are attached to a singular root journal.

Entry Rules

1. *EntryId* is the primary key and must be unique. For all but the core entries, *EntryId* will be a positive-valued INTEGER. Only for the core entries that are manually inserted is *EntryId* allowed to be a negative value. This partitioning of the domain of the integers is used to divide the core, system world from the user world.
2. The *UserId* field must reference a valid row in the *Users* view.

3. The *ContentTypeId* must reference an entry that is either the master content type or a content type definition that directly references the master content type.
4. The *JournalId* must reference an entry that is in fact a journal.

The Master Types

The three *MASTER_* types in the BCJ are the foundation for all other types in the system. A clear understanding of the philosophy behind these types will help in understanding many of the other design attributes of the system. The first of the master types necessary to build the type hierarchy is the *MASTER_CONTENT_TYPE*. This is the basis of all types in the system. The *EntryId* for this type is a negative number meaning that it can only be created via manual insertion as an administrator. The *UserId*, consequently, is that of the user 'CJ', the owner of the schema. The *ContentTypeId* is a recursive reference to the master content type itself. This self referential nature is yet another reason why this entry must be manually inserted into the table without any rules being applied. In a tabular format the information about the core types can be represented as:

EntryId	UserId	ContentTypeId	**Title**	JournalId
-100	CJ	-100	"MASTER_CONTENT_TYPE"	0
-101	CJ	-100	"MASTER_JOURNAL_TYPE"	0
-102	CJ	-100	"MASTER_ENTRY_RELATION_TYPE"	0
-103	CJ	-101	"ROOT_JOURNAL"	0

Table 2: Core Types Table

In the diagram below, showing the linkages between the core types, the left side of each box has three green ports (dots), for the start of a pointer to another entry. The right side of each box has three red ports (dots), for the end of a pointer to another entry. The box has four rows of information. The first row is the title; the second row, EID, is the *EntryId*; the third row, CTID, is the *ContentTypeId*; the fourth row, JID, is the *JournalId* and references the entry's parent. The ports are on the same horizontal row as the ID they are associated with.

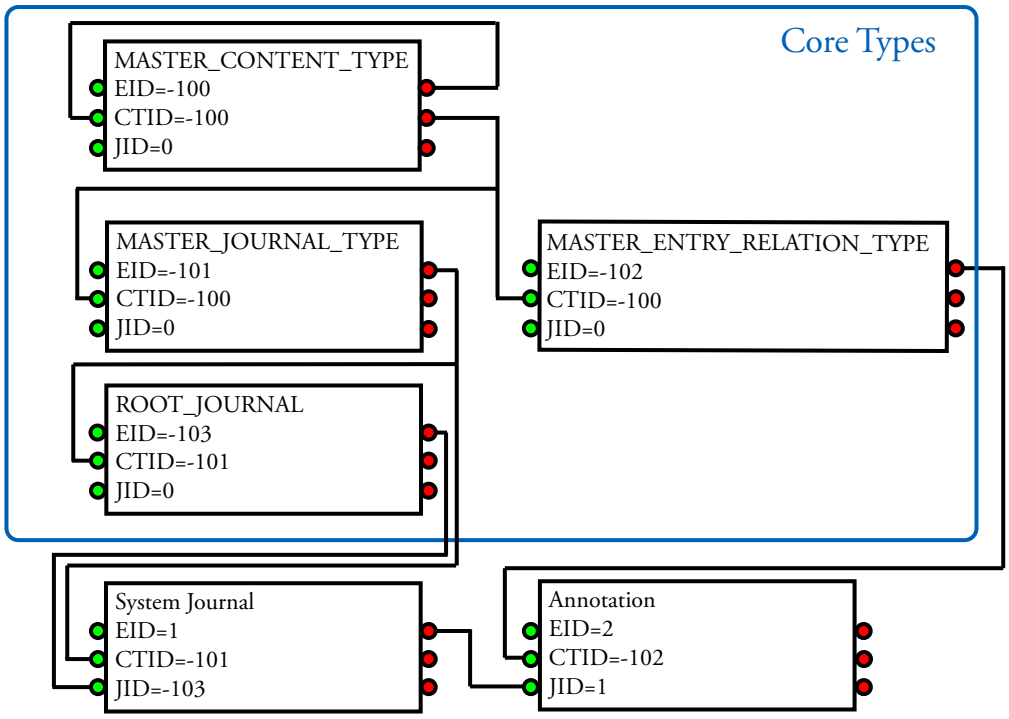


Figure 4: Core Types Diagram

The two additional entries shown in the figure "Core Types Diagram", with the *EntryId* of 1 and 2, are an example of user-generated entries in the BCJ-DB. The entry titled "System Journal" shows how the *ContentTypeId* (CTID) is linked to the entry "MASTER_JOURNAL_TYPE.". Second, this same entry has a parent journal of the entry "ROOT_JOURNAL." The other entry, "Annotation," is an example of a new relationship type being defined. This entry has a CTID of -102, the entry "MASTER_ENTRY_RELATION_TYPE." Lastly, we see how this new entry has been placed in the new "System Journal."

Interacting with Entries

The four primary actions to consider when interacting with entries are:

1. Creating a new entry
2. Reading from an entry
3. Writing to an entry, and
4. Deleting an entry.

All four methods will be discussed, but throughout this chapter and the next four chapters, the creation of an entry including source code and a flow chart of the steps involved

predominates. This is done because entry creation is always the first step in interacting with entries and, therefore, serves well to characterize nearly all aspects involved with entry management. The proceeding figure contains a flow chart showing the complete version of the *createEntry* function. At present the flow chart might not make sense. However, after reading the next few chapters, the *createEntry* function will become clear as further details are provided. Reference to this diagram, when appropriate, is recommended.

Lastly, some mention of code conventions is in order. In Oracle PL/SQL code (often referred to simply as Oracle code), tags preceding the semantic name associated with a variable are used to indicate the type of variable.

Tag	Meaning
g_	Global Variable
p_	Parameter to a Function/Procedure
v_	Local Variable in a Function/Procedure

Table 3: Program Tag Definitions

The entry rules specify that an entry must have a unique id to be used as the primary key. To ensure the primary key is unique, a sequence that produces unique integers in a sequential manner is utilized. This sequence has the name *EntryId_Seq*. In Oracle it is specified as:

```
CREATE SEQUENCE ENTRYID_SEQ INCREMENT BY 1 START WITH 1
NOCYCLE;
```

An entry must have a content type specified. A *ContentTypeId* must reference an *EntryId* that both exists and has a *ContentTypeId* that references the master content type. The *JournalId*, as with the content type, must reference an *EntryId* that exists and the entry referenced must have a *ContentTypeId* that references the master journal type. Lastly, so that the caller can find the new entry, the function should return the *EntryId* that was retrieved from the sequence.

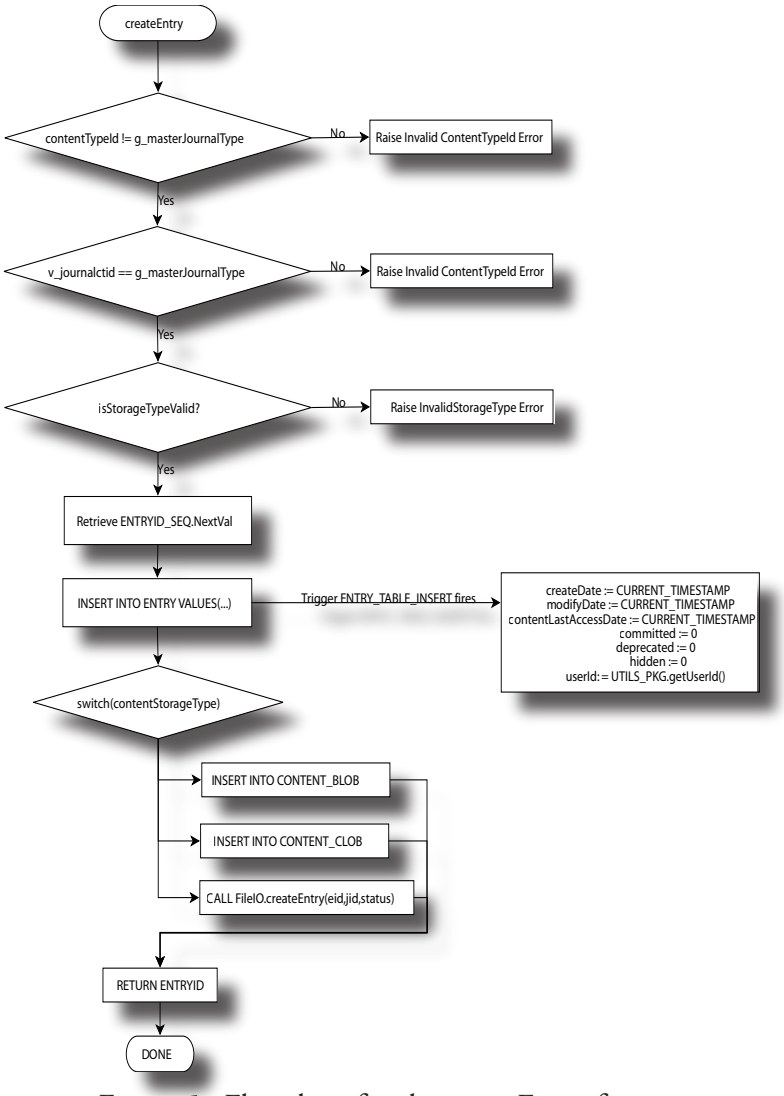


Figure 5: Flowchart for the createEntry function

Given the above fields, a reasonable prototype for an entry creation method would be:

```
createEntry (
   journalid,
   contentTypeId)
```

While not required by the entry model, this prototype could be enhanced through two additional parameters, the title and the system generated flag. The *Title*, as discussed earlier, is a string. The *SystemGenerated* flag, on the other hand, has not been discussed previously. This flag is implemented as a BOOL that can signal true and false. An entry requires this field to be set at creation time. This metadata field specifies if the entry was created by the system or by a user. Examples of system generated entries include built-in content types (including the master types) and error files from experiments. While

this field can be helpful, it is not essential or even necessary for proper system operation. Rather, it is a convenience to limit the scope during searches. The function prototype is now:

```
createEntry (
  journalId,
  contentTypeId,
  systemGenerated,
  title)
```

In Oracle, with the proper type references, this becomes:

```
FUNCTION createEntry (
  p_journalId        IN TYPES.ENTRYID,
  p_contentTypeId    IN TYPES.ENTRYID,
  p_systemGenerated  IN BOOL.BOOL,
  p_title            IN TYPES.ENTRYNAME)
  RETURN             TYPES.ENTRYID;
```

The keyword FUNCTION specifies that this is a function definition to Oracle. The next string is the title of the function. Within the parentheses is the parameter listing. The parameters specify (in order):

1. the parameter name,
2. if the parameter is an input, output, or both, and
3. the type the parameter is.

Lastly, the return value type is specified. For the createEntry method all parameters are INputs. In the BCJ-DB, the type specification generally references a package and a type definition within the package. The notation for this specification is "PACKAGE.TYPEDEF". This function uses the TYPES package and the BOOL package. From TYPES, the ENTRYID and ENTRYNAME type definitions are used. From BOOL, only the definition of BOOLean is used. If an invalid reference for *p_journalId* or *p_contentTypeId* is provided, an exception is thrown and an entry is not created. In the next chapter the final parameter present in the createEntry method will be discussed.

With the method for creating an entry established, the next goal is to read from and write to an entry. This is done by interacting with the table that stores the metadata for an entry. The name of this table, not surprisingly, is ENTRY. The content for an entry is stored in separate tables. The easiest way to read an entry is to select all fields from ENTRY where the ENTRYID matches the desired entry and read from the result. To write to an entry, simply perform the appropriate SQL update on the fields to be changed. In SQL, reading an entry could be described as:

```
SELECT * FROM ENTRY WHERE ENTRYID=123456;
```

As an example of a write operation, consider updating the title of an entry:

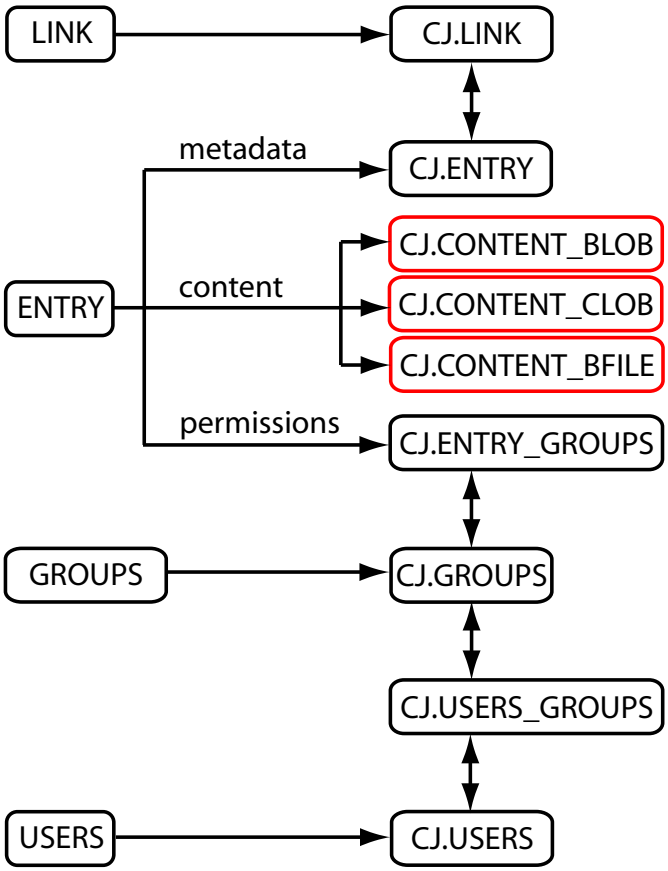
```
UPDATE ENTRY SET TITLE='New Title' WHERE ENTRYID=123456;
```

Clearly if this were the full extent of the work done by the BCJ-DB the system would hardly be worth discussing. Not all fields of an entry may be modified for security reasons. In subsequent chapters interesting and unique features governing operations, including interacting with content, will be discussed.

When erasing an entry, care needs to be taken because the content is detached from the metadata. For this reason, only a function interface is provided. An entry can be erased by referencing the primary key of an entry, *EntryId*, and calling the *eraseEntry* method. This method returns a Boolean indicating if the deletion were successful. The signature for *eraseEntry* is:

```
PROCEDURE eraseEntry (
  p_eid          IN TYPES.ENTRYID);
```

If the entry does not exist, an exception indicating that fact is thrown.



Chapter IV: Storage Architecture

Introduction

The storage demands on the BCJ-DB to support both rapid search and local cluster-based computation were the primary drivers to develop the data abstraction model. This model splits an entry into two components: metadata and content-data. The metadata of an entry is stored in the ENTRY table. This facilitates rapid access via indexing and quick searching. The content of an entry, however, has a hybrid storage architecture utilizing both tables in the database and files on a file system accessible by the database and the cluster.

Content Storage Type

The two driving factors for the design were latency and volume. A third attribute, the need for search, was added to reduce unnecessary load on the database from indexing. To achieve these goals, the storage architecture stores entry content in three locations. Two of the storage types are stored inside the database while the third is stored on a shared

file-system. Character Large Object (CLOB) and Binary Large Object (BLOB) type content are stored internally. Binary File (BFILE) data is stored externally.

	Latency	Volume	Searchable
CLOB	Low	Low	Yes
BLOB	Low	Low	No
BFILE	High	High	No

Table 4: Comparison of content storage types

From the table, it becomes clear that CLOB is useful for annotations and other metadata schema used for informational provenance because it is indexed. BLOB's are used to store binary objects that may need frequent reference and are small in size. Examples include all content types, workflow and experiment definitions, and journal definitions. BFILEs are used to store files, such as the human genome, that are large, require rapid cluster access, and will rarely, if ever, be viewed or modified extensively in the BCJ client.

Entry Interaction

The three storage types available in the BCJ enable a more flexible storage system that provides higher responsiveness to the tasks involved in computational biology through specialization. Fortunately the impact of this specialization is minimal on the interaction with entries.

Entry Creation

With the addition of this field, the interface to create new entries changes only slightly. Namely, a content storage type must be specified. The Oracle PL/SQL is:

```
FUNCTION createEntry (
  p_journalId        IN TYPES.ENTRYID,
  p_contentTypeId    IN TYPES.ENTRYID,
  p_systemGenerated  IN BOOL.BOOL,
  p_title            IN TYPES.ENTRYNAME,
  p_storageType      IN TYPES.STORAGETYPE)
  RETURN             TYPES.ENTRYID;
```

This is the complete specification for creation of an entry in the BCJ-DB. All parameters are still inputs. The return value is the *EntryId* of the new entry. And, as before, an exception is thrown if an invalid p_journalId or p_contentTypeId is passed in. Lastly, if a *StorageType* that is unrecognized is provided an exception indicating that fact is thrown.

When the createEntry method is called, it first checks all parameters to determine if any rules are broken. If any are, an appropriate exception is thrown. If no rules are broken,

the *EntryId* to be assigned to the new entry is retrieved from the sequence ENTRYID_SEQ and the new entry is inserted into the entry table. Lastly, based on the content storage type specification, one of three things is done. If the storage type is BLOB, written TYPES.ST_BLOB, the ENTRYID and an empty BLOB are inserted into the CONTENT_BLOB table. Similarly for CLOB (TYPES.ST_CLOB), the ENTRYID and an empty CLOB are inserted into the CONTENT_CLOB table. Lastly, for a BFILE (TYPES.ST_BFILE), something similar in significance but different in implementation, happens—a Java Stored Procedure (JSP) created in Oracle is called. This JSP will create a file on the shared file system. Before proceeding with the JSP, the code to this point is:

```
FUNCTION createEntry (
  p_journalId        IN TYPES.ENTRYID,
  p_contentTypeId    IN TYPES.ENTRYID,
  p_systemGenerated  IN BOOL.BOOL,
  p_title            IN TYPES.ENTRYNAME,
  p_storageType      IN TYPES.STORAGETYPE
)
  RETURN             TYPES.ENTRYID
AS
-- Local variables
  v_cval          TYPES.ENTRYID;
  v_journalctid   TYPES.ENTRYID;
  v_status        INTEGER;
BEGIN
  -- Is the ContentTypeId != g_masterJournalType
  -- If it is, then they are really creating a journal and should call createJournalEntry
instead.
  IF p_contentTypeId = g_masterJournalType
  THEN
    errpkg.raise(errnums.en_invalid_contenttypeid);
  END IF;

  -- Is the Entry represented by journalId really a journal?
  SELECT ContentTypeId INTO v_journalctid FROM ENTRY
   WHERE ENTRYID=p_journalId;
  IF v_journalctid != g_masterJournalType
  THEN
    errpkg.raise(errnums.en_invalid_contenttypeid);
  END IF;

  -- Is the StorageType valid?
  IF(p_storageType != TYPES.ST_BLOB) AND
    (p_storageType != TYPES.ST_CLOB) AND
    (p_storageType != TYPES.ST_BFILE)
```

```
THEN
  errpkg.raise(errnums.en_invalid_storagetype);
END IF;

/* All checks have been meet, so we can proceed */
-- Get the next value in the sequence for the primary key
EXECUTE IMMEDIATE
'SELECT ENTRYID_SEQ.NextVal FROM dual' INTO v_cval;

-- Insert the values passed in into the ENTRY table. This fires a trigger as well.
INSERT INTO ENTRY(
  EntryId, JournalId, SystemGenerated, ContentTypeId,
  StorageType, Title)
 VALUES(
  v_cval, p_journalId, p_systemGenerated,
  p_contentTypeId, p_storageType, p_title);

-- Initialize the corresponding row in the CONTENT_ table
IF (p_storageType = TYPES.ST_BLOB) THEN
  INSERT INTO CONTENT_BLOB(EntryId,Data)
   VALUES(v_cval, EMPTY_BLOB());
ELSIF (p_storageType = TYPES.ST_CLOB) THEN
  INSERT INTO CONTENT_CLOB(EntryId,Data)
   VALUES(v_cval, EMPTY_CLOB());
ELSIF (p_storageType = TYPES.ST_BFILE) THEN
  FileIO.createEntry(v_cval,p_journalId,v_status);
ELSE -- Invalid Storage Type
  errpkg.raise(errnums.en_invalid_storagetype);
END IF;

COMMIT;
```

Figure 6: createEntry source code

Read & Write LOB Content

Reading and writing of the two Large Object (LOB) types is done through a functional interface. When reading or writing an entry, a pointer to a temporary LOB that is a copy of the actual data is used for interaction. This is done for security reasons to prevent modification of the actual content present in an entry. Thus, when a user requests to read the content of an entry that has a content type of BLOB or CLOB, permissions for access are checked by the security framework (discussed in chapter seven). If access is granted, the BCJ-DB copies the content from the appropriate table (CONTENT_BLOB or CONTENT_CLOB) to a temporary LOB. The return value for the read function is a pointer to the temporary LOB. When writing, a similar procedure takes place. The user writes into a temporary LOB and then calls the appropriate write function. The

BCJ-DB will then evaluate the security rules for writing to an entry and if the user is allowed to perform the write, the content will be copied into the appropriate table. In the case of both read and write, if the user is not allowed to perform the action, an exception is thrown. If the *EntryId* specified does not exist, an exception indicating that fact is thrown. Because *EntryId* is the primary key, it is used to reference the source/destination. The Oracle PL/SQL for reading an entry is quite clear:

```
FUNCTION readEntryContent (
  p_eid             IN TYPES.ENTRYID)
  RETURN              BLOB;
```

Read & Write BFILE Content

BFILE storage has a central base directory where the directories for all of journals that exist on the system are located. Entries are stored as files placed inside the directory with the same name (actually a number) as the *JournalId* value. Security requirements dictate that the file referenced by the BFILE be placed in a directory only accessible by the BCJ computational server and the BCJ-DB. This prevents modification or deletion of entry content of storage type BFILE.

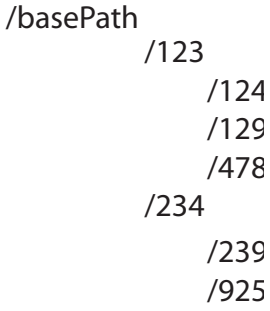


Figure 7: BFILE Base Path

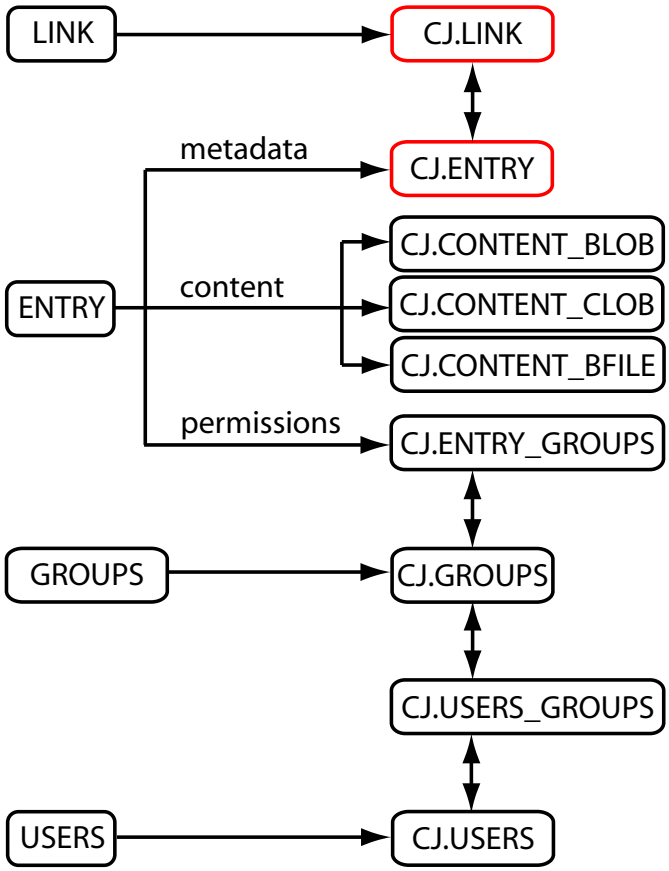
The figure "BFILE Base Path", represents a situation where there are two journals, with the *JournalId* values of 123 and 234. There are a total of five entries. Three of the entries, 124, 129, and 478 are in journal 123. The other two, 239 and 925, are in journal 234. This type of mapping between the database and a UNIX file system is done to better load-balance the number of files in a directory. Although this design has limitations with respect to shortcomings present in the underlying filesystem, it should be robust under nearly all circumstances. A more robust method using hashing could easily be developed by modifying the source code in the JSPs that handle BFILE access.

Reading a BFILE is done by calling a method with a similar prototype to that for the two LOB types. However, to permit easier access to the large data typical of BFILEs, the addition of two parameters, one that specifies the offset into the content to begin reading from and another that specifies the number of bytes to be read, have been added. This allows selectively reading a portion of the file. For instance, a user may wish to preview the first 10,000 bytes of a file to see the header information on the output from an experiment. This can be easily accomplished via this method. The prototype is as follows:

```
FUNCTION readEntryContentBfile (
   p_eid          IN TYPES.ENTRYID,
   p_offset        IN INTEGER,
   p_length         IN INTEGER)
   RETURN           BLOB;
```

This method, as with the LOBs, copies the content requested to a temporary LOB and returns a pointer to that temporary. The only difference being that this time only a portion of the content may be contained in the temporary.

To write to a BFILE, the user writes to a temporary BLOB. This temporary BLOB is then passed as a parameter into the write function for a BFILE. The primary difference between the read and write methods is that partial writing is not allowed for BFILE. The temporary BLOB must contain all data that is to be placed into the BFILE. This decision was made because of the inherent difficulty in handling large files. The BCJ-DB design did not invision a scenario where this type of access would be important. When large data sets are used in bioinformatics, it is typically as the input to an experiment, as with the human genome, or the output of an experiment that will later be processed by further algorithms or visualization techniques, as with molecular dynamics. In either case, user modification of large content outside of the computational environment was not a use-case that was felt to be important.



Chapter V: Provenance

Introduction

To maintain provenance information, the BCJ-DB employs a number of techniques. The two key areas involved in provenance are:

1. The metadata present in an entry stores attribution data and
2. Relationships (links) between entries.

The first part of this chapter will discuss the metadata fields of an entry used in support attribution provenance. The latter part will discuss the use of links to maintain informational provenance. Using RDF triples for storing semantic relationships is hardly unique. However, the addition of two fields in support of provenance integrity proves to be unique. Chapter six discusses additional information provenance fields for use in searches. Finally, the focus on integrity continues in chapter seven with a discussion of the security model.

Attribution Metadata

To this point, four entry fields have been discussed in detail. In chapter three, the *EntryId*, *ContentTypeId*, *JournalId*, and *Title* were the focus. Chapter 4 discussed *StorageType*. This chapter presents an additional five fields, thereby doubling the number of available fields. The five new fields are:

1. *UserId*,
2. *CreateDate*,
3. *ModifyDate*,
4. *CommitDate*, and
5. *Committed*.

The first field, *UserId*, identifies the user that created this entry. An entry is assigned this value at creation and it is unalterable after creation. The *UserId* is used both for attribution and for security, regulating who may modify an entry. Synonyms include "entry owner" and "entry creator".

CreateDate is a high-precision timestamp indicating the instant the entry was created. The purpose of this field is to signal when work began on an entry. The timestamp used is retrieved during the invocation of the createEntry method. After createEntry inserts the new row into the database, the *CreateDate* field is unalterable.

ModifyDate is also a high-precision timestamp and specifies the last time the content for this entry was written to. This field can be used to track when the last work was done on an entry. This field has a role in both attribution and informational provenance that may be used in searches. During the invocation of a method to write to the content, the timestamp is retrieved and the *ModifyDate* field of the entry is updated.

The last two fields, *CommitDate* and *Committed*, are closely related and form the final piece of an entry used to maintain the attribution integrity. The concept of committing an entry was developed so that the content could be locked, preventing further alteration. Committing changes ensures accurate records indicating the date a particular piece of work was completed. Entries are created in an uncommitted state. For example, before being committed the content may be modified at will without restriction by the entry's owner. When the owner is pleased with the content of the entry, it may be committed, signaling the completion of work. A trigger on the ENTRY table detects alteration of

the value of the Boolean *Committed*. If *Committed* changes from false to true, the current timestamp is retrieved and the *CommitDate* field is set with the retrieved value.

After committed, only modification of entry metadata fields related to organizational convenience is allowed. These organizational fields will be discussed in chapter six. Thus, at present, the example of the *rating* field, used to indicate the importance or usefulness of an entry, will suffice. The *title* field may not be altered after an entry has been committed.

Entry Relationships

Entry relationships, also known as links, form the basis of many provenance systems. Provenance management systems typically use RDF 3-tuples for the storing the relationships between items. While good at describing relationships, the RDF 3-tuple does not address concerns about maintaining the integrity of the provenance semantics they imply. To address this limitation, the BCJ-DB adds two additional fields to the RDF 3-tuple model to indicate if the subject or predicate has an essential dependency on the other.

In the BCJ-DB, the subject-predicate-object nomenclature transforms into *ToEntry-LinkType-FromEntry*, respectively. The *ToEntry* is a reference to the *EntryId* of the subject of the relationship. Likewise, *FromEntry* references the *EntryId* of the object of the relationship. And, lastly, the *LinkType* is a reference to an entry that defines a relationship type. As discussed in chapter 3, an entry that defines a relationship type has a *ContentTypeId* that references the master entry relationship entry. All relationships use an "IS-A" type of wording. This decision, though not purely arbitrary, was not seen as necessarily superior to a "HAS-A" wording. However, it was felt that having both could lead to confusion and might necessitate the use of inverse relationships in searches to handle situations where the opposite wording was used. For example, the inverse of "John has a son Sam" is "Sam is the son of John." Both sentences specify the same type of familial connection but swap the location of the subject and object. All relationship types should be able to conform to the structure:

FromEntry **is a** *LinkType* **of** *ToEntry*

Using the example above, there could be a *LinkType* of "son" defined. The *FromEntry* would be "Sam" and the *ToEntry* would be "John." This would be written as: "Sam **is a** son **of** John." This specification should ease searches for additional relationships that may

exist in the system. For instance, it might be desirable to find all records that are "son of John" or find all sons in general simply by looking for the specific relationship type.

The primary key for the *Link* table is the 3-tuple of *ToEntry-LinkType-FromEntry*. Few rules regulate the table. As should be evident, the value in *ToEntry* should not be present in *FromEntry* or *LinkType*. Similarly, the value in *FromEntry* should not be present in *ToEntry* or *LinkType.*

The two new flags to indicate the dependencies between the *ToEntry* and the *FromEntry* (the subject and the object) are given the names *ToRequiresFrom* and *FromRequiresTo*. Establishing a dependency between entries signifies if an entry requires the other to form an unbroken provenance trail. An easy way to understand this concept is with a concrete example. A researcher defines a workflow to test a hypothesis. The workflow "WF" takes as input an entry "In" and outputs entry "Out". Clearly if "In" were deleted both "WF" and "Out" would no longer have the appropriate information to provide replication capability or contextual understanding. Similarly, if "WF" were deleted "Out" would no longer have a workflow to define how it was generated. The consequence in both cases is data without appropriate metadata to describe where it came from and how it was generated.

The table below describes the valid combinations of flag values.

ToRequiresFrom	*FromRequiresTo*	Valid?
False	False	Yes
False	True	Yes
True	False	Yes
True	True	No

Table 5: Link table dependency flag values

As described in the table above, it is not possible for both the *ToRequiresFrom* and the *FromRequiresTo* flags to be simultaneously set to true. The rationale behind this decision is that this would cause an inextricable link between the two entries with no possibility to break the linkage. It should be noted, however, that a circular dependency chain between entries could be created. The primary difference between the two, however, is that in the case of the dependency chain the loop can be broken if the dependency requirement of any one of the entries can be altered. In the case of both values being set to true, however, there is no possibility of ever breaking the dependency loop.

Entry Relationship Interaction

To keep the use of the entry relationship simple, only two methods were added to the interface: *createLink* and *deleteLink*. To create a link, five parameters are required:

1. *FromEntry*—the entry the link is from,
2. *LinkType*—the type of link (referenced as an *EntryId*),
3. *ToEntry*—the entry the link points to,
4. *FromReqTo*—if the *FromEntry* has an essential dependency on the *ToEntry*, and
5. *ToReqFrom*—if the *ToEntry* has an essential dependency on the *FromEntry*.

The method *createLink* does the obvious and simply inserts into the *Link* table the appropriate values. The prototype for this method is:

```
PROCEDURE createLink (
  p_linkType        IN TYPES.ENTRYID,
  p_fromEntry       IN TYPES.ENTRYID,
  p_fromReqTo       IN BOOL.BOOL,
  p_toEntry         IN TYPES.ENTRYID,
  p_toReqFrom       IN BOOL.BOOL
)
```

The method *deleteLink*, similarly, simply removes a link from the table. This is an unfortunate weakness of the new model. The prototype for the method is:

```
PROCEDURE deleteLink (
  p_linkType        IN TYPES.ENTRYID,
  p_fromEntry       IN TYPES.ENTRYID,
  p_toEntry         IN TYPES.ENTRYID,
)
```

Link deletion only requires a unique reference to the primary key of a link and is not dependent on the two new flags added.

An accurate method for establishing who could delete a link when there were different owners for the *ToEntry* and the *FromEntry* was never fully established and so this functionality is not implemented at present. The major challenge with coming to a decision on how this should be handled focused on the compromise between the desire for data integrity and the wish to remove the potential for abusive and/or overzealous creation of links between entries owned by different users causing data that may be irrelevant being locked into the system. A compromise solution, disallowing the removal of a link where the user attempting the deletion is not the owner of both the *ToEntry* and the *FromEntry* in conjunction with the usage of the *Deprecated* field of an entry seems to be appropriate. To explain further, the problem with the link dependency exists for the user who

has had their information linked to. For example, if researcher A ran an experiment that produced an output useful to researcher B, who subsequently linked to the output entry and used it in an experiment of their own, that entry would then have an essential dependency. Eventually, a more encompassing approach wherein the database could be run in two modes seems workable. In one more the data is strictly managed (committed entries could never be removed). In the other more pragmatic relaxed security mode, committed entries could be removed, but some guarantees on provenance maintenance would be lost. This dual security mode is discussed in the next chapter. To finish up this discussion, the use of *Deprecated* should allow tagging information that has become irrelevant to the creator of the information even if they are unable to delete it as a result of a strict security mode.

Provenance Collection & Usage

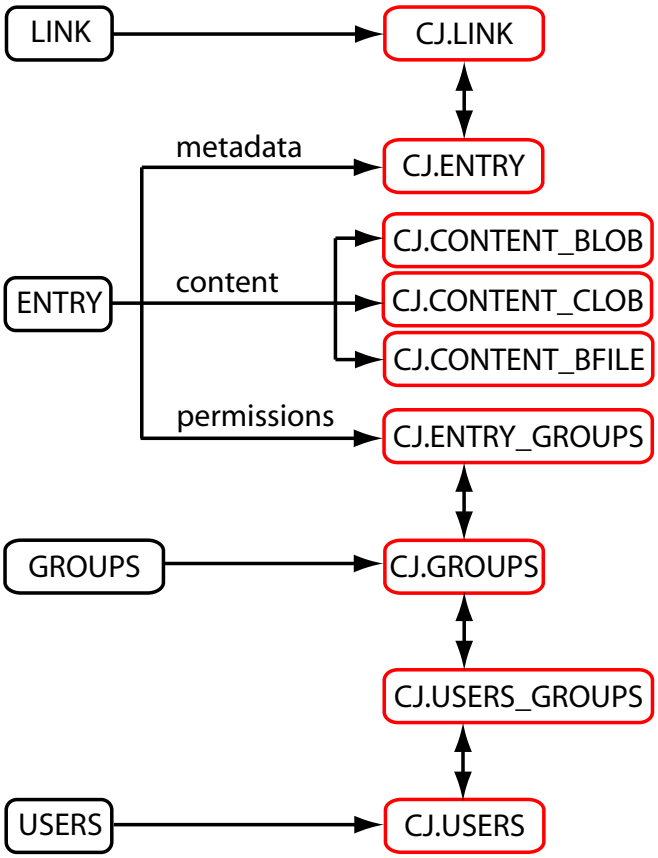
Provenance information is gathered in two ways: through automatic provenance gathering and through an API. These two collection methods are complementary and do not overlap. The automatic collection of provenance is done through typical interaction with the BCJ-DB, such as creating an entry, updating a field of an entry, or updating the content of an entry. The provenance information input through an API typically comes from BCJ plug-ins. For instance, when a user executes an experiment and output is generated, the experiment plug-in will create links indicating the essential dependency of the experiment outputs on the experiment definition. Likewise, the quick text annotation tool available in the BCJ navigator makes an annotation relationship between the text entered and the entry that has focus in the navigator.

Once collected, provenance information may be retrieved by navigators and tools to provide pertinent information to the user while reducing unnecessary clutter. The current BCJ navigator allows filtering by author, all date types, flags such as deprecated, rating, and hidden, and categorization by content types. Additionally, an auxiliary pane in the navigator uses entry relationships to display a type-categorized tree view of the entries related to the current selection. All of the following examples of BCJ usage of provenance information derived the provenance via SQL queries against the E*ntry* and *Link* tables. By providing an open interface using views instead of a purely functional interface, the BCJ-DB makes it possible to perform complex join queries relating entries to one another.

Remarks

Typical research without a provenance framework utilizing flat files stores provenance information in headers or throughout the file. Although this approach provides some form of provenance, mainly attribution of a data source, it is weak even in providing that limited provenance because it requires careful management of files to ensure they are associated with the correct experiments. In the world of bioinformatics with rapid information expansion, this solution rapidly becomes unmanageable by an individual. The time required for management will begin to out-weigh the time spent performing productive research. Thus, an integrated system such as the BCJ that tracks provenance as a user approaches their daily work can be an essential aid in combating the data management overhead.

The BCJ-DB's rigorous authorship and entry create/commit timestamps make it capable of providing non-repudiation assurances. The enhancements to the basic RDF 3-tuple model foster collaboration between researchers by ensuring that the use of data provided by a fellow researcher is without fear of its becoming inaccessible at a later date.



Chapter VI: Search

Introduction

With the volumes of information the BCJ clients automatically produce in the form of relations between entries, the BCJ-DB must provide adequate search capabilities. In fact, the BCJ-DB views search not just as a feature. Rather, it views search as a way of life for presenting useful information to BCJ users in every action they perform. The BCJ navigator concept is the most obvious representation of this. Navigators perform many searches to present a view of the data present in the BCJ-DB. These views are typically user-specified "slices" through the provenance information stored in the BCJ-DB. For example, limiting the navigator to a particular journal or journals through use of the working set feature limits the scope of a view to a subset of information that has been manually associated by users through their choice to place one or more entries within a journal. Navigators might use creation or modification dates to sort entries. In other areas, such as with the workflow editor and experiment editor, search is used to find blocks that can be placed and available data sources that provide valid input to a particular block. For

example, if a block takes FASTA as its input, when searching for inputs to that block, the scope will be limited to the types that would make a valid workflow/experiment.

"Helper" Metadata Fields

The BCJ-DB features a number of metadata fields that are designed to help a user find what they are looking for. To this point, a total of eight have been discussed. They are *UserId, ContentTypeId, CreateDate, ModifyDate, CommitDate, Committed, Title, and SystemGenerated*. These fields are all important to provenance and thus are equally important to search. A system that maintains provenance that is not searchable is merely keeping provenance for an "after the fact" type of analysis. The use of provenance at creation time facilitates better results that are easier to obtain by limiting the potential data that must be dealt with. Without further ado, the last four entry metadata fields are:

1. *ContentLastAccessDate,*
2. *Deprecated,*
3. *Hidden*, and
4. *Rating*.

ContentLastAccessDate specifies the last time the owner accessed the content of the entry. The date is modified anytime the content is accessed, even after the entry has been committed. This field's primary use is in a BCJ entry navigator for sorting based on the last time an entry's content was accessed. The date could also be useful when displaying a list of possible inputs for a port in a workflow by sorting the most recently used items to the top of the list.

Deprecated indicates that the owner has marked the entry as no longer being relevant. The reasoning could simply be that the entry is no longer up-to-date or perhaps has been found to be incorrect and cannot be deleted due to an essential dependency. In either case, this field can prove very useful for reducing the visual clutter present in a navigator and for searching for annotations, results, workflows, or other information when performing global searches of the BCJ-DB.

Hidden indicates that an entry, while not deprecated, is for some reason cluttering the search space and a user wishes to limit it from appearing. This field is used by tools in situations where information needs to be stored but would be distracting and cause "information overload" if displayed in the navigator or other results. A common use of this flag is with the experiment execution debug file (actually an entry). The debug entry

contains the scripts, program standard in and out, and other useful information about an experiment execution. While this entry can be useful in investigating a potential problem, it is typically more of a visual distraction and is unnecessary. By making this entry hidden, important audit trail provenance is maintained—the fourth type of provenance the BCJ maintains.

Lastly, *Rating*, indicates an integer value from 0 to 5 of the rating the user wants to assign to an entry. This rating could be used for any purpose. While the original specification did not include such a field, discussion with a number of users led to the creation and inclusion of this field. The value '0' is used when the value is unset. The range from 1 to 5 specifies set values, where 1 is the lowest and 5 is the highest rating. In the navigator scenario, a rating on an entry could be used in ordering the entries. Another possibility would be for users to use the flag to assign a value indicating the quality of the data. Users could then search, for example, for data that was of a particular content type, had particular contents, and had a rating greater than 3.

Content Indexing

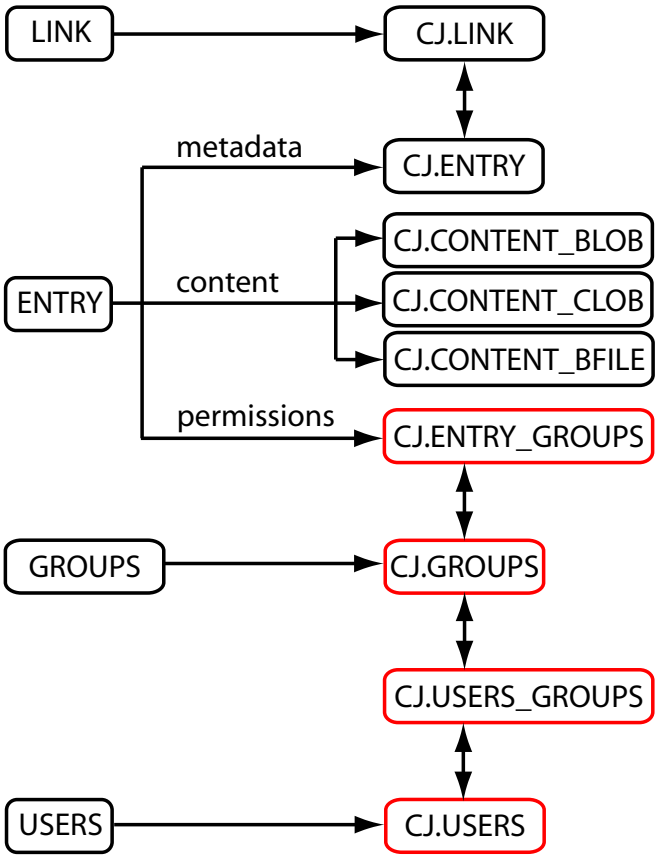
The three-level storage architecture described in chapter four was an important design decision. The purpose of the architecture, as described earlier, was to enable search while maintaining excellent performance for local cluster-based computation. By partitioning storage into three levels, data that needs indexing can be indexed and data where indexing would have a high burden with limited or no benefits is left unindexed. For example, large gene sequences are typically stored as BFILEs. Because indexing gene sequences would requires large computational and memory resources from the database with no real benefits to the user, BFILE storage type is not indexed. BLOB data is typically a serialized Java object, a picture, or another format that is unsearchable and is also left unindexed. CLOB data, however, is textual information that is typically of high value. Thus, partitioning the storage makes it possible to only index the entry content that needs to be indexed. The index specification is:

```
CREATE INDEX CONTENT_CLOB_IDX ON CONTENT_CLOB(DATA)
INDEXTYPE IS CTXSYS.CONTEXT
PARAMETERS ('sync (every "sysdate+(1/24/60)") MEMORY 128M');
```

The last line of the specification, with the parameters, simply indicates that the index is updated once every hour and that the updating process can use at most 128 megabytes of memory.

Complex Join Queries

The end product of the many features discussed so far is a database that is flexible, extensible, and provides comprehensive provenance information stored within. The "trick", of course, is how to get access to the information. The BCJ-DB answer is through queries. While many systems only provide rudimentary query functionality, where a simple question such as, "Tell me all entries authored by user Alice?" is asked, the BCJ-DB supports much more complicated queries. Because all information that is accessible to a user is available, queries asking questions such as, "Show me all entries authored by Alice that were output from a workflow that used the BLAST block?" This query would be a search that joined the *Entry* table to the *Link* table and then joined back to the *Entry* table. Fundamentally, all tables are accessible (given the user has appropriate permissions to the specific row) and thus may be joined together. A query could ask for "All annotations on workflows that contain the text 'searchForMe'." A good formulation of this query would be to first find the *EntryId* of all entries that have the text "searchForMe" joined with *ToEntry*, with the rows in the *Link* table where the *LinkType* references the entry relationship definition titled "Annotation," and, lastly, to then join the *FromEntry* to all entries that are of content type "Workflow."



Chapter VII: Security

Introduction

Security conscious system design supporting authentication for data access has long been the standard security model present in systems. Typical systems support only very coarse-grained security where permissions are assigned to entire projects or journals, similar to assigning permissions on a top-level directory (one attached to the root) and having the permissions inherited by all files and directories within it. In this model to share a specific piece of information with select actors, a new top-level directory would need to be created with the desired permissions and the information copied into that directory. The problems with coarse-grained security is that the top-level becomes polluted with many directories to describe all possible combinations of permissions, provenance information may be obfuscated or lost, and maintenance is complicated by having duplicate files.

By providing advanced security features, the BCJ-DB strives to accomplish three things:

1. Create a secure, private workspace for individuals,
2. Encourage collaboration through the ability to selectively share information with fine-grained controls, and
3. Maintain the integrity of all provenance information the BCJ collects.

To accomplish these goals, the BCJ-DB uses a number of approaches. In chapters three and four the data abstraction, content storage type, and functional interfaces were introduced. In chapter five, the concept of committing an entry was introduced along with the fundamental actor in the BCJ-DB, a user. This chapter furthers these ideas through discussion of the group-based access controls, the BFILE storage type, and the functional interface.

Before proceeding further, an overview of the entire Oracle portion of the BCJ-DB interface is in order. Figure eight shows all of the tables, methods, and procedures discussed up to this point. Additionally, it shows the three new tables used for storing security-parameters and the six views that implement the security specified by the underlying tables. In this figure, the thin arrow heads represent foreign key relationships with the arrows pointing to the foreign key referenced. The broad arrows (filled in arrow-head) point from an underlying table to the exported view of that table.

Users & Groups

The primary actors in the BCJ-DB are users. Users are defined by two characteristics:

1. They have an Oracle account that grants them connect privileges, and
2. They have been granted the 'CJ_USER' role.

A single schema, 'CJ', contains all tables, views, packages, and sequences necessary for operation of the BCJ-DB. Two Oracle roles, *CJ_User* and *CJ_Admin* provide two levels of access for other Oracle users to this schema. The *CJ_User* role defines a limited set of access permissions to the CJ schema. In particular, *CJ_User* is able to execute the packages:

1. TYPES: type aliases,
2. BOOL: defines the Boolean type,
3. USER_PKG: functions & procedures used by users, such as createEntry, and
4. UTILS_PKG: functions to find out information about a user.

CJ_User is granted select on all six views, namely:

1. ENTRY_VIEW,
2. ENTRY_RESTRICTED,
3. CONTENT_CLOB_VIEW,
4. GROUPS_VIEW,
5. USERS_GROUPS_VIEW, and
6. USERS.

The *CJ_Admin* role is for administrating the CJ schema and performing advanced operations from the BCJ environment, such as creating and removing users and groups and associating a user with one or more groups.

Because Oracle roles are used to isolate the BCJ-DB from the rest of the Oracle database and grant access, users are not stored in a table. Rather, they are a view generated by querying against the Oracle DBA_USERS table (stores all users on this Oracle database) and the DBA_ROLE_PRIVS view (associates roles with users and lists their privileges). The SQL for creating this view is:

```
CREATE OR REPLACE VIEW USERS
AS
 SELECT USERNAME AS USERNAME,USER_ID AS USERID
 FROM SYS.DBA_USERS, SYS.DBA_ROLE_PRIVS
 WHERE GRANTEE=USERNAME AND GRANTED_ROLE='CJ_USER'
 WITH READ ONLY;
```

This query can be understood as finding all rows in the *DBA_Role_Privs* view where the role that has been granted is *CJ_User* and joining the *Grantee* column of that sub-query with the *UserName* column of the *DBA_Users* table. The end product is a view that contains all Oracle users that have been granted the role *CJ_User*.

BCJ-DB groups specify a collection of users. This is accomplished through an indirect reference to a list of users as specified in the *Users_Groups* table. Creation of new groups and management of group membership is the role of the *CJ* user or a user with the *CJ_Admin* role. The *Groups* table has two columns, *GroupId* and *GroupName*. The *GroupId* comes from a sequence specified identically (other than name) to that used for generating unique *EntryId* values. The *GroupName* is assigned as a parameter in the *addGroup* procedure. The *Users_Groups* table contains two foreign keys—one to the *Users* view referencing *UserId* and another to the *Groups* table referencing *GroupId*. To associate a user with

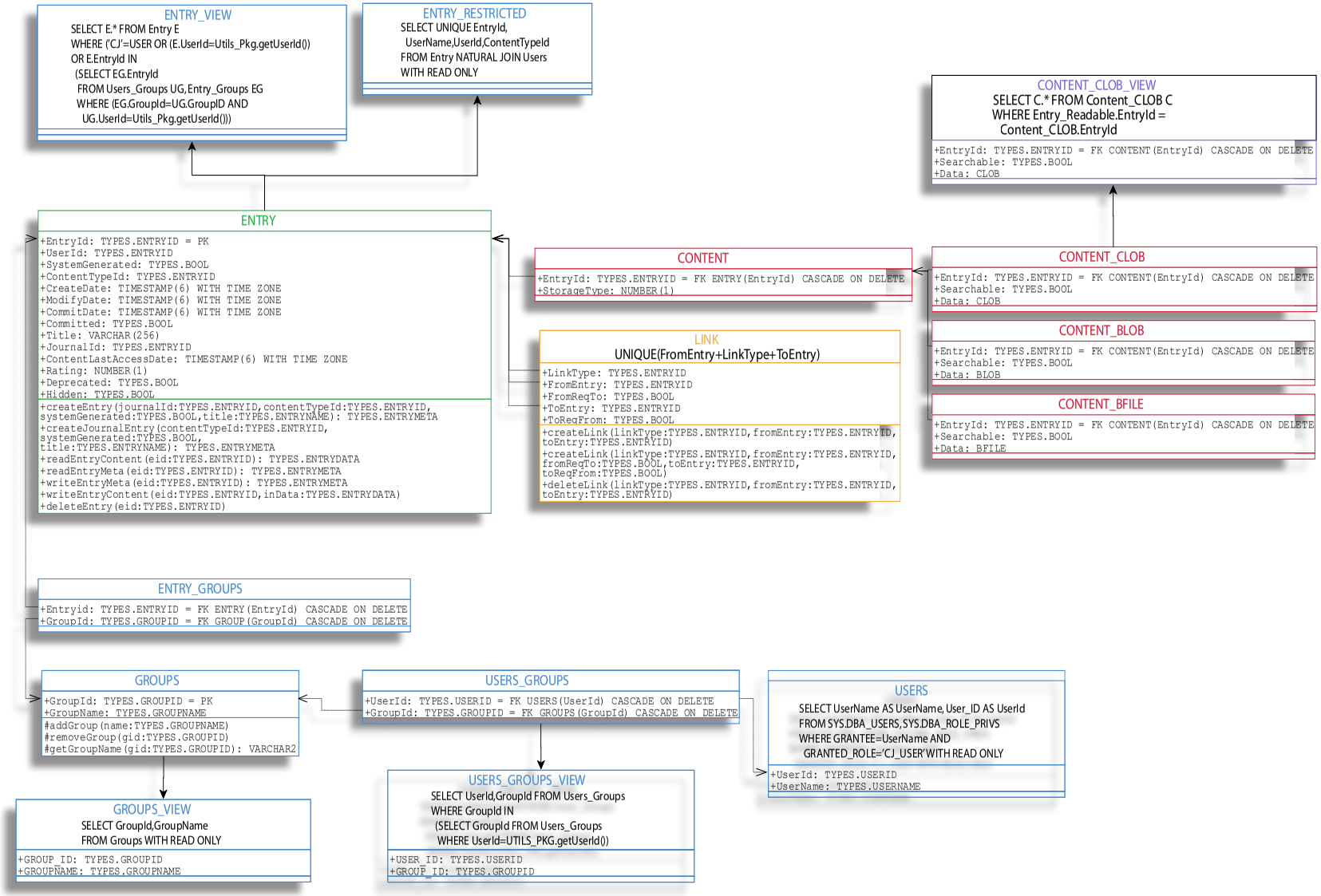


Figure 8: CJ Database Diagram

Figure 9: CJ Schema Packages

a group, an entry is placed in the *Users_Groups* table indicating the *UserId* that is to be a member of a group, represented by its *GroupId*.

Below is a hypothetical set of users, groups, and associations between the two. There are two groups beyond the standard 'ALL' and 'NONE' group specifications. There are four users defined: Lance, Ed, Gary, and Victor. All four users are a member of the group 'ALL'. Lance is a member of the ECBC group. Ed is a member of both the ECBC and the ITTC groups. Gary is a member of no additional groups. Lastly, Victor is a member of only the ITTC group.

GROUPS	
GroupId	GroupName
0	NONE
1	ALL
2	ECBC
3	ITTC

USERS_GROUPS	
GroupId	UserId
1	1
1	2
1	3
1	4
2	1
2	2
3	2
3	4

USERS	
UserId	Username
1	Lance
2	Ed
3	Gary
4	Victor

Table 6: *Users_Groups* Example

Entries, Groups, and Actions

With the basic framework for describing the actors in place, the objects must now have access control lists specifying the actor(s) that may access the objects and what actions they may perform on the object. To keep things simple, the BCJ-DB does not concern itself with complicated actions. The model only supports control over reads and writes at the entry-level.

Create Access Control

Entry creation can be performed by any BCJ user with any *JournalId* specified as the journal the entry is to be contained within. This decision has positive implications for collaboration (everyone may create entries anywhere) and the only potential negative implication of other users having entries in a journal you "own" is easily rectified through the navigator's use of queries to populate its view, thus allowing filtering out of entries by particular users. This global shared data space is one of the primary ways the BCJ benefits users in their collaborative efforts. The challenge, of course, is balancing the desire for collaboration with the need to have a private workspace. The BCJ addresses this via the

navigator and filters. This allows "virtual" private workspaces where others can contribute at will, but a user can choose to ignore the contributions of another user or users at any time.

Read Access Control

A third table, *Entry_Groups*, specifies the association between *EntryId* and the *GroupId*s granted read access of that *EntryId* and associated content.

The rules for processing of the *Entry_Groups* table are as follows:

1. Ownership takes precedence over group membership.
2. If a user owns an entry that is uncommitted, the user:
    a. Can read and write to all user fields, and
    b. Can read and write to the content.
3. If a user owns an entry that is committed, the user:
    a. Can read all entry fields,
    b. Can write to certain user fields,
    c. Can read the content, and
    d. Cannot write to the entry content.
4. If an action related to a particular *EntryId* locates no rows in the *Entry_Groups* table this:
    a. Is equivalent to having the group 'NONE' assigned to that *EntryId*.
    b. Thus, only the owner may access the entry, as defined by rules 1 and 2.
5. If a user attempting to perform an action on a particular *EntryId* is a member of at least one group that has been granted access to that entry and does not own the entry, the user:
    a. Can read all entry fields,
    b. Cannot write to any entry fields,
    c. Can read the entry content, and
    d. Cannot write to the entry content.
6. If a user attempting to perform an action on a particular *EntryId* is not a member of at least one group granted access to an entry and does not own the entry:
    a. The user cannot read or write to any entry fields,
    b. The user cannot read or write entry content, and
    c. An appropriate exception is thrown in response to the action.

To summarize, entry ownership is the primary criterion. If a user owns the entry, group permissions are irrelevant. Only if user does not own the entry do groups need to be considered. Permissions are additive. A user only needs to be a member of one of the groups specified in the ACL for an entry to be granted read access.

Continuing with the previous example, the tables below contain the same users, groups, and group membership information as previously. There are four entries represented in the *Entry_Groups* table. In the case of *EntryId* = 1, *GroupId* 1 and 2 (ALL and ECBC) have been granted access. Therefore, the users Lance, Ed, Gary, and Victor are allowed to read this entry. For *EntryId* = 2, *GroupId* 2 and 3 (ECBC and ITTC) are specified and thus Lance, Ed, and Victor may read this entry. For *EntryId* = 3, *GroupId* 2 (ECBC) is allowed access, thus Lance and Ed may access it. Lastly, *EntryId* = 4 has the 'NONE' group specified, indicating that no user other than the entry's owner may access it.

USERS	
UserId	Username
1	Lance
2	Ed
3	Gary
4	Victor

GROUPS	
GroupId	GroupName
0	NONE
1	ALL
2	ECBC
3	ITTC

USERS_GROUPS	
GroupId	UserId
1	1
1	2
1	3
1	4
2	1
2	2
3	3
3	4

ENTRY_GROUPS	
EntryId	GroupId
1	1
1	2
2	2
2	3
3	2
4	0

Table 7: *Entry_Groups* Example

Entry Metadata & Commit

While all entry metadata fields are readable by users granted access, not all fields are writable. In particular, two categories of metadata fields exist:

1. System-controlled, and
2. User-controlled.

The system-controlled fields may only be modified by the 'CJ' schema itself. Modification is accomplished through the use of a functional interface and via triggers. For instance, methods such as *createEntry* automatically determine the caller's *UserId* and place the value in the *UserId* field when inserting. These system-controlled fields are very important for BCJ-DB integrity and non-repudiation of attribution provenance. Within the system-controlled fields, there are two times where the ability to alter the value stored in a field changes. The first time is at creation. During the creation process fields such as *UserId* and *CreateDate* are fixed. In the interval between an entry being created and committed, all fields but those fixed at creation may be altered. After an entry has been committed,

only a small set of "helper" fields that are designed to improve search capability, such as *Deprecated* and *Rating*, may be altered.

	System-Controlled	
	Fixed at Creation	Fixed at Commit
EntryId	Yes	X
UserId	Yes	X
CreateDate	Yes	X
ModifyDate	No	Yes
CommitDate	No	Yes
ContentLastAccessDate	No	No

Table 8: System-Controlled Fields

	User-Controlled	
	Fixed at Creation	Fixed at Commit
Commited	No	Yes*
ContentTypeId	No	Yes
Deprecated	No	No
Hidden	No	No
JournalId	No	No
Rating	No	No
SystemGenerated	Yes	No
Title	No	No

Table 9: User-Controlled Fields

Write Access Control

As mentioned in earlier chapters, write access to an entry is limited to the entry's creator (owner). This policy decision means that multiple writers to a single entry are not allowed. In a system that uses page concepts, limiting the ability of multiple writers to a single page could prove quite negative for collaboration. In the BCJ-DB, however, the fine granularity of the information stored within a single entry mitigates the potential benefits of allowing multiple writers to a single entry. Instead of writing to the same entry, other users make annotations on entries to comment on the work being done by others.

Owner-initiated writes can be to either the metadata or the content of an entry. When writing to the metadata of an entry, a functional interface is not presented because this might limit the search functionality. Instead, triggers are used on the entry view and table. The triggers on the entry table are standard triggers, of which there are three. They are on the insert, update, and delete calls. These triggers are fired by functions or by

trigger rewrites from *Entry_View*. The insert and delete triggers are only fired by function calls. The trigger on update, however, can be invoked because of a rewrite present on *Entry_View* that simply passes on a call to update for the user-controlled fields (see section later in this chapter) if the current user is the entry's owner. The insert trigger fires before the insert and sets important fields, such as *UserId*, *CreateDate*, and other fields that must be non-null at creation time.

```
CREATE OR REPLACE TRIGGER ENTRY_TABLE_INSERT
 BEFORE INSERT ON ENTRY
 REFERENCING OLD AS OLD NEW AS NEW
 FOR EACH ROW
DECLARE
 uid  TYPES.USERID;
BEGIN
 /* Find userid of person performing INSERT on the table */
 uid := UTILS_PKG.getUserId();
 /* Update appropriate Date fields to current system date */
 :new.CreateDate := CURRENT_TIMESTAMP;
 :new.ModifyDate := CURRENT_TIMESTAMP;
 :new.ContentLastAccessDate := CURRENT_TIMESTAMP;
 :new.Committed := 0;
 :new.Deprecated := 0;
 :new.Hidden   := 0;
 /* Update UserId field to the USER_ID of the person performing the INSERT */
 :new.UserId := uid;
END ENTRY_TABLE_INSERT;
/
```

The update trigger on entry has a bit more complex role. It must handle the behavior related to what is allowed to happen with an entry that is committed or has dependents. In particular, if an entry has dependents, the *Committed* field may not be altered to uncommit the entry. This is one of the ways removal of an entry that is referenced by other entries is prevented until the other entries are removed—by not allowing *Committed* to change. The second method, of course, takes into account the addition to the RDF 3-tuple to ensure essential dependencies are enforced. The function *hasEssentialDependents* takes one parameter, *EntryId*, and determines if essential dependents upon the entry exist by checking if either of two scenarios exist in the *Link* table:

1. If the entry is referenced as the *ToEntry* and there is a *FromReqTo* that is true, or
2. If the entry is referenced as the *FromEntry* and there is a *ToReqFrom* that is true.

If either scenario occurs, *hasEssentialDependents* returns BOOL.TRUE.

Delete Access Control

With the owner-only policy for write permission on an entry established, the delete rules may now be specified. The delete rules can be summarized as follows:

1. If the current user is not the owner, the deletion is never allowed.
2. If the entry has essential dependents, the deletion is never allowed.
3. If the user owns the entry and the entry is committed but has no essential dependents, it may be deleted.*
4. If the entry is owned by the current user, is not committed and has no essential dependents, it may be deleted.

* See *compromise* under Strict vs Relaxed Security

The table below summarizes these rules. The 'X' stands for "don't care."

Current User == Owner	Committed	hasEssentialDependents()	Deletion
FALSE	X	X	Denied
X	X	TRUE	Denied
TRUE	TRUE	FALSE	Allowed*
TRUE	FALSE	FALSE	Allowed

Table 10: Entry Deletion Rules

Strict vs. Relaxed Security

The asterix on rule 3 is there because of a compromise that was later introduced into the BCJ-DB. The original specification for the BCJ-DB only allowed for a very strict model where once an entry was committed, it could never be deleted—regardless of essential dependents or ownership. The suggestion to users was to make use of the *deprecated* flag in the entry metadata and along with filters to reduce visual clutter in the navigator and perform what amounts to a deletion from the visual field, but not from the database. While very appealing conceptually, the problem relates to the database itself and the management of large volumes of deprecated data that could perhaps otherwise be removed from the system. For example, every time an experiment is executed, the outputs generated are committed automatically to prevent the user from altering the results. Unfortunately, given the large potential number of iterations a researcher may perform while trying to form a hypothesis, the amount of low-value (in terms of usefulness) and high-cost (in terms of storage) data will blossom dramatically. The solution to this problem was to develop a more relaxed set of rules that could be applied to the deletion of an entry. The rules in the table above and in the text represent this relaxed security mode.

While beneficial for storage reasons, this mode does have its drawbacks. In particular, the relaxed security mode allows the deletions of an entry that is committed so long as there are no essential dependents. Unfortunately, at present no control over who may remove a relationship between entries exists. Thus, even though a second user may have placed an essential dependency on an entry, the first user could remove that essential dependency and then proceed to delete a committed entry. As a result, the database has two modes it can run in, a strict mode where even if an entry has no essential dependents the entry cannot be removed once committed and a relaxed mode where the essential dependent concept takes can override the committed concept if an entry has no essential dependents.

Managing Access Permissions

Only the owner of an entry may alter the groups allowed to access an entry. This is done through triggers on the *Entry_Groups* table. Whenever a user attempts an insert, update, or delete of a row in this table, the user attempting to perform the action is checked to verify that they are the owner of the *EntryId* being referenced. If user is not the owner, an exception is thrown. If the user is the owner, the trigger checks to ensure that the *GroupId* referenced by the call in fact exist. If it does not, an exception is thrown.

BFILE Security

The creation of the BFILE storage type, while offering performance enhancing benefits, had the potential to harm the BCJ-DB's ability to ensure data integrity. To enhance data integrity, a special file system on the cluster was created that only the database server and a special user that submitted experiments for execution to PBS could access. To further prevent modification of the files, the Oracle server changes the mode bits when an entry changes from being uncommitted to committed and vice-a-versa.

Mode bits are used to control the permissions on UNIX files. There are three actors the mode bits address; each actor has three actions. The first digit in the mode bits specifies the rights for the owner of the file; the second digit specifies the rights for the group associated with the file; and the last digit is for all other users not included by the previous two categories. The precedence in evaluating the mode bits, from highest to lowest, is: user, group, other. The two modes specified below are '660', which means both the user and group can read from and write to the file but cannot execute it, and '440', which

means that both the user and group can read the file but cannot write to it. In both '440' and '660', the other mode bits specifies no read, write, or execute permission is granted.

Of the three triggers on the entry table, two of them have functionality to assist the BFILE implementation for security reasons. First, the update trigger on the *Entry* table will call a Java Stored Procedure (JSP) whenever the committed field changes from '0' to '1' (false to true) to change the mode bits on the associated file to '440' from their original value of '660'. If the update trigger detects a committed entry is being uncommitted, an inverse function that changes the mode bits from '440' to '660' is called. When an entry is deleted, the other trigger is activated and if the storage type is BFILE, then appropriate JSP is invoked to remove the file from the file-system. Even though the file-system should be secure, taking the precaution of immediate deletion of the content is prudent. Additionally, the immediate deletion policy saves disk space that might otherwise be wasted.

Security Views

With the structure in place for assigning access controls to entries, all that remains is to use the values set to implement the rules. This is done through six views. The first view, *Users*, has already been discussed extensively.

Groups_View

The view *Groups_View* simply presents the *Groups* table in a read-only format.

Users_Groups_View

The view *Users_Groups_View* presents a slightly more complicated picture by using the *UserId* of the current user to limit the view presented to the users and groups that the current user is also a member of. Using the example from this chapter, the user Gary was only a member of the 'ALL' group. As such, Gary could only see the other users/group pairs where the user was a member of the 'ALL' group. The rows from *Users_Groups* where *GroupId*=1 would be the only rows displayed in the view. Victor, on the other hand, is a member of 'ALL' and 'ITTC'. Therefore, Victor can see all rows from *Users_Groups* where *GroupId*=1 or 3. The code for this simple but useful view is:

```
SELECT UserId,GroupId FROM Users_Groups
WHERE GroupId IN
    (SELECT GroupId FROM Users_Groups)
    WHERE UserId=Utils_Pkg.getUserId()
```

Entry_View

Entry_View is the most complicated view and ties together four tables. They are *Users_Groups*, *Groups*, *Entry_Groups*, and *Entry*. *Entry_View* has three paths that could result in an entry being viewable:

1. The current user is 'CJ',
2. The current user is the owner of the entry, or
3. The current user is a member of a group that has been granted access to the entry.

The first two should be rather obvious. *Entry_View* first finds all groups that are attached to the *EntryId* by using the *Entry_Groups* table. *Entry_View* then retrieves the *UserId* of the current user and then finds all groups that user is a member of by consulting *Users_Groups*. The view then compares the two sets of groups to determine if any intersection between the two exists. If the two are conjoint, the user is granted access. If the two sets of groups are disjoint, then *Entry_View* will not display that row of the *Entry* table to the current user. The code for the view is:

```
SELECT E.* FROM Entry E
WHERE (
    'CJ'=USER OR
    (E.UserId=Utils_Pkg.getUserId()) OR
    (E.EntryId IN (SELECT EG.EntryId from Users_Groups UG, Entry_Groups EG
    WHERE (
            EG.GroupId=UG.GroupId AND
            UG.UserId=Utils_Pkg.getUserId()))
```

Exception Handling & Entry_Restricted

The last topic has do with what happens when a user is unable to access an entry. The BCJ-DB attempts to take a pragmatic approach to the problem while still maintaining good security and preventing excessive information leakage. The general consensus was that if an entry were inaccessible but a user was trying to access it, the reason for this occurrence was a change in the permissions assigned to the entry. To assist the user, a view that would present some limited information about the entry that might identify the owner and direct the user to the entry was desired. This view presents the *EntryId*, *UserName*, *UserId*, and *ContentTypeId*, of all entries but excludes the *Title* field and is specified as:

```
SELECT UNIQUE EntryId,UserName,UserId,ContentTypeId
FROM Entry NATURAL JOIN Users WITH READ ONLY
```

Overall, this simple view provides the ability to trace the owner of an entry who changed the access controls.

Content_CLOB_View

The final view integrates with the BCJ-DB search capabilities. This view presents the CLOB content present to those who have read-access to the associated entry. The view does this by performing a simple, but effective, join on an internal view used to determine read access permissions. The code is:

```
SELECT C.* FROM Content_CLOB C
WHERE Entry_Readable.EntryId = Content_CLOB.EntryId
```

Concluding Remarks

The security model, while simplistic in nature, is flexible and efficient and provides far greater security assurances than many competing provenance management systems do while still affording good search capabilities. The use of views is fundamental to making all of the features of the BCJ-DB accessible.

Chapter XIII: Analysis & Conclusion

Analysis

The Bioinformatics Computational Journal, unlike its predecessors and many of its contemporaries, truly provides an integrated computational biology platform. Within the BCJ ecosphere, the BCJ-DB is responsible for all data and meta-data management. In order to perform these duties, the BCJ-DB addressed a number of important areas.

First, a flexible, extensible type hierarchy was implemented. The four core entries achieve those attributes through a simple but elegant design. The core concept is that data can and should be decomposed into a single entity to be managed—an entry. This entry should be identified by a globally unique identifier within the BCJ-DB environment. Each entry is associated with one simple type definition. While other systems have developed a hierarchy to represent relationships between types, the BCJ-DB shunned that approach for the a simpler flat structure where all types are linked to a single master type. This link to a master type is done only to distinguish type definitions from other entries in the system. Unfortunately, in the bioinformatics field having a type hierarchy can be complicated to manage and also impedes the easy addition of new types.

To help manage this flat structure, the concept of a journal was added to the system. Although the entry-journal structure could be implemented through the use of relationships, instead it was done through a single field in the entry meta-data. Links offer flexibility but at the cost of lower performance. Static allocation of the parent journal field in the entry meta-data improved performance and promoted the concept from a 2nd order to a 1st order data management technique. A similar decision was made with the content type field for an entry.

At present, the BCJ-DB's primary limitation is the breakdown of absolute security in the essential dependency concept. This breakdown occurs when an unwitting or malicious user places a large number of links that create an essential dependency on entries that are inaccessible to the user. The problem is that an essential dependency link should only be allowed to be made to a committed entry. However, the current BCJ-DB implementation does not perform this check and should do so. When a user attempts to create a new relationship that will create an essential dependency, checks should be performed to ensure that this principle is not violated.

BCJ-DB performance has not been analyzed thoroughly. Simplistic benchmarks evaluating the read and write performance to various content types were performed with a single reader/writer. These results were favorable, but further investigation into multiple-user benchmarks should be performed. However, in its goals to develop an integrated computational biology framework with powerful provenance and security features while maintaining excellent cluster performance, the BCJ was very successful. Experiments run through the BCJ had identical performance to their command-line counterparts. The big win with the BCJ, of course, is the automatic provenance tracking, job submission, results management, and ability to easily share inputs, workflows, and results with others.

Conclusion

When comparing the biological research performed with notebooks in a wet-lab and the more modern computer-based bioinformatics research, the most critical problem is the impedance mismatch between the paper world and the computer world. With paper devices such as laboratory notes and books, conveniently flipping through pages in search of useful information can be done using "fuzzy" searches and other pattern recognition techniques humans are born with innately and developed further over the subsequent years.

In the electronic world, unfortunately, these techniques are not generally applicable. Computational sciences nearly always result in "information overload," where an individual is overwhelmed by the sheer volume of data presented and can no longer critically evaluate the information present because the throughput necessary to obtain results in a reasonable time is unattainable. Two, often complementary, solutions to information overload exist. The first is the development of better computer algorithms to filter or otherwise reduce the information to a manageable form. The second is furnish better tools for data management. Better data management makes it possible to know that your pages are well organized with full metadata and are always accessible. When combined with powerful search, by analogy, the system is like a dictionary where it is always easy to flip to the right page when looking up a word of interest.

The data management techniques that enable this capability are comprehensive collection of all five types of provenance metadata along with powerful search capabilities to ask insightful questions about the data and metadata that has been collected. While many systems address some limited form of informational or replication provenance, few ad-

dress questions of accurate, secure attribution and auditing. No current systems answer questions of data quality in a comprehensive manner for bioinformatics because of a lack of suitable metrics with which to evaluate data quality.

To address these challenges, the BCJ-DB supports four of the five types of metadata and can perform powerful searches involving metadata and data. The extensible content and relationship types allows users to add to the data and metadata search features of the BCJ-DB. Subsequently, the user can use those search features without having to abandon prior research accomplished with the BCJ. The managed storage provided by the BCJ-DB ensures that vital information is retained forever, even if the user is not the owner. The powerful search capabilities encourage users to use found data to further their own work, perhaps then deriving new results from the original data and subsequently sharing of the new results with others.

Consequently, the globally shared workspace of the BCJ-DB provides an area for collaboration and sharing of ideas beyond that typically seen. By having a single shared location for storage of all resources, users can easily share by simply modifying the access permissions on an entry. The ability to share not just data but workflows in a common space greatly improves collaboration. Workflows encapsulate ideas and processing concepts, not just raw data. By working in the same journal, work can be organized for easy access by all members of a project. Powerful annotation capability comes from the ability to define new types of content that can be attached. Powerful annotation helps remove the potential for a plot of the potential energy in an molecular dynamics energy minimization experiment becoming separated from the inputs, workflows, parameters, and outputs it was derived from.

Flexible scoping of shared information encourages users to share because they know that only small bits of information will be shared with a select set of users and groups they specified. Unlike with coarse-grained security, users are not forced to share an entire journal when a colleague simply needs one piece of information. On the other hand, if a user wants to share all entries with all users, that can also be done.

Furthermore, the BCJ-DB extension of the RDF 3-tuple into a 5-tuple not only prevents loss of important provenance but also encourages collaboration because users can use

information provided by others in their research since the researcher is guaranteed that information cannot be removed.

By combining a globally shared workspace with powerful search capabilities, an extensible type hierarchy, and a fine-grained security model, and the concept of essential dependency relationships between entries, the BCJ-DB makes a truly powerful ally for the bioinformatics research community.

Citations

[Anon95] "Information revolution calls for changes in behavior," *Chemical and Engineering News*, American Chemical Society, March 27, 1995. Available HTTP: http://pubs.acs.org/hotartcl/cenear/950327/art04.html

[Bran03] D. Brandon, K. Vandivort, R. Brunner, G. Budescu, "2003 BioCoRE Survey Report," Theoretical and Computational Biophysics Group, Beckman Institute for Advanced Science and Technology, University of Illinois at Urbana-Champaign, 2003. Available HTTP: http://www.ks.uiuc.edu/Research/biocore/eval/surveyData/BioCoRE_2003_Survey_Report.pdf

[Bune06] P. Buneman, A.P. Chapman, J. Cheney, "Provenance Management in Curated Databases," ACM SIGMOD Conference on Management of Data, 2006.

[Butl05] D. Butler, "Electronic Notebooks: A new leaf," *Nature*, Issue 436, pp. 20-21.

[Fitz00] M.C. Fitzgerald, "The evolving, fully loaded, electronic laboratory notebook," *Chemical Innovation*, American Chemical Society, January, 2000. Available HTTP: http://pubs.acs.org/hotartcl/ci/00/jan/inet.html

[Eric96] T. Erickson, "The Design and Long-Term Use of a Personal Electronic Notebook: A Reflective Analysis," Human Factors in Computing: The Proceedings of CHI '96, Vancouver, BC, CA, April 1996. Available HTTP: http://www.visi.com/~snowfall/notebook.html

[Gobl02] C.Goble, "Position Statement: Musings on Provenance, Workflow and (Semantic Web) Annotations for Bioinformatics," in *Workshop on Data Derivation and Provenance, Chicago,* 2002.

[Gray02] J. Gray, A. Szalay, A. Thakar, C. Stoughton, J. vandenBerg, "Online Scientific Data Curation, Publication, and Archiving," Microsoft Research Technical Report MSR-TR-2002-74.

[Gray05] J. Gray, D. Liu, M. Nieto-Santisteban, A. Szalay, D.J. DeWitt, G. Heber, "Scientific Data Management in the Coming Decade," RECORD, Association for

Computing Machinery Special Interest Group on Management of Data (ACM SIGMOD), vol. 34, no. 4, December, 2005.

[Gupt05] A. Gupta, B. Ludascher, L. Raschid, "Report on the 2nd International Workshop on Data Integration in the Life Sciences (DILS'05)," ACM SIGMOD Record, Volume 35, Issue 2, June, 2006, pp. 56-58.

[Gwiz98] J. Gwizdka, "Categorization Is Difficult: Use of an Electronic Notebook for Organizing Design Meeting Notes," in Proceedings of Human Factors and Ergonomics Society 42nd Annual Meeting, Chicago, October 5-10 1998, pp. 516-520.

[Hala87a] F.G. Halasz, T.P. Moran, and R.H. Trigg, "NoteCards in a Nutshell," in Proceedings of CHI+GI 1987, 1987, pp. 133-143.

[Hala87b] F.G. Halasz, "Reflections on NoteCards: Seven Issues for the Next Generation of Hypermedia Systems," Communications of the ACM: Hypertext '87 Papers, November, 1987, pp. 836-852.

[Myer01a] J.D. Myers, E. Mendoza, A. Geist. Scientific Annotation Middleware: Technical Overview. [Online] Available HTTP: http://collaboratory.emsl.pnl.gov/sam/samtechoverview.html

[Myer01b] J.D. Myers, E. Mendoza, B. Hoopes, "A Collaborative Electronic Laboratory Notebook," in *Proceedings of the IASTED International Conference on Multimedia Systems and Applications* (IMSA 2001), August 13-16, 2001, Honolulu, Hawaii. Available HTTP: http://collaboratory.emsl.pnl.gov/presentations/papers/ELN.IMSA.final.pdf

[Myer03a] J.D. Myers, A.R. Chappell, M. Elder, "Re-Integrating the Research Record," Computing in Science and Engineering, May/June, 2003.

[Myer03b] J.D. Myers, C. Pancerella, C. Lansing, K.L. Schuchardt, B. Didier, "Multi-Scale Science: Supporting Emerging Practice with Semantically Derived Provenance," in Proceedings of the Semantic Web Technologies for Searching and Retrieving Scientific Data Workshop, Sanibel Island, FL, Oct 20, 2003.

[Shan05] S. Shankar, A. Kini, D.J. DeWitt, J. Naughton, "Integrating Databases and Workflow Systems," RECORD, Association for Computing Machinery Special Interest Group on Management of Data (ACM SIGMOD), vol. 34, no. 3, September, 2005.

[Simm05a] Y.L. Simmhan, B. Plale, D. Gannon, "A Survey of Data Provenance in e-Science," RECORD, Association for Computing Machinery Special Interest Group on Management of Data (ACM SIGMOD), vol. 34, no. 3, September, 2005. Available HTTP: http://www.sigmod.org/sigmod/record/issues/0509/p31-special-sw-section-5.pdf

[Simm05b] Y.L. Simmhan, B. Plale, D. Gannon, "A Survey of Data Provenance in e-Science," Technical Report IUB-CS-TR618, Computer Science Department, Indiana University, 2005. Available HTTP: http://www.cs.indiana.edu/pub/techreports/TR618.pdf

[Skid98] J.L. Skidmore, M.J. Sottile, J.E. Cuny, A.D. Malony, "A Prototype Notebook-Based Environment for Computational Tools Computational Tools," *IEEE/ACM Conference on Supercomputing, 1998*, pp. 22-32.

[Taub06] J. Tauberer. (2006, July). What is RDF. [Online]. Available HTTP: http://www.xml.com/pub/a/2001/01/24/rdf.html

[Rose06] L. Randall, Interview with Charlie Rose, Charlie Rose, PBS, New York City, NY, USA, December 12, 2006.