THE REPRESENTATION AND STORAGE OF COMBINATORIAL BLOCK DESIGNS

by

Hatem Nassrat

Submitted in partial fulfillment of the
requirements for the degree of
Master of Applied Computer Science

at

Dalhousie University
Halifax, Nova Scotia
APRIL 2009

DALHOUSIE UNIVERSITY

FACULTY OF COMPUTER SCIENCE

The undersigned hereby certify that they have read and recommend to the Faculty of Graduate Studies for acceptance a thesis entitled "THE REPRESENTATION AND STORAGE OF COMBINATORIAL BLOCK DESIGNS" by Hatem Nassrat in partial fulfillment of the requirements for the degree of Master of Applied Computer Science.

Dated: APRIL 15, 2009

Supervisors:

P. Bodorik

P. Dobcsányi

Reader:

C. Watters

DALHOUSIE UNIVERSITY

DATE: APRIL 15, 2009

AUTHOR: Hatem Nassrat

TITLE: THE REPRESENTATION AND STORAGE OF COMBINATORIAL BLOCK DESIGNS

DEPARTMENT OR SCHOOL: Faculty of Computer Science

DEGREE: M.A.C.Sc. CONVOCATION: MAY YEAR: 2009

Table of Contents

Abstract

Combinatorial block designs are in essence a multiset of subsets of a base set with certain properties. Many statistical and combinatorial properties are associated with block designs. These properties are often computationally intensive to generate and therefore capturing them once generated is important. This project, composed of three phases, aims to create a system to represent, store and allow searching for a large collection of combinatorial block designs. The first phase of the project dealt with the representation of block designs. To fulfil this step, the External Representation (Ext Rep) of Block Designs was extended and re-implemented to use JavaScript Object Notation (JSON), creating version 3 of the Ext Rep. Fulfilling this phase dealt with complexities in transforming designs in the predecessor version, which was implemented in Extensible Markup Language (XML), to the newer version. The complexities arose due to the inherent differences between the two languages. Moreover, the new Ext Rep contains extensions to the functionality that were not available prior. Block designs (represented as Ext Rep structures) required a storage scheme that was created in the second phase of the project. A carefully designed database schema was created along with the choice of a suitable database engine. The final phase dealt with an implementation of a web interface to the database that hosted over two and half million designs which are searchable by any of the Ext Rep criteria.

Chapter 1

Introduction

Combinatorial block designs are used in various applications that combine permutations, combinations and partitions of element sets. Such applications are largely seen in the field of experimental design and various fields within computer science. Combinatorial block designs, also referred to as experimental and statistical block designs, are of great scientific importance and are in need of a massive collection and archiving scheme. Having an appropriate archive allows users to search and browse the library of combinatorial block designs. This project deals with the issues involved with archiving large sets of combinatorial block designs, storing them in a searchable database and providing an interface to such a database.

1.1 Combinatorial Block Designs

A *combinatorial block design* is a mathematical object of great importance in many computational fields of study including statistics and computer science. Block designs are described generally as a set system of a particular nature. They were first used in the design of statistical experiments, as a systematic method of dealing with differences in experimental material [19]. To statistically analyse the experimental results, test points (*treatments*) are partitioned into blocks, hence originating the term "Block Design".

Combinatorialists and statisticians see block designs in different ways. To a statistician, a block design is a set of "*plots*" ("*experimental units*") which is partitioned into "*blocks*", with a function from the set of blocks to the set of "*treatments*" to be experimented on. To combinatorialists, the set of treatments is known as "*points*" identifying each block as a subset of treatments occurring on plots in that block. The block design D is thus viewed as a set of points V and a multiset of subsets of points (multiset of blocks (B)) [21].

The following example, from [19], emphasizes how block designs are used. There

are seven varieties of seeds (treatments), and they are to be tested in an agricultural experiment. If there were 21 identical plots of land for the test, then it would be clear that three plots may be planted for each seed type. However, if the plots were spread on 7 different farms in different regions, where each farm had three plots, then the experiment's design would be modified. The following scheme would be a good design for the experiment:

```
Blocks = [
    [0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5],
    [1, 4, 6], [2, 3, 6], [2, 4, 5]
]
```

Each element in the set of *blocks* would represent one farm and each element within a block would represent the variety of seed to be tested on that particular farm. Therefore on $farm[0]$: $seedvariety[0], seedvariety[1], seedvariety[2]$ would be planted on each of the three plots. The above design is statistically optimal which means that performing each of the experimental units allows for the comparison of the seven seed varieties.

The block design above is a special design famously known as the "*Fano plane*" and is part of a group of designs called *balanced incomplete-block designs* or **BIBD**s [23]. More specifically, the Fano plane is a member of set of designs known as t-designs. All t-designs have optimal statistical properties, and therefore are the best candidates for experimental design [21][29].

The Fano plane is labelled **2-(7,3,1)** meaning that there are seven varieties, each block contains three of them and every two varieties occur in the same block exactly one time. The label is a grouping of four general properties of a block design, t, v, r, λ where $t = 2$, $v = 7$, $r = 3$, $\lambda = 1$. Usually the parameters b and k are also used to label block designs, where b is the number of blocks and k is the size of each block (which is applicable only when the design has a constant block size). The *Fano plane* design may be also labelled using the template $t - (v, b, r, k, l)$ giving the label **2-(7,7,3,3,1)**.

Combinatorial block designs are being utilized in many fields of study, with deep roots in experimental design. Other areas where block designs have been used include

error-correcting codes, graph packing and covering problems [22], and finite geometry [19]. Hamming codes are a great example of block design application as they were discovered by Fischer as a design 5 years prior to R. W. Hamming's discovery of them in the field of error correction [19]. Many areas requiring combinations, permutations, partitioning, and other functions on point sets find block designs useful in their application.

1.2 Project Phases: Motivation, Contributions, and Outline

This section outlines the three project phases that include External Representation, database design, and system interface. For each of the phases, motivations and contributions are mentioned.

1.2.1 External Representation of Block Designs

Many statisticians, scientists and programmers utilize Combinatorial block designs in their work. Prior to the work of Cameron et al. [21] no standard format for representing block designs was available. Cameron et al. published the *External Representation of block designs* version 1.1 [21] in 2003 to fill this void. The External Representation served as documentation for block design users on how to represent a block design in a unified way. One of the reasons for the invention of this standard was that it can be utilized by all components of the research resource server at DesignTheory.org. The External Representation protocol allowed any component on the server to communicate with any other component in a standard manner. The protocol was designed to allow external *users* and applications to utilize resources provided by the resource servers and any servers that were designed to be compatible with this format for representing designs and their properties. The concept of Ext Rep "users" is described as both human and software agents [21].

The External Representation version 1.1, and later version 2.0, Extensible Markup Language (XML) was chosen as the language to be used to represent block designs and the design's various properties. The External Representation paper [21] explains the choice of XML over a few alternatives that were suitable for the implementation of the External Representation. Their main motivation was due to the popularity of XML and the availability of tools that worked with XML.

Although XML had its advantages it did not seem to be a great match for combinatorial block designs. XML seems best suited for text; However, a given dataset of a mathematical nature requires a better suited representation language [36][53][10]. For this reason, a segment of this project deals with writing a new External Representation of Block Designs that is based on a more suitable representation language.

In the following Chapter, labelled *External Representation Version 3*, I discuss the External Representation in general, the previous versions shortcomings, how Version 3 solved these issues. Moreover, specific examples of how I implemented the Version 3, and how the designs already produced in the version 2 format were converted into the newly developed version 3 format is also discussed.

1.2.2 Searching for Block Designs

Historically combinatorial block designs were published in papers and text books published by the researchers that discovered the particular designs. Block designs were sometimes available in survey publications that grouped a particular class of designs. To find a design users had to locate the document within which the design was published. Moreover, many publications only displayed the blocks of the design or additionally particular properties of a design that were used within the paper. For the user to see the properties they are interested in they had to compute those properties. The time required to compute these properties cannot be predetermined due to the combinatorial nature of the designs. For that reason it is useful to store those properties whenever they are computed, to reduce the overhead of recomputing them when they are needed.

Aiming to gather the designs into a central repository, a group of researchers at the Queen Mary University of London, under a fund from The Engineering and Physical Sciences Research Council, UK, generated around two and a half million block designs and stored them using the External Representation of block designs version 2 (explained in section 1.2.1) format. This marked the birth of DesignTheory.org, where their collection is currently stored along with other research documents and papers relating to the field of combinatorial designs. The designs in the collection have been grouped in files by their t, v, b, r, k, λ (see Section 1.1) parameters (when available).

The DesignTheory.org collection is considered the largest single collection of block designs available for public use. This collection was not placed in a database system (relational or otherwise), which makes it hard to search for a given design. Rather, the collection was organized in a simple manner as files on a file system. To effectively search the entire collection for a particular design, the user must download the entire collection and recursively search the file contents to find the design.

Moreover, a dynamic searchable database of combinatorial block designs for any collection was never attempted. Aiming to create a functional central repository of combinatorial design, this project deals with placing DesignTheory.org files in a searchable database. In Chapter 3, I discuss the choice of the most suitable database engine, the design of the internal structure of the database that hosts the design collection, the process of populating the database, and the design and implementation of the Query Engine sub-system used to query the database.

1.2.3 System Interface

In this project, a web interface was also designed and implemented to allow users to utilize the database and find designs they are interested in. Chapter 4 discusses the choices made to implement the web interface, the design and implementation of the interface code, and various screen shots displaying how the web interface is utilized to query the database. The web interface with respect to browsing the library and viewing the search results is further described in chapter 6. In particular section 6.1 discusses how my system helped with a particular user's search for a specific design.

The objective of creating a centralized repository for combinatorial block designs is to present a one stop shop for finding block designs, thus making it simple for users to find the designs they are looking for. Moreover, this project provides the framework for external users to upload their personal designs to the central repository.

1.3 Outline

Before proceeding with the next chapters, the final section of this chapter discusses the choice of programming language.

Chapter 2 describes the External Representation in detail, along with the shortcomings of its version 2. Moreover, the chapter describes the decisions made to come

up with the version 3, along with some of the tools implemented to aid with this process.

The discussion continues with chapter 3, where the design, implementation, and population of the searchable combinatorial design database is described. Moreover, the decisions that lead to the choice of a suitable database management system is described, along with a unique query language that is used to search the implemented database.

I follow with the description of the web interface to the system in chapter 4. Some of the various options available to implement the web framework are discussed, along with brief implementation details and screen shots to display the interface.

In chapter 5, I discuss the various techniques utilized to create a system that is easily deployable. Moreover, the choice of web server, interface to the implemented web application and the tools implemented to automate the deployment on Unix based machines are discussed. Furthermore, the results of a brief experiment to determine the maximum system load is displayed in that chapter.

The overview of the full system and how each of the sub systems integrate is explained in chapter 6. Also included is a description of how my system should be utilized, along with a brief example of a specific search for a certain design. Moreover, the usage counts of the system for the year 2008 and the month of January in 2009 are displayed in this chapter.

Finally I conclude with a summary of the effort that took place along with future possibilities and extensions to my project.

1.4 Implementation Programming Language

To implement this project many programming languages were briefly considered. Since this project has a wide variety of aspects that require code to be written, including the parsing of documents, database creation and utilization, and a web application, it was best if the languages of choice was capable of performing all these aspects. From the candidates considered Perl, PHP, Python, and Ruby best fulfilled these criteria. After considering each of these candidates, Python [50] was chosen as the main programming language for this project for the following reasons:

Large Standard Library

Similar to many programming languages, Python contains a large standard library which allows for fast paced code development.

Numerical and Scientific Packages

Many packages that work with combinatorial designs have been either written or wrapped in Python. Examples of such software include *PYDESIGN* [30] and *block-design* [31]. Such packages were required for the work that was done.

High Level

Python is a very high level programming language, which allows for expressive scripts in a few lines of code. Such a style is shared by a few other programming languages, including Scheme [49] and Ruby [34].

Web Programming

Python is growing to be the web programming language of choice. Many organizations are porting their web applications to Python [46][47]. The advantages of Python definitely contribute toward today's large-scale push for its use. This could arguably be similar to the boost that Ruby started experiencing a couple of years ago. However, they both are high level languages with frameworks that allow for decoupled web applications.

These features directed the choice toward the Python programming language. Speed of development was the largest contributing factor, which is contributed to by each of the mentioned advantages. Such characteristics are needed to complete such a scale of a project within the given time frame and resource constraints.

Chapter 2

External Representation Version 3

The External Representation (Ext Rep) as defined in section 1.2.1 is very important when discussing designs. It provides a common protocol for parties to be able to communicate designs.

Essentially a *block design* is a list of blocks, each of which is a list of points. To utilize designs, in the various fields such as combinatorics and experimental designs, the combinatorial and statistical properties of designs often need to be considered. For example when performing experiments designed utilizing combinatorial designs, the robustness properties (an entry under statistical properties) are important. The robustness properties state the ability of a design to maintain its statistical significance in the case of a loss of particular plots or entire blocks [21][29]. Combinatorial design's properties, in general, are computationally intensive to calculate and often rely on an entire set of pairwise non-isomorphic designs (sometimes available when the design is generated), in which they are part of, to be available. Therefore, design properties must be stored along with the design once found since their re-computation is a waste of time and resources and it is for that reason the Ext Rep allowed for possible properties of block designs to be placed within an *Ext Rep structure*.

The notion of *block design document* is simply a set of blocks along with the properties associated with the blocks. The Ext Rep documentation [21][29] describes how to create *Ext Rep compliant* documents. The Ext Rep documentation goes into detail explaining each of the properties such that it is useful even for readers that are not familiar with combinatorial block designs.

Since a combinatorial design is simply a set of blocks, presenting a set of blocks is the minimum needed to represent a design using the Ext Rep structure. However, as mentioned above, there are many statistical, combinatorial properties, and functions (such as generators that produce the isomorphic designs from a given design) encoded within the Ext Rep structure. The Ext Rep protocol is designed to allow users to input

as much, or as little information, depending on their specific needs or requirements. This chapter describes the various shortcomings of the Ext Rep $v2$ protocol and the new version of the External Representation [29] that I made to overcome these problems of version 2. The full Ext Rep v3 schema can be seen in Appendix A of this document and in [29].

2.1 Shortcomings of Version 2

The previous version, External Representation $v2$, is based on the Extensible Markup Language (XML). After using this language its shortcomings become apparent. The need for a data representation language that would be closer to the mathematical structure, in various mathematical systems, had become essential. Examples of such systems include R [33], GAP [38], Mathematica [52], and SAGE [48]. Also the verbosity required for XML with regard to opening and closing tags to denote entities has been seen to make the document quite large and unreadable.

2.2 Solutions and Variations in Version 3

Multiple representation languages had been investigated and JSON [25] had been found to be the most suitable for the representation of block designs. Where XML is aimed to best serve text, JSON [4] allows for basic data types and data structures (arrays and associative arrays) [36][53]. Moreover the syntax used in JSON is very close to the syntax of the same data structures in the previously mentioned mathematical systems ([33][38]) along with a few interpreted programming languages such as Python [50]. JSON's syntax is also much less verbose than its predecessor XML as it does not contain the opening and closing tag system of XSLT based languages. However, where XML can be directly 1-1 mapped from a tree structure, JSON contains *list* structures that allow for a more condensed representation of repeated nodes within a tree.

The External Representation $v3$ was written to closely follow the design of the External Representation $v2$. However, due to the inherent differences between XML and JSON some changes had to be made. Moreover, there were some changes that were made to enhance the functionality of the Ext Rep document, its usefulness, and

```
<root>
  <nodes>
    <node> <z>1</z> <node/>
    <node> <d>1.5</d> <node/>
    <node> <q>2/3</q> <node/>
  </nodes>
</root>
```

(a) XML

```
{
  "root": {
    "nodes": [1, 1.5, {"Q": [2, 3]}]
  }
}
```

(b) JSON

Figure 2.1: Snippets showing the same structure in XML and JSON

level of readability.

This section describes the core differences between the *Ext Rep* versions with examples to highlight these changes. An Extended BackusNaur Form (E-BNF) schema for the Ext Rep $v3$ is appended to this document (see Appendix A). It may also be found along with the complete specification of the Ext Rep $v3$ in [29].

2.2.1 From XML to JSON

A great advantage of the new Ext Rep is that it was written in JSON. The integrated standard data types in JSON along with its two main data structures (lists and objects) allow for much more, and in a much simpler form, than XML allows. It was clear that JSON was better suited for mathematical structures. Figure 2.1 displays snippets of an XML tree (2.1a) and its respective implementation in JSON (2.1b).

2.2.2 Structural Manipulation

The basic differences in the underlying representation of the data begin with the ability of JSON to store the basic data types, namely *numbers* and *strings*. In the previous version of Ext Rep, implemented in XML, these basic data types were not implicitly present and were required to be explicitly declared using wrapper tags that reflect each of the data types. Since these are no longer required, these tags were all "*lifted*". The only wrapper remaining that was used to define a data type in the Ext Rep $v2$ was the *rational number* wrapper as rational numbers are also not a basic data type in JSON. Figure 2.1 displays how each of the datatypes were represented in Ext Rep $v2$ and $v3$.

```
<list_of_designs design_type="block_design" dtrs_protocol="2.0" no_designs="1"
 pairwise_nonisomorphic="true" xmlns="http://designtheory.org/xml-namespace">
    <designs>
        <block_design b="7" id="t2-v7-b7-r3-k3-L1-0" v="7">
            <blocks ordered="true">
                <block><z>0</z><z>1</z><z>2</z></block>
                <block><z>0</z><z>3</z><z>4</z></block>
                <block><z>0</z><z>5</z><z>6</z></block>
                <block><z>1</z><z>3</z><z>5</z></block>
                <block><z>1</z><z>4</z><z>6</z></block>
                <block><z>2</z><z>3</z><z>6</z></block>
                <block><z>2</z><z>4</z><z>5</z></block>
            </blocks>
        </block_design>
    </designs>
</list_of_designs>
```

Figure 2.2: the root nodes from an Ext Rep v2 document

The Ext Rep $v2$ contained tags special tags that were used to denote a repeated structure (i.e. a list). For example more than one *block* were placed under the tag *blocks*. The tag *block* has been *lifted*, since lists are standard structures in JSON documents (see Fig 2.1).

Another tag that went through such a *lifting* process is the *list_of_design* tag. This tag is found at the root of every Ext Rep $v2$ document and was previously required because XML requires a root node for every document. Fig 2.2 and 2.3 display the differences between the root nodes of the Ext Rep trees in the respective versions.

Some structures in the previous Ext Rep version seemed to be modelled after structures that are implicitly available in the JSON representation language. Examples include the various repeated tags which were transformed systematically into a list of elements. Moreover, the Ext Rep contains a structure known as "*Index Flags*". Such flags are index based structures and seemed to better fit with the JSON object structure. Therefore the list of *Index Flags* were transformed into mappings of indices to the data represented by each index. This transformation is displayed in Fig 2.4 of this document.

Another process that was performed systematically in the transformation into the Ext Rep $v3$ was labelled "*Merging*". A few sub-tress in Ext Rep $v2$ contained pairs of attributes which were dependant on on one another. These entries were systematically

```
{
  "external_representation_version" : "3.0",
  "design_type" : "block_design",
  "number_of_designs" : 1,
  "pairwise_nonisomorphic" : true,
  "designs" : [
    {
      "type" : "block_design",
      "id" : "t2-v7-b7-r3-k3-L1-0",
      "v" : 7,
      "b" : 7,
      "blocks" : [
        [0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4, 6],
        [2, 3, 6], [2, 4, 5]
      ]
    }
  ]
}
```

Figure 2.3: the root nodes from an Ext Rep v3 document

```
<t_wise_balanced>
    <index_flag flag="true" index="1">
    </index_flag>
    <index_flag flag="true" index="2">
    </index_flag>
</t_wise_balanced>
```

(a) v2

```
"t_wise_balanced" : {
  "1" : {
    "lambda" : 3
  },
  "2" : {
    "lambda" : 1
  }
}
```

(b) v3

Figure 2.4: *t_wise_balanced* in different Ext Rep versions

```
<indicators>
    <repeated_blocks flag="false">
    </repeated_blocks>
    <resolvable flag="false">
    </resolvable>
    <affine_resolvable flag="false">
    </affine_resolvable>
    <equireplicate flag="true" r="3">
    </equireplicate>
    <constant_blocksize flag="true" k="3">
    </constant_blocksize>
    <t_design flag="true" maximum_t="2">
    </t_design>
    <connected flag="true" no_components="1">
    </connected>
    <pairwise_balanced flag="true" lambda="1">
    </pairwise_balanced>
    <variance_balanced flag="true">
    </variance_balanced>
    <efficiency_balanced flag="true">
    </efficiency_balanced>
    <cyclic flag="true">
    </cyclic>
    <one_rotational flag="false">
    </one_rotational>
</indicators>
```

(a) v2

```
"indicators" : {
  "repeated_blocks" : false,
  "resolvable" : false,
  "affine_resolvable" : false,
  "equireplicate" : {
    "r" : 3
  },
  "constant_blocksize" : {
    "k" : 3
  },
  "t_design" : {
    "maximum_t" : 2
  },
  "connected" : {
    "no_components" : 1
  },
  "pairwise_balanced" : {
    "lambda" : 1
  },
  "variance_balanced" : true,
  "efficiency_balanced" : true,
  "cyclic" : true,
  "one_rotational" : false
}
```

(b) v3

Figure 2.5: *indicators* in different Ext Rep versions

merged into a single entry. For example the indicator *t_design* can be *true* or $false$, in addition when it is true it has an associated value (the value of the maximum t for that design). Both the boolean value and the property maximum t have been merged into a single entry which can either be $false$ or the key-value pair $t_{maximum} : int$. Figure 2.5 displays this transformation.

Finally the "*cleaning*" process was systematically applied when applicable. In the Ext Rep $v2$ some sub-trees contained static values, which their presence was previously enforced. For many of such instances their presence was more aesthetic than useful. Such instances were safely removed without losing any information conveyed in the Ext Rep document. An example is the attribute *ordered* on the *blocks* of a design (as can be seen in Fig 2.6). This attribute was defined to always be true. In

```
<blocks ordered="true">
    <block><z>0</z><z>1</z><z>2</z></block>
    <block><z>0</z><z>3</z><z>4</z></block>
    <block><z>0</z><z>5</z><z>6</z></block>
    <block><z>1</z><z>3</z><z>5</z></block>
    <block><z>1</z><z>4</z><z>6</z></block>
    <block><z>2</z><z>3</z><z>6</z></block>
    <block><z>2</z><z>4</z><z>5</z></block>
</blocks>
```

(a) v2

```
"blocks" : [
  [0, 1, 2], [0, 3, 4],
  [0, 5, 6], [1, 3, 5],
  [1, 4, 6], [2, 3, 6],
  [2, 4, 5]
]
```

(b) v3

Figure 2.6: *blocks* in different Ext Rep versions

the new version, the importance of keeping the blocks ordered has been placed in the documentation and removed from each of the complying Ext Rep structures.

In addition to the above, there were a few more aesthetic changes made to the names of certain entries in Ext Rep $v3$ specification with respect to $v2$. Moreover, there were changes introduced to enhance the functionality of the Ext Rep from a combinatorial point of view, they are discussed in section 2.2.3.

2.2.3 Functionality Enhancement

In combinatorial block designs, a *Balanced Incomplete Block Design* would contain different values for the property λ depending on the level of $t; 0 <= t <= t_{maximum}$ that is being investigated. For version 3 of the Ext Rep it was desired to display the values of λ for each applicable value of t. In the previous Ext Rep, the *t_wise_balanced* subtree contains a set of *index flags*. These flags stated that a given design is a *t-design* for a stated value of t (see Fig 2.4a). The *index flags* have been transformed to allow the additional placement for the value of λ where the given design is a *t-design*. An example of this transformation can be seen in Figure 2.4.

Another fundamental addition made in the Ext Rep $v3$ was the specification of *invariants* of an Ext Rep document. This sub-structure was left for future consideration by the previous Ext Rep version. As implied by its name, it represents properties within a list of designs that are maintained throughout each design. Therefore the invariant sub-structure may be used to categorize the group of designs presented in the given list of designs, specifically when looking at the general parameters t, v, b, r, k, λ

that are invariant within the list.

2.3 Implementing Ext Rep v3

Moving from version to version of Ext Rep specification can be a complicated task, specifically when there are vast differences between the versions. Similar to most designed systems, a version bump of a partial version change (e.g. 2.0 to 2.1) denotes changes that keep compatibility to the previous version. Moreover, a version bump of a full version number denotes large changes that generally break compatibility to older versions as is the case with Ext Rep $v3$ [29]. There has been many changes to this version of the protocol and a system was created to make the transformation process as simple and intuitive as possible.

2.3.1 Schema Language

When discussing a data representation, a schema language is essential in order to explain how the data looks and how it may be formed. Although examples may help explain how the data should be presented, a formal schema is required to guide the users and serve as a reference when writing complying documents.

The Ext Rep $v2$ has been implemented in XML and therefore an XML compliant schema language was used to document the structure of complying documents. Due to the clear and simple design of the Relax NG [7] Schema language for XML, it has been chosen to describe the Ext Rep $v2$. The Ext Rep $v2$ schema has been published in [20]. In implementing the $v3$, there was an obstacle due to the lack of a clear and simple schema languages to describe the new protocol. For that reason I utilized a novel schema language, built as an E-BNF, to describe Ext Rep $v3$. The full Ext Rep $v3$ schema can be seen in Appendix A and in [29] where it is described in detail.

2.3.2 Conversion from v2

It is often hard to upgrade software or systems from a given version to one that breaks compatibility with the given version. However, developers of the new version often try to aid users of older versions in their move to the newer version to cater for as many users as possible. Software systems such as Python [50] decided to write scripts that

would automate the transformation of version 2 code to their new version 3 (*Py3k*) [17]. Other software systems such as *SqlAlchemy* [24] have produced documentation describing their changes required to transform old code into code that works with their new version 5 [1].

To complete the conversion of Ext Rep from $v2$ to $v3$ it has been decided to adopt a similar standpoint as the Python programming community. A software utility was developed to convert the XML $v2$ [21] into the JSON $v3$ [29]. This software utility is extremely important to be able to transform the current documents that have been previously produced in Ext Rep $v2$. Such documents include a collection of over 2.5 million designs that have been produced by the researchers at DesignTheory.org. Moreover the full collection of Ext Rep documents has been transformed into compliant Ext Rep $v3$ documents available at [3]. The conversion of the full set of Ext Rep files takes around 3.6 hours (215.6 minutes).

The conversion utility mentioned here may be found in my design database software package [42] under the *extrep/utils* directory.

2.3.3 Ext Rep v3 Parser

Many factors were considered when choosing JSON language to implement the Ext Rep $v3$ specification. Factors such as readability, size and simplicity, have been mentioned in the earlier sections of this chapter. Nevertheless, another decisive factor that lead to choosing JSON is the parsability of the JSON language. In addition, there are a large number of available software libraries implemented in various programming languages that are able to parse JSON [4]. Having such a wide variety, further encouraged the decision since users of combinatorial designs have the flexibility of choosing their preferred language and architecture to implement specific application that utilize these designs.

Although there are many applications written in the *Python* programming language to parse JSON, they all rely on the full JSON structure being loaded into memory prior to processing. This is not feasible when dealing with Ext Rep structure as many of the documents are extremely large in size. To solve this issue a few incremental parsers for JSON in other programming languages have been considered. The "Yet Another JSON Library" (YAJL) [41] library, implemented in *C*, has been

found most suitable for my application. YAJL is a very compact library that has multiple advantages including fast parsing and good usability.

To use YAJL in my Python based project, the YAJL code had to be encapsulated in a Python friendly wrapper. After considering multiple options, the *C-types* [37] module was chosen. The C-types module was added to the Python standard library as of version 2.5 [2]. After wrapping the YAJL Library with the correct code, the encapsulation allowed for the import of the *YAJL* library as if it was a Python library. Moreover, it allowed for an incremental parser in Python which did not exist prior. The *C-types* wrapper for YAJL 0.4 can be found in the software package accompanying this document (found under the *extrep* module of the *ddb* package).

Another required feature was the strict ordering of key value pairs while reading and transmitting JSON *objects* which are generally implemented as a hash map. Since hash maps do not guarantee ordering, an ordered hash map was used to fulfil the strict ordering requirement. Since all of the standard JSON parsers for python use a standard hash map to represent a JSON *object*, it was yet another advantage of implementing a custom parser built on the callback based YAJL library.

Similar to the other stages of my project, the utilization of open source software provided a mutual benefit for my project along with the open source community. Due to the use of the YAJL library, a couple of bug reports and patches were sent to the author of YAJL which contributed toward the release of version 0.4 of the library.

Chapter 3

Database of Combinatorial Designs

Combinatorial designs, as mentioned earlier, are very useful structures that are utilized in many fields (see 1.1). Therefore it is essential that users of such structures be able to locate a specific design on demand. Moreover, it has been mentioned that block designs (and their associated properties) are usually computationally intensive to generate (see introduction of chapter 2). Therefore it is important to store these designs and their relevant properties once found. It is also important to be able to organize the collection in such a way as to allow for searching. For these reasons, a database of combinatorial designs is required.

It is very strange that no effort has been placed prior for such a database as it is quite essential for the users. My project aims to fill the gap. The following sections describe the decisions and implementations that took place in order to materialize the searchable database of combinatorial block designs.

3.1 Database Engine

When it comes to database systems there are many proprietary software packages and many more that are freely available (open source [28]). These packages tend to lie in one of two collections, either *Hierarchical* or *Relational* database management systems. The following subsections describes the different options considered for use as the DB Engine. Only open source database engines have been considered, due to the reduced cost and faster speed of updates witnessed with open source software in general.

3.1.1 Hierarchical DBMS

Ext Rep structures, formed by organizing combinatorial designs and their properties, are in essence hierarchical structures. It seemed necessary that an attempt be made

to place the designs into a hierarchical database. *HDF5* [35] was considered due to the availability of a Python interface.

The HDF5 database is a very powerful hierarchical databases system used by many researchers in the scientific realm to store their results. Similar to all hierarchical systems, it is based on tables. In object oriented terms, an HDF5 table represents a set of objects. In the ideal setup the combinatorial designs database would consist of a single hierarchical table that host all the combinatorial designs. Each combinatorial design would be represented as an object, where some of the objects may vary in the amount of information withheld inside it. The variance is due to the optional subtrees of an Ext Rep design that may have been left out for particular designs [29].

HDF5, and hierarchical systems in general, allow for storage and access of a large amounts of data. However, what hierarchical systems miss out, from the relational perspective, is a quick (computationally efficient) method of being able to perform relations between tables and objects (rows).

The reasons for choosing HDF5 over other hierarchical DBMS, is mainly due to the availability of a Python interface to it. The Python package PyTables [11], aims to link HDF5 (written in C) with Python and the Python scientific libraries (such as NumPy [44]). This interface provides all of the functionality available by HDF5 except for allowing for variable length fields within tables. This is one of the most important features needed by the application being developed for my project due to the possible variation in size of the combinatorial design structure. For that reason some research is needed in order to find the most suitable way of storing the variable sized content within HDF5.

3.1.2 Relational DBMS

Relational databases have been a corner stone in many projects when dealing with data. Since the introduction of the relational data model in the early 1970s by E. F. Codd, it has become more popular and more sophisticated. Today there is a large market for and a large number of providers of *Relational Database Management Systems*. Companies such as Oracle are leading the food chain and have been deploying many systems around the world that are pivoted around RDBM Systems.

For this project, only open source software systems were considered for the same

reasons mentioned earlier. Since there is a large number of open source Relational DBMS available, I only considered mature and stable RDBMS. These restrictions limited the available choices to *SQLite*, *MySQL*, and *PostgreSQL*.

The storage structure for blocks was the initial concern as I aimed to keep the design for storing blocks as simple as possible. For this reason PostgreSQL [12] was chosen as the RDBMS of choice, as it contains an Array datatype that can efficiently store multi-dimensional arrays of a single datatype. However, only the highest dimension may be of variable size, all sub dimensions have to be of constant size.

Some designs may not have a constant block size for their blocks and, for that reason, an approach had to be developed to manipulate the structure such that it can be stored using the Array datatype. Each of the blocks was padded with *NULL*s such that it became the same size as the largest block of the design. This transformation is being done while inserting design blocks into the database, and is being reversed as the blocks are being read from the database.

Representing a design as a Python object was an advantage of HDF5 that I wanted while using PostgreSQL. Manipulating objects is much simpler than creating a large number of SQL templates to manipulate the database. For that reason an Object Relational Mapper (*ORM*) was utilized to wrap the SQL tables into Classes and rows into instances.

There were a few options for ORMs available for Python, but SqlAlchemy [24] was the final choice. Like many ORMs, it makes working with RDBMS simpler. The mapper contains information about all foreign keys and related tables, thus allowing the full Ext Rep design to be abstracted as a single object. This feature was a great advantage for the development of this application, due to the variation in size of each of the Mathematical Objects (Ext Rep block designs) being processed. Each of the optional sub-trees of an Ext Rep design was separated into different tables. However, at the application level, the complete Ext Rep structure is abstracted as a single tree (mapped to an Ext Rep Design object).

3.1.3 Choosing between HDF5 and PostgreSQL

Prototypes were created for both HDF5 and PostgreSQL for initial experimentation. The general target of the prototypes was to create a collection of Ext Rep designs using only the root level properties (design id, v, b, blocks) and the indicators subtree. Due to the variable nature of the blocks, and the unavailability of a simple method of storing these blocks in HDF5, three prototypes were built. Each of the prototypes differed only in the method of storing the blocks. For PostgreSQL a single prototype was implemented.

Two types of experimental runs took place on each of the prototypes, reading and writing. There were 5 iterations of each of the experimental runs where the outliers were removed and the middle 3 runs were averaged. The experiment consisted of a write to the database followed by a read. The first run wrote 100 records, while the last one wrote 10,000 records to the database, using increments of 100 between each of the experimental runs. The time elapsed for both writing and reading the databases for each of the experimental runs was recorded and was used to compare HDF5 and PostgreSQL performance.

The following sub-subsections describe the specifics of each of the experiments and the results.

HDF5 Experiments

Combinatorial Design blocks are in nature variable in size with respect to number of blocks and the size of each block. To decide weather HDF5 was suitable to store the blocks of a design three experiments were set-up. Each of these initial experiments only dealt with storing the root node of each design tree along with only the indicators subtree, rather than the full tree with all the possible variations. As mentioned earlier, PyTables only allows fields of a predetermined fixed length, which required some design of how the blocks are to be stored. PyTables and HDF5 allow an *Array* structure, alongside the Table structure. There are multiple types of arrays that PyTables provides each of which formed the basis of the three experiments.

In each of the experiments the blocks were stored using different HDF5 arrays implementations such that the best internal representation can be chosen. All the implementations stored the blocks in an array structure, external to the designs table,

while storing the array index in the designs table. Therefore to find the blocks of a particular design, the index would be taken from the design object and used to lookup the external array.

The first (PyTables Experiment 1) stored the design in an HDF5 Variable Length Array (*VLArray*). The list of blocks were serialized (*pickled*) using the Python *Pickle* library [51] and directly stored into the array.

The second variation stores the design blocks into an HDF5 Enlargeable Array (EArray). The blocks were similarly *pickled* and stored character by character into the large array. In the designs table the begin and end offset for the blocks were stored. Therefore, all the blocks for all the designs were stored in a single array and each design had the begin and end offsets for its particular set of blocks. According to a conversation between myself and the main developer of PyTables, this approach seems to be close to how the VLArray structure is implemented internally [15].

Similar to the first design, for the last prototype each design had a record within a large 2-dimensional VLArray, with an index inserted into the particular record in the designs table. To store all the blocks into a VLArray, all the points for each block of a given design were placed within a single list (i.e. the boundaries between the blocks were removed). A second VLArray was created to carry the offsets of the blocks (i.e. the boundaries) of each design, thus allowing the regeneration of the block boundaries.

The final prototype was considered to be the most suitable due to the fact that the data was stored in its numerical form rather than a serialized form. Following the general experiment guide (section 3.1.3) the various PyTables experiments were run. Figure 3.1 shows the results of these experiments.

The experiments show a slight variation between the different underlining implementation. However, due non-binary nature of the third implementation, it is the most reasonable for storing *design blocks* under PyTables/HDF5.

PostgreSQL Experiment

Using the above concepts, a simple experiment was designed to do an initial comparison with the PyTables prototypes discussed earlier. This comparison was used to complete the decision of which DBMS to use for this project. Figure 3.2 shows the

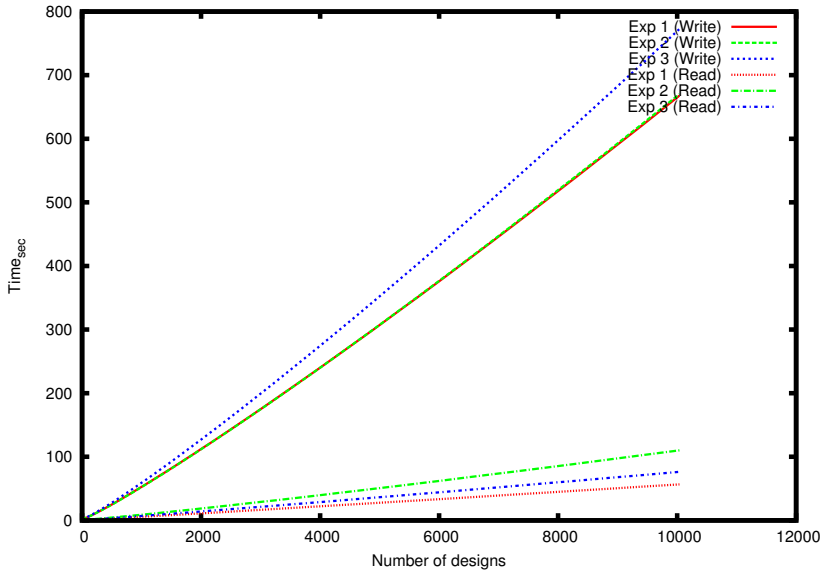


Figure 3.1: Results of the PyTables experiments

results of the similar experiment (see experimental guidelines in section 3.1.3).

The results, as expected, show that writes to the SQL database are slow, compared to reads. However, since the application is a read-intensive application where the database is built first, then the users of the database would browse its contents, the read speeds to the database are the main concern with respect to performance.

3.1.4 Prevalent DBMS

After preliminary tests, and practical assessment (from a development perspective) it was clear that with the current state of both database management systems that PostgreSQL is the right choice. The main factors that lead to this decision is due to the faster read speed of PostgreSQL and the simplification of how blocks are stored internally. Moreover, it was apparent that relational queries would be faster with the RDBMS than with HDF5. Most of the users would be reading from the database and thus it was necessary to consider that carefully.

Alongside the relational database, the use an object relational mapper (ORM) would bring the best of both worlds. ORM would allow the full Ext Rep design to be viewed as a single object, while allowing the use of the powerful relational nature of SQL to query the database.

It has not been decided that the Hierarchical Database system, or PyTables should

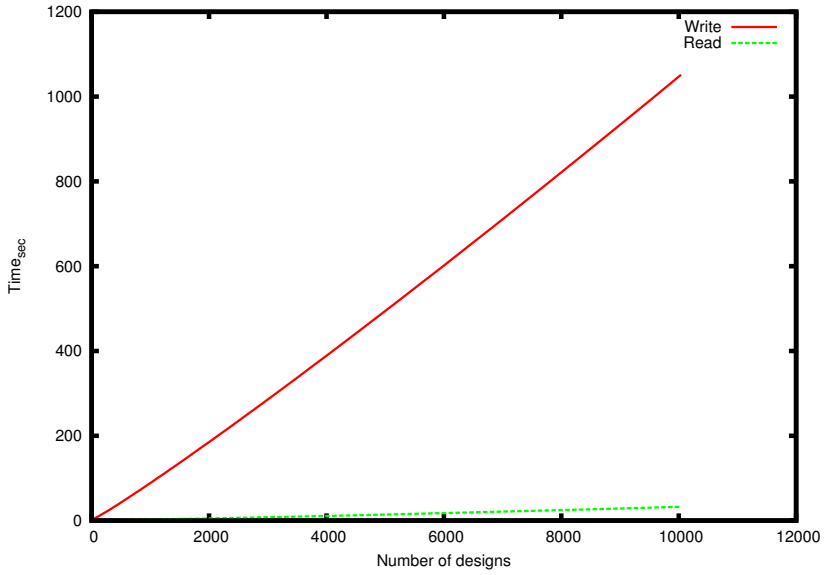


Figure 3.2: Results of the PostgreSQL Experiment

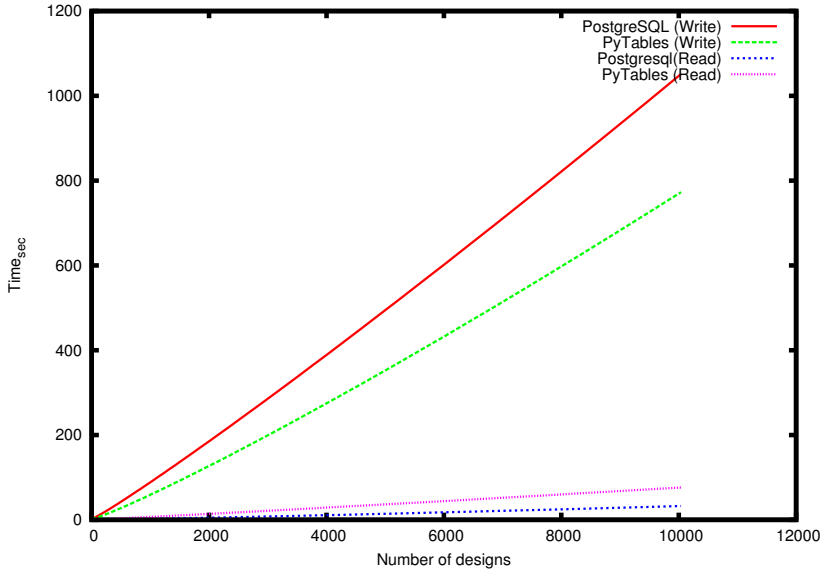


Figure 3.3: Combined Preliminary Results of the DBMS Experiments

not be reconsidered in the future. On the contrary, the belief is that with future releases and added functionality, the performance of hierarchichal databases in general, and HDF5 in particular, when hosting Ext Rep designs may exceed the performance of RDBM Systems (see Section 7.2).

3.2 Designing the DB

In chapter 2 the Ext Rep structure has been described as a complicated tree. Moreover, it is a tree with many optional sub-trees, such that each object conforming to the Ext Rep specification may vary greatly in what it contains. Some objects may contain all the sub-trees, while some may not. There a few reasons for such variation, the simplest being that the person who generated a particular design neglected to provide all the statistical and combinatorial information, and simply just provided the blocks. However, the more practical and realistic reason is that many combinatorial designs do not have some of the specific properties that others may. For example, the design *resolutions*[1] are only available when a design is *resolvable*[2]. Since the majority of designs are not *resolvable*, they would not contain the *resolution* sub trees, while a few designs may have one or more than one *resolution*. Therefore the database had to be designed to allow for such variation between the structure.

After analyzing the Ext Rep $v3$ schema [29], it was decided that each sub-tree would be placed in a separate table linking back to its parent node. However, some of the sub trees contain segments that can recur a variable number of times. For example the *Automorphism Group* of a design is represented in its own table, but it may have a variable number of *generators*[3]. To insert the *Automorphism Group* subtree into the database, a second table was created to host the *generators*. The *generator* table has a relationship of one to many with the parent table (*Automorphism Group*).

From a programming perspective, the database structure (as described above) would be viewed using the ORM as the original tree structure, thus allowing the generation of the original Ext Rep representation with ease and simplicity. Moreover,

[1] A resolution is a partition on the blocks of a design such that a point may only appear in a single partition

[2] A design that has one or more *resolutions*

[3] Automorphism Group generators are functions that act on the points of a given design to permute them and create isomorphic designs (equivalent designs).

the Entity Relationship (ER) Diagram for this database would be similar to displaying the Ext Rep structure as a Tree. This was important to maintain the simplicity of the system. The Entity Relationship Diagram for this database can be seen in Appendix B of this document.

Using the available tools, applications were written to be able to create the database and generate the objects that would map back to the database tables. This allowed for easy manipulation, tweaking and fixing of the schema as the system progressed. Similar to all areas of the system, the code written here was structured to allow for easy manipulation and modularity. Such that any segment may be replaced at any point within the development. Similarly the object relational mappers were loosely coupled to the system, and the ORM system itself allowed for configuring and using a different RDBMS back-end easily.

3.3 Database Population

Combinatorial and Statistical designs are in general very hard to find. Once they are found it is a good idea to store then, for quick access in the future. This was the driving force behind my project. Section 1.2.2 mentioned that DesignTheory.org had generated and stored a collection of over 2.5M designs. Section 2.3.2 describes the conversion utilities written to convert files written in the previous Ext Rep version. Following the conversion of the Ext Rep $v2$ documents generated by DesignTheory.org to $v3$, the designs were uploaded to the PostgreSQL *combinatorial designs* database.

3.3.1 Storage

The upload process involved the integration of the Ext Rep parser described in section 2.3.3. The parser callbacks were written such that each design was transformed into an ORM instance (database mapped object). These objects were then saved in the database. After the completion of this stage, the database was populated with the full set of designs from the DesignTheory.org collection. This collection did not guarantee that all of its designs are pairwise non-isomorphic. Therefore filtering out the isomorphic designs was necessary.

The storage of $2,559,638$ designs (full Ext Rep trees) into the database takes a total time of 63.8 hours, which is about 11.15 Ext Rep objects per second.

3.3.2 Isomorphic Rejection

Two block designs are considered isomorphic if there exists a permutation of the point set which transforms the blocks of the first design into the blocks of the second design. Basic analysis of the blocks of a design do not indicate that two particular designs are non-isomorphic, since a permutation on the point set may generate a design with a new set of blocks.

The isomorphism testing of block designs can be easily reduced to the isomorphism testing of graphs. The graph isomorphism problem belongs to NP but is not known (neither is believed) to be NP-Complete. NAUTY [9] is the best available software package for graph isomorphism testing. For this reason the *blockdesign* [31] Python package, used to test isomorphism, is based on NAUTY.

It is required to keep a single copy of a design due to the fact that isomorphic designs can be generated from a given design. Moreover, the Ext Rep specification allows for defining the automorphism group of a design as a "generator" function. Thus allowing the users to generate all the isomorphic design from a given design. Generating isomorphic designs may be useful in specific applications such as cryptography [43].

The *blockdesign* Python package was used to compute a certificate for each of the designs. All designs within the same isomorphism class have the same certificate. The computed certificate is quite large and hence was hashed into a standard sized integer (32 bits). This hash value was then stored in the database for later reference.

Two designs can only be isomorphic if they have the same parameters (number of blocks, points and block sizes when applicable). The isomorphic rejection process was designed to look at all designs that have the same parameters and hash value. Designs matching that query would further trigger the certificate generating routine to check if the designs are isomorphic (rather than just having a collided hash value, due to the hashing function used).

The isomorphic rejection process reduced the original input set of $2,559,638$ to $2,556,583$, rejecting $3,055$ designs. This pruning completes in under 57.0 hours.

3.3.3 Design Classification

Design may be classified into categories using various methods and techniques. The classification system used to categorize the original set of block designs (DesignTheory.org collection, see section 1.2.2), is done using the $t, v, b, r, k, \lambda, q, x$. Parameters v (number of points) and b are the only ones that must be available for any block design [29].

For block designs the categories created using (suitable) subsets from the general parameter set t, v, b, r, k, λ (described in section 1.1) are arranged hierarchically [13], such that the set with the least number of designs is at the bottom. This concept may be represented as an inverted pyramid (see fig 3.4), such that the bottom (tip of the pyramid) has the least number of designs. Traversing upwards the parameter set becomes less strict including more designs from the universal set of designs.

Once any level in this pyramid is known to be complete, meaning all non-isomorphic designs from the universal set that are categorized (for the given parameter set) have been discovered, all of the levels lower than it (with a stricter version of the parameter set) must also be complete. This is because designs in a stricter category, are a strict subset of the set of designs with a looser parameter set.

As mentioned earlier the original collection of block designs was divided into categories (using parameters $t, v, b, r, k, \lambda, q, x$). Moreover, the users of combinatorial designs are very interested in browsing a certain category of designs. Therefore it was essential to have the same classification schema available from my system. Note that lower categories are more restrictive (more specific) than higher categories in the pyramid diagram. Figure 3.4, displays an example of the $v = 6, b = 40, k = 3$ general category going down to the $t = 3, v = 6, b = 40, r = 20, k = 3, \lambda = 2$ specific category.

The numbers next to each category in figure 3.4 represent the number of designs available in the database. The label complete is attached to the category that is known to be complete (i.e. we have all possible designs for that particular category in the database). All categories below (stricter) than the one labelled complete are also complete (by the definition mentioned earlier), while the ones above it may or may not be complete.

Knowledge that a category is complete (or not complete), meaning all designs for the given parameter set have not been found, requires generating all the pairwise

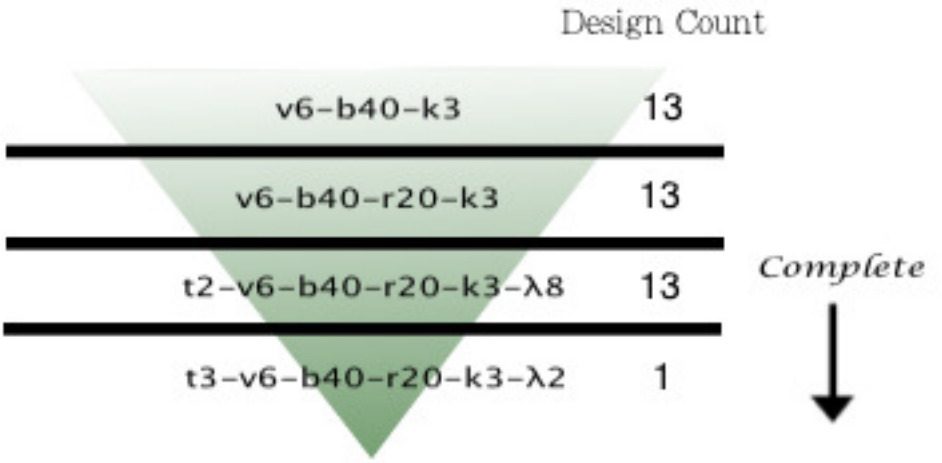


Figure 3.4: Levels of a v6-b40-k3 design category from the Design DB. For the last two levels it is known that all designs are discovered

non isomorphic designs for the given parameter set (category). This is due to the combinatorial nature of designs and how they are generated. The only way to prove that a given set of designs is not complete, is by presenting a design that belongs to the category which was not placed in the set. Since the system will be displaying all designs it has knowledge of, design categories will either be complete, or in an unknown state. Thus the only situation where designs would be in an "incomplete" set is while yielding query results.

To produce a *Summary Table* of the categories available in the database a few obstacles were overcome. Due to the method used to generate the designs from the DesignTheory.org collection (see section 1.2.2), which is simply described as a search in a large combinatorial space that is narrowed down based on the initial parameter set, some runs of the application took a very long time to complete and were killed before they could search the entire space. However, when the application that generates those designs was killed, the designs that were found up to that point are saved. For that reason some designs were found using the specific parameter set, yet were not when using the generalized parameter set (due to it taking too long).

To solve the mentioned issue, designs were linked to all categories which they belong to. This lead to the generation of categories that were not considered in the

original collection. These new categories needed to be checked whether they contain all designs for a given parameter set. A category can be marked as complete if and only if a more general category was marked complete in the original collection.

Adhering to the "Keep it Sweet & Simple" (KISS) principle [5], The *Summary Table* was designed as a table with its columns being the parameters making up the categories, along with count and completeness information. A many to many relationship was setup with between the summary table and the designs, such that a design may belong to multiple categories, and a category contains multiple designs. The completeness information, was propagated from the old collection by checking whether the collection was previously marked complete in the original collection. Moreover, any new ones that were theoretically complete (due to parent level completeness) were also marked complete. Finally, after all designs were processed, aggregate queries were run to determine how many designs are in each summary entry (class), and these numbers were stored in the summary table for a quick reference.

The main *Summary Table* calculation process, occurred during the design upload phase and thus the timing for this stage is enclosed within the average time displayed in section 3.3.1. As for the count aggregate phase, it consumes less than 0.5 hours to process $1,757$ categories.

3.4 Query Engine

Having such a large database makes querying the database complicated. Although SQL is a very powerful yet simple language, building queries to search all areas of the database is arguably more complicated a task than building the database. A query interface was required to modularize the task of querying this large database.

3.4.1 Query Protocol

To allow querying of all areas of the database a language or protocol to simplify this task was required. This protocol would be used by the web interface designed for the general users of the system, and in the future would be directly used by power users to communicate with the database using an application programming interface (API).

The aim was to make a simple query protocol, while allowing the user to query for any design in the system via any of its parameters. Moreover, the Ext Rep specification, includes notes about further research into having a query language written in Ext Rep [21][29], and thus the query protocol discussed in this section can be considered the first prototype for the desired query language.

The standard Ext Rep $v3$ [29] structure implemented in JSON, was modified to act as a query language (or protocol) to the database system. It was first modified such that the leaf nodes of the trees would contain an relational conditions rather than literal values. These conditions are represented in JSON as a two element list. The first being a conditional operator drawn from the set $=, !=, <=, <, >, >=$, and the second being the right hand operand of the conditional operator. The left hand operand is the parent node of this list structure which represents the database column to be queried. Figure 3.5a, displays a query, designed using this method, to search for designs with canonical variance > 0.22.

The query language design described above was the first iteration, which only allowed for one set of relations and thus was not practical enough. It was further extended to allow more flexibility when searching for a result set. The second variation allowed for having multiple conditions for each leaf node. These conditions are joined using the implicit *&* ("and") operation. Figure 3.5b, shows how you would have searched for designs with $0.22 <$ canonical variance $<= 0.34$. To use the || ("or") operation, multiple queries would be issued to the system and the union of each result set would be the full query result.

The final obstacle for the design of the query language was the translation between this language and the SQL query language which is understood by the relational database system. Modules were built with the aid of the object relational mapper to traverse a query (which is an Ext Rep tree like structure), and convert each level of the tree and apply the correct join condition needed to fulfil the query.

The advantage of this query language is that it is still JSON, thus allowing the reuse of the available parsing code for JSON. Moreover, it adheres to the structure specified in the Ext Rep document, as was advised in [21][29], and allowing for a common and familiar method of communication to and from the database. Users would be able to issue Ext Rep like queries and be able to get their results back as

```
{
  "designs" : [{
    "statistical_properties"  :   {
      "canonical_variances"   :   {
        "value" :   {
          "canonical_variance": [">", 0.22]
        }
      }
    }
  }]
}
```

(a) Iter 1

```
{
  "designs" : [{
    "statistical_properties"  :   {
      "canonical_variances"   :   {
        "value" :   {
          "canonical_variance": [
            ["<", 0.34], [">", 0.22]
          ]
        }
      }
    }
  }]
}
```

(b) Final

Figure 3.5: Snippets Showing the Experimental Ext Rep Query Language

an Ext Rep document (set of designs).

3.4.2 DB User Management

Since many queries return a large number of records, my system has to be able to handle and manage the user queries. Users would be able to use my system to query the database, followed by calls that would allow them to browse their queried result set.

To reduce unnecessary load on the database system, the standard cursor was extended such a subset of the results were cached. An actual database query request on the cursor would occur only if the user requested a query result that was not already in the cache. This allowed for traversing both forward and backward within the result set. Following that a session management system was implemented such that users would issue a system call that would return a DB cursor id and any subsequent queries would include the issued cursor id. Therefore subsequent queries, and browsing requests, would go through the issued cursor.

Chapter 4

Web Interface

In general, web applications require no special configurations on users PCs and many people today are familiar with the internet and are acquainted with web browsers.

Deploying an update to a system that utilizes a web interface is arguably much easier than having to rollout updates to each of the users. The application is updated on the server side and the users instantly can view the updated system. This is done with very limited downtime thus allowing users to be able to access the application 24 hours a day, seven days a week.

Viruses and malicious code distribution are less likely to happen with web applications. Many users are paranoid about installing applications on their machines. Web applications generally avoid this issue, thus allowing for a larger potential user base.

Data centralization, which is an advantage of all network based applications, is an important advantage. My system, requires this feature, as it will not be feasible to have each user deploy the combinatorial design database on their local machines.

For the above reasons it was decided that a web interface would best suit my system. The following sections describe the choices that were made and the applications used to develop the web interface to the combinatorial design database.

4.1 Python Web Frameworks

Python is a powerful language that has been fairly used in the web application realm in the past. Today more and more Python based web frameworks are being developed, all of which aim to make development easier and faster. With powerful Model-View-Controller (MVC) architectures they allow for decoupled systems that are easy to maintain.

The MVC design pattern splits an application into seperate layers labelled *model*, *view* and *controller*. The *model* layer is responsible for the data access, while the *view*

layer is responsible for the user interface. In the middle comes the *controller* which is responsible for the business logic of the application [6].

The availability of a wide range of web frameworks for Python did not make the search an easy one. These frameworks have a large range of features that make development easier, increase security and allow for the inclusion of various open source and customized middle ware to the system.

Python web frameworks are divided into two schools, *Glue* and *Full-stack* frameworks. Glue frameworks, as the name implies, glue together various available open source components (such as a template renderer, object relational mapper, development http server) in order to establish the framework. Examples of such frameworks are *TurboGears* and *Pylons*. Full stack frameworks on the other hand are custom built from the ground up, they do not use any other projects as components within their project. Examples of such frameworks are *Web2Py*, *Quixote*, *Zope*, and *Django*.

Although frameworks that use a variety of minor components to make up the framework (i.e. glue frameworks) arguably have a faster releases and more combined resources, frameworks such as Django (full-stack) are moving very fast. Nevertheless, many frameworks including Django, are moving away from the strict full stack methodology and allow for the swapping of the standard components. In this section a few frameworks that have been looked at are described. Moreover, the chosen framework is described in detail along with the reason for choosing it.

4.1.1 Quixote

Quixote is a very simple, yet powerful full-stack web framework. Like all web frameworks to be discussed here, Quixote contains a standalone http server to directly serve the application on the web. Quixote also features session management, a simple template language known as *PTL*, and allows for the integration of any template language of choice.

Quixote allows for integration with various database interfaces, and allows to be run under the Simple Common Gateway Interface (SCGI). The advantage of this framework is that it has a very small code base and allows for easy auditing and bullet-proofing.

4.1.2 TurboGears

TurboGears is an example of a glue framework that utilizes many of the popular web libraries and components. These components include *MochiKit*, *Kid*, *CherryPy*, *SQLObject*, *ElementTree*, *FormEncode*, *Nose*, and *json-py*.

The libraries utilized here are powerful and popular, yet TurboGears seems to combine them in an awkward way. The modules have been modified (i.e. hacked) to integrate with the TurboGears design. This is mainly why it has been labelled as an "Awkward-Glue" framework by the python community [16].

4.1.3 Django

Django is a Full-Stack framework, with a wide community base and a large numbers of developments. Django offers a steady release schedule due to the fast paced development on the project. It has all the features that TurboGears (Section 4.1.2) offers, yet all the components have been specifically built for the Django project. The project has recently been moving toward allowing the ability to integrate external libraries with the framework. They have started with allowing users to easily incorporate *SqlAlchemy* instead of their standard ORM.

4.1.4 Pylons

Pylons is built as a special configuration (grouping of packages) using the Python *Paste* project. Paste allows the integration of various components and libraries into a framework. The standard setup includes many popular components, however, also allowing users to easily swap in other components. This has been achieved through the use of *Paste* to couple the components, rather than the code modification approach that TurboGears utilized. It is therefore labelled as a "Decoupled-Glue" framework [16]. Moreover, Pylons allows the use of *Flup* package which is a contains a fully compliant HTTP 1.1 web server. As mentioned, *Pylons* allows for any available web library to be utilized, the following set of components were chosen for this project:

SQLAlchemy

Powerful, flexible and clean Object Relational Manager (ORM), that is DB-API

compliant. Allows to connect to various database engines including PostgreSQL, Oracle, MySql, Firebird, Sqlite, and other RDBMS engines.

Mako

Pythonic, lightweight and fast template system. Uses the Python syntax in the template language, which allows for a powerful and easy to use system.

Prototype

JavaScript framework, that has gained popularity, in the web development community. Contains a very easy to use AJAX library, and support for class driven applications.

Routes

URL Router that allows for the configurable mapping between the applications code base and URL templates. *Routes* is a re-implementation of the Rails (Ruby) routes system.

Pylons is a very powerful framework that is cleanly designed such that one may hook in their code with ease. The sophisticate deploy methods is also one of its great advantages, as it allows for all types of deploy methods.

Due to the flexibility of Pylons through allowing the choice of any sub component, it was chosen as the framework for this project. Although such a framework has a higher learning curve to start, due to having to learn each of the various components separately while also learning how they integrate, it was apparent that the flexibility offered by the decoupled nature of pylons would be helpful to integrate the various components of the project, such as the custom JSON parser and query engine.

4.2 Query Interface

An Ext Rep structure is one that is a large tree structure with many branches and leafs (Chapter 2). Moreover, the requirement to query the collection using criteria in any subtree complicates how the interface would be designed to allow for such flexibility. A query method was required that would also work well with the query language (Section 3.4.1).

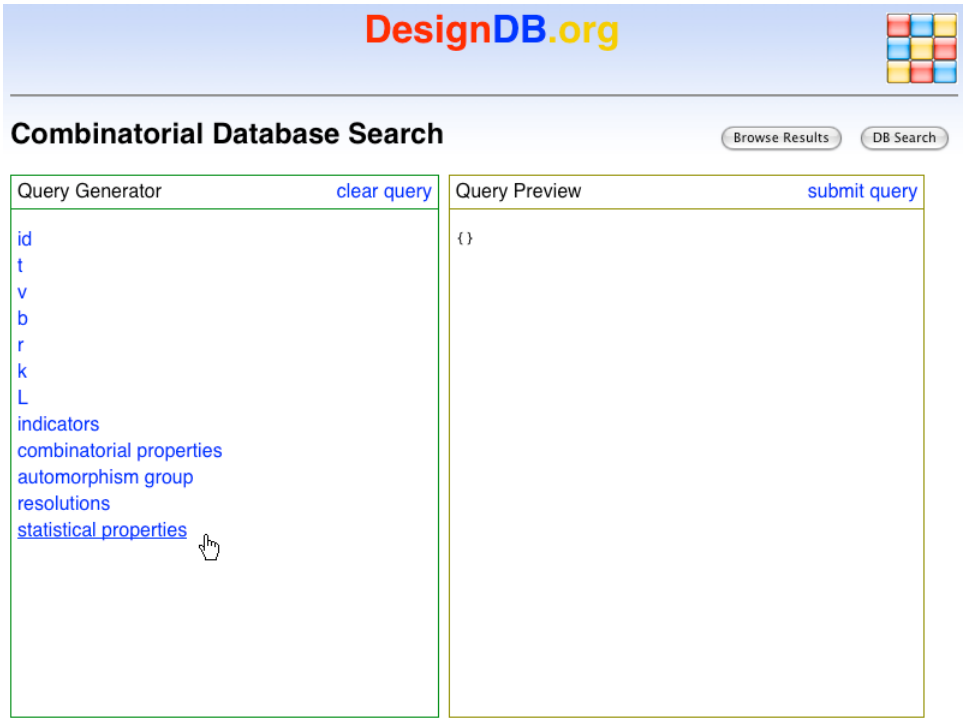


Figure 4.1: [Adding Canonical Variance] Step 1_a: Locating $CanonicalVariance$

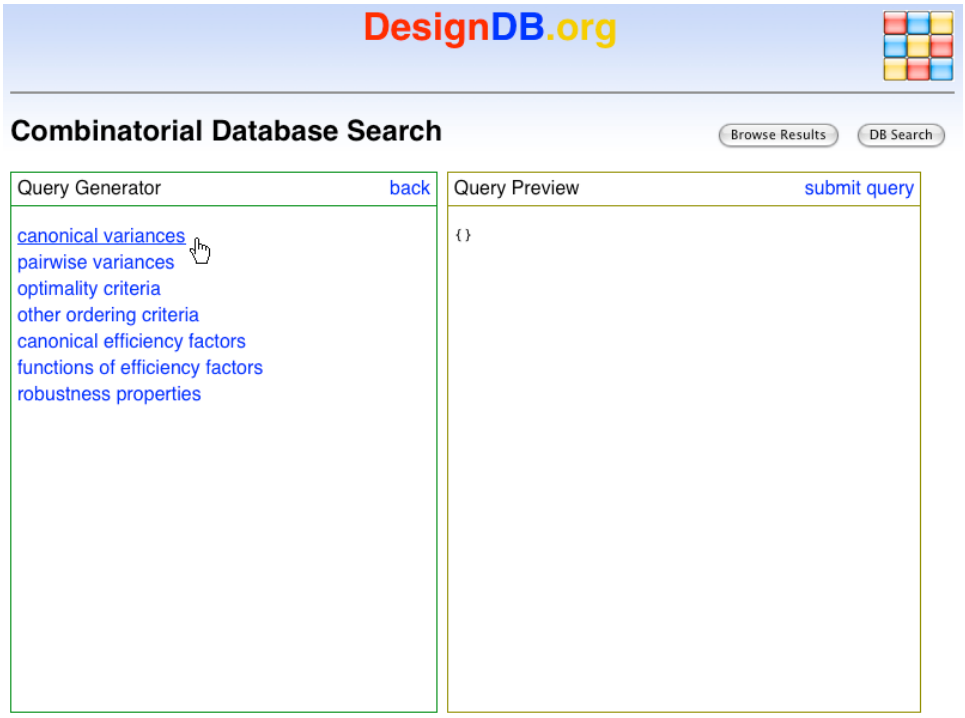


Figure 4.2: [Adding Canonical Variance] Step 1_b: $StatisticalProperties$

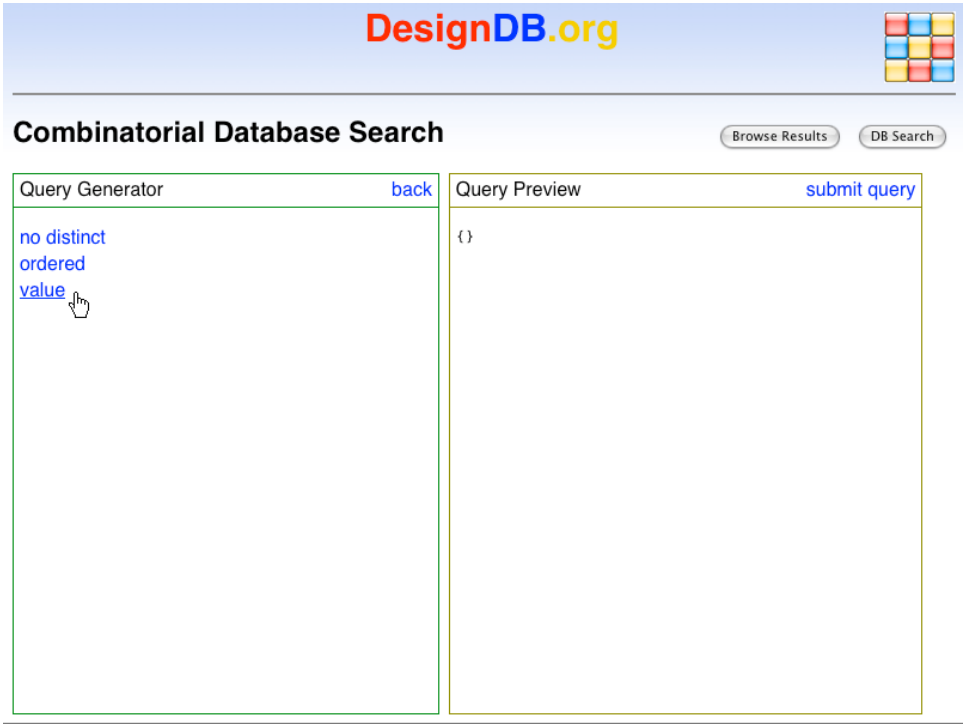


Figure 4.3: [Adding Canonical Variance] Step 1_c: $CanonicalVariances$

Figure 4.4: [Adding Canonical Variance] Step 1_d: $CanonicalVariances \rightarrow Value$

The solution adopted aiming to make query generation user friendly, was to implement a JavaScript based application to help generate queries. The mini application grabs the query schema from the web server using an asynchronous JavaScript call (AJAX). Parsing the result of this call, it proceeds by showing the top level subtrees in click-able link form. Clicking on any "branch", the next level subtrees are displayed. Following the tree in that manner to reach the leaf nodes, the user would be able to enter the relational condition for a leaf node, and proceed to add that field to the query.

The first step in generating a query (Figures 4.1, 4.2, 4.3, 4.4) is to select the entries at the top levels of the Ext Rep tree that lead to the desired leaf node. Following that the user enters the desired condition (Figure 4.5), and clicks the *Add* button, thus leading to the condition being added to the query. Clicking *Add*, immediately takes the user back to the root of the Ext Rep query system and allows the user to preview the generated query (Figure 4.6). In figures 4.1 and 4.6 it is visible that the parameters t, r, k, L are available at the root. These parameters are not in the root of the Ext Rep tree, however, they were promoted to the root of the tree due to their frequent use.

Alongside the "Query Generator", the user sees the "Query Preview", which shows the user the JSON query that will be sent to the database. Moreover, modifying the parameters for a given leaf node for that query is as simple as going through and re-adding that leaf node's parameters.

Utilizing this approach results in displaying a brief amount of options at each tree level, and clicking down to the deepest leaf in the Ext Rep tree, would require 5 clicks. This approach is less overwhelming to the user than displaying all the possible query parameters at once. On the other hand, the devised query method would only be beneficial to users that are familiar with the Ext Rep structure. However, requiring that the users be familiar with the Ext Rep structure, is not a blatant requirement, since the Ext Rep structure was designed to be intuitive to the users, and logical in its structure [14]. Figures 4.1, 4.2, 4.3, 4.4, 4.5, and 4.6 show an example utilization to add the condition "$0.22 <$ canonical variance < 0.34" to the query.

Another approach in searching the current database, is available on the main page of the web application (Figure 4.7). The "Summary Table" contains links to

Figure 4.5: [Adding Canonical Variance] Step 2: Define & Add Canonical Variance to Query

Figure 4.6: [Adding Canonical Variance] Viewing the Updated Query Preview

Figure 4.7: Searching using the basic design parameters

the various design categories in the database. Users are able to view the designs in each category by clicking those links. Moreover, users would be able to generate their own category, based on a mini search, using the t, v, b, r, k, λ parameters of a set of designs.

4.3 Interface With Query Engine

Pylons allows for a multi-threaded web application. However, with multi-threaded applications come some thread safety concerns. Subsequent requests may be handled by different threads, and therefore relying on thread-local data cannot be done. Nevertheless, this is the case with web development in general.

Many databases have restrictions regarding this issue as well. Connections are generally not allowed to be used by different application threads. However, using PostgreSQL and SqlAlchemy integrated with Pylons allows for simplicity when dealing with the issues related to multi-threading, as they are all handled implicitly.

The development of the Query Engine (Section 3.4) allowed for simple "black-boxed" utilization. The session management feature, implemented in the Query Engine, allows for users to issue queries and return to browsing them, as long as they keep their session with the web application.

Due to the Query Engine design, controllers would easily be able to dispatch user queries to the model. Similarly, controllers would be quickly built to generate user results, sending them to the view, thus maintaining the decoupling promoted by the MVC architecture paradigm (see Section 4.1).

4.4 Displaying Designs on the Web

The task of displaying combinatorial and statistical designs on the web is a complicated task. The task is similar in complexity to displaying the query interface to the database on the web. The reason is that both tasks deal with the large Ext Rep structure.

Initial experiments of "prettifying" the designs using HTML constructs were rather discouraging. One of these attempted approaches was to maintain the nested structure of the Ext Rep, however, enclosing each segment in a bounding box (similar to the HTML *fieldset* tag), with a label to denote its name. This approach quickly transformed the new Ext Rep $v3$ design to something hideous, huge and unreadable.

The initial "prettifying" attempt led me to take the route implied in [14] and implemented in [21] [29], to display designs as Ext Rep structures. The main driving force behind this approach (and a major driving force behind the move to Ext Rep v3, see Chapter 2) is maintaining a level of readability.

Displaying designs as Ext Rep $v3$ compliant structures has multiple advantages, the most important after readability being that JSON happens to be valid syntax for multiple programming languages (Section 2.2). Allowing power users to copy the sections they seek from the Ext Rep document directly from the web page into an interactive shell, or programming environment.

Chapter 5

System Deployment

A Unix based operating system was chosen to host this project due to the various advantages Unix has over other operating systems. Moreover, this choice conforms to the decision of utilizing only open source software for this project.

The project deployment was done such that implemented software would not interfere with the rest of the system, allowing for a clean installation. This chapter discusses the various deployment techniques applied in my project.

5.1 Installing the package

Abiding by the lazy programmer paradigm [39], it was not desirable to issue large sets of commands in order to install and configure the different segments of the system. To avoid issuing many commands to install the system, Makefiles were implemented in every subdirectory of the project that required a Makefile. The Makefile in the root directory of the project is responsible for calling the other Makefiles and installing the full project.

Since the project deals with a long running process that interacts with users via the web, a special system user was created under which the application would run. The special user has very limited privileges and thus the risk from foreign attacks to the host machine is reduced. Therefore even in the unlikely event of a security hole in the implemented application, the security of the Unix user privileges system, and database privileges, would secure the host system.

Since my project is a Python project, the code has been designed in a modular, object oriented fashion. Moreover, the code has been packaged, to fit nicely with Python *SetupTools* library such that the code base would be easily accessible as a Python module.

The installation of the code as a Python package also allows for the external use of any of the sub-packages within the system, such as the JSON parser. Python scripts

would be able to import the YAJL wrapper (Section 2.3.3), and quickly streamline the parsing of a JSON document. Similarly scripts may be written on the host machine to access the database, and do special or diagnostic queries.

5.2 Interfacing to the Web

Since this project is WSGI conformant [8], there are a variety ways to deploy the application to be visible by the web users. Options start with direct deployment through any of the available mini WSGI servers, and range up to using the Goliath known as Apache2. Since deployment under Apache would be considered the most secure and flexible server that may handle other web requests, this section discusses deployment methods compatible with Apache. Please note that *mod-Python* has not been discussed due to that project no longer being actively maintained.

5.2.1 Web Server Gateway Interface

Web Server Gateway Interface (*WSGI*) is a specification for application and web servers to be able to communicate with web applications [32]. This standard promotes web application portability across a variety of web servers without the pitfalls of the Common Gateway Interface (*CGI*) and it successor the Fast Common Gateway Interface (*FCGI*). The WSGI specification is currently a Python standard and is implemented for many webservers including Apache.

Mod WSGI is an Apache module that aims to be fully WSGI compliant and hence allows serving of any WSGI application through Apache. Since the web application runs through the Apache web server, it utilizes Apache's forking technique allowing the application to be run under a configured Unix user and group. WSGI is currently the preferred method of deployment due to the popularity of the WSGI standard within the Python community and its replacement of the *mod-Python* method of deployment.

5.2.2 Simple Common Gateway Interface

Simple Common Gateway Interface (*SCGI*) is a very light protocol, which communicates using netstrings. Netstrings are readable packets that allow for a simple and

easy to debug protocol. SCGI was designed to overcome the deficiencies in FCGI. SCGI is available to apache using the *Mod SCGI* module.

Deploying a Python WSGI application with SCGI requires a WSGI-to-SCGI adaptor. Such an adaptor is available in the *flup* library which can used via the paste package, that was installed with Pylons.

5.2.3 Fast Common Gateway Interface

Fast Common Gateway Interface (*FCGI*) is the initial protocol designed to overcome the speed deficiencies of standard CGI. It is argued that FCGI aims to over solve the problems with CGI with the inclusion of multi-threading and multi-processes in its protocol. On many production systems, it is visible that once FCGI starts spawning processes it places the machine it is running on in a very fragile state and in most circumstances in a DOS (Denial of Service) state which is undesirable. For production use of FCGI many people choose to turn its super features off or limit them in order to get a usable system and thus is not favoured in most web communities that are able to avoid its use. To use FCGI, a WSGI-to-FCGI adaptor is available in the flup package, and a module (*Mod FCGI*) is available for Apache.

5.2.4 Apache JServ Protocol

Apache JServ Protocol (*AJP*) is a binary protocol written primarily for the Jakarta project. AJP may be used to interface to a Python WSGI application. The AJP protocol aims to be as complicated as FCGI however without its bugs. The authors are amongst the ones involved with SCGI. The main reason netstrings were not used this time around is that this protocol contains more overhead then the Simple Common Gateway Protocol (SCGI) and would be quite slow using readable packets. To interface a Python WSGI application with AJP, once again a converter can be found in the flup package and can be setup with *mod_proxy_ajp* under Apache.

5.2.5 Reverse Proxy

There exists many standalone WSGI servers that can serve this project, including the standard Pylons Paste server, which claims to be the fastest deployment method for Pylons. One may also look at using Apache simply as a proxy for the web application.

Although this is a plausible option, it is not really a desirable one, as it forces the maintenance of this external process to allow Apache to communicate with it. To utilize reverse proxying with Apache *Mod Proxy* and *Mod Rewrite* would be used.

5.2.6 Final Deployment

Due to the portability of the WSGI standard and the availability of the Mod WSGI Apache module, WSGI and Mod WSGI were chosen as method of deployment of my web application.

5.3 Performance

The system was tested to see how it handled under stress and load. Three experiments were setup to test the performance of the system. The following subsections describe the conditions for the experiment and its results.

5.3.1 Experimental Setup

The system was deployed to an *INTEL Pentium 4* Machine, machine running a *Debian* flavour of Linux (see Fig 5.1a). The database was built using *PostgreSQL 8.3*. The latest versions of the dependencies were installed, including *Pylons 2.6.2*, *SqlAlchemy 0.5*, *Apache 2.2.8*, *mod-wsgi 2.3* and *blockdesign 0.1.5*.

To complete the performance testing, the load/stress testing software *Tsung* [40] was used. This software allows for recording sample user sessions and replaying them simulating hundreds of simultaneous user access. Tsung was setup on the client machine (Fig 5.1b) to record information about both the server and the client. The response rates of the clients are measured, and the server is monitored by an Erlang monitor for CPU and memory consumptions. Any errors generated by the server, typically HTTP 500 (Internal Server Error) codes are recorded to show the breaking point of the system.

5.3.2 Experiment Details

Three different experiments were run in order to test the performance of the system. All of the experiments tested the system usage via the web interface, which is currently

```
CPU Model        : Intel(R) Pentium(R) 4 CPU 2.40GHz
CPU MHz          : 2400.185
CPU Cache        : 8KB (L1), 512KB (L2)
Memory           : 512MB * 2 (Running at 333Mhz)
Ethernet         : 3Com 3c905C-TX/TX-M [Tornado]
Ethernet Size    : 100 MB/s
Hard Disk        : Western Digital WDC WD600JB-00CR
Operating System : Ubuntu 8.04.1 (Hardy)
Unix Kernel      : 2.6.24-19-generic
```

(a) Server Specs

```
CPU Model        : Intel(R) Pentium(R) M processor 1400MHz
CPU MHz          : 600.000
CPU Cache        : 64KB (L1), 1MB (L2)
Memory           : 512MB (Running at 133Mhz)
Ethernet         : PRO/Wireless LAN 2100 3B Mini PCI adaptor
Ethernet Size    : 10 MB/s
Hard Disk        : Toshiba MK4025GA
Operating System : Ubuntu 8.04.1 (Hardy)
Unix Kernel      : 2.6.24-19-generic
```

(b) Machine Running Clients Simulator

Figure 5.1: OS and Hardware Information for Performance Tests

the only external access method to the system.

There are two main methods to query the system (see Section 4.2). All the simulated clients in each of the first two experiments used one of these methods, while the third experiment simulated a fairly random access strategy that aims to simulate the daily usage of the system.

The *Categorical Access Experiment* dealt with access to the combinatorial designs corpus via the summary table on the home page. Since the summary table is searchable via their summary reference id, Tsung was configured to randomly pick an id to search, simulating a user click on one of the links available on the home page.

The second experiment (*Search Form Experiment*) dealt with using the search form also available from the home page. Tsung was setup for this experiment to simulate users accessing the design corpus via the simple search form.

These first two experiments are to test the major entry points to the system. The entry points described in the experiments above are considered major because they are available on the home page of my web application. Moreover, they are the simplest methods of locating a particular design, and therefore most users would be interested in using them.

Lastly, *Random Access Experiment* simulates users querying and browsing the database through all entry points, simulating a generic full usage of the system. The user sessions simulated by this experiment were of both users that are browsing the System without any particular design in mind, and users that are looking for a particular design. All the web interface's pages are accessed in this test, thus load testing the entire web application.

Tsung allows for the configuration of thinking time in order to simulate real user behaviour. Each of the test runs were one hour in length. The following sub-sections discuss the results of each of the experiments.

5.3.3 Categorical Access Experiment

When looking at querying the database via the summary table, on average, the result set was available to the user in 3.07 sec. The fastest being returned at 1.20 sec and the slowest at a delta of 25.70 sec. For this test, server connection was established on average after 3.3 msec, with a lower bound of 1.2 msec and an upper bound of 0.11

sec. In summary the average session (total of connect and request) took 3.078 sec to complete.

Internal server errors begin to appear when the number of users exceeded 1000 users per hour (1 users every 3.6 seconds). For the given optimal run, querying and browsing a subset from the returned result set resulted in a throughput of 26.75 MB from the server and 154.87 KB to the server. The server CPU usage ranged from 0.9% to 100%. Memory consumption for this test was 27.05 MB.

5.3.4 Search Form Experiment

Querying the database via the simple form, on average, the result set was available to the user in 4.65 sec. The quickest completed at 1.94 sec and the slowest at a delta of 16.1 sec. For this test, server connection was established on average after 5.88 msec, with a lower bound of 1.3 msec and an upper bound of 0.39 sec. In summary the average request (total of connect and page) took 4.654 sec to complete.

Similar to the *Categorical Access Experiment*, Internal server errors begin to appear when the number of users exceeded 1000 users per hour (1 users every 3.6 seconds). For the given optimal run, querying and browsing a subset from the returned result set resulted in a throughput of 27.66 MB from the server and 497.98 KB to the server. The server CPU usage ranged from 0.7% to 100%. Memory consumption for this test was 13.55 MB.

5.3.5 Random Access Experiment

This experiment focused on simulating the utilization of all entry points to the system via the web interface. On average the response was available to the user in 90.5 seconds. The fastest page returned at 0.226 sec after being requested and the slowest at a delta of 254.7 sec. For this test the server connection was established on average after 4.25 msec, with a lower bound of 1.3 msec and an upper bound of 36.7 sec. In summary the average request (total of connection establishment and server response) took 90.5 sec. The reason for this high number is that each client had multiple requests, simulating a real live user, and therefore many simultaneous users were connected at the same time (which hit a maximum of 423 simultaneous connected users). Moreover, the main summary page, which contains a table that is made up

of more than 3,000 rows, was included in this test. Since this page takes some time to download it had considerable stress on the network throughput and the server.

Since this test fully utilizes and stresses the system, internal server errors begin to appear earlier than in the previous tests. The break point experienced in this test seemed to hover around 710 users per hour (1 users every 5 seconds).

The usage scenario simulated by this experiment resulted in a throughput of 88.38 MB (773.78 Kbps) from the server and 4.59 MB (28.83 Kbps) to the server. The server CPU usage ranged from 1% to 100%. The memory consumption for this test was 195.99 MB.

5.3.6 Experimental Summary

The experiments show that the system can successfully handle between 700-1000 users per hour depending on their utilization of the system. Section 6.2 displays the current number of users my system is serving. Since this number is far less than 10 users per hour, it is safe to say that the system can successfully handle the current load and substantially bigger loads. In the future, when more users begin using my system, a more complex setup may be considered, which is described in section 7.4.

Chapter 6

System Interfaces and Usage

This project implements a fairly large system with many sub-systems that work together to serve the user. To aid in seeing the abstract interaction of the different basic subsystems a system diagram is shown in Figure 6.1.

The system implemented by this project is one that is awaited by the combinatorial and statistical design users since it was never implemented before. Due to this reason, the application was fairly well publicized on DesignTheory.org, DesignDB.org and WikiPedia with various links leading to the web server, which at the time of this writing is `http://nassrat.cs.dal.ca/ddb2`. Section 6.2 displays the utilization of the system for the months prior to the publishing of this document.

6.1 Searching For a Design

A user of this combinatorial design database, Mr. Brian Moore from Johann Radon Institute for Computational and Applied Mathematics (RICAM), has had a recent problem that required a combinatorial design. The problem his colleagues and himself were working on seems to have been quite complex, thus he has mapped it into a simpler form in his request for help. Mr. Moore formulated his search as follows:

> In our case, we are interested in a slightly different problem (I am not sure if it has a name). For this, let me explain the problem: We have 18 players playing a certain sport (let's say curling) playing on 3 different allies (6 players per alley) at the SAME time. They play 17 games and we want that every combination of 2 players play exactly 5 times together. That is, we want to build a schedule.[1]

Mr. Moore's description leads to designs with parameters $v = 18, k = 6, \lambda = 5, t = 2, r = 17, b = 51$. This is because of the following mapping:

[1] Excerpt from an email conversation with Brian Moore.

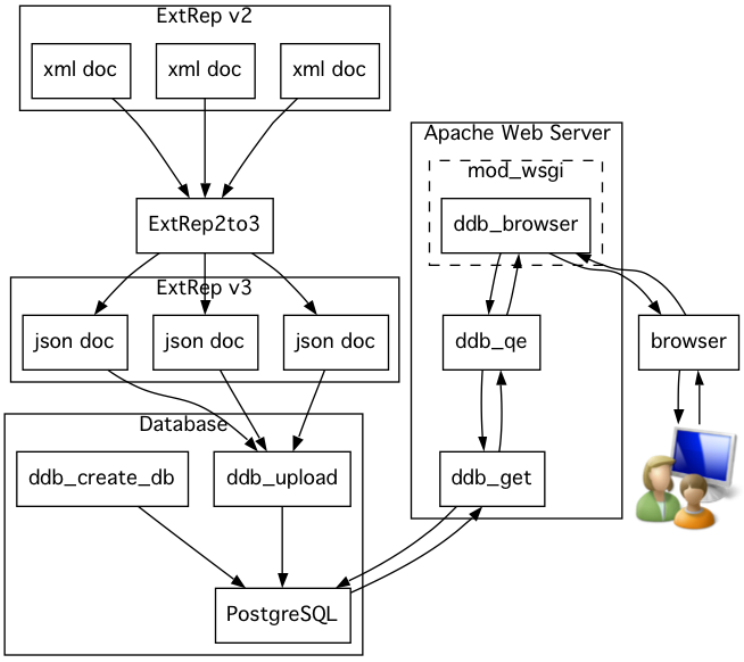


Figure 6.1: Design DB Overview Diagram

v number of players

k numbers of players per game

λ number of times each pair (for t=2) repeats within all blocks

r number of games played by each player

b number of different games in total

Moreover, Mr. Moore's problem contains an extra constraint that the design must be resolvable (simply defined as partition-able) [29]. The following subsections describe how the system can be utilized to search for the desired design.

6.1.1 Utilizing the Summary Table

The summary table allows the users to access the designs through choosing a top level category. The designs are categorised based on the basic parameters $t, v, b, r, k, \lambda, q, x$ (Section 3.3.3). Explanation of these parameters is in the sytem in a table to aid users in understanding the meaning of the parameters. Some of these parameters are

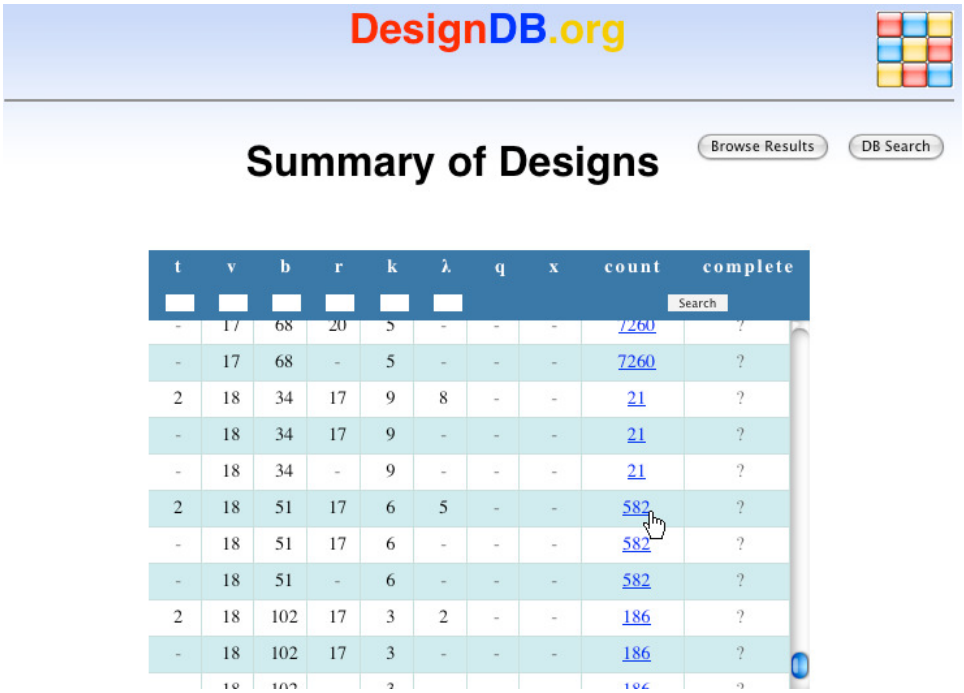


Figure 6.2: Finding a group of designs from the summary table

not applicable for all categories, which is indicated using the symbol "-" to replace the value. Figure 6.2 displays a section of the summary table that contains the design category $v = 18, k = 6, \lambda = 5, t = 2, r = 17, b = 51$. To locate this entry the user would need to move the scrollbar on the left side of the lexicographically sorted table.

Clicking on the link displayed in the "Number of Designs" column for a given category sends an appropriate query to the database server to return a cursor to the given result set. Figure 6.3 displays the result of clicking on the category within which the design (specified in the problem above) would be located. As mentioned before, the link for the category is the one pointed to by the cursor in figure 6.2, indicating that there is 582 results in that given class.

As can be seen in figure 6.3, the user has two main options, the first is to download the result set, in which case the results will be streamed to the user as a JSON Ext Rep Document. The user has a second option, which is to browse the result set. In the browsing mode, the user may go forward or backward through the result set. Moreover, the user may download a single design as a compliant Ext Rep document. Figure 6.4 displays the third result within this result set.

Figure 6.3: Query Result Summary Page - For the given search

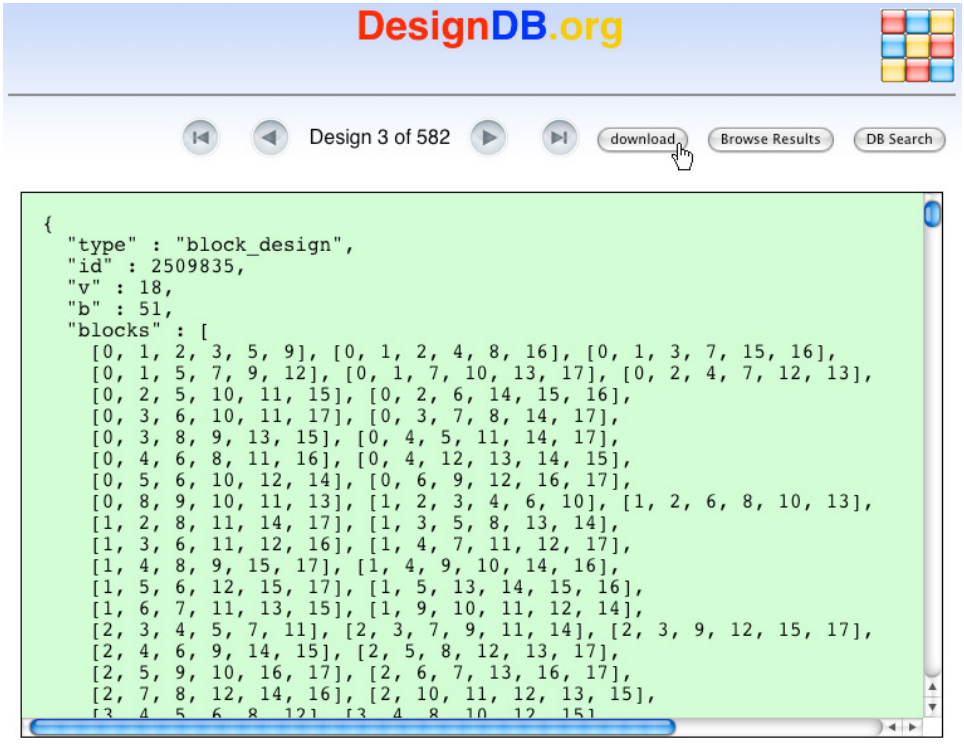


Figure 6.4: Third result for $v = 18, k = 6, \lambda = 5, t = 2, r = 17, b = 51$

Parameter	Condition
L	= 5
b	= 51
k	= 6
r	= 17
t	= 2
v	= 18

Figure 6.5: Query Results for $v = 18, k = 6, \lambda = 5, t = 2, r = 17, b = 51$

6.1.2 Using the Brief Design Search

A second method of searching for the design database is through the mini search attached to the top of the summary table. This search bar can be seen in figures 4.7 and 6.2. Filling the applicable search boxes and clicking on "search" sends the query to the database server. This feature allows for the creation of classes "on the fly", or for quicker access than locating the appropriate class in the summary table.

For the problem described above (Section 6.1) the result summary after doing a search for $v = 18, k = 6, \lambda = 5, t = 2, r = 17, b = 51$ is displayed in figure 6.5.

6.1.3 Using the Full Design Search

There is yet another method to search the database, as mentioned in section 4.2. This method of searching is flexible to allow querying the database via any of the Ext Rep parameters. The advantage of this feature is to be able to quickly find the design being targeted because the method allows for the resolvable flag in the search. Figure 6.6 shows the query preview, while figure 6.7 displays the search results.

As can be seen from the search results (Figure 6.7), no results were found in the database for Mr. Brian Moore's problem. However, looking at the summary table

Figure 6.6: [Brian Moore's Search] Query Preview

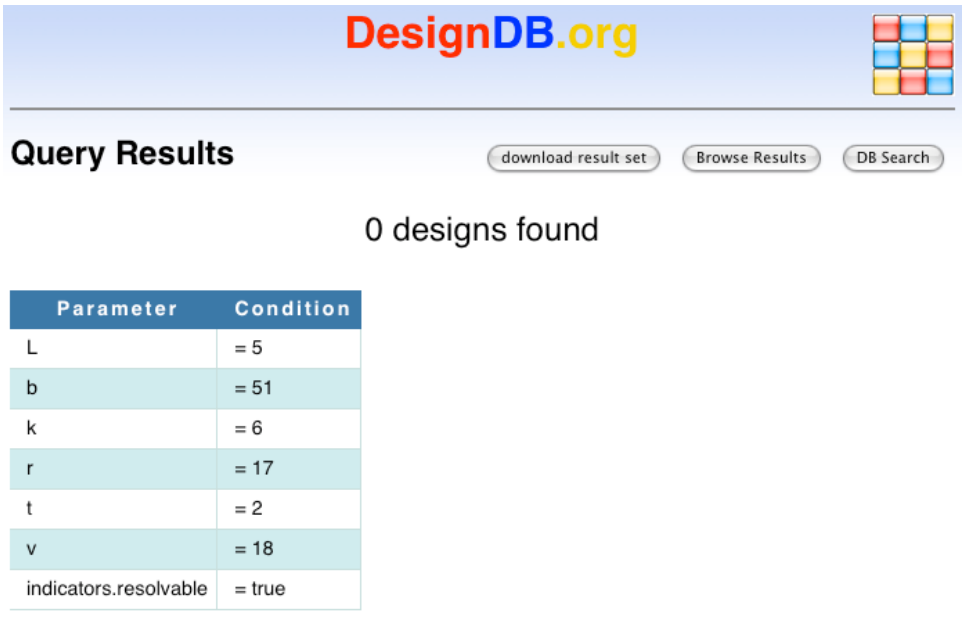
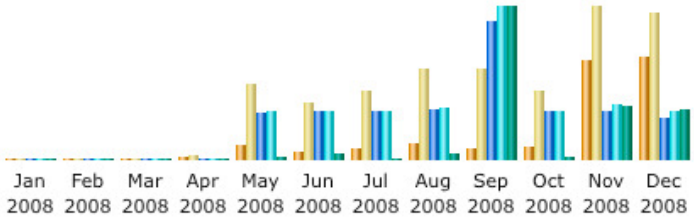


Figure 6.7: Query Results for $v = 18, k = 6, \lambda = 5, t = 2, r = 17, b = 51, Resolvable$



Month	Unique visitors	Number of visits	Pages	Hits	Bandwidth
Jan 2008	0	0	0	0	0
Feb 2008	0	0	0	0	0
Mar 2008	0	0	0	0	0
Apr 2008	6	9	15	30	678.96 KB
May 2008	36	188	16694	17595	12.11 MB
Jun 2008	20	141	17381	17561	27.63 MB
Jul 2008	26	169	17730	17781	2.24 MB
Aug 2008	41	226	18286	18481	25.06 MB
Sep 2008	29	226	50252	55400	658.36 MB
Oct 2008	30	170	17599	17676	8.35 MB
Nov 2008	247	378	17449	20002	228.39 MB
Dec 2008	252	362	15153	17540	217.56 MB
Total	687	1869	170559	182066	1.15 GB

Figure 6.8: Web Visit Stats for 2008

entry (Figure 6.2), the database may not have all the entries for this particular design class due to the "?" (unknown) flag in the complete column. If Mr. Moore solves the problem, his solution can be entered into the database for other people to discover and use.

6.2 Usage Counts

It was expected that my system was urgently needed by its stake holders (combinatorial design users). For this reason the web application was made public and was publicized shortly after the system was deemed stable. This section displays the usage statistics of the system in the months prior to the publishing of this document.

This system was made public in late April 2008. Figure 6.8 shows the monthly visit and page hit history for the web interface in 2008. As can be seen in the figure, the number of unique visits increased with time. The abrupt increase in the number of visits in November correlates with placing links to the web application on the famous website Wikipedia in the pages that discuss block designs.

To be able to analyse the stats further, the most recent statistics for the full month of January 2009 will be used. Figure 6.9 displays detailed statistics for each day in January 2009.

As can be seen, there seems to be a steady rate of visits per day with a consistent slight decrease over the weekends. This is considered normal as most users of the database would utilize combinatorial designs in their work, which usually occurs on weekdays. The average page hits for each day of the week can be seen in figure 6.10.

Most visitors did not spend a large amount of time on the website. This could mean two things, either that they did not enjoy the web application, or that they were quickly able to utilize the system to either get what they were looking for or determine that the block design is not in the database. Nevertheless a 20% of the *vists* lasted for over one hour. Figure 6.11 displays the durations for the month of January 2009.

The main entry point url to the webserver in the month of January is the design database web application. Although the webroot had many more hits, it seems that this was the result of users roaming around the website after utilizing the design database. Figure 6.12 displays the different hits for the top urls on the webserver. Note that the urls starting with "ddb2" point to the Design DB.

The design database has users from all over the world. The majority of the users seem to be located in the United States and Great Britain. However, there seems to be a significant number of users from other countries as well. Figure 6.13 displays the locations by country of *IP* addresses that accessed the Design DB.

There seems to be a wide variety of operating systems and web browser clients that accessed the Design DB. However, the majority of the users were utilizing Microsoft Windows and Microsoft Internet Explorer. The number of hits given an operating system and web browser software for January can be seen in figures 6.15a and 6.15b respectively.

Day	Number of visits	Pages	Hits	Bandwidth
01 Jan 2009	12	605	671	5.12 MB
02 Jan 2009	10	583	627	5.39 MB
03 Jan 2009	8	579	625	4.08 MB
04 Jan 2009	9	584	641	5.72 MB
05 Jan 2009	7	578	608	3.04 MB
06 Jan 2009	13	594	685	9.21 MB
07 Jan 2009	14	687	767	7.38 MB
08 Jan 2009	18	608	714	8.66 MB
09 Jan 2009	14	601	697	9.14 MB
10 Jan 2009	7	575	582	2.11 MB
11 Jan 2009	6	572	574	138.87 KB
12 Jan 2009	10	581	623	4.83 MB
13 Jan 2009	19	648	757	13.27 MB
14 Jan 2009	18	685	818	10.69 MB
15 Jan 2009	13	595	670	6.26 MB
16 Jan 2009	12	601	659	9.33 MB
17 Jan 2009	10	583	609	2.97 MB
18 Jan 2009	14	594	643	5.12 MB
19 Jan 2009	19	623	721	10.95 MB
20 Jan 2009	23	617	733	12.20 MB
21 Jan 2009	25	621	779	15.89 MB
22 Jan 2009	20	659	895	16.09 MB
23 Jan 2009	10	650	719	6.02 MB
24 Jan 2009	7	400	440	4.07 MB
25 Jan 2009	7	397	433	4.57 MB
26 Jan 2009	24	581	909	17.60 MB
27 Jan 2009	20	422	579	15.49 MB
28 Jan 2009	15	415	523	9.91 MB
29 Jan 2009	15	419	521	7.71 MB
30 Jan 2009	16	470	678	9.53 MB
31 Jan 2009	13	404	473	8.42 MB
Average	13.81	565.52	657.19	8.09 MB
Total	428	17531	20373	250.89 MB

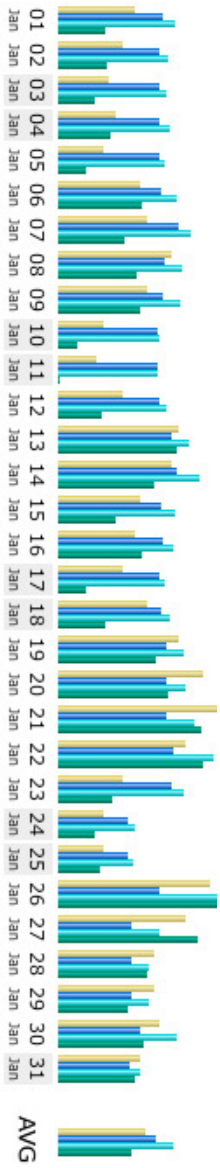
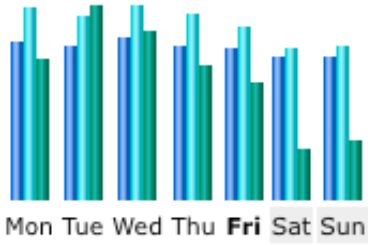


Figure 6.9: Web Visit Stats for January 2009



Day	Pages	Hits	Bandwidth
Mon	590.75	715.25	9.10 MB
Tue	570.25	688.50	12.54 MB
Wed	602	721.75	10.97 MB
Thu	577.20	694.20	8.77 MB
Fri	565.80	648	7.61 MB
Sat	534.25	564	3.31 MB
Sun	536.75	572.75	3.89 MB

Figure 6.10: Web Visit Aggregate for Weekdays in January 2009

Visits duration		
Number of visits: 414 - Average: 852 s	Number of visits	Percent
0s-30s	236	57 %
30s-2mn	40	9.6 %
2mn-5mn	23	5.5 %
5mn-15mn	16	3.8 %
15mn-30mn	4	0.9 %
30mn-1h	8	1.9 %
1h+	84	20.2 %
Unknown	3	0.7 %

Figure 6.11: Web Visit Durations for January 2009

134 different pages-url	Viewed	Average size	Entry	Exit
/	15852	11 Bytes	164	163
/ddb2/	259	672.35 KB	220	134
/ddb2/query	158	6.87 KB	5	23
/ddb2/design_browser/browse	137	13.45 KB	2	26
/ddb2/design_browser/query_schema	94	3.84 KB		23
/ddb2/design_browser/search	69	20.02 KB	6	
/ddb2/design_browser/browse/2	37	41.92 KB	1	1
/ddb2/design_browser/query	36	6.85 KB	2	6
/ddb2/design_browser/download /results.json	35	281.42 KB		7

Figure 6.12: Top Urls for January 2009

Countries		Pages	Hits	Bandwidth
Unknown	ip	10088	10151	6.74 MB
United States	us	6313	7761	135.38 MB
Great Britain	gb	155	443	18.54 MB
European country	eu	128	231	7.97 MB
Canada	ca	107	316	10.61 MB
Italy	it	75	101	2.67 MB
Germany	de	53	187	12.80 MB
Australia	au	27	105	11.18 MB
Spain	es	18	94	6.40 MB
France	fr	15	46	2.93 MB
Others		72	325	25.90 MB

Figure 6.13: User Countries - January 2009

Total: 19 different pages-url	Pages	Percent	Hits	Percent
http://en.wikipedia.org/wiki/Block_design	61	25.5 %	61	25.5 %
http://en.wikipedia.org/wiki/Design_of_experiments	56	23.4 %	56	23.4 %
http://en.wikipedia.org/wiki/Combinatorial_design	50	20.9 %	50	20.9 %
http://designdb.org	35	14.6 %	35	14.6 %
http://en.wikipedia.org/wiki/Randomized_block_design	12	5 %	12	5 %
http://en.wikipedia.org/wiki/Combinatorics	5	2 %	5	2 %
http://en.wikipedia.org/wiki/Symmetric_design	3	1.2 %	3	1.2 %
http://en.wikipedia.org/wiki/BIBD	2	0.8 %	2	0.8 %

Figure 6.14: Top Referrals for January 2009

Operating Systems	Hits	Percent
Windows	18784	95 %
Macintosh	544	2.7 %
Linux	218	1.1 %
Unknown	101	0.5 %
BSD	70	0.3 %
Sun Solaris	43	0.2 %

(a) Top Operating Systems

Browsers	Grabber	Hits	Percent
MS Internet Explorer	No	17333	87.7 %
Firefox	No	1892	9.5 %
Safari	No	301	1.5 %
Opera	No	106	0.5 %
Unknown	?	73	0.3 %
Mozilla	No	27	0.1 %
Netscape	No	15	0 %
BonEcho (Firefox 2.0 development)	No	12	0 %
- FDM Free Download Manager	**Yes**	1	0 %

(b) Top Web Browser Software

Figure 6.15: User Software - January 2009

Chapter 7

Conclusion and Future Considerations

7.1 Ext Rep Extensions

The External Representations $v3$ is not the end of this protocol. An attempt was made to cater for all the needs that were missing from $v2$; However, there appears that more will be required from such a protocol. It was only after the deployment and usage of the Ext Rep $v2$ [21] when it was apparent that there were issues that required attention. Similarly, issues would arise after the use of the newly developed protocol $v3$.

A query language was proposed in the described system, however, it is not fully compatible with the Ext Rep protocol as it changes the structure of the document. In the future, research will be made, to either modify the external representation to cater for querying the database, or try to devise a method of using the currently designed protocol as a query language.

7.2 RDBMS Alternatives

In this project, a Relational Database System was used to host the combinatorial designs, with a nice wrapper to give it an Object Oriented feel. However, further research should be considered with a different type of a database engine.

HDF5 and its Python wrapper *PyTables* have been considered here but have been found inappropriate for the current requirements and design. When *PyTables* progresses to be in a fully compatible state with *HDF5* this options should be revisited for consideration.

Moreover, consideration should be given to Object Oriented databases such as Python's infamous *ZODB*. Similarly, one should also consider *Schevo*, *Durus*, *Metakit* and *BuZhug*.

Nevertheless, a document database should also be considered, namely *Couch DB*.

Couch DB is a JSON database that uses map reduce [27] to filter through the documents and return the relevant document set. This strategy has been found fast and useful in the search technology by Google [27]. The classification and grouping of designs may require some extra consideration to integrate with Couch DB but it may lead to a much simpler system and can be looked at as a future sub-project.

7.3 Interface Personalization

Without users my system would be useless, and therefore one must aim to cater for the users of the system. Although this is the first and only database of combinatorial and statistical designs on the web, users may not use this application if it is not easy to use.

To understand better how the users are using the system, further research will need to take place. Examples of technologies that may aid in such research are tracking tools that would report back how long users have been dwelling on certain pages and their idling times. Similarly looking at the exit points from the system may help identify pages with an unintuitive user interface (UI) that may have led to user frustration with the system.

Indeed when the number of users grows, professional ethnographers may be required to identify the bottlenecks of the user interface and what can be changed to facilitate the user experience.

It is obvious that to query using any parameter in the Ext Rep is not fast. A user may need to perform up to 5 clicks to add a single condition to the query. In the near future the query interface will be revisited, and research will take place to make the interface more intuitive and simple.

Interface personalization is another area of interest. Research into this area would lead to improving the Web 2.0 look and feel of the system.

7.4 High Performance

The design database is expected to grow in the number of designs and users. Since performance degrades with the number of users, the system may be rendered useless with extreme traffic. Therefore prior to reaching the breaking capacity, consideration

must be made for high performance and availability setups, such as having multiple databases and web servers, for both load balancing and failsafe measures.

7.5 Software Upgrade

As this document is being published, the state of the open source tools that were used in this project have been advancing. In October 2008 the new version of Python 2.6 was released. This new version has a few extra features over Python 2.5 and has deprecated a few of the older functionalities. Similarly Pylons 0.9.7 is coming close to being released. These two specific pieces of software are part of the core of the design database system, thus updating to these new versions will be the next priority.

In general, open source tools have fast release cycles which makes this task an ongoing one. However, this is not a disadvantage, on the contrary, it is a great advantage of using open source software.

7.6 RESTful API

The current system is lacking an application programming interface (API) such that programmers can effectively utilize combinatorial designs. An example of such a user is Brian Moore (Section 6.1). Mr. Moore asked if there was an easy way for programmers to interact with the system. Since mathematicians and computer scientists find combinatorial designs useful, such an interface would allow them to quickly and easily access the system. A design for such an interface is currently in a β (beta) stage, and will be part of the focus for the next phase of this project.

7.7 Accepting Contributions

The success of this project lies in the fact that it is currently the only database of combinatorial designs on the web. Thus my system acts as the sole online combinatorial design repository. For this reason it is essential that an automated system is deployed that will accept new designs. Such a system has to be carefully designed to check if the design is pairwise non-isomorphic to all designs in the database. Following that the design would be accepted and integrated into the system. Although

many of the building blocks for this feature are available, it is a considerably sized task to complete and will also be a major focus for future phases of this project.

7.8 Conclusions

My project made three large contributions to the field of combinatorial and statistical designs. The new Ext Rep $v3$ was created and is expected to be more popular than its XML predecessor. A categorized database of pairwise non-isomorphic designs from the set of designs gathered by DesignTheory.org was created. Finally an interface was produced and deployed on the web as a gateway to the system such that the users may interact with the system and search for combinatorial designs.

Prior to this project such a database did not exist, therefore forcing many users to recreate efforts and regenerate designs. The initiation of this project is an aim to create a central repository of combinatorial and statistical designs accessible at all times to all users.

Moreover, this project had contributions in the field of computer science in the integration of the subsystems and the evaluation and use of bleeding edge technologies. Therefore, it may be used as an example for other projects utilizing open source software, or ones aimed at designing databases to host mathematical objects. Similarly projects that are considering implementing a data representation may find this project valuable.

As with all systems, and especially software engineering, the process is an iterative one. Although much has been done to make such a system flourish and be usable by its users, the users will always request new features and simpler interfaces. Hopefully this project will have many phases in the future with many contributors such that it remains the largest single repository for combinatorial designs on the web.

Bibliography

[1] 05migration - sqlalchemy [online, cited 2008 June 17]. Available from World Wide Web: http://www.sqlalchemy.org/trac/wiki/05Migration. Migrating from SA 0.4 to 0.5.

[2] 14.14 ctypes – a foreign function library for python. [online, cited 2008 September 21]. Available from World Wide Web: http://docs.python.org/lib/module-ctypes.html.

[3] Collection of Designs. Available from World Wide Web: http://designdb.org/extrep/json-files/.

[4] Javascript Object Notation (JSON). Available from World Wide Web: http://www.JSON.org.

[5] KISS principle [online, cited 1 January 2009]. Available from World Wide Web: http://en.wikipedia.org/wiki/KISS_principle.

[6] Model-View-Controller [online, cited 2009 February 19]. Available from World Wide Web: http://en.wikipedia.org/wiki/Model-view-controller#Pattern_description.

[7] Relax ng schema language for xml. Available from World Wide Web: http://relaxng.org.

[8] Specifications - wsgi wiki [online, cited 2008 December 1]. Available from World Wide Web: http://wsgi.org/wsgi/Specifications.

[9] Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981. Available from World Wide Web: http://cs.anu.edu.au/~bdm/papers/pgi.pdf.

[10] Dion Almaer. Ajaxian JSON vs. XML: the debate [online, cited 2009 April 4]. Available from World Wide Web: http://ajaxian.com/archives/json-vs-xml-the-debate.

[11] Francesc Alted and Ivan Vilata. PyTables - hierarchical datasets in python. Available from World Wide Web: http://www.pytables.org.

[12] Paul M. Aoki. Implementation of extended indexes in postgres. *SIGIR Forum*, 25(1):2–9, 1991.

[13] R. A. Bailey and Peter J. Cameron. What is a design? how should we classify them? *Des. Codes Cryptography*, 44(1-3):223–238, 2007.

[14] R. A. Bailey, Peter J. Cameron, Peter Dobcsnyi, John P. Morgan, and Leonard H. Soicher. Designs on the web. *Discrete Mathematics*, 306(23):3014–3027, 12/6 2006.

[15] Ivan Vilata Balaguer and Hatem Nassrat. Re: VLArray inside a table. Available from World Wide Web: http://article.gmane.org/gmane.comp.python.pytables.user/811.

[16] Ian Bicking. Full Stack vs. Glue [online]. February 2007 [cited 2008 December 1]. Available from World Wide Web: http://blog.ianbicking.org/full-stack-vs-glue.html.

[17] Georg Brandl, Peterson Benjamin, Cannon Brett, Winter Collin, and Loewis Martin. Repository - directory - projects: sandbox/trunk/2to3. Available from World Wide Web: http://svn.python.org/view/sandbox/trunk/2to3/.

[18] Peter J. Cameron, editor. *Encyclopaedia of DesignTheory*. 2006. Available from World Wide Web: http://designtheory.org/library/encyc/.

[19] Peter J. Cameron. Designs. In W. T. Gowers, editor, *Princeton Companion to Mathematics*. Princeton University Press, March 2008. ISBN: 0-691-11880-9.

[20] Peter J. Cameron, Peter Dobcsányi, John P. Morgan, and Leonard H. Soicher. Dtrs protocol version 2.0, 2004. Available from World Wide Web: http://designtheory.org/library/extrep/.

[21] PJ Cameron, P. Dobcsányi, JP Morgan, and LH Soicher. *The External Representation of Block Designs*. December 2003. Available from World Wide Web: http://designtheory.org/library/extrep/ext-rep.pdf.

[22] Charles J. Colbourn and Paul C. van Oorschot. Applications of combinatorial designs in computer science. *ACM Comput. Surv.*, 21(2):223–250, 1989.

[23] Charles J. Coldourn and Jeffrey H. Dinitz. *The CRC Handbook of Combinatorial Designs*. CRC press edition, 1996.

[24] Rick Copeland. *Essential SQLAlchemy*. O'Reilly, 2008.

[25] Douglas Crockford. RFC4627: Javascript Object Notation, 2006. Available from World Wide Web: http://www.ietf.org/rfc/rfc4627.txt.

[26] Kevin Dangoor. Webpae about turbogears [online, cited 2007 October 15]. Available from World Wide Web: http://turbogears.org/about/.

[27] Jeffrey Dean and Sanjay Ghemawat. Google research publication: MapReduce. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, December 2004. Available from World Wide Web: http://labs.google.com/papers/mapreduce.html.

[28] Chris DiBona, Sam Ockman, and Mark Stone, editors. *Open Sources: Voices from the Open Source Revolution*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1999.

[29] P. Dobcsányi and Hatem A. Nassrat. *The External Representation of Block Designs v3 β*. 2008. Available from World Wide Web: http://designdb.org/extrep/ext-rep.pdf.

[30] Peter Dobcsányi. PYDESIGN version 0.5, November 2004. Available from World Wide Web: http://designtheory.org/software/pydesign/.

[31] Peter Dobcsányi. block_design v0.1.5, June 2008. Available from World Wide Web: http://www.sagemath.org/doc/reference/sage/combinat/designs/block_design.html.

[32] Philip Eby. Pep: 333, python web server gateway interface v1.0. http://www.python.org/dev/peps/pep-0333/.

[33] Brian Everitt and Torsten Hothorn. *A Handbook of Statistical Analyses Using R*. Chapman & Hall/CRC, Boca Raton, FL, 2006. Available from World Wide Web: http://cran.r-project.org/src/contrib/Descriptions/HSAUR.html. ISBN 1-584-88539-4.

[34] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language*. O'Reilly, 2008.

[35] Mike Folk and Elena Pourmal. Hdf software process. In *NOBUGS*, volume NOBUGS 2004, December 2004. Available from World Wide Web: http://www.hdfgroup.uiuc.edu/papers/papers/Software_Engineering_at_HDF1.pdf.

[36] Arun Gupta. Language-neutral data format: XML and JSON [online, cited 2009 April 4]. Available from World Wide Web: http://blogs.sun.com/arungupta/entry/language_neutral_data_format_xml.

[37] Thomas Heller. The ctypes package. Available from World Wide Web: http://python.net/crew/theller/ctypes/.

[38] A. Konovalov. Computer algebra system gap. *"CHIP" Magazine*, (9), 2001. Supplementary article for the GAP 4.2 distribution on the CD-appendix to the magazine.

[39] Philipp Lenssen. Why good programmers are lazy and dumb, August 2005. Available from World Wide Web: http://blogoscoped.com/archive/2005-08-24-n14.html.

[40] Zhen Liu, Nicolas Niclausse, and César Jalpa-Villanueva. Traffic model and performance evaluation of web servers. *Perform. Eval.*, 46(2-3):77–100, 2001.

[41] Lloyd. Yet Another JSON Library, 2008. Available from World Wide Web: http://lloydforge.org/projects/yajl/.

[42] Hatem Nassrat. Design Database version 2, 2009. Available from World Wide Web: http://hg.nassrat.ca/ddb2/.

[43] Wakaha Ogata, Kaoru Kurosawa, Douglas R. Stinson, and Hajime Saido. New combinatorial designs and their applications to authentication codes and secret sharing schemes. *Discrete Mathematics*, 279(1-3):383 – 405, 2004. Available from World Wide Web: http://www.sciencedirect.com/science/article/B6V00-49M6T2B-4/2/41d82f427f342ce9a7a08ff3a61f281f. In Honour of Zhu Lie.

[44] Travis Oliphant. *Guide to NumPy*. http://numpy.scipy.org/numpybook.pdf.

[45] Kenneth H. Rosen and John G. Michaels. *Graph Invariants and Isomorphism*. CRC Press, 2000.

[46] Leo Simons. Our choices for python web applications [online, cited 2008 March 17]. Available from World Wide Web: http://lsimons.wordpress.com/2007/12/11/our-choices-for-python-web-applications/.

[47] Steve Spez. on lisp [online, cited 2007 December 12]. Available from World Wide Web: http://blog.reddit.com/2005/12/on-lisp.html.

[48] William Stein. *Sage: Open Source Mathematical Software (Version 2.10.2)*. The Sage Group, 2008. Available from World Wide Web: http://www.sagemath.org.

[49] Gerald Jay Sussman and Jr. Guy Lewis Steele. Scheme: An interpreter for extended lambda calculus. Technical Report AI Lab Memo AIM-349, MIT AI Lab, December 1975. Available from World Wide Web: http://repository.readscheme.org/ftp/papers/ai-lab-pubs/AIM-349.pdf.

[50] G. van Rossum and J. de Boer. Linking a stub generator (ail) to a prototyping language (python). *Proceedings of the Spring 1991 EurOpen Conference, Troms, Norway*, pages 20–24, 1991.

[51] Guido van Rossum and Fred L Drake. pickle – python object serialization [online, cited 2008 October 13]. Available from World Wide Web: http://docs.python.org/lib/module-pickle.html.

[52] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, 2002. Available from World Wide Web: http://www.wolframscience.com.

[53] Kris Zyp. XML vs JSON [online, cited 2009 April 4]. Available from World Wide Web: http://www.authenteo.com/page/XML_vs_JSON.

Appendix A

Ext Rep v3 Schema

```
#
# The External Representation of Block Designs
#
# an EBNF based schema
#

<list_of_designs> = {
    <header> ,
    <designs>
};

<header> =
         external_representation_version : "3.0"
         design_type                     : ( block_design | latin_square | mixed )
         number_of_designs               : ( $integer | unknown ) )
    ?( , <invariants> )
    ?( , pairwise_nonisomorphic          : ( $boolean | unknown ) )
    ?( , complete_upto_isomorphism       : ( $boolean | unknown ) )
    ?( , number_of_isomorphism_classes : ( $integer | unknown ) )
    ?( , precision                       : $integer )
    ?( , <info> )
;

<invariants> = invariants : {
    # relations which hold over the list of designs
    # conjunction implicitly assumed
    ?( relations : [
             [ <rel_arg>, <rel_op>, <rel_arg> ]
```

```
        *( , [ <rel_arg>, <rel_op>, <rel_arg> ] )

    ]
    # optional invariant description as needed
    ?( invariant_description : $string )
};

<rel_arg> = <design_parameter> | <scalar> ;

<design_parameter> = t | v | b | ... | resolvable | ... ;

<rel_op> = "=" | "!=" | "<" | "<=" | ">" | ">=" ;

<designs> = designs : [ <design> *( , <design> ) ] ;

<design> = <block_design> | <latin_square> ;

<block_design> = {
         type      : block_design ,
         id        : ( $string | $integer ) ,
         v         : $integer ,
    ?(   b         : $integer , )
    ?(   precision : $integer , )
         <blocks>
    ?( , <point_labels> )
    ?( , <indicators> )
    ?( , <combinatorial_properties> )
    ?( , <block_design_automorphism_group> )
    ?( , <resolutions> )
    ?( , <statistical_properties> )
    ?( , <alternative_representations> )
    ?( , <info> )
```

```
};

<blocks> = blocks : [  <block> *( , <block> ) ];

<block> =  [ $integer *( , $integer ) ];

<point_labels> = point_labels : [
      ( $integer *( , $integer ) )
    | ( $string *( , $string ) )
];

<latin_square> =  { latin_square : <to be defined later> };

<indicators> = indicators : { <indicator> *( , <indicator> ) } ;

<indicator> =
      repeated_blocks     : $boolean
    | resolvable          : $boolean
    | affine_resolvable   : ( $boolean | { mu            : $integer } )
    | equireplicate       : ( $boolean | { r             : $integer } )
    | constant_blocksize  : ( $boolean | { k             : $integer } )
    | t_design            : ( $boolean | { maximum_t     : $integer } )
    | connected           : ( $boolean | { no_components : $integer } )
    | pairwise_balanced   : ( $boolean | { lambda        : $integer } )
    | variance_balanced   : $boolean
    | efficiency_balanced : $boolean
    | cyclic              : $boolean
    | one_rotational      : $boolean
;

<combinatorial_properties> =
    combinatorial_properties : {
        <point_concurrences> ,
        <block_concurrences> ,
```

```
        <t_design_properties> ,
        <alpha_resolvable> ,
        <t_wise_balanced>
};

<point_concurrences> = point_concurrences : [
    <function_on_ksubsets_of_indices> *( , <function_on_ksubsets_of_indices> )
];

<block_concurrences> = block_concurrences : [
    <function_on_ksubsets_of_indices> *( , <function_on_ksubsets_of_indices> )
];

<t_design_properties> = t_design_properties : {
    t_design_properties_member *( , t_design_properties_member )
};

<t_design_properties_member> =
      parameters : {
          t      : $integer ,
          v      : $integer ,
          b      : $integer ,
          r      : $integer ,
          k      : $integer ,
          lambda : $integer
      }
    | square                : $boolean
    | projective_plane      : $boolean
    | affine_plane          : $boolean
    | steiner_system        : ( $boolean | { t: $integer } )
    | steiner_triple_system : $boolean
;

<alpha_resolvable> =  { <index_flag> *( , <index_flag> ) };
```

```
<index_flag> = "$integer" : ( $boolean | unknown ) ;

<t_wise_balanced> =  { <t_index_flag> *( , <t_index_flag> ) } ;

<t_index_flag> = "$integer" : ( $boolean | unknown | { lambda: $integer } ) ;

<resolutions> = resolutions : {
    pairwise_nonisomorphic  : ( $boolean | unknown ) ,
    all_classes_represented : ( $boolean | unknown ) ,
    value                   : [ <resolution> *( , <resolution> ) ]
};

<resolution> =  {
         function_on_indices: <function_on_indices>
    ?( , <resolution_automorphism_group> )
};

<resolution_automorphism_group> = resolution_automorphism_group : {
         <permutation_group>
    ?( , <resolution_automorphism_group_properties> )
};

<resolution_automorphism_group_properties> =
    resolution_automorphism_group_properties : <to be defined later> ;

<block_design_automorphism_group> = automorphism_group: {
    <permutation_group>,
    <block_design_automorphism_group_properties>
};

<permutation_group> = permutation_group : {
         degree : $integer ,
         order  : $integer ,
```

```
        domain : points ,
        <generators>
    ?( , <permutation_group_properties> )
};

<permutation_group_properties> = permutation_group_properties : {
    <permutation_group_properties_member> *( , <permutation_group_properties_member> )
};

<permutation_group_properties_member> =
      primitive             : $boolean
    | generously_transitive : $boolean
    | multiplicity_free     : $boolean
    | stratifiable          : $boolean
    | no_orbits             : $integer
    | degree_transitivity   : $integer
    | rank                  : $integer
    | <cycle_type_representatives>
;

<block_design_automorphism_group_properties> =
    automorphism_group_properties: {
        block_primitive           : ( $boolean | not_applicable ) ,
        degree_block_transitivity : ( $integer | not_applicable ) ,
        no_block_orbits           : ( $integer | not_applicable )
};

<cycle_type_representatives> = cycle_type_representatives : [
        <cycle_type_representative> *( , <cycle_type_representative> )
];

<cycle_type_representative> = {
    permutation          : <permutation>,
    cycle_type           : [ $integer *( , $integer ) ] ,
```

```
    no_having_cycle_type : $integer
};

<generators> = generators : [ ? ( <permutation> *( , <permutation> ) ) ];

<permutation> = [ $integer *( , $integer ) ];

<alternative_representations> = alternative_representations : {
    <incidence_matrix>
};

<incidence_matrix> = incidence_matrix: {
    shape   : points_by_blocks ,
    <matrix>
};

<matrix> =
       no_rows    : $integer ,
       no_columns : $integer ,
    ?( title      : $string , )
       matrix     : [ <row> *( , <row> ) ]
;

<row> = [ <number> *( , <number> ) ];

<statistical_properties> = statistical_properties: {
          precision: $integer
    ?( ,  <canonical_variances> )
    ?( ,  <pairwise_variances> )
    ?( ,  <optimality_criteria> )
    ?( ,  <other_ordering_criteria> )
    ?( ,  <canonical_efficiency_factors> )
    ?( ,  <functions_of_efficiency_factors> )
    ?( ,  <robustness_properties> )
```

```
};

<robustness_properties> = robustness_properties: {
    ?( <robustness_properties_member> *( , <robustness_properties_member> ) )
};

<robustness_properties_member> =
      robust_connected_plots     : <robust_connected_value>
    | robust_connected_blocks    : <robust_connected_value>
    | robust_efficiencies_plots  : <robust_efficiencies_value>
    | robust_efficiencies_blocks : <robust_efficiencies_value>
;

<robust_efficiencies_value> = {
    precision: $integer ,
    robustness_efficiency_values: [
        <robustness_efficiency_values> * ( , <robustness_efficiency_values> )
    ]
};

<robustness_efficiency_values> = {
         number_lost                : $integer ,
         loss_measure               : ( average | worst ) ,
    ?( , phi_0                      : <robustness_efficiency_values_value> )
    ?( , phi_1                      : <robustness_efficiency_values_value> )
    ?( , maximum_pairwise_variances : <robustness_efficiency_values_value> )
    ?( , E_1                        : <robustness_efficiency_values_value> )
};

<robustness_efficiency_values_value> = {
         self_efficiency       : <number>
    ?( , absolute_efficiency   : ( <number> | unknown ) )
    ?( , calculated_efficiency : ( <number> | unknown ) )
};
```

```
<robust_connected_value> = {
    number_lost : $integer ,
    is_max      : ( true | unknown )
};

<functions_of_efficiency_factors> = functions_of_efficiency_factors: {
    geometric_mean : <number> ,
    minimum        : <number> ,
    harmonic_mean  : <number>
};

<canonical_efficiency_factors> = canonical_efficiency_factors: {
    no_distinct: $integer | unknown | not_applicable ,
    ordered: true | unknown ,
    value: [
             { multiplicity                : ( $integer | not_applicable ) ,
               canonical_efficiency_factor : ( <number> | blank ) }
        *( , { multiplicity                : ( $integer | not_applicable ) ,
               canonical_efficiency_factor : ( <number> | blank ) } )
    ]
};

<other_ordering_criteria> = other_ordering_criteria : {
    ?( <other_ordering_criteria_member> *( , <other_ordering_criteria_member>  ) )
};

<other_ordering_criteria_member> =
      trace_of_square_of_C              : <ordering_criteria_value1>
    | max_min_ratio_canonical_variances : <ordering_criteria_value1>
    | max_min_ratio_pairwise_variances  : <ordering_criteria_value1>
    | no_distinct_canonical_variances   : <ordering_criteria_value2>
    | no_distinct_pairwise_variances    : <ordering_criteria_value2>
;
```

```
<ordering_criteria_value1> = {
         value                 : ( <number> | not_applicable )
    ?( , absolute_comparison   : ( <number> | unknown ) )
    ?( , calculated_comparison : ( <number> | unknown ) )
};

<ordering_criteria_value2> = {
         value                 : ( $integer | unknown | not_applicable )
    ?( , absolute_comparison   : ( <number> | unknown ) )
    ?( , calculated_comparison : ( <number> | unknown ) )
};

<optimality_criteria> = optimality_criteria: {
    ?( <optimality_criteria_member> *( , <optimality_criteria_member> ) )
};

<optimality_criteria_member> =
      phi_0                     : <optimality_criteria_value>
    | phi_1                     : <optimality_criteria_value>
    | phi_2                     : <optimality_criteria_value>
    | maximum_pairwise_variances : <optimality_criteria_value>
    | E_criteria                : { <E_value> *( , <E_value> ) }
;

<optimality_criteria_value> = {
         value                 : ( <number> | not_applicable )
    ?( , absolute_efficiency   : ( <number> | unknown ) )
    ?( , calculated_efficiency : ( <number> | unknown ) )
};

<E_value> =  "$integer" : {
         value                 : ( <number> | not_applicable )
    ?( , absolute_efficiency   : ( <number> | unknown ) )
```

```
    ?( , calculated_efficiency : ( <number> | unknown ) )
};

<pairwise_variances> =
    # function with domain_base=points and k=2
    pairwise_variances: <function_on_ksubsets_of_indices> ;

<canonical_variances> = canonical_variances: {
    no_distinct : ( $integer | unknown | not_applicable ) ,
    ordered     : ( true | unknown ) ,  # do we need this?
    value: [
             { multiplicity       : ( $integer | not_applicable ) ,
               canonical_variance : ( <number> | blank | not_applicable ) }
         *( , { multiplicity       : ( $integer | not_applicable ) ,
               canonical_variance : ( <number> | blank | not_applicable ) } )
    ]
};

<function_on_ksubsets_of_indices> = {
       domain_base       : ( points | blocks ) ,
       n                 : $integer ,
       k                 : $integer ,
       ordered           : ( true | unknown ) ,
    ?( image_cardinality : $integer , )
    ?( precision         : $integer , )
    ?( title             : $string , )
        maps             : [ ?( <map> *( , <map> ) ) ]
};

<function_on_indices> = {
       domain            : ( points | blocks ) ,
       n                 : $integer ,
       ordered           : ( true | unknown ) ,
    ?( image_cardinality : $integer , )
```

```
    ?( precision         : $integer , )
    ?( title             : $string , )
        maps              : [ ?( <map> *( , <map> ) ) ]
};

<map> = {
    ( <preimage> | <preimage_cardinality> | blank ) ,
    image   : ( <number> | not_applicable )
};

<preimage> = preimage : [
      $integer *( , $integer )
    # Removed: word ksubset, substituting with a seq of lists
    | [ $integer *( , $integer ) ] *( , [ $integer *( , $integer ) ] )
    | entire_domain
];

<preimage_cardinality> = preimage_cardinality : $integer ;

<info> = info : {
         software  : [ $string *( , $string ) ]
    ?( , reference : [ ?( $string *( , $string ) ) ] )
    ?( , note      : [ ?( $string *( , $string ) ) ] )
};

# scalars
<scalar> = <number> | $boolean | $null | unknown
<number> = $integer | $float | <rational> ;
<rational> = { Q : [ $integer, $integer ] } ;
```

Appendix B

Design DB Entity Relational Diagram

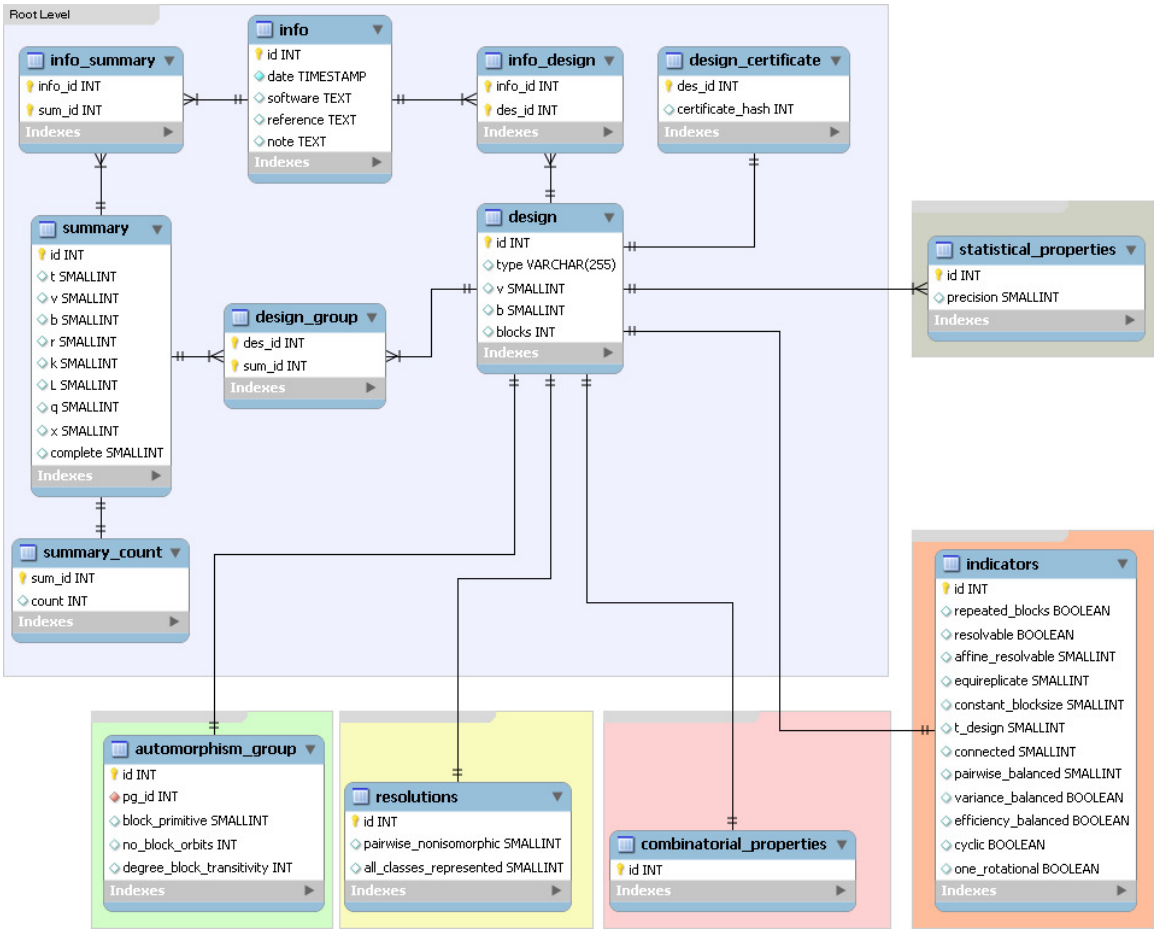


Figure B.1: Root Node of A Design

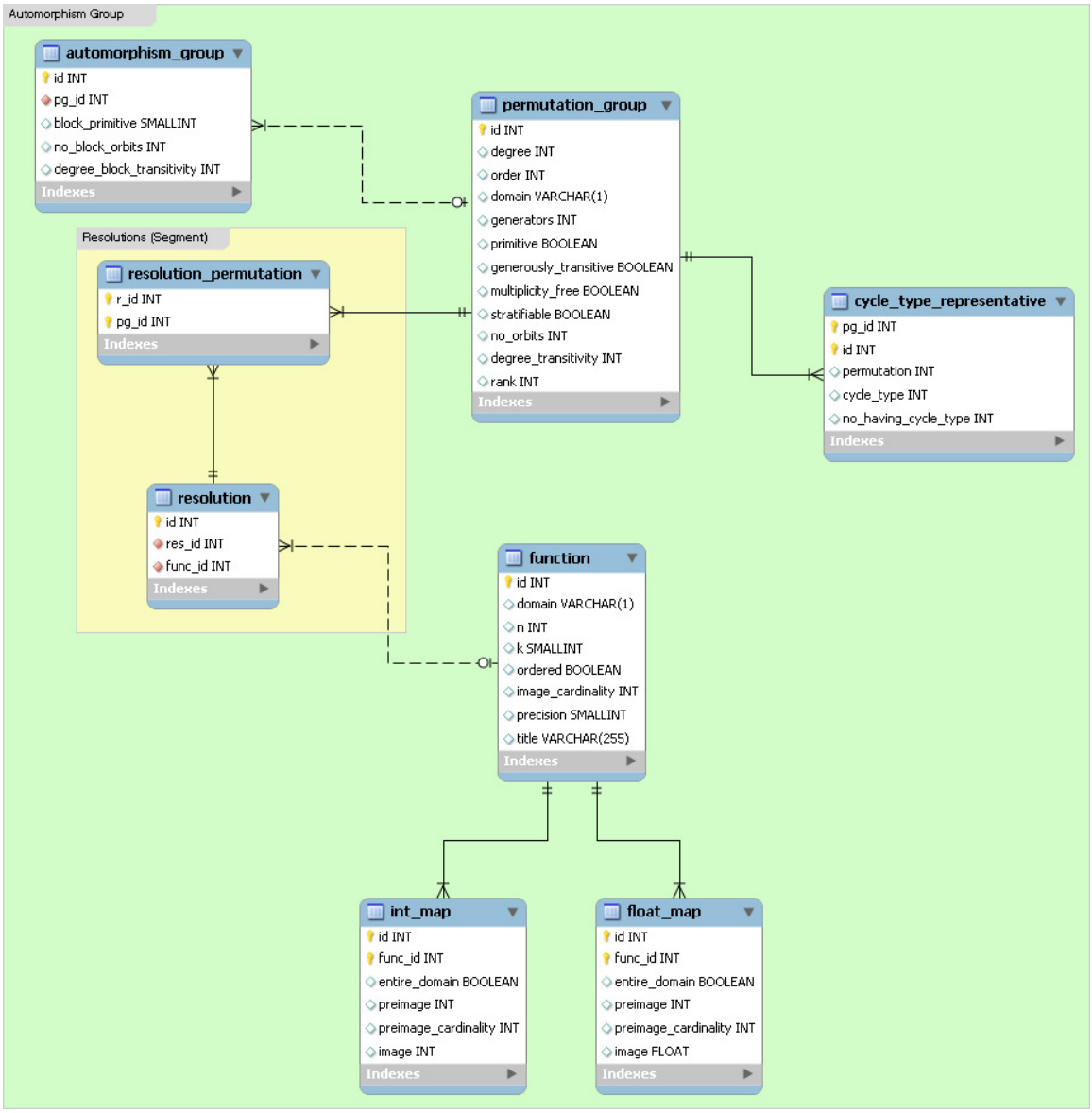


Figure B.2: Design Automorphism Group

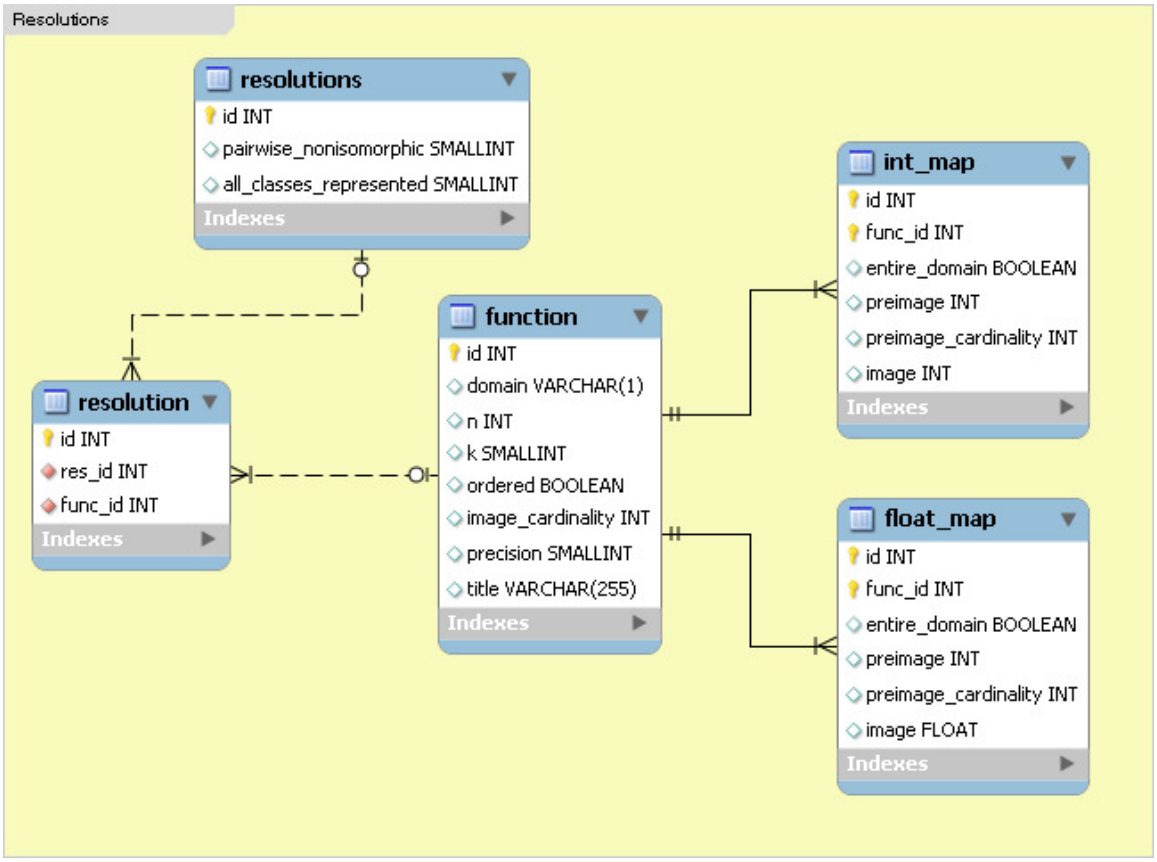


Figure B.3: Design Resolutions

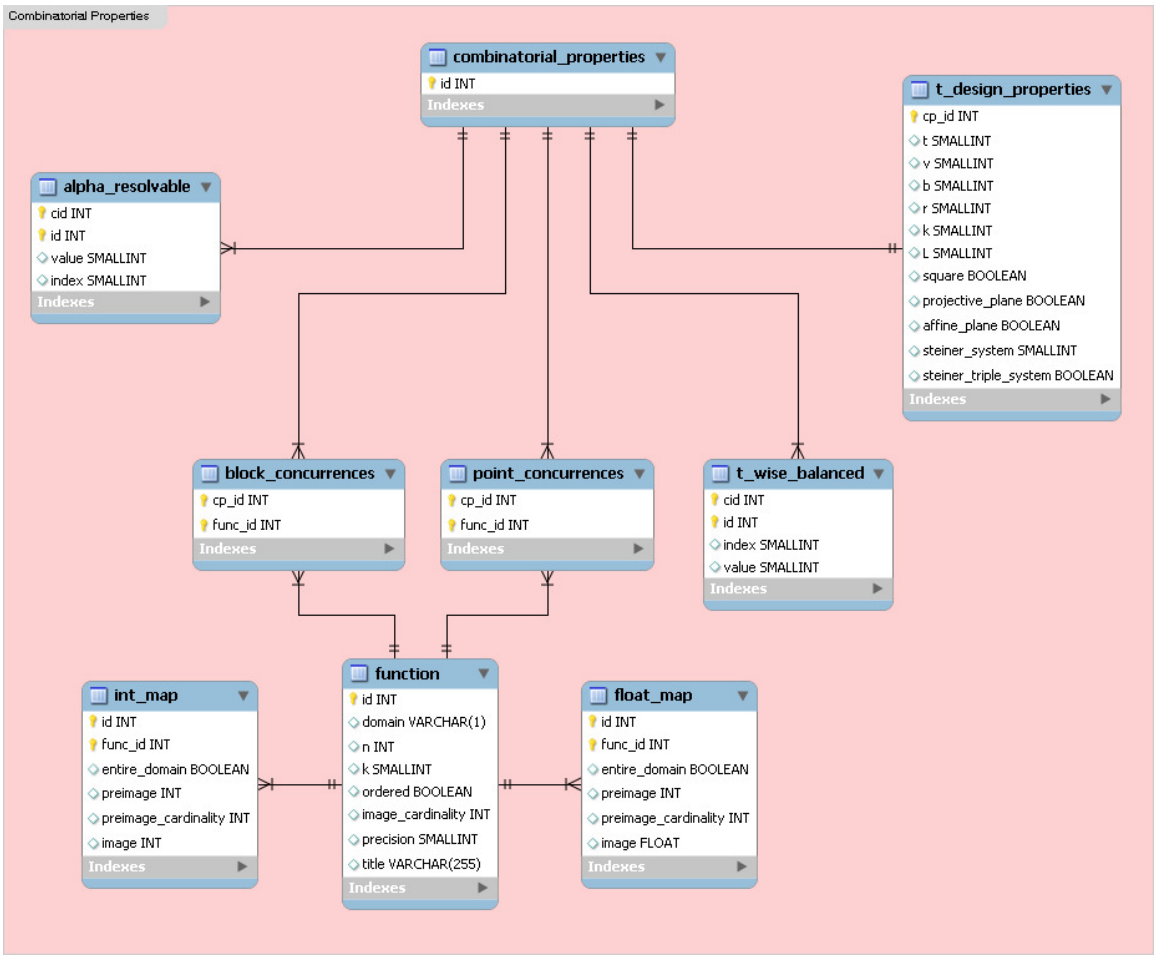


Figure B.4: Design Combinatorial Properties

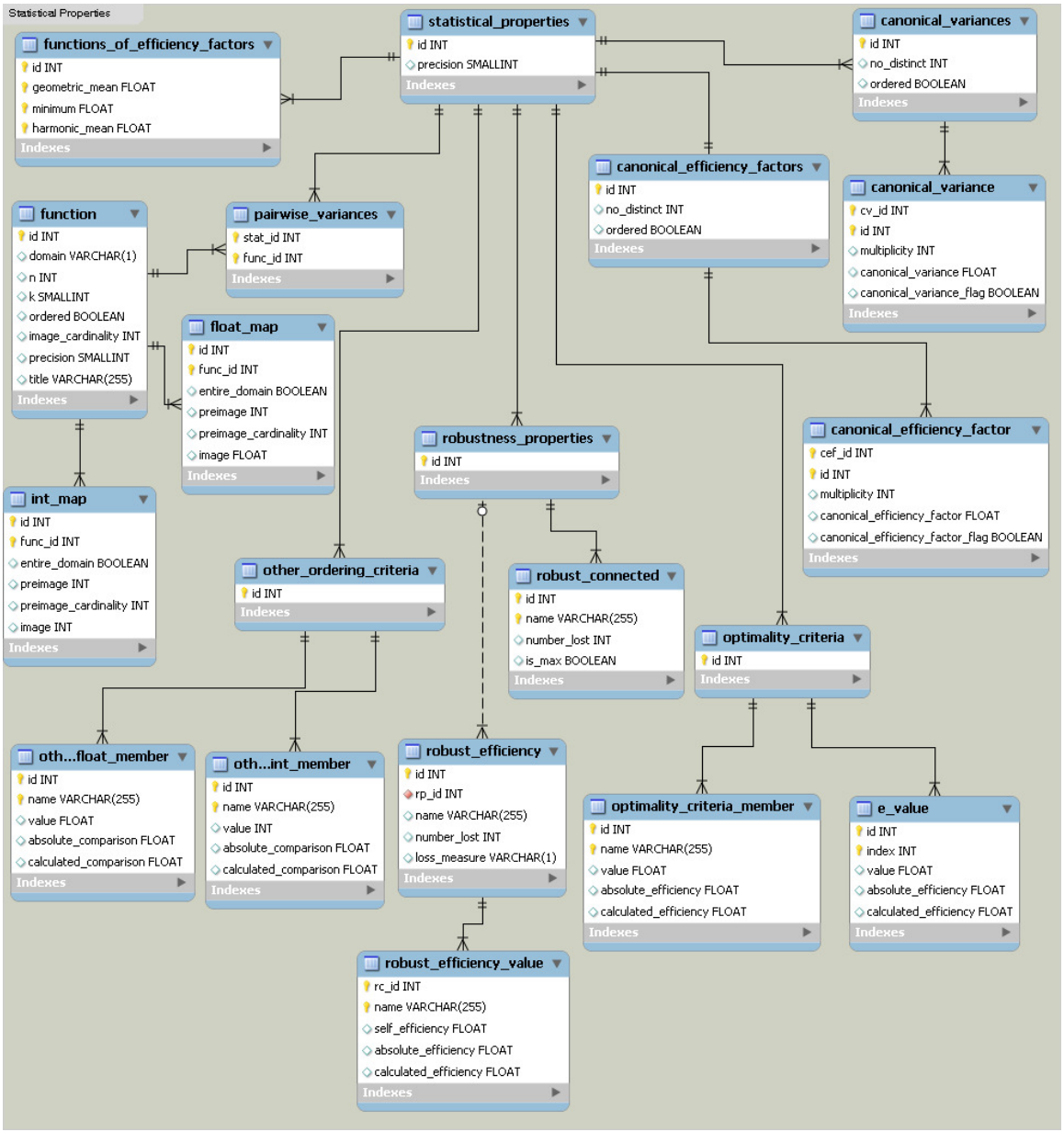


Figure B.5: Design Statistical Properties

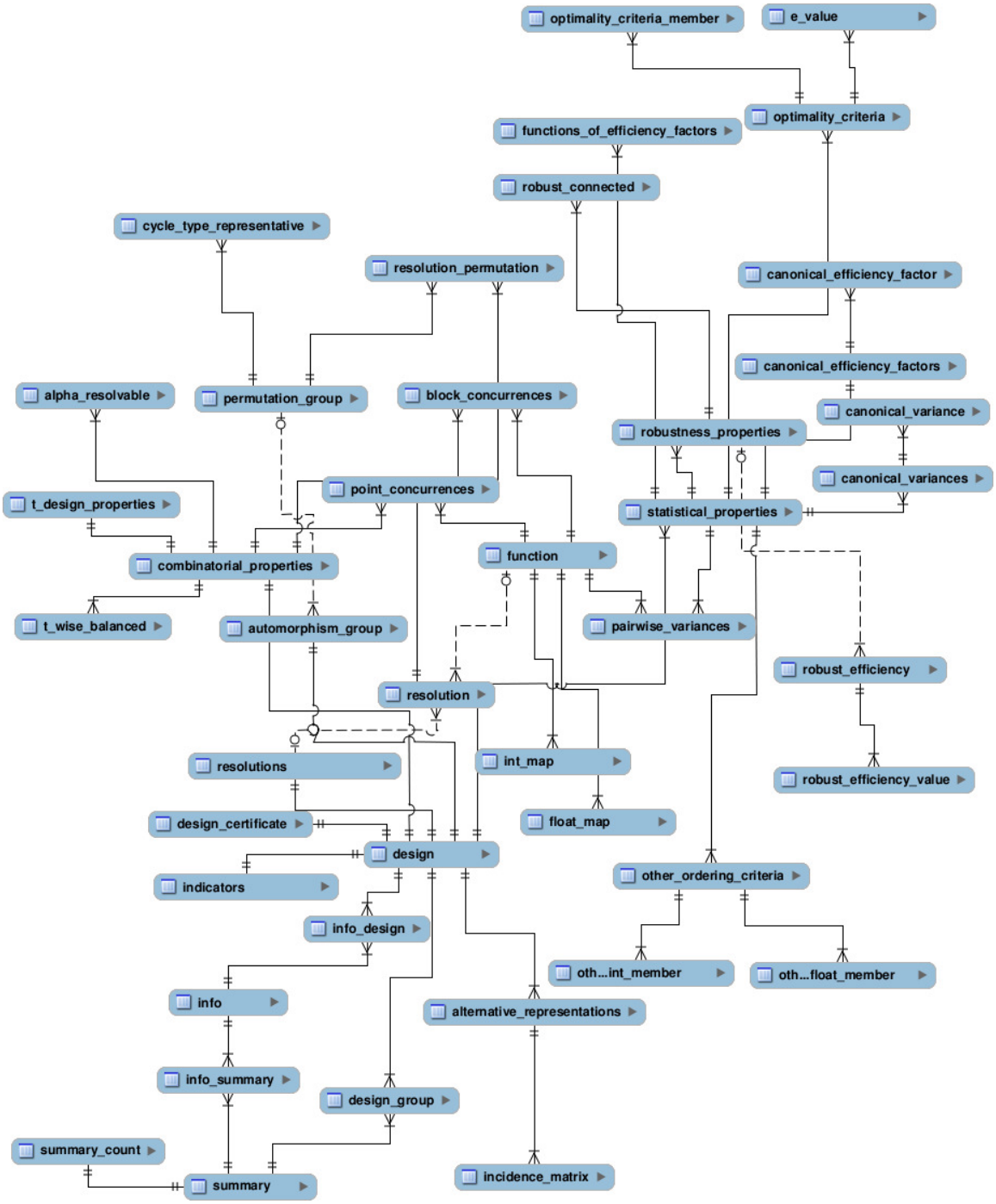


Figure B.6: Full Design DB ERD