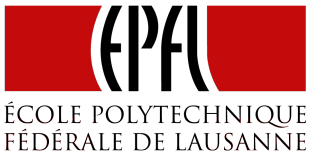
Indico: Behind the Scenes of CERN events



Ferhat Elmas
CERN Supervisor: **Pedro Ferreira**
EPFL Supervisor: **Karl Aberer**
School of Computer and Communication Science
EPFL

Master Project Report
Distributed Information Systems Laboratory
14 Mar 2014

Abstract

Indico is a web application which is used to schedule and organise events, from simple lectures to complex meetings, workshops and conferences with sessions and contributions. The tool also includes an advanced user delegation mechanism, paper reviewing, archiving of event materials. More custom tailored features such as room booking and collaboration are provided via plug-ins.

Indico is heavily used at CERN[1] and, in more than 10 years, very different features were added according to the needs of an even increasing number of users. However, the data store, ZODB, has never changed. ZODB has been very vital in accelerating development of many features but it's now a bottleneck in terms of scalability and elapsed time in feature development.

Indico currently requires a more scalable back-end to serve an increasing number of events while providing advanced features. The aim of the project is to replace ZODB with a highly scalable data layer to face this demand. The first part of project involves the analysis of different databases in the light of Indico schema, scalability and migration cost. The second part is bringing chosen database into production by development of a prototype that works in an incremental transition environment.

Keywords: back-end, CERN, databases, Flask, NoSQL, PostgreSQL, Python, scalability, SQL, SQLAlchemy, Web Development, ZODB

[1] `http://home.web.cern.ch/`

Contents

1 Introduction **1**
- 1.1 Overview - What is Indico? . . . 1
- 1.2 Motivation - Why change? . . . 2
- 1.3 Approach - How to change? . . . 3
- 1.4 Outline - What is Done and Next? . . . 4

2 Data Storage in Indico **5**
- 2.1 The Early Days . . . 5
- 2.2 The Decade of ZODB . . . 6
  - 2.2.1 The Age of Caching . . . 7
- 2.3 The Need for a New Storage . . . 7
  - 2.3.1 Indexing . . . 8
  - 2.3.2 An Example: The Dashboard . . . 8
- 2.4 Selection Criteria of New Database . . . 11

3 Survey of Candidates **13**
- 3.1 Main Database Types . . . 13
- 3.2 Object-Oriented Databases . . . 13
  - 3.2.1 WakandaDB . . . 14
  - 3.2.2 ZODB . . . 14
- 3.3 Relational Databases . . . 16
  - 3.3.1 PostgreSQL . . . 17
  - 3.3.2 MySQL . . . 18
  - 3.3.3 SQLite . . . 19
- 3.4 Column Family Databases . . . 21
  - 3.4.1 Cassandra . . . 22
  - 3.4.2 HBase . . . 24
- 3.5 Document Oriented Databases . . . 24
  - 3.5.1 MongoDB . . . 25
  - 3.5.2 CouchDB . . . 25
- 3.6 Key-Value Stores . . . 25
  - 3.6.1 Redis . . . 26
  - 3.6.2 Riak . . . 26
- 3.7 Graph Databases . . . 27
  - 3.7.1 OrientDB . . . 27
  - 3.7.2 Neo4j . . . 31
  - 3.7.3 Titan . . . 32

3.7.4 Broadness and Validity of Survey . . . 32

4 Databases In Action **41**
4.1 Relational Databases . . . 41
4.1.1 PostgreSQL . . . 41
4.2 Column Oriented-Databases . . . 42
4.2.1 HBase . . . 42
4.3 Document Oriented-Databases . . . 43
4.3.1 MongoDB . . . 43
4.4 Key-Value Store . . . 44
4.4.1 Redis . . . 44
4.4.1.1 Conclusion . . . 45
4.5 Graph Databases . . . 46
4.5.1 OrientDB . . . 46

5 Decision **48**
5.1 Chosen Database Type . . . 49
5.2 Chosen Database . . . 50

6 Implementation **52**
6.1 Big Picture . . . 52
6.2 Scope of Project . . . 53
6.3 Room Booking Schema . . . 53
6.4 ORM - Object Relational Mapper . . . 55
6.5 MVC - Model View Controller . . . 56
6.6 Models . . . 56
6.7 Distributed Queries . . . 57
6.8 Queries . . . 57
6.8.1 Complexity . . . 58
6.8.2 Portability . . . 58
6.9 Testing . . . 59
6.10 Controllers . . . 59
6.10.1 Forms . . . 60
6.10.2 More Flask . . . 60
6.11 Views and Templates . . . 61
6.11.1 Unicode . . . 61

7 Validation of PostgreSQL Decision **63**
7.1 Comparison of PostgreSQL to Document-Oriented Databases in Room Booking . . . 63
7.2 Database Size . . . 63
7.3 Documentation and Tooling . . . 64
7.4 Summary . . . 65

8 Conclusion **66**

References **68**

1

Introduction

1.1 Overview - What is Indico?

Indico (Integrated Digital Conferencing) is a web application for event management, initially developed as a joint initiative of CERN[1], SISSA[2], University of Udine[3], TNO[4], and University of Amsterdam[5]. Currently, most development is happening at CERN with occasional contributions from the outside, such as time zone support by Fermilab[6].

Indico events are divided into 3 types: lecture, meeting and conference as from simplest to the most complex, respectively. Lectures are one time talks while conferences provide a wider range of features.

In addition to events having a type, each event belongs to a specific category, which enables category-driven navigation between events. However, active categories can contain many events in which finding events may be cusbersome so time-driven (day/week/-month) view is combined with category view for a more powerful navigation. Sometimes time-driven navigation isn't enough, as any user, extensively uses Indico, can easily end up with tens of events in a daily view. To remedy this problem, personal dashboard was developed in which upcoming events are shown in sorted order to the user. Moreover, users can mark some categories as their favorites and Indico recommends categories to users by analyzing this information and the event history of users.

Indico has a huge number of features and some important ones:

- Customization of the event's website.
- Easily setting up access and modification rights: Every part of conference management requires different access rights. One contributor should be able to modify her contribution but she shouldn't be able to change others, for instance.
- A powerful fully customizable WYSIWYG registration form for events
- Custom forms such as preparing a survey to get feedback about event.
- Call for abstracts

[1] The European Organization for Nuclear Research
[2] International School for Advanced Studies
[3] http://www.uniud.it/
[4] Nederlandse Organisatie voor Toegepast Natuurwetenschappelijk Onderzoek
[5] `http://www.uva.nl/en/home`
[6] `https://www.fnal.gov/`

- Paper reviewing
- Rich text editing in Latex and Markdown
- Archiving files: pictures, slides, videos, etc. It's a document store as much as it's a meta-data store.
- Converting documents between different formats via an external conversion server such as automatically generating PDF from Microsoft Office slides.
- Exporting event meta-data in multiple formats: HTML, XML, iCal, PDF, etc.
- Extensible plug-in system to provide more advanced features and to integrate with other systems. Some widely-used plug-ins:
  - A powerful Room Booking system which enables management and reservation of rooms in a large organization
  - Powerful video conferencing by integration with Vidyo[1] and EVO [2]
  - Using instant messaging, XMPP, is easy from Indico
  - Payment processing for registration using different providers (PayPal, PostFinance, WorldPay, etc.)
  - Search for categories and events via Invenio[3] integration
- Fully embracing mobile with mobile version and mobile check-in application for events

As seen, Indico is absorbing new feature sets by integrating with other services and steadily advancing to become the de facto interface for most of the needs in terms of event organization and management.

1.2 Motivation - Why change?

The main driving force in development of Indico was the need for a service to manage events, a successor of CERN's own CDS[4]. Thus, rapid development of production ready software was a necessity. Since there were many features to be implemented, time spent in other parts of the system had to be minimized.

The beginning of the 2000s was the time objects and object oriented programming languages became mainstream. As a result, there was a tendency to try objects everywhere and data stores had their own lion's share. Thus, an object database as a back-end for web application was a reasonable decision at that time. However, object databases never took off as general-purpose solutions.

ZODB (Zope Object Database) was a viable option for Python in 2002. Since ZODB is an object database, it enables direct manipulation and transparently storage of objects.

[1] `http://www.vidyo.com/`

[2] `http://evo.caltech.edu/`

[3] Invenio, also developed at CERN, is a digital management library by providing classification, indexing and curation of documents

[4] CERN Document Server

This capability of certain data stores eliminates a layer between objects in the programming language and data in store which in turn saves a lot of time, money and complexity in development.

Fast forward to 2014, ZODB is far from having a medium sized community. What was saved before by using ZODB is now being paid to work around its own limitations. In the last couple of years, the net gain turned out to be negative which meant replacing ZODB with a more standard, powerful and scalable data store which will be the basis for new developments in the next decade of life of the tool.

1.3 Approach - How to change?

We are sure about the necessity of the change but change is costly and getting costlier every day because more feature are being developed, more systems are being integrated and scale is increasing. Thus, a wrong decision taken today may even kill the development of the tool so it should be well-planned: past mistakes should be analysed to prevent them happening again and emerging technologies should be examined to utilize.

Firstly, limitations of ZODB should be extensively studied because most of the time, leaving aging production system behind and writing it from scratch results in failure [1][2]. It's developed for more than 10 years, there is a huge body of accumulated expertise with working ZODB. As a result, many work around solutions are developed and which should be studied to make clear where it failed. Unless problems are made ridiculously clear, clean solutions can't be found[3].

We will present these reasons and how we tried to solve these limitations as in the existing system and technologies. After becoming certain that refactoring would be more cost-effective instead of pushing ZODB more and more, next step is to define features, that new data layer should be capable of. We have mainly used two things: where ZODB failed and interpolation of the growth.

In the golden era of data stores, started by Google[4][5][6] and Amazon[7] who needed to implement their own systems with extreme requirements; there are zillions of candidates, and even barely knowing each of them is a really difficult task. The goal is to collect and classify candidates that conform to criteria as much as possible.

After a first elimination and classification, the next step is to get familiar with each one via implementing a very small subset of Indico. Implementation is important because it is hard to estimate from abstract concepts alone. Even if the feature to be implemented is small, it makes it easier to predict the schedule and complexity.

When know-how about each system is acquired, the tipping point is choosing one or a subset of the candidates because, in big applications, each part has different characteristics and requirements which entail suboptimal solutions when using only one data store.

In either case, the transition phase is long and painful so it should be well-planned in terms of bugs should be fixed immediately, needs of the community and release cycle. Indico is an open source project and there is a vibrant community engaging worldwide. Thus, unlike a closed-source product, decisions can't be given only by CERN, the final decision maker, though. Once the schedule is carefully studied, the final bit is to implement the system conforming to the plan.

1.4 Outline - What is Done and Next?

In the following chapter, we will provide an exhaustive description of the work done from the study of limitations of ZODB to a new implementation using a new polyglot database system. Moreover, database change involves refactoring a huge body of code base so this project also aims to pay years of technical debt and to complete what we wish we had had in the beginning, such as a fully modular Indico. In conclusion section, how important steps are taken to modernise Indico code base and to prepare transition of Indico to Python 3 can be seen in addition to getting scalable by new back-end.

Even if important steps are accomplished, that's not enough because software is very volatile and requirements and technologies are quickly changing. While keeping basic principles like DRY and KISS, extending data system for specific cases will help with better scalability and facilitate the development of certain functionality such as statistics and social features.

2

Data Storage in Indico

In this chapter, we first look at the evolution of Indico's storage system so we mainly turn to ZODB to understand why it was chosen, what it accomplished and where it failed. In this chapter, the insight provided by studying ZODB will be the basis for a comparison of databases in the next chapter.

2.1 The Early Days

Since the early days, Indico was built around an object-oriented philosophy. In the early 2000's, when the project started, the Object-oriented world and Java in particular seemed to be heading towards a hegemonical position in the software world. Object-oriented philosophy managed to permeate into practically all fields of computer science, including, of course, web applications, which until then had been no more than collections of CGI scripts in several interpreted languages[1].

Choosing Python as the development language for a new web application could have seemed a risky option in 2002, when buzzwords like *Django* or *Flask* were everything but real, but risk has shown to pay off in the long term: the Python web ecosystem is now flourishing and Indico is part of this movement. This means that months of development time can now be saved in the implementation of certain complex features since a full ecosystem of libraries and applications that were not there one decade ago emerged.

From the beginning, one of the most important (if not the most important) dependencies of Indico has been ZODB. While practically all the main dependencies have been replaced over time, from XML libraries to web frameworks, not to mention JavaScript libraries and templating engines, ZODB has stayed in. It's not hard to imagine that this was due to the fact that ZODB occupies a primordial role in the software package, by providing a transparent persistence layer over which the business logic of Indico is implemented.

Once again, the choice of ZODB as data storage solution for Indico was daring, to say the least. Despite the growing popularity of the project in the early 2000's[2], non-relational databases were far from being considered mainstream. The enthusiasm around object-oriented data stores seemed to be increasing, but the following decade would show a clear predominance of relational databases over the competition.

[1] That may be the reason of popularity of PHP today

[2] mainly due to the increasing usage of the Zope framework

2.2 The Decade of ZODB

In 10 years of ZODB usage, a single serious incident due to a database problem was reported. It caused a service interruption of around one day, the only one to report in a decade, and absolutely no data loss. Many other minor incidents have happened, but never related to the DB itself or the technology behind it. In addition to that, there are no examples of data corruption to talk about. This said, it is not unfair to say that ZODB is a reliable product, a solid storage solution that can be trusted upon.

ZODB can be shortly described as a *glorified pickle store*. This is both its strength and its weakness: it is intrinsically bound to Python and its Object-oriented subsystem. This provides a great degree of transparency and the expressiveness of an Object-oriented approach, but sacrifices, first of all, portability and, additionally (but also of great importance), the performance of some operations.

As an example, let's take a very simple DB operation that is usually taken for granted in the relational world: getting the list of names of all registrants in a conference:

```
1 SELECT last_name, first_name
  FROM conference_attendants
3 WHERE conf_id = 1234;
```

This type of query can be performed in ZODB, but it will involve:

- Querying the DB for all the objects relative to attendants of the conference in question. This involves:
  - Retrieving the Conference object from the database
  - Reconstructing the object from its serialized form (pickle)
  - Extracting the list of objects from the Conference object and requesting each one of them from the DB
- Reconstructing (unpickling) the objects client-side
- Retrieving the attributes in question from the reconstructed object.

These are by no means slow operations. Specifically, Point 1 involves a significant amount of network latency (given that each object is requested separately) and Point 2 can be significantly slow if a large amount of objects is involved.

This is the main issue with ZODB: objects are separate entities and, even though there are tricks that allow us to group/retrieve them together, those are not without their own disadvantages.

Of course there are advantages to the object-oriented approach: it's easy to work with and there is no need for conversion mechanism that will map classes in the object domain to database tables and rows. This mechanism is normally called ORM[1] and is nowadays a common feature of many web frameworks.[2]

[1] Object-Relational Mapping
[2] Ruby on Rails, Django, Symfony, etc.

2.2.1 The Age of Caching

Over the last 5 years, Indico has gradually become ubiquitous at CERN. The addition of a room booking module and later video-conferencing capabilities has for sure contributed to the broadening of the tool's audience. This has quickly increased the load over the system, and consequently the database. After a round of DB optimization fixes and several studies into ZODB, developers quickly realized that future growth needed to be sustained by a different strategy.

Performance improvement could be easily achieved in two different ways:

- Faster DB access - this would depend on both the speed of the network connection between the DB and the web workers and the performance of the DB server instead. The choice was made of equipping the Indico DB server with SSD devices.
- Less frequent DB access - this would rely on keeping more data on the client side and asking for DB content less often, also known as *caching*, and ZODB already has a client cache. Unfortunately, it is per-process cache and state is not shared between caches. An application-level caching layer was implemented, which can use different caching backends.
- DB replication - this could provide some relief to the main DB server during activity peaks, specifically by adding a second ZODB server that would mirror it. Web workers wanting to retrieve data could then contact to the second server instead of the first one, thus freeing the latter from keeping additional connections open for them.

Unfortunately, no simple and stable ZODB/ZEO replication mechanism was available at the time. The only exception was ZRS, which was a commercial product. An evaluation version was requested and it proved to be an effective solution. Its high price was the only deterring factor. It was decided that, for the moment, the first two solutions would be implemented. ZRS has recently been open sourced by *Zope*, which makes it a quite viable option for the future if Indico sticks with ZODB.

After a first implementation of caching with recourse to files, memcached support was finally added, which allowed for further performance improvements. Later, *Redis* support would be added.

At the time of the writing of this document, caching is being used in several parts of the application, including conference timetables and the room booking system. Furthermore, new (more experimental) features such as the Dashboard are making use of specific forms of caching (*Redis*, in this case) in order to keep a secondary, fast, easy to query secondary store. However, it also comes with its own problems; data must be synchronized with the main database, which requires extra overhead for writes; and the maintenance cost of extra code.[1]

2.3 The Need for a New Storage

The evolution of Indico's feature set has been increasingly towards a more personal, user-centric application with an emphasis on the events that matter to each particular user.

[1] 7 Lua scripts for server-side queries

The User Dashboard is a perfect example of the kind of feature that users can expect in the future from the system: a page that allows them to see which events are going on that involve them, as well as those taking place in their favourite categories.

Needless to say that the implementation of this kind of functionality implies the existence of mechanisms that allow a significant volume of information to be quickly queried and all matches retrieved. It is necessary to find out which events match a specific condition in a universe of hundreds of thousands (or potentially more) of them. This brings us to the problem of indexing.

2.3.1 Indexing

One of the most important features of a relational database is that arbitrary queries can be executed at any time. This allows for a great degree of flexibility, but makes things less predictable on the server side - a server has to know what to expect, otherwise it will take too much time going through all elements and checking whether they fit the criteria. Fortunately, most RDBMSs[1] support database indexes and allow them to be created at the stroke of a key. Indexes allow results to be retrieved much more quickly at the cost of a small performance penalty on write operations.

ZODB provides developers with several database-targeted data structures. Examples of that are *B-Tree* and *TreeSet* (implemented using B-Trees), two structures aimed at large volumes of data that need to be stored in chunks and accessed in sub-linear time. B-Trees are implemented on top of the Object-Oriented storage paradigm of ZODB (in separate *buckets*, the nodes of the B-Tree). One important detail is that the ZODB server application implements no additional logic regarding these data structures - it handles objects in a pretty transparent way, simply retrieving them as they are requested. This is not completely true, as conflict resolution is done by the database itself and can be overridden by custom library code. All searches, comparisons and even changes on DB objects are, however, performed on the client side, and only then reflected in the database.

The simplified logic on the server side together with the absence of a feature that can be compared to "stored procedures" leave the developer no other option than implementing everything on the client side. While this can be seen as a way of keeping things simple, it is highly inefficient as it implies the presence of all the relevant objects in the local context. This can be extremely slow for large data structures.

As a result, while implementing indexes in ZODB is possible, these are also potentially much slower than their relational counterparts, not to mention that they have to be implemented at the application level, even though ZODB can persist them for future use. There are tools that help developers with this task, such as *zope.index* and *repoze.catalog*, but they do not solve the problem of having to retrieve all needed information to the client side.

2.3.2 An Example: The Dashboard

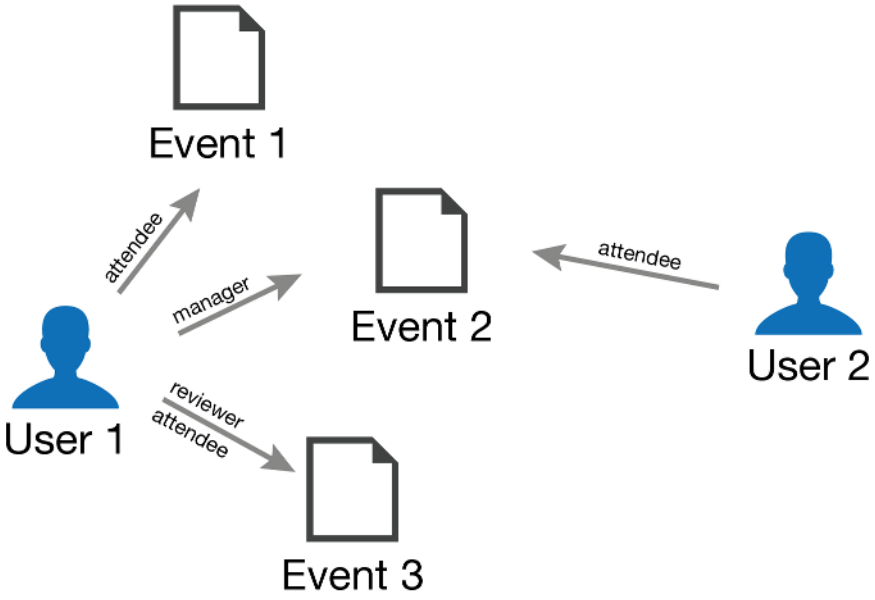
The User Dashboard was the first Indico module to be implemented on a storage mechanism that is not ZODB-based (Redis). While ZODB remains as the primary source of information for user-event relations, Indico is fetching this data directly from a Redis

[1] Relational Database Management System

server, and updating it in parallel with Indico. Redis acts here as a *write-through* cache, a secondary data store that is, however, always up-to-date.

There are reasons for this choice. They are mostly connected to the relation between the different entities in the problem.

Figure 2.1: Simple User and Event Relationship Example



The Dashboard is basically a time-ordered display of the relationships between a particular user and certain events. For instance, *User 1* is an *attendee* of *Event 1* or the *manager* of *Event 2*. Logically, other users can share the same role in the context of the same event - it's a many to many relationship. ZODB has no problem in defining many-to-many relationships - its object-oriented approach allows references to be added pretty much anywhere, very much like in a graph - that is not the issue. However, there are actually two problems:

1. It has already been mentioned before that ZODB fetches objects separately, so it is impossible to retrieve 100 pickled persistent Conference objects in one go.

2. Indexing, once again, is hard and not optimal

A possible (naive) approach for a ZODB-based solution to this problem would be creating a B-Tree, ordered by *time* and *event ID*, containing the conference objects that relate to a particular user. If we wanted to concentrate everything in a single B-Tree, we could just use a composite key such as (`user_id, timestamp, event_id, role`). This would allow us to effectively query events by *user/timestamp*. Still, it would be necessary to ask for and unpickle every single object for which a reference is returned (problem 1). Now, let's assume that problem 1 has minimal impact in terms of performance (which is not exactly true, but let's assume it just for the sake of the argument):

- Querying by user would be OK - It's a simple range query on `(user_id, 0, '', '')`

- Querying by timestamp for a particular user would be OK: range query on `(user_id, timestamp, '', '')`
- Creating a new entry would be trivial - it would just be a matter of adding a new key to the tree, and the worst case scenario would be a bucket split/join operation, which would be far from disastrous performance-wise
- Deleting a user would be OK - it would be a matter of finding out all corresponding keys using a range query and deleting them (*O(N)*)
- Detaching a user from a role would be possible in *O(N)*
- Completely detaching a user from a particular event would be simple, as it is a sub-problem of the previous problem

There are, however, some operations that are not as simple:

- If **an event gets deleted**, one will need to remove all index entries that refer to it − using this approach, that means going over all `user_ids` and checking whether the event is present
- Same happens if an event changes start date - the timestamp will change and one will need to update all corresponding entries

These are potential blockers, since event deletion is not such an uncommon operation. Of course there are known solutions to this problem, such as:

- Marking the event as deleted and excluding deleted events from query results. The deletion of "old" events from the index could then be made in the background by a periodic job that would go over all users. This wouldn't solve the start date issue, though.
- Having a reverse-lookup index that would map event ids to the corresponding keys in the primary index. This means that we would know that *Event 2* has index keys `(User 1, 12365, Event 2, manager)` and `(User 2, 12345, Event 2, attendee)`. This helps with both situations, but it's far from ideal as one would have to maintain two indices instead of one.

Similar problems have already been solved in the past using the second approach. In fact, a helper module (*Catalog*) was written, based on ZODB B-Trees, that makes it easier for developers to deal with indexes, by abstracting some of the needed operations (like, for instance, maintaining a reverse-lookup index). However, debugging such index code is not an easy task.

To sum up, saved data is hierarchical and contains a high amount of relations between entities so that they can be accessed from many different points. Therefore, the ability to query arbitrarily and to do projections[1] is vital from the performance point of view. Overcoming this problem is possible via caching but it introduces the problem of consistency of data and brings in the burden of maintaining extra code which means developers are working to solve issues created by the database, not for new features which

[1] getting subset of properties of object without loading whole objects

Table 2.1: Limitations of ZODB

Limitation	Explanation
Ad-hoc queries	Objects can't be queries by properties
Indexing	Not automatic and left to application developer
Caching on the Server Side	No cache on the server side, only operating system cache
Replication	Mostly single-point of failure. ZRS provides master-slave replication. It was commercial but open-sourced.
Community	Niche product so small number of people are using it. As a result, documentation is lacking.

take advantage of it. Rapid development, the reason why ZODB was chosen at first isn't so any more. Moreover, ZODB's community is small, documentation and development speed is lacking. Last but not least, it is still a niche product even after 10 or so years.

It seems to be clear, from what was mentioned above, that ZODB is not an optimal solution for this class of data problem.

2.4 Selection Criteria of New Database

We have looked at ZODB limitations and here, we try to define criteria which will guide us to compare different databases.

We have come up with 6 main categories and small deal breakers: Here is main categories:

1. *Availability*: Replacement technology must be open source since Indico is an open source application and users shouldn't be forced to use a commercial solution. Moreover, the license used by the database in question should take into account future plans for the project (such as possible commercial services, etc.)

2. *Scalability / Replication*: Scalability is the main driving factor for the replacement of ZODB so it is a must in the system due to the increasing usage and popularity of Indico. Te database should have room for extensibility for the foreseeable future.

3. *Easiness of Use / Development*: The chosen database should have good tool support for Python, main development language of Indico, and it should facilitate the time to implement new features so it should be transparent as much as possible like ZODB. The complexity of a feature should come from application logic itself, not the database.

4. *Transactions / Consistency*: Writes touch many entities at the same time due to dependencies between entities. Thus, possessing transaction capabilities is very important in a database , as a way to keep data consistent.

5. *Community / Momentum*: Indico aims to solve the conference management problem, not the nitty gritty use-case details of a specific database. When a problem is encountered, a respective solution should normally have been found before. Successful deployments of the products in other contexts are an important measure of this requirement.

6. *Cost / Exit Strategy*: Transition costs should be minimized and the project should not in any way become a "hostage" of the chosen technology to decrease the exit cost when the time to pay the price of *easy solutions* aka. *technical debt*[1] has come.[8][9][10]

It is easily seen that ZODB isn't doing good because it is clearly successful at 2 criteria of 6; namely, being open source and possessing transaction support.

[1] eventual consequences of poor software architecture and development in code base

3

Survey of Candidates

In this chapter, we try to look into as many databases as possible, in order to compare their strong and weak points and to find the best possible fit.

3.1 Main Database Types

Database systems specified in this document are mainly divided into 6 categories, which try to group current technologies according to the data structures and storage/querying strategies they employ:

1. Object-Oriented
2. Relational
3. Column-Family
4. Document-Oriented
5. Key-Value
6. Graph

Each database type and possible candidates from each type are explained further.

3.2 Object-Oriented Databases

Object-Oriented databases bring the capabilities of object-oriented programming languages into the database world. Objects in programming languages can be directly stored, modified or replicated because these databases use the same object models as the corresponding programming languages. Unlike relational databases, there is no layer that converting objects back and forth. However, that requires database to be tightly integrated with programming language so that it provides a more transparent storage.

Object databases have been around since object-oriented programming was made popular by Smalltalk in 1970s. Although they have a long history, they become prominent with the Java "invasion" of software world. Even if the place of Java and object-oriented programming in software world are obvious, object-oriented databases have never been able to become mainstream like relational databases. As a result, there are not very

successful products which leaves us with small number of choices to consider. ZODB and WakandaDB are the main viable systems. These are the ones that fit our context, not necessarily the most "viable". For instance, *db4o*[1] is a popular object database for Java and .NET but it's unusable in Python ecosystem.

3.2.1 WakandaDB

WakandaDB is very new - only half year passed since first public stable release at the time of writing. It pushes forward the motto *JavaScript is everywhere*, which is a relatively recent phenomenon, pioneered by *Node.js*.[2]

WakandaDB is an ambitious project because developers are trying hard to create a suite of tools that will work together seamlessly and will come handy in the life cycle of an application. Currently, WakandaDB comes with:

- *Wakanda Studio*: IDE[3] with full featured debugger
- *Model Designer*: Easy creation of classes and schemas
- *GUI Designer*: Makes it easier to create visual interfaces
- *Web Application Framework*: Plug and play framework to use WakandaDB with a REST API

It is much more than a simple database product. This may be seen as an advantage since these tools save developers from tons of problems and details but it isn't actually perfect in terms of Indico. It forces its own custom stack but chosen database should coexist with other parts of Indico. Database change is already a big project in its own so tool chain and framework replacement aren't expected. Furthermore, its JavaScript push isn't nicely fits into Python stack of Indico but we should also note that JavaScript percentage in the code base of Indico has been increasing clearly with interface renovations.

If we were starting a web project now, WakandaDB seems a promising option with tooling and feature of totally embracing web. However, it is a bit dangerous even that situation since it's too early to play on WakandaDB as seen from lack of success stories and big deployments.

3.2.2 ZODB

Firstly, ZODB is also considered as a candidate because if we couldn't find a better option that is worth of replacing, then we would continue with it and look for work around solutions to its problems.

The Zope Object Database is an object-oriented database that transparently persists Python objects. In the beginning, it was just part of web application framework but later it is extracted to be used independently.

ZODB is a directed graph of Python objects where its starting vertex is a *dictionary*[4] called *root*. Objects must be attached directly or indirectly to the *root* in order to be

[1] `http://www.db4o.com/`
[2] A platform to build scalable network applications, which is itself built on Chrome's runtime
[3] Integrated Development Environment
[4] map, hash, etc.

Table 3.1: WakandaDB Object-Oriented Database

Criteria	Feature
Availability	Dual license, AGPLv3 and commercial.
Scalability Replication	It's in infancy of development and stabilization. There aren't enough successful and big deployments. Since its target is specific, there are no benchmarks to compare its performance characteristics.
Ease of Use Development	JavaScript is the only "first class" supported language. Server supports REST API so in theory every language can be leveraged via HTTP but Python isn't fully ready for the time being.
Transactions Consistency	Transactions, even nested, are fully supported with the benefits of ACID guarantees.
Community Momentum	New project, so it's normal that the community is just starting to get together but as seen from version control history, development isn't going on fast enough. Using only one language in every part of the stack is a huge time-saver and JavaScript is living its golden age so the choice of *JavaScript Everywhere* makes sense, but results aren't obvious yet.
Cost Exit Strategy	Adapting the whole tool chain would be too much but if only the server component is utilized, cost can be reduced. Even using only the server part makes requires extensive JavaScript knowledge and leaving accumulated Python know-how. Exit would be easier compared to ZODB because there is an official MySQL connector which can export all data to be used in MySQL with little effort.
Score	★★★☆☆☆

persisted. Thus, persisted objects can be accessed by starting from *root* and following links its children.

ZODB has powerful features in addition to being a transparent store:

1. ACID transactions[1]
2. Version control for objects, which implies the need for periodic *packing* operation. Every version of an object is stored, which dramatically increases database size. Packing operation includes taking back-up of database first so as to keep every version of the objects and then shrinking the database back to the bare minimum. Unless this operation is repeated regularly enough, disk space will easily be filled. Under heavy write traffic, this may be a serious problem and it does for Indico.
3. Pluggable storage - choice of different back-ends
   (a) File: Only one file in the file system and it's currently in use in most (if not all) Indico servers.

[1] A set properties: Atomicity, Consistency, Isolation and Durability

   (b) Network aka. ZEO (Zope Enterprise Objects): It allows multiple client processes to access database concurrently which enables easier scaling and it also implements ACID but the ZEO server itself becomes a bottleneck and a single point of failure.

   (c) RelStorage: Brings independence from one particular product at the expense of extra complexity and overhead. It allows replication and fault-tolerance which is good.

4. Client caching: A partial solution for the lack of caching on the server side

5. MVCC: Reads aren't blocked while only one write is allowed such that multiple writes are allowed as long as they don't cause version conflicts.

6. Replication via *ZRS* (Zope Replication Services): It was recently open sourced (May 2013). It was totally out-of-picture before due to its elevated price. ZRS recovers single point of failure by providing hot-backups for writes and load-balancer for reads.

It has been around since late 90s, so it is reasonable to call ZODB as the most mature Python object database in the wild. It powers some successful products, Indico being one of them as well as Plone[1].

One interesting drawback of ZODB being very tightly coupled with the code base is that each object is stored using its full class name on disk. That prevents moving classes around without a migration step. *MaKaC*, Make a Conference, was the name that was chosen for Indico in its early development stages. As a result, source code still contains a lot of classes under `MaKaC` package or prefixed by `MaKaC`. They should be moved into `indico` package. The migration to a new back-end will hopefully help remove most (if not all) of these references.

3.3 Relational Databases

A relational database is a collection of tables of data items (*rows*) that follow a relational model. Each table has a strict schema which specifies data parts, their types and their relationships to each others. Each row has a *primary key*, composed of one or multiple columns, and which uniquely identifies itself. Relationships can be established within a table or between tables via a *foreign key* which is a pointer to primary key of some table. Foreign keys provide various levels for normalization of tables. Normalization is a methodology that avoids data manipulation anomalies and loss of integrity by preventing redundancy and non-atomic values in table design. After the schema is normalized, which potentially makes it less intuitive compared to non-normalized data, arbitrary queries can be executed on the schema. This ad-hoc query capability is the most important feature of the relational world and nicely fits into evolving and agile software development. However, what is won by dividing data is lost when data spanning multiple tables is requested because data can only be recreated by joining tables, combining rows with referenced foreign keys, which are very costly in terms of performance. Even though normalization requires touching multiple tables at the same time to join data, relational

[1] `http://plone.org/`

Table 3.2: ZODB Object-Oriented Database

Criteria	Feature
Availability	Zope Public License
Scalability	Although there are solutions like ZEO, RelStorage and ZRS (recently); lack of serve cache, ad-hoc query capability and indexing limits scalability.
Ease of Use Development	ZODB is more like a library than a standalone program, which makes setup easy. It's tightly integrated with code, which is both an advantage and a disadvantage. While it makes things easier and most update functionality automatic, it can be at times tricky. A developer should know its inner workings in order to optimise performance.
Transactions Consistency	ACID
Community Momentum	Even if it's the most well-known and mature object database in the Python world, it's a niche project. The documentation and surrounding tool chain are not very complete despite its long history.
Cost Exit Strategy	Starting to use ZODB is easy at its simple setup and tight integration with Python. Getting rid of it can be very complicated.
Score	★★★☆☆☆

database systems are fully ACID compliant with tunable guarantees at e.g. row or column level. Furthermore, with the advent of big data requirements, replication is supported by default. Finally, schema design and normalization level are extremely important for sharding. Having both replication at default and sharding by default or application level, relational database can scale to relatively huge traffic.

Relational databases are some of the most battle-tested products of the last 20 years and have been well-studied for the last 40 years. As a consequence, there are multiple very successful deployments, and a huge active community with extensive know-how, and powerful tools.

3.3.1 PostgreSQL

Some general properties are provided here but an explanation of in-depth features will be given later since PostgreSQL was the winner of our survey.

Table 3.3: PostgreSQL Relational Database

Criteria	Feature
Availability	PostgreSQL License (BSD or MIT-like) and clear roadmap and open development
Scalability	Scalability was tested with successful deployments: Amazon, Dropbox, Heroku, Instagram, Reddit, Skype, NASA, Yahoo
Ease of Use Development	Object-relational mapping (ORM) layer is needed but there is a well developed and supported Python library, SQLAlchemy[1], which is in production at Dropbox, Yelp, Uber, OpenStack.
Transactions Consistency	ACID
Community Momentum	One of two large communities in the current database ecosystem, rivaling with MySQL. However, MySQL is losing blood since the acquisition of Sun Microsystems[2] by Oracle[3] because main developers[4] employed by Sun Microsystems left the company and forked the project such as MariaDB. Developers concern about strategy followed by Oracle and possible commercialization. Although many frameworks and PaaS providers support relational databases in general, they recommend PostgreSQL usage notably as a result of standards compliance.
Cost Exit Strategy	The integration of the ORM layer makes the code back-end agnostic unless back-end specific extensions are used. However, special features and types make the code more efficient, such that especially *array* type enables ORM to load tree-like relations into Python collections.
Score	★★★★★★

3.3.2 MySQL

MySQL is the most popular relational database by far. MySQL owes its popularity to an investment in performance more than in features. PostgreSQL did it the other way around. This is a long story which has a dramatic end because PostgreSQL increased its performance in recent versions while MySQL has implemented most of the lacking features. Even if the gap is closing in simple queries, PostgreSQL performs better in complex queries involving sub-queries and/or multiple joins via its well-studied query optimizer.

There is a subtle detail in this comparison. PostgreSQL is an integrated database, a single block of code base but MySQL is composed of two layers, SQL (parser, optimizer, etc.) and storage layer (responsible for actual data storage, modification, etc.). Performance characteristics, features and transactional behaviour of storage layers are highly different. Thus, carefully establishing which storage is used is important for a fair com-

parison. In that respect, MySQL supports multiple storage engines; namely, InnoDB[1], BerkeleyDB[2], TokuDB[3] and MyISAM[4], etc. but considering ACID requirements of Indico, development activity, and the need for a far comparison with PostgreSQL, InnoDB is our main concern.

Table 3.4: MySQL with InnoDB vs PostgreSQL Comparison

Winner	Features
=	MySQL (InnoDB) is very similar to PostgreSQL in terms of stored procedures, triggers, indexing and replication.
PostgreSQL	Richer set of data types, better sub-query optimization, fully customizable default values and better constraints for foreign keys and cascades. Standard compliance. PostgreSQL has a clearer roadmap and a more permissive license compared to MySQL because, after acquisition by Oracle, MySQL has remained as open-source product but the community hasn't remained intact. As a solution, some concerned experienced MySQL developers forked the project, which resulted in multiple MySQL-like products in the market, such as MariaDB[5], Drizzle[6], etc. Moreover, with the help of MySQL's layered architecture, different companies are providing similar but diverse products which reuse the SQL layer and plug in a custom storage layer such as Percona Server, TokuDB, etc. As a result, many tailored products emerge but this divides community while PostgreSQL community as a whole has focused on making one code base perfect.
MySQL	Advanced concurrency primitives like REPLACE - check and set atomically, and richer set options for horizontal partitioning.

In short, MySQL is a very successful relational database that has a flexible design and is targeting performance at the expense of a full standard compliance and additional features. Its acquisition by Oracle in 2010 introduced, however, uncertainty regarding its long term future.

3.3.3 SQLite

SQLite is an embedded relational database management system. In layman terms, it is a tiny C library that can be dynamically linked. Unlike MySQL or PostgreSQL, it's not a separate process that is waiting for requests. On the contrary, it is a part of the application and can be pictured as an independent module within the latter. and can be used through function calls.

All information related to the database is stored into a platform agnostic file. This is similar to the file storage of ZODB, and as by just copying that file, that same database can easily moved to wherever it is required.

[1] `https://dev.mysql.com/doc/refman/5.7/en/innodb-storage-engine.html`
[2] `http://www.aosabook.org/en/bdb.html`
[3] `http://www.tokutek.com/products/tokudb-for-mysql/`
[4] `http://dev.mysql.com/doc/refman/5.7/en/myisam-storage-engine.html`

Table 3.5: MySQL Relational Database

Criteria	Feature
Availability	Dual license (GPLv2 and commercial)
Scalability	Scalability has been proven to be good, is used by companies such as Facebook, Google and Twitter, at a large scale.
Ease of Use Development	An ORM layer is needed but code targeting for PostgreSQL in SQLAlchemy is generally usable by MySQL unless dialect-specific features of PostgreSQL are used extensively.
Transactions Consistency	ACID in InnoDB
Community Momentum	According to various rankings[1], MySQL is the most popular open source database with a still huge community. Its acquisition by Oracle has damaged its reputation for openness. Forks emerged and have benn following closely with upstream changes until now but after the stabilization of the long waited 5.6 version, first version to come out during the "NoSQL craze", forks started to diverge. MariaDB versions were one-to-one mappable with MySQL versions until 5.5 but since 5.6, they even chose to change their version scheme to start with 10.0.
Cost Exit Strategy	The ORM layer makes the replacement of a relational database easy, so what is said for PostgreSQL is also valid for MySQL. Their differences are getting subtle and unimportant. Moreover, MySQL can be easily replaced by any of its forks, at least while development is still synchronized with upstream.
Score	★★★★★☆

Writes lock this file which means writes are executed sequentially, but reads don't require such a locking mechanism, which enables concurrent reads. If multiple writes happen concurrently, only one of them can gather the lock and others will fail with a specific error code. It's easy to see why it is much simpler than PostgreSQL and MySQL.

It's a self-contained, server-less[2], zero-configuration relational database which makes it a universal solution for small needs. But it's not capable of serving high load in production. Since it's easy to use, it is used by many high caliber products such as Android and Firefox. In the context of Indico, it makes perfect sense for testing. However, many features provided by PostgreSQL and MySQL are missing in SQLite so, while keeping MySQL or PostgreSQL as main data back-end, using SQLite even for tests brings additional complexity in what contains back-end dependent operation such as date and time handling.

[2]no daemon for waiting client connections

Table 3.6: SQLite Relational Database

Criteria	Feature
Availability	Public Domain, Universal and even comes with Python.
Scalability	A big problem because it is designed to be an embedded light database system. It may be thought as a testing-only solution since no configuration is required and the database can totally be kept in memory for better performance. The faster tests are runnable, the more frequently test are run.
Ease of Use Development	An ORM layer is needed, since it's a relational database system and luckily, it is supported by SQLAlchemy.
Transactions Consistency	ACID
Community Momentum	Due to its popularity (Android, Chrome, Firefox, Opera, Skype, ThunderBird, etc.), the community is huge and it is well tested[1] and documented.
Cost Exit Strategy	The ORM layer provides good abstraction. Although SQLite implements most of SQL standard, some important ones are omitted.[2] such as triggers, full outer joins (left outer join is implemented and will be used in Indico), writable views and security (most vital lacking feature). We can somewhat live without these features but implementing Indico schema with exclusive database-level write lock doesn't scale.
Score	★★★☆☆☆

3.4 Column Family Databases

Column-family databases are designed to handle large amount of data across many commodity servers at very large scale (e.g. Facebook). They provide auto-sharding and master-master replication so as to provide low latency and no single point of failure at the expense of dropping joins (favoring denormalization) and ACID guarantees. Column-family databases differentiate their records via a row id, as RDBMS does, timestamps for conflict resolution and/or versioning, column-family and column name. A column-family is a container for columns and columns can be added and deleted whenever needed without any down time. Moreover, every row generally has the same set of columns but it doesn't need to have same columns as RDBMS rows.

There are also column-oriented relational databases. They present data in terms of columns instead of rows because they physically store all data of one column together. Therefore, they are better in data-warehousing and customer relationship management (CRM). Except for this design difference, their features are more or less that of well-known row-oriented RDBMSs. A commercial product, Microsoft SQL Server, supports both storage designs, for instance. Vocabulary may be similar but column-oriented databases are very different designs than column-family databases.

3.4.1 Cassandra 

Apache Cassandra[11] is an open source distributed (mainly) column oriented key-value store achieving high performance and availability. According to the CAP theorem[12], a distributed system can't provide consistency, availability and partition tolerance at the same time. As a distributed database, Cassandra chooses to provide availability and partition tolerance (aka. distributed) with no single point of failure by the assigning same role to each node in the cluster. In addition, Cassandra supports master-less asynchronous inter-cluster replication for higher availability guarantees. At the same time, the third dimension, consistency, can also be adjusted according to the needs of particular applications. Since there is no difference between nodes, performance linearly scales as new nodes are added to the cluster, without interrupting existing nodes.

Due to its data model, Cassandra may actually be seen as a hybrid between key-value stores (inspired by Amazon Dynamo[7]) and column stores (inspired by Google BigTable[4]) because each row (on a record, a variable number of columns) is identified by a unique key which is distributed across the cluster using random partitioning or order preserving partitioning so that similar keys are closer to each other. Partitioning according to this key means assigning rows to physical nodes. Primary keys can be composed of multiple columns and after rows are partitioned by row id, they are clustered by the remaining columns.

Row partitioning requires a unique id to identify and to keep track of records. In the relational world:

- Single server: a unique id is just an auto incrementing counter.
- Multiple servers: application logic, 3-rd party service or ticket servers. These options bring extra components and they may also be a single point of failure and write bottlenecks. It can also be seen as "escaping the problem" instead of solving it on the server side.

As seen, solving this problem in a distributed environment is hard. Relational options are used in master-slave fashion but Cassandra nodes are identical so this problem should be solved while keeping this architecture. Allowing each node to generate its own ids may result in inconsistencies and a lock is needed to prevent them. However, locking means waiting and performance degradation while the main objective of Cassandra is high throughput.

Cassandra tries to solve it not much differently than its relational counterparts:

- It generates id from data
- It uses Universal Unique ID (UUID)[13], easily produced by the client

The structure of tables is pre-defined like relational databases but unlike RDBMS, they can be modified on the fly without any downtime (no blocking for queries). The number of columns can vary from 1 to 2 billion columns where each column has a name, value, timestamp and TTL (for expiration). A row doesn't need to contain exact set of columns.

In RDBMS, full normalization is recommended in order to prevent update anomalies. In Cassandra, there are higher level abstractions to store 1 to many relations in a column; namely, set, list and map. Cassandra doesn't support joins and sub-queries, except

Hadoop tasks (Pig and Hive are also supported), so de-normalization is encouraged by using these higher level constructs.

Table 3.7: Cassandra Column-Oriented Database

Criteria	Feature
Availability	Apache Software License 2.0 and Cassandra is one of the most influential top level projects of Apache Software Foundation.
Scalability	It's a clear winner in terms of throughput with increasing number of nodes by supporting clusters spanning multiple data centers with master-less replication. Especially, it's still perform well under heavy write load. Other databases requires locks for writes but Cassandra doesn't.
Ease of Use Development	It's designed to handle very massive datasets and shines under a huge deployment with thousands nodes. Highly distributed nature requires some excellence to leverage it. However, its data model is highly similar to relational counterparts and supports collections, which are higher level primitives. For instance, one room can have multiple equipments. In relational, there is a need for a table that maps rooms and equipments since this is a many to many relationship. In Cassandra, this can easily be handled by *set* collection. Moreover, de-normalization with collections is encouraged in Cassandra since there is no join. Therefore, converting deeply nested objects of Indico into Cassandra data model is easier than fully normalized relational tables.
Transactions Consistency	There was no support for transactions but there are good efforts to bring and as a result, latest version supports lightweight transactions, which are far from ACID because it's only very specific case of generic transactions. Transactions enable to do multiple operations on multiple tables without any interference but lightweight transactions (very misleading naming) only support doing two things on the same row by a compare and swap operation.
Community Momentum	Cassandra is the most popular column store. Companies that have high write loads like Facebook, Reddit and Twitter are users of Cassandra.
Cost Exit Strategy	Cassandra uses its own query language, *CQL*, which isn't a standard SQL but pretty similar to SQL so making an intra-class change is much harder than other types of database systems. However, there are some projects recently to use Cassandra as a back-end such as MySQL/MariaDB and Titan, a graph database. Thus, transition to these products may require less effort.
Score	★★★★☆☆

3.4.2 HBase

Apache HBase is an open source clone of Google's BigTable[4]. It's a fault tolerant column oriented database with compression and version control for data. HBase is a CP system according to CAP Theorem.

Records are accessed, sorted and distributed by row key. A master node keeps tracks of assignments to slaves, *region* servers, because records are sorted and an assignment is a continuous range so clients can easily learn where to look for data by knowing lower and upper bounds of regions.

It's easily thought that HBase isn't scalable due to the existence of a master node. On the contrary, the system is actually scalable because the master node isn't involved with data requests. On the other hand, reassignments of regions according to size (split or combine) and table operations require the master to be available. Thus, if master is down, data can be served by region servers because clients (can) cache the boundaries of region servers. However, a cold client can't learn assignments because assignments are kept in the meta table stored in the master which isn't up by then.

HBase runs on top of Hadoop Distributed File System (HDFS)[1] while BigTable runs of top of Google File System (GFS)[14]. Adaptation phase of HBase is complex due to the HDFS setup and configuration but when it's there, replication of data comes for free with HDFS. HBase is designed to crunch very big data, peta-bytes of it. That's why HDFS requirement seems reasonable. But if we think about Indico, HBase is a much more complex solution than needed. It's basically "killing a mosquito with an atom bomb". It's in production in data-rich parts of high load systems such as Facebook, Twitter and Yahoo.

Google has its own version, BigTable and it was very successful as a universal back-end at Google. Google is now working on the "next generation" of BigTable called *Spanner*[15] which is the storage engine used for Drive, GMail, Groups and Maps.

HBase doesn't support transactions but atomic operations are supported at the row level. ACID-like guarantees provided by relational databases are achievable via de-normalization. However, nesting data requires at least superficial knowledge of access patterns. Otherwise, ad-hoc queries would be a necessary but they aren't possible due to lack of transactions. The transaction problem is solved in the design phase with well-planned row key and nesting data into one row.

3.5 Document Oriented Databases

Document-oriented databases are a middle point between key-value store and RDBMS. Key-value stores identify records using a key and values aren't interpreted, only seen as a blob of bytes instead. On the other hand, RDBMS requires a strict schema even for values ("columns" in relational terminology). Document databases try to provide benefits of both world because they don't have a strict schema for values and can also give a meaning to the values so that they can index and query by values.

[1] http://aosabook.org/en/hdfs.html

3.5.1 MongoDB

MongoDB is the most popular document-oriented database. It stores flexible JSON-style documents and provides a set of functionalities that is closer to that of RDBMS, namely a full-fledged query interface, full indexing support and in-place updates. The schema of a MongoDB is basically non-existent - there is the concept of *collections* that can be compared to relational tables, but objects can have whatever fields and values the programmer sees fit. An object is not constrained in any way just because it belongs to a particular collection.

3.5.2 CouchDB

Apache CouchDB is a multi-master document-oriented database that is completely ready for the web. From CAP Theorem, it's closer to AP system because it favors availability with its multi-master architecture while resolving conflicts are delegated to clients. MongoDB vs CouchDB has caused many heated debates but this discussion actually can be reduced to whether consistency (CP) or availability (AP) is more important for the application.

Document model of CouchDB is very similar to the model of MongoDB. Documents, key-value pairs as in JSON, has a unique id to differentiate themselves. Values can be primitive values but also more complex types like lists or associative arrays.

CouchDB doesn't support joins but can support ACID semantics in the document level via implementing MVCC not to block reads by writes. Joins are possible via Map/Reduce tasks implemented as JavaScript functions. This tasks are called views in CouchDB terminology and they can also be indexed and synchronized to updates to documents.

CouchDB provides eventual consistency in addition to availability and partition tolerance. Each replica has its own copy of data, modifies it and sync bi-directional changes later. This design brings offline support at default which may be very useful for smart phones since smart phones frequently go offline and come back a later time.

Other use case is running pre-define queries on top of occasionally changing data. Compared to ad-hoc query capabilities of MongoDB, views of CouchDB require a bit more design beforehand. However, if data structure is hardly changing, data is accumulating and versions of data are also important, then CouchDB is the perfect solution.

CouchDB is in production at Credit Suisse, dotCloud and Engine Yard but there are also some failure stories such as Ubuntu One.

3.6 Key-Value Stores

Key-value stores are distributed hash table implementation for larger data. Each object has a unique key to be accessed with and values can be anything, seen as pure byte streams, so key-value stores don't enforce any schema. Moreover, object databases can be as an example of key-value stores since value is just an object of a specific programming language.

Key-value stores don't try to interpret bytes of values. As much as they give meaning to the value, they are closer to document-oriented databases. Nature of design makes them highly scalable but only scalability isn't enough because ease-of-use, usage difficulty and quering capabilities are also important. The more ad-hoc query capabilities key-value

store has, the closer it is to document-oriented databases and the more usable it is in general so different products try to give a compromise in-between.

Due to simpler architecture, this is the category that has the most number of products. Products are very different in terms of consistency characteristics, key structure, back-end and sorted-ness of keys. Main competitors are BerkeleyDB, Hazelcast, LevelDB, redis, Riak, Voldemort, memcached, Tarantool. Even document-oriented databases, MongoDB and CouchDB, can be seen as very capable key-value stores.

3.6.1 Redis 

Redis is a very simple key-value storage solution. It can be compared to memcached on steroids - a key-value store with extra [data structures] and additional features such as transactions and PubSub. Contrarily to e.g. memcached, it also allows developers to user server-side Lua scripts, akin to stored procedures. Redis will by default keep its data only in memory, but data can also be persisted on disk if desired. Redis Cluster is recently released to bring important features such as replication and strong consistency.

3.6.2 Riak 

Riak is an open-source fault-tolerant key-value store inspired by Amazon Dynamo. It's very configurable in terms of trade-off proved by CAP Theorem.

Riak replicates data in master-less fashion into n_val nodes which is *three* at default. Where data will be written is achieved by consistent hashing. In case of node failure or network partition, keys can be written to neighbouring nodes (hinted hand-off) and when failing nodes are back, new nodes off-load their part and data, updated while they are unavailable, is rewritten to them.

Reads requires R number of nodes to be read so cluster can tolerate $N - R$ node failures.

Consistent hashing enables distributing data evenly which makes query latency very predictable even in node failures. To make division more evenly, if there are low number of physical nodes, each Riak physical node creates *vnodes* virtual nodes. Moreover, it also enables assigning different amount of data to nodes by changing *vnodes* for the respective physical nodes if physical nodes have different specifications (more RAM, SSD, etc.). Riak is totally designed for distributed environment so generally, adding more nodes makes operations faster.

All nodes are equal and there is no master so any request can be served by any node. Consistency of data under concurrent writes is achieved by vector clocks.

Like CouchDB, Riak is written in Erlang and fully talks REST and supports MapReduce via JavaScript. In addition, Riak leverages Apache Solr to provide search capabilities and links to traverse objects via MapReduce to provide graph-like features. Moreover, even if Riak is a Key-value that requires access to objects via keys, it has more than that. Riak uses multiple backends; namely, memory, Bitcask, LevelDB and any combination of them together. If LevelDB or memory is in use, objects can queried by secondary indexes which are integer or string values to tag objects. Queries support exact match or range retrieval. Result can be paginated, streamed or even be given as input into MapReduce job.

Like HBase, Riak support inter-cluster replication but Riak has a master where HBase is master-less. Replication is done in two modes, real time and full-sync in 6 hours.

Riak is the choice of the majority of first 100 traffic websites such as GitHub (URL shortener and pages) and Google via acquisitions.

3.7 Graph Databases

Graph Databases focus on graph structured datasets where adjacency of data is important and relations between entities make them closer to each other. Graph databases try to optimize retrieval of these relations.

Graph databases are composed of two main parts; underlying storage and processing engine:

Underlying storage may be a native graph storage which relations directly point to physical location of entities. This is very different than foreign keys in RDBMS because there is no need of computation to retrieve related entities since entity itself contains links to physical location of its neighbours which can be used in *O(1)* time.

Computing engine implements standard graph algorithms on top of the storage engine. Performance and capabilities of this layer highly depends on storage. If storage is a native graph, graph can't be easily separated into parts which hoists scalability concerns. Even if database supports distributed architecture, whole data must be replicated to each node. However, having non-native storage enables processing giant graphs in the order of trillion edges via sharding capabilities of lower level storage, Cassandra or HBase, for instance. Unless graph partition quality is enough, performance characteristics can have noticeable differences since while native storage has whole data via *O(1)* access, non-native storage may need to talk to many nodes over network.

Supporting a standard graph processing interface is important because it makes transition easier between databases according to changing needs of the domain. Therefore, even if there are graph databases with non-native graph processing in the market, they aren't listed (FlockDB is an exception due to being open source but it seems to be dead at the time of writing due to activity and Scala version, main development language) because they are commercial, old (first examples of graph databases) and have much smaller user community.

Graph databases seems to be de-facto go to database in the implementation of personal dashboard and recommendations, and also next generation of features of Indico such as complete socialization and gamification.

3.7.1 OrientDB

OrientDB is a hybrid between document-oriented and graph databases. By default, it works like a Document-oriented DB in which records are JSON-like documents. However, there is a *graph* layer implemented on top of this that allows for elaborate relations to be established between documents. In this layer, both vertex and edges are documents that can be transparently manipulated.

There are 3 versions of OrientDB:

1. Standard (Document-oriented and Graph)
2. Graph (TinkerPop Stack)
3. Enterprise (coming with auto sharding/replication capabilities)

Graph functionality is actually included in the standard edition - it's a common misunderstanding. The graph edition differs from the the standard one in the fact that it includes a TinkerPop stack. OrientDB is 100% compliant with this graph processing stack. OrientDB can also be used as a key-value store, since document-orientation is a super-set of key-value stores. That is made possible by allowing records to be documents or flat strings. The data can be fully kept in memory if desired.

We have played a bit more with OrientDB compared to other databases since its features seemed to be a better fit for Indico.

Concepts

There are some concepts that make OrientDB quite different from its relational companions:

- **Record** - an instance of data as row in RDBMS or document in document-oriented databases.
- **Record ID** - auto-assigned unique number. It directly specifies the place of a record in disk, so it's not the equivalent to a logical ID (primary key) in RDBMS. That's why retrieving a record, when the ID is known, works in constant time, while it's O(log(n)) in RDBMS, using an index based on the id.
- **Cluster** - collection of links to records. A Record ID is composed of a Cluster ID and a sequence number within the cluster.
- **Class** - an abstraction of cluster used to group records. However, being a member of class doesn't put any constraints on records. Classes actually work in a similar way to their counterparts in Object-Oriented Programming and can bring new functionality via inheritance. By default, there is 1-to-1 mapping between classes and clusters and clusters can be seen as a table in an RDBMS that is used to group same type instances. However, in advanced usage, mapping may be n-to-m. Some examples:
  - cluster *person* to group all records from *person* class (1-to-1)
  - cluster *cache* to group most accessed records (1-to-n)
  - cluster *car* to group all records belonging to the "car" class by type; suv, truck, etc. (n-to-1)
  - cluster *daily* to group all records by creation day (n-to-m)

Security

OrientDB has powerful security mechanism compared to other NoSQL stores; namely, rule (bitmask for CRUD), role (group of rules) and user (executor of rules). Rules can be defined server, database, cluster or record level. These features brings OrientDB closer to RDBMS, which are strict and powerful in security.

Transactions

Since we heavily rely on transactions, transaction support is important in a database. MongoDB allows atomic operations at the document level. However, we may need operations that span multiple documents if the data is to be split across collections. On the contrary OrientDB is fully ACID-compliant:

- It allows multiple reads and writes on the same server
  - client-scoped and no lock on the server
  - implemented by a special property @*version* tracked for each record. Mismatch means roll-back (MVCC).
- Distributed transactions aren't supported. However, OrientDB got master-less replication via Hazelcast with latest release and as seen from code base, they have the base for distributed transactions. It's seen a must for horizontal scaling by the development team and planned for 2.0 version. Thus, it's only a matter of time.
- Nested transactions aren't supported. It's highly probable that they will be implemented while distributed transactions.

Speed

OrientDB seems to be really fast compared to other graph databases: 2 to 3 times faster than Neo4j in some benchmarks. In a fairly recent analysis (2012) by the Tokyo Institute of Technology and IBM, OrientDB was considered the fastest graph database from a group that included AllegroGraph, Fuseki and Neo4j (market leader).

Why is OriendDB fast?

- Cache
  - Level 1 - Thread Level (for each open connection) in client and Database Level in server
  - Level 2 - JVM Level (shared between all connections) in client and Storage Level in server
  - Advanced Storage cache depends on implementation
  - OS cache for Memory Mapped files
- Indexing
  - Tree Index - MVRB-Tree (proprietary but open source algorithm that combines best parts of RB-Tree and B+ Tree). Tree Index is heavier and works in $O(logn)$ time but enables range queries.
  - Hash Index - Lighter and works in *constant time*.
  - Composite Index (usable partially)
- Hard Links
  - Record ID is a pointer to a real physical place, not a logical concept so unlike RDBMS, it needs no calculation to access. As a result, Data loading is $O(1)$ time by record id.

- Memory-mapped files
  - Java New I/O (JSR 51 aka. NIO)
  - No system calls, since there is no switch between kernel and user modes
  - In-place update (*seek* only required if a record is beyond the current page's boundaries)
  - Lazy loading (efficient usage of memory)
  - Adjusted page size via collected statistics about database
  - The downside is that 32-bit, being only to address at most 4 Gb in their virtual address space, are limited in terms of file size.
- Transactions
  - No lock for reads (MVCC)
- Tools to tweak
  - explain (standard as PostgreSQL or MongoDB)
  - profiler

Scaling

Scaling is done by multi-master scheme, each instance auto-magically synchronize each other. There are two modes:

- synchronous: slower writes. The larger cluster size is, the slower writes because writes wait for each node to acknowledge success and quorum as in Cassandra or Riak is not configurable for now.
- asynchronous: faster but naturally weaker consistency

Each node has the whole database which may be a problem in some cases if database is so large (in a 64-bit system, it is very unlikely since virtual address space huge enough). Sharding is possible by grouping nodes and forming clusters according to shard keys but that possibly would bring some logic to client for whom to talk.

API

Java and Scala are the main supported languages. However, Python is also fully supported by 3 drivers:

- Native driver, a wrapper over C library (liborient)
- HTTP REST driver
- *Rexster* driver (only if graph edition is in use)

There is a powerful console (native) with auto-complete that makes trying commands/concepts fairly easy. Moreover, there is another open-source project called OrientDB studio to accomplish administrative tasks easier like what *PHPMyAdmin* does for MySQL/MariaDB.

The company that is supporting the development of database and providing database in the cloud as a service, is closed to focus more on the development side. Whether it is a good thing is a big mystery because it is more likely that they couldn't keep up with a sustainable business due to market share of OrientDB and very established competitors.

Extensibility

There are two ways supported:

- Server-side functions (different than triggers): totally generic functions that are also capable of executing sql queries on the database.
  - Java
  - JavaScript on Mozilla Rhino
- Hooks (Java plug-ins to core)

Backup/Restore

There is import/export mechanism to easily move the data/schema or upgrade. Automatic regular backup is native but it should be enabled.

There is also an emailing module which may be used for emergency, statistics, etc.

Competitors

There are a lot of graph databases but there are a few in active development; namely, DEX, Neo4j and Titan. DEX is out of comparison since it is commercial. Neo4j is the leader. Titan is quite new but could catch up with others. Titan goes pretty good with other systems such as Cassandra, HBase, ElasticSearch, etc. which is very important in big data era, to be able to deploy a polyglot storage system. Moreover, it is getting popular faster than OrientDB by providing better scalability with underlying storage engine.

3.7.2 Neo4j

Neo4j is the most popular and mature graph database which is deployed at Cisco, HP, Huawei, etc. Neo4j adapts dual license, community and enterprise. There are huge differences between them in terms of scalability. Enterprise is fully ready for horizontal scalability with default cache, auto-sharding, master-slave replication where none of them available in community edition. Documentation and resources are plentiful compared other graph databases but lacking features between different versions risks accessibility.

Both versions support ACID transactions and leverage Lucene for fast searching. OrientDB supports documents everywhere so as to vertexes and edges can theoretically be annotated as it's requested. Neo4j delegates it with integrating Lucene. Since OrientDB does itself, it is faster but being an important Apache project, Lucene has much much faster development.

3.7.3 Titan 

Titan is a graph database that targets scalability with supporting scalable NoSQL databases. Graphs are connected so dividing graph into multiple nodes is extremely difficult without data-aware solutions. Thus, main scalability idea is to replicate the graph into multiple nodes via master-slave or multi-master scheme but this prevents scaling beyond capabilities of weakest node. While in terms of query latency, this solution can provide reasonably good performance as Neo4j and OrientDB. While two unrelated points which are very far from each other are being updated, synchronization and locking are unnecessary. This is main problem Titan is trying to solve at the expense of losing ACID transactions. Moreover, scalability is more than throughput such that databases replicates whole graph in every node miserably fail scaling graphs in size. There are theoretical limits like 2 billion vertexes in Neo4j and OrientDB but these numbers can only be accessed in Titan practically.

In short, its architecture addresses an important problem which, in turn, draws attention of many users but it's still not widely used. Furthermore, the schema of Indico isn't mappable into a graph and while we are already trying to be away from column-oriented stores due to complexity, Titan even increases it.

3.7.4 Broadness and Validity of Survey

We try to give basic architecture details of each database but there are normally much more details to be considered. However, database ecosystem is quickly changing. We have seen multiple versions same database even while this writing. That's why we tried to keep more tiny details online in a living document. This enables us to easily update data and since it gives comparison more exposure, small mistakes are found and fixed earlier.

Table 3.8: HBase Column-Oriented Database

Criteria	Feature
Availability	HBase may be the biggest kahuna of the systems that can handle massive data. HBase is also a top level project of Apache Software Foundation, licensed under Apache License 2.0 and it's a part of Hadoop ecosystem so runs on top of HDFS.
Scalability	It follows the design of Google's BigTable and highly scalable. Like TaskTracker and JobTracker in Hadoop, or NameNode and DataNode in HDFS, HBase also has a master and slaves. Master handles administrative operations for auto-sharding and making assignments for slaves, region servers. This mapping is stored in META table at ZooKeeper node. Client need to know where each region is stored but they don't need to talk to master every time when they need data, instead clients can keep a cache and can directly connect to slaves. If there is a reassignment in regions, clients will get an exception to invalidate cache and will be required to relearn region assignment. Therefore, even if it seems it doesn't scale linearly due to a master-slave architecture, it is actually scales because master is only there to coordinate and master doesn't involve in data requests. However, there are some availability concerns related to master. If master is down, administrative operations can't be accomplished such as table creation/update but slaves can continue serving data operations even if master is down.
Ease of Use Development	Since it's a part of Hadoop, it requires Hadoop machinery in place. Setup process of HBase is the most involving of all candidates. However, if the system is installed, it can easily handle peta-bytes of data and provides a giant table view of data to clients irrespective of where data is stored.
Transactions Consistency	HBase isn't an ACID compliant database but it provides some of properties such as atomic mutation in the row level even if mutation includes multiple column families but that isn't enough in the context of Indico because Indico has a pretty complex schema and whole schema can't be nested so that one magic row key will enable atomic operations.
Community Momentum	HBase is a successful product and has a lot of well-known users but migrations seem to be cumbersome. That's why users that don't really have (really) big data problems in the scale of HBase (tens of peta-bytes) are going away from HBase in the favor of simpler solutions.
Cost Exit Strategy	Entering and exiting from HBase are costly because it lives within Hadoop ecosystem and there is no direct replacement as powerful as HBase for very big data.
Score	★★★★☆☆

Table 3.9: MongoDB Document-Oriented Database

Criteria	Feature
Availability	AGPLv3
Scalability	Replication and sharding comes at default which are cornerstones of a scalable system. MongoDB supports master-slave replication via replica sets to provide redundancy and higher availability. Auto-sharding is possible with configuration servers, sharded meta-data holders, and query routers, *mongos*. Moreover, it is faster compared to competitors since it loosened some constraints. However, these weak guarantees can bite the application developer if application is very strict about data lost/consistency even under high load. MongoDB applies exclusive read-write lock so multiple reads can happen at the same time but a write requires a exclusive lock. There were problems related to the extent of write lock like being for whole *mongod* instance but it is recently improved to be for a collection which should be improved for document level.
Ease of Use Development	Python is one of the officially supported languages. MongoDB doesn't have a schema at all and very flexible. As a result, using MongoDB is a pleasure and development with MongoDB takes less time for the same feature in RDBMS.
Transactions Consistency	MongoDB gives guarantee of BASE, eventual consistency and it doesn't support transactions which is a really bad disadvantage for Indico where there are multiple inter-relations between entities. Some parts of Indico nicely fit into a document but putting everything into a document is difficult in terms of design complexity and size constraint. Then, inter-relations discourage MongoDB as main storage.
Community Momentum	Without any doubt, MongoDB is the most popular NoSQL database in the wild and it is very well documented.
Cost Exit Strategy	Replacing MongoDB with documented oriented databases wouldn't be much difficult because all of them; CouchDB, RethinkDB, RavenDB are similar at the core. Moreover, documents are close to objects as in ZODB. That's why coming from object-oriented world into document databases is a smoother path to follow.
Score	★★★★★☆

Table 3.10: CouchDB Document-Oriented Database

Criteria	Feature
Availability	CouchDB, licensed under Apache License 2.0, is one of the high priority projects in Apache Foundation and more mature compared to other NoSQL prouducts because it has longer history and is developed by more inter-company developers.
Scalability	CouchDB shines at scalability with incremental multi-master replication and also implements MVCC to prevent writes to block reads. Sharding may be done in application level but there are couple of products to extend CouchDB such as auto-sharding by CouchBase.
Ease of Use Development	It is developed for web and provides HTTP API so can be easily used with any language. Data is stored in documents as MongoDB which is more natural and these documents are very flexible due to lack of schema.
Transactions Consistency	ACID-like eventual consistency is guaranteed. This is far from the transactions of relational world but closer than MongoDB to what is needed by Indico.
Community Momentum	Compared to MongoDB, CouchDB has a smaller community and it is preferred by companies in more niche categories such as PaaS providers while MongoDB is in production at eBay, Foursquare, New York Times and SourceForge.
Cost Exit Strategy	Coming into CouchDB from object-oriented world is easier than relational since documents are similar to objects. Consistency guarantees of CouchDB is more analogous to ZODB than MongoDB but this makes CouchDB replacement more difficult compared to MongoDB with other document databases.
Score	★★★★★☆

Table 3.11: Redis Key-Value Store

Criteria	Feature
Availability	According to rankings, Redis is the most popular key-value store in use and BSD license is suitable for Indico. However, Redis is still developed by only one developer, in the core.
Scalability	Redis is developed as a single threaded in-memory database. Later, optional durability is added by flushing memory in intervals aka. snap-shotting or asynchronous append-only operation, *journaling*. In December 2013, redis cluster, distributed version of Redis, is released but it is unstable and there is no success stories, at least for now.
Ease of Use Development	Python is a well supported language. However, there is no rich data structure that can easily map objects of ZODB. Thus, converting complex hierarchy of objects to Redis primitives will be a pain and time taking.
Transactions Consistency	Redis supports transactions but since it's single threaded, transactions are executed in a blocking fashion which deteriorates performance. It's also important to note that there is no roll-back for failed transactions.
Community Momentum	Community size is medium since Redis can't be the only database system, it's more suited to work as a helper for another main storage engine but there are also a few successful only Redis deployments which have very very high write loads. Redis is in production at craiglist, flickr, stackoverflow.
Cost Exit Strategy	Using Redis is a bit difficult due to lack of complex structures but when time has come to exit, it is also difficult because Redis provides some powerful data structures compared to other key-value stores and if these structures are used, they may not be replaced easily. Moreover, main memory is the limiting factor for the size of the data because data managed by Redis can't be bigger than available memory.
Score	★★★★☆☆

Table 3.12: Riak Key-Value Store

Criteria	Feature
Availability	It has Apache License which is suitable for Indico and it is one of most popular key-value stores with Redis.
Scalability	Riak is inspired by Amazon Dynamo paper. It's developed for scalability and high availability so it is very easy to add/remove machines to cluster. Inter-cluster replication is even possible in master-slave fashion in two modes, asynchronous and synchronous. Intra-cluster, every piece of data is stored in multiple nodes and consistent hashing is used to divide the data between machines so Riak has very predictable latency.
Ease of Use Development	Python is one of the officially supported languages but mapping complex object structures to very primitive data types is difficult. Moreover, lack of proper transactions like in relational databases puts burden on readers.
Transactions Consistency	Riak implements vector clocks and reads use them to decide which write is the latest under concurrent updates so reads require extra work but writes always succeed.
Community Momentum	Riak is in production at very popular websites due to its highly fault-tolerant nature which is vital for a business takes huge traffic, a result of being big.
Cost Exit Strategy	Entering into Riak as a main storage is difficult due to lack of complex data structures because complex structures of Indico should be converted into simple ones. There is no direct replacement of Riak in provided distributed fault tolerancy support so leaving Riak is also difficult.
Score	★★★★☆☆

Table 3.13: Architecture of Graph Databases

Graph DB	Storage	Engine
Twitter FlockDB	non-native	non-native
neo4j	native	native
OrientDB	native	native
Titan	non-native	native

Table 3.14: OrientDB Graph Database

Criteria	Feature
Availability	OrientDB is licensed under Apache License 2.0.
Scalability	With newest release in November 2013, OrientDB supports multi-master replication via Hazelcast but sharding is offloaded to application. Within graph databases with native graph storage, OrientDB seems to be the fastest but for a healthier evaluation, comparison with other databases from different disciplines is missing.
Ease of Use Development	Python is supported by drivers provided from community (binary or HTTP). Being document and graph database enables flexible schema which is easier for refactoring and intuitiveness.
Transactions Consistency	ACID with MVCC
Community Momentum	OrientDB is getting popular fairly quickly because it isn't a native graph database. It's a hybrid of document and graph databases where each node or edge is also a document as in MongoDB or CouchDB. Thus, it is able to provide benefits of both worlds at the same time. However, there aren't enough successful big deployments which brings the question of if OrientDB is really production ready.
Cost Exit Strategy	If it is compared to Neo4j, picking up OrientDB is easier because parts that are problematic in Neo4j can be mapped via more intuitive document nodes and edges.
Score	★★★★★☆

Table 3.15: Neo4j Graph Database

Criteria	Feature
Availability	Neo4j is licensed under GPL and AGPL (for enterprise) and it is by far the most stable graph database in the market.
Scalability	Scalability is lacking because horizontal scaling of a graph is really difficult due to relationships in a tightly connected graph. Neo4j Clustering technology provides horizontal scalability but only for enterprise version.
Ease of Use Development	Some parts of Indico such as roles of users in events, links between conferences, sessions and contributions, etc. perfectly match links in a graph but not everything is mappable.
Transactions Consistency	Neo4j has ACID transactions like relational databases.
Community Momentum	Neo4j is the most popular graph product but in general, graph databases are niche products and mainly used by communication companies, where connections between entities are deadly important to business.
Cost Exit Strategy	Starting to use Neo4j isn't difficult due to 1-to-1 mapping from ZODB objects to vertexes and edges of Neo4j. Graph databases supports *Gremlin*, a graph traversing and manipulation language like SQL in relational world. If Gremlin is used, Gremlin creates an abstraction layer on back-end storage like *SQLAlchemy* does for relational databases so back-end can easily be changed with another Gremlin-compliant database such as OrientDB or Titan.
Score	★★★★☆☆

Table 3.16: Titan Graph Database

Criteria	Feature
Availability	Apache License 2.0
Scalability	Titan has a notably different architecture compared to other graph databases via pluggable storage back-ends where scalable back-ends such as HBase or Cassandra brings their scalable nature into Titan.
Ease of Use Development	Python is officially supported in binary protocol but while pluggable storage engine, which itself isn't easy to use, is bringing scalability, it also makes the system much more complex compared to OrientDB or Neo4j due to many moving parts.
Transactions Consistency	ACID and eventual consistency are possible but it mainly depends on the back-end characteristics.
Community Momentum	Graph databases are on the rise and data is getting bigger and bigger. Other graph databases can't divide their data but Titan can divide by following a different path. Thus, Titan can scale to the numbers OrientDB and Neo4j can't in terms of graph vertex and edge sizes. If GitHub popularity of graph databases is ranked (quite rational since all three are hosted on GitHub), then Titan seems to be the most popular one. However, this may be misleading because this popularity may be a result of popularity of pluggable storages.
Cost Exit Strategy	Start phase to use Titan may be probably difficult since required storage engine. Exit is comparably easier since Titan is also 100% compliant with TinkerPop stack. If application is using TinkerPop stack, when graph database is changed, there is nothing to be changed from the application point of view.
Score	★★★★☆☆

4

Databases In Action

In this section, we recreate dashboard example, which is used to explain limitations of ZODB, to interpolate complexity. Since we did a survey a great number of databases, we couldn't do it for each database. That's why strongest candidates are chosen from each category; namely, PostgreSQL, MongoDB, Redis and OrientDB.

4.1 Relational Databases

4.1.1 PostgreSQL

Table 4.1: PostgreSQL users

users		
id	first_name	last_name
1	John	Doe
2	John	Smith

Table 4.2: PostgreSQL users and events association

users_events		
user_id	event_id	role
1	John	Doe
2	John	Smith

Table 4.3: PostgreSQL events

events		
id	title	timestamp
1	CERN Open Days	12345
2	CERN Higgs Event	12365
3	TEDxCERN	12377

Conclusion

± Plus vs Minus ±

Pros:

Very stable and huge community (business, cost, docs) Important successful deployments Huge list of features and customizable ACID but fast Security and data protection Cons:

Strict schema (new features?) Joins (complex data retrieval via SQL)

4.2 Column Oriented-Databases

4.2.1 HBase

Table 4.4: HBase users

users			
row key	timestamp	column families	
user_1	t1	personal_info:name="John Doe"	events:event_1="attendee", events:event_2="manager", events:event_3="reviewer, attendee"
user_2	t2	personal_info:name="John Smith"	events:event_2="attendee"

Table 4.5: HBase events

events		
row key	timestamp	column families
event_1	t5	info:title="CERN Open Days", info:timestamp="12345"
event_2	t6	info:title="CERN Higgs Event", info:timestamp="12365"
event_3	t7	info:title="TEDxCERN", info:timestamp="12377"

4.3 Document Oriented-Databases

4.3.1 MongoDB

```
Collection 'users':
  [
    {
      "id": "user_1",
      "name": "John Doe",
      "events": [
        { "id": "event_1", "roles": ["attendee"] },
        { "id": "event_2", "roles": ["manager"] },
        { "id": "event_3", "roles": ["reviewer", "attendee"] }
      ]
    },
    {
      "id": "user_2",
      "name": "John Smith",
      "events": [
        { "id": "event_2", "roles": "attendee" }
      ]
    }
  ]

Collection 'events'
  [
    {
      "id": "event_1",
      "title": "CERN Open Days",
      "timestamp": 12345
    },
    {
      "id": "event_2",
      "title": "CERN Higgs Event",
      "timestamp": 12365
    },
    {
      "id": "event_3",
      "title": "TEDxCERN",
      "timestamp": 12377
    }
  ]
```

Conclusion

Pros: Good documentation, big community and successful examples (Sourceforge, eBay, Forbes, CERN) Javascript and JSON are first class citizens Replication (auto-failover with replica sets) and sharding (built-in) In-place updates Built by most valuable open source startup Aggregation Framework and Map-Reduce Cons:

Lack of transactions Unsafe writes by design (data loss possibility unless be sure of propagation) Not fast comparable to expectations, even if constraints are loosened to gain speed Memory constraints (document size, working set, etc.) Possible Use-Cases:

User-account details management: Requires medium heavy of read and writes Required Conference details management: Much more read than write Installation manage-

ment: Tracking indico installations needs flexible schema, since tracked properties may change frequently by new versions

4.4 Key-Value Store

4.4.1 Redis

The *Redis schema* (if fair to say) that was used for our implementation of the dashboard feature was the following:

```
avatar_event_roles: (<AVATAR_ID>, <EVENT_ID>) -> SET(ROLE)
event_avatars: <EVENT_ID> -> SET(AVATAR)
avatar_events: <AVATAR_ID> -> SORTED_SET(EVENT_TS -> EVENT_ID)
```

The first namespace establishes a link between an avatar and an event, much like named edges of a graph. The second one ($event_avatars$) maps a particular event to a set of avatars (so that we can have a quick list of all the people involved in a specific event), while $avatar_events$ does the opposite (maps an avatar to all the events it's connected to) with a little twist: events will be ordered by timestamp. This makes it easier to retrieve an ordered list of events relative to a particular user.

Here is the representation of the example Dashboard we've shown before using this schema:

```
avatar_event_roles: (user_1, event_1) -> attendee
                    (user_1, event_2) -> manager
                    (user_1, event_3) -> reviewer, attendee
                    (user_2, event_2) -> attendee

event_avatars: event_1 -> user_1,
               event_2 -> user_1, user_2
               event_3 -> user_1

avatar_events: user_1 -> 12345 -> event_1
                         12365 -> event_2
                         12377 -> event_3
               user_2 -> 12345 -> event_2
```

There are a series of operations that need to be executed over this "schema":

- **Adding a new role** (involves all the three namespaces)
- **Getting all the events in a specific time interval**, with the respective role information (involves $avatar_events$ and $avatar_event_roles$)
- **Deleting an event** (involves all three namespaces)
- **Deleting an avatar** (involves all three namespaces)
- **Deleting a role link** (involves all three namespaces)
- **Updating event time** (involves $event_avatars$ and $avatar_events$)

- **Merging avatars** (involves all three namespaces)

In this particular implementation, most of the work is done client-side, using LUA scripts (similar to *stored procedures*). This really speeds up query times, but has the inconvenience of locking the server, thus not allowing anything else to run in parallel. This can be problematic in the case of slow scripts.

The operations shown above have quite reasonable time complexity boundaries: Redis guarantees at least linear time in almost all operations, and most hash-related operations are logarithmic. *Set* and *get* of single values tend to be executed in constant time. A possible *worst-case scenario* could be *deleting an event* when all users are connected to all events. This would imply going over all users in $event_avatars[deleted_event]$ and deleting them from $avatar_events$ ($O(log(N))$) as well as $avatar_event_roles$ ($O(1)$). This results in $O(Nlog(N))$.

4.4.1.1 Conclusion

Redis is a simple, fast and highly powerful key-value store. It is very versatile and can be used to solve a large set of problems. Its simplicity is, however, indicative of the role that it should occupy in an application stack. Its memory-oriented nature makes it ideal for caching and other use cases that require storing short-lived data (even though this data can be persisted) and the "key-value" philosophy that is behind it makes it hard to model complex relations using the simple data structures that are provided. There are sites successfully [using Redis as their primary DB][2], but the complexity of their DB schema cannot be compared to that of Indico's.

It is, then, clear that Redis would be a viable option in a "double database" scenario (primary DB using a different technology), in which it would act as a cache or possible primary storage for "simple" information that could be easily decoupled from the applications' business logic. Examples of that are:

- Cached JSON objects (already used)
- Web Session information (already used, ¿=1.2)
- Cached templates
- Job scheduler tasks
- Plugin cache (plugins that are available, corresponding extension points...)
- Short URL registry
- Other possible cases could be:
  - User account information
  - Versioned minute storage

4.5 Graph Databases

4.5.1 OrientDB

```
(V, E) -> (Vertex, Edge)
$ create class User extends V
$ create class Event extends V
$ create class Role extends E
$ create vertex User set name = 'John Doe'
$ create vertex User set name = 'John Smith'
$ create vertex Event
  set title = 'CERN Open Days', timestamp = '12345'
$ create vertex Event
  set title = 'CERN Higgs Event', timestamp = '12365'
$ create vertex Event
  set title = 'TEDxCERN', timestamp = '12377'
$ create edge Role
  from
  (select from User where name = 'John Doe')
  to
  (select from Event where title = 'CERN Open Days')
  set roles = ['attendee']
```

```
Class 'User':
  [
    {
      "@rid": "#1:0",
      "name": "John Doe"
    },
    {
      "@rid": "#1:1",
      "name": "John Smith"
    }
  ]

Class 'Event':
  [
    {
      "@rid": "#2:0",
      "title": "CERN Open Days",
      "timestamp": 12345
    },
    {
      "@rid": "#2:1",
      "title": "CERN Higgs Event",
      "timestamp": 12365
    },
    {
      "@rid": "#2:2",
      "title": "TEDxCERN",
      "timestamp": 12377
    }
  ]

Class 'Role':
  [
    {
```

```
        "@rid": "#3:0",
        "@out": "#1:0",
        "@in": "#2:0",
        "roles": ["attendee"]
      },
      {
        "@rid": "#3:1",
        "@out": "#1:0",
        "@in": "#2:1",
        "roles": ["manager"]
      },
      {
        "@rid": "#3:2",
        "@out": "#1:0",
        "@in": "#2:2",
        "roles": ["reviewer", "attendee"]
      },
      {
        "@rid": "#3:3",
        "@out": "#1:1",
        "@in": "#2:1",
        "roles": ["attendee"]
      }
    ]
```

Conclusion

OrientDB is a multi-disciplinary graph database but is it really ready for production?

Pros:

Multi-disciplines; document, graph and key-value Flexible schema Fast (index, cache) ACID Security Customizable Interface (SQL and studio) Getting traction Cons:

Slow development Buggy Small community No big successful production Java and drivers Competitors; Neo4j and Titan Lack of speed comparison to other databases from different disciplines

5

Decision

After surveying and then trying out of some these databases at a small scale[16], next logical step was to decide on using one (or a poly-glot) database system. We decided on running PostgreSQL for main data storage. However, transition will be incremental, so we will be using ZODB and PostgreSQL together for some time. Since *Redis* is very flexible and blazingly fast, it will be leveraged to offload some traffic from *ZODB* and *PostgreSQL*. This is the first phase of a database change.

Secondly, there is one more important goal, to make Indico more user-friendly and it includes:

- Making Indico installable with one command
  - Installing Indico for cloud now isn't easy since there is no pre-built package.
- Providing Indico as SaaS [1]
  - Each installation of Indico is running independently so each one is requiring its own resources and administration. Thus, not all of servers use the latest version due to update/migration costs which gives birth to the need of support for old versions.
  - Everybody doesn't need full featured installation of Indico for one time event, for instance.
  - Collecting usage statistics and patterns is difficult.
- To be able to use more generic accounts such as Google, Yahoo, etc.
- Making Indico more user-centric and social.

Different kinds of problems require different solutions, and other storage paradigms are being used within the Indico ecosystem. For instance, we have decided to use a document-oriented database for statistics collection and caching of event meta-data that will aggregated from servers around the world, which is called project *indic8r*. Inside of available document-oriented databases, we decided on *MongoDB* since it's the most accessible in terms of know-how, popularity and features. Indico has a mobile version which is read-only - thus, it's reasonable to see it as a cache for *ZODB*. This mobile version uses *MongoDB*, since there is no need for ACID, ad-hoc queries are supported

[1] Software as a Service

and the lack of schema enables faster development. Due to such nice fit between use-case and features provided by MongoDB, Indico mobile has been very successful. The data manipulation flow of these new planned use-cases are very similar to that of mobile version. That's why *MongoDB* is the selected option.

The introduction of social network-like features in Indico can be considered as an application of graph processing in which graph databases excel. But at the time of writing, there is a decision: we aren't using them, at least for now. The main focus right now is to change the de-facto back-end and it will take a great amount of effort. Until it's complete, graph databases will probably change a lot and there are no successful big deployments with graph databases except Neo4j and custom/proprietary solutions such as Facebook.

5.1 Chosen Database Type

We have first checked object-oriented databases because one good object-oriented database could reduce transition costs a lot by keeping same paradigm, enabling easy development and tight integration with Python. However, there is no good option in the market which will be able to replace *ZODB*.

Column-family databases aren't considered since they are just too complex at Indico's scale.

Key-value stores are overly simplistic. It's very difficult to convert Indico's schema to key-value storage and still keep it manageable. References between entities or higher level collections are needed to easily store/retrieve related data. However, there are specific cases where managing data is easy, such as url and session caching. Since the less the main database is used, the better it is; a key-value store, *Redis*, is used and it'll be employed wherever it makes sense but purely as secondary storage.[1]

Graph databases meet the majority of needs of Indico, such as ACID transactions, replication and mapping complex relationships. However, they are niche products lacking community and not every entity of a complex Indico schema isn't mappable because entities are usually like trees and putting these *contains* relationships into a graph database exponentially increases number of edges. Therefore, storing *contains* relationship into documents on vertexes and using edges for far separated entities is favourable. This design is extensively supported by *OrientDB*. That's why *OrientDB* is studied in detail but as a result, we find it to be in a too extreme development phase and not production ready.

After exhausting any other types, we had two options; namely, document-oriented and relational databases. This is basically a trade-off between easiness-of-use (lack of strict schema and faster development) and ACID transactions. For specific cases, ACID transactions may be unnecessary because document-oriented databases are capable of supporting ACID guarantees at a document level and what are multiple tables in fully normalized relational databases will be nested documents within one huge document in document-oriented databases such as room booking schema, which will be explained later in the condition of prototype. However, when other parts start to have huge documents, we can't work on document level any more, there is a need for cross references between tables.

[1] Twilio Billing Incident in Redis back-end

As a result, we are left with relational databases and relational databases are the best fit for Indico's use-case and schema because:

- Schema is composed of many cross-referenced entities which requires ACID
- The size of the data set is perfectly manageable by relational databases
- Indico provides many flexible search interfaces which execute ad-hoc queries

5.2 Chosen Database

Since we were looking for a relational database, *MySQL* cloud or *PostgreSQL* were obvious candidates. The future of *MySQL* is confusing, while *PostgreSQL* is gaining popularity steadily and a much more tightly-connected community is coming to life around it. In addition to these community issues, getting *PostgreSQL* boxes at CERN is possible via a service provided by database team[1]. The burden of managing a database system will be reduced, even though management of ZODB was not too hard: most operations (like packing and backup) were done by automatic scripts.

PostgreSQL has many more features than MySQL, which makes it closer to document-oriented databases. Between features and speed, we have chosen features because we didn't experience notable speed difference between them and PostgreSQL has been getting faster steadily.

Finally, notable features of *PostgreSQL* (not an exhaustive list, of course):

- Most compliant DBMS with standards
- Most extensible and enormous number of features
  - Concurrency [2]
  - Standard SQL [3]
  - Index [4]
    - B-Tree
    - Hash
    - Functional
    - Inverted [5]
  - Triggers
  - Schema
    - Powerful types
      - array
      - IP
      - XML

[1] Database on Demand Service
[2] ACID, 64 cores
[3] fully 92-compliant
[4] bitmap index can utilize multiple indexes at the same time
[5] content → row

        - JSON
        - range
    - Table Inheritance
    - Auto Master-Slave Replication
    - Views
        - Materialized
        - Modifiable
        - Recursive
    - Notifications
    - Fine-grained security
        - Kerberos
        - LDAP
        - RADIUS
    - Regular Expressions
    - Online Backups
    - Full-text Search
    - Administrative tasks and Analytics
        - pgAdmin
    - Custom background workers
        - default
        - pgQ
    - Rich plug-in ecosystem
- Well-know successful deployments
    - Amazon
    - Apple
    - Disqus
    - Facebook (Instagram)
    - Heroku
    - Microsoft (Skype)
    - Nasa
    - Yahoo

In the light of that study, *PostgreSQL* was chosen to replace *ZODB*.
One final remark is that *PostgreSQL* went from being "an ugly academic project"[17] in early 2000s to a complete renaissance in terms of features, which have attracted many small businesses and enterprises. Thus, knowing how to use *PostgreSQL* has become an important skill and a valuable asset in a career path. The fact that Indico is an open source product with PostgreSQL may have contributed to the increase of outside contributions.

6

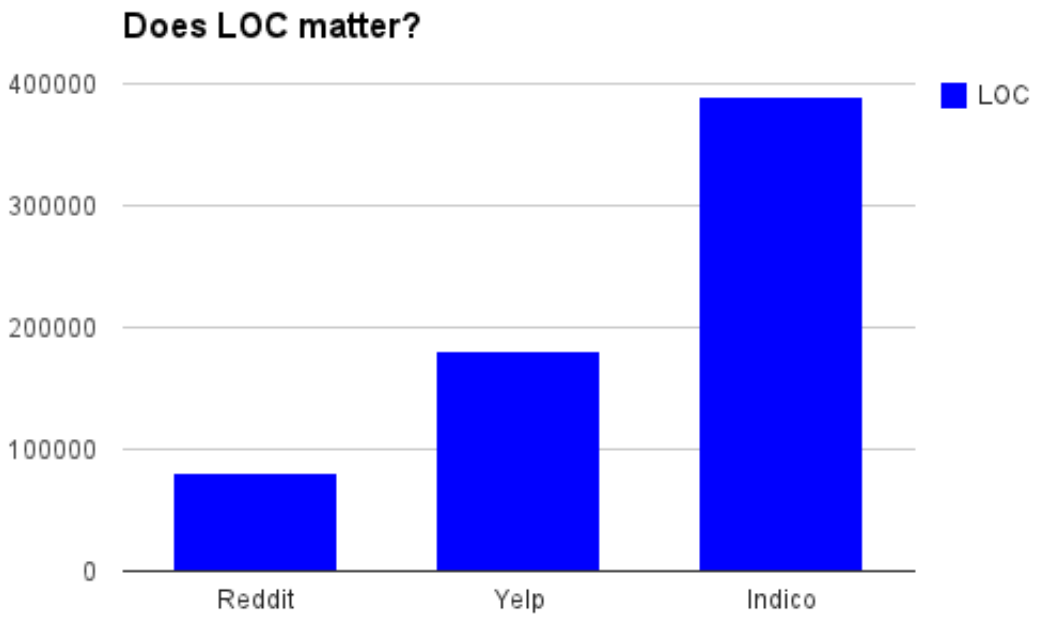
Implementation

In this chapter, we explain the implementation plan and the work that has been done so far.

6.1 Big Picture

Indico is a complex product with many features. As a result, it as a large code base which is easily seen figure 6.1[1] in comparison to some popular services.

Figure 6.1: Line of code comparison between Reddit, Yelp and Indico



[1]Reddit on Ohloh, Yelp: Developing a Python service stack at Twitter University, Indico on Ohloh

This kind of complexity makes a database change a non-trivial operation that has to be done in step. Since we would like to see the result of our changes all the time, we need a working system. Therefore, the main database machinery, is first integrated and then each module is refactored into using the new database. At the same time, tests are being written to validate the behaviour of code. When a module is ready, after careful testing, it is put in production. Other modules follow, being gradually refactored and moved to the production environment.

6.2 Scope of Project

When ZODB started to perform badly, incremental optimizations were done. For instance, Room Booking which enables location and room management, and reserving rooms for events, is one of the most complex and involving modules and has its own database (in CERN's production system). Since it's not very tightly integrated into the main database, it is a good starting point that allows the development team to gain experience in the frameworks and tools in question as well as in refactoring code to the new database. It also allows for more precise estimates to be established.

6.3 Room Booking Schema

Since we are going to use a RDBMS, a schema must be designed up-front.

We started inspecting database with external tools to get the object structure. However, ZODB being an object database, inspecting it is much harder, not only because the inspection tool needs run-time information about used classes but also because data is organized in a graph-like fashion. Even if three different tools are available for such job, none of them has shown to be stable and well-supported:

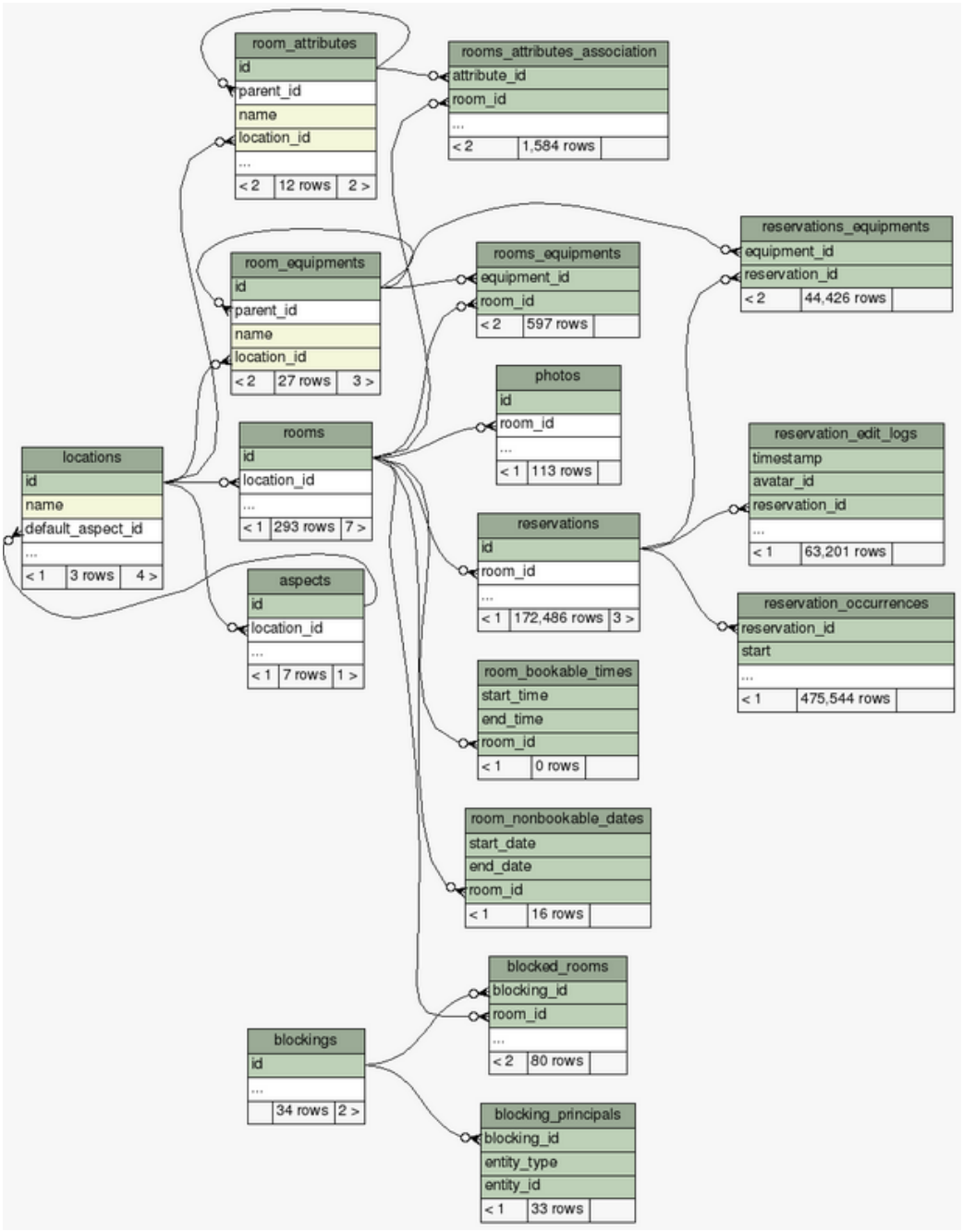
- `zodbbrowser`
- `eye`
- `z3c.zodbbrowser` (unfinished GSOC[1] project)

We have tried all of them but *eye* was the most stable in our opinion and our experience. But even *eye* wouldn't open our production database, which can be as big as 40 GB. Even a packed version won't be smaller than 15 GB.

The viewer could be patched, but the interaction with upstream developer showed to be slow. Te choice was, then, to use a subset of this database instead - data was loaded incrementally in order to find the rupture point. Next logical step was to analyse the mapped classes which wasn't an easy task, since some of the classes were as big as ∼2000 lines.

[1] Google Summer of Code

Figure 6.2: Room Booking Schema with CERN production data



Normalizing such a data set is a challenge, because most of the ZODB classes in question use the *set*, *list*, *dictionary* data structures of Python and there are places where data is replicated within objects. They should be fully normalized into their own tables.

There are places which work nicely in small number of cases but don't scale:

- Recurrence of room bookings: currently, there are only 6 options so their text representations and dates are calculated explicitly. Moreover, their implementation is not always precise. Some corner cases result in results that are not intuitive at all. Adding one more repetition type isn't easy so text and date generation needs

some *humanization* and a better computing algorithm.

- Location and Room attributes and equipments: These are replicated for each one, they all mean the same thing, though.
- Location and Room attributes and equipments: They have a hierarchy but this hierarchy is hard coded into the source code. Each level of this hierarchy is one list in room object, for instance.
- Location and Room attributes and equipments: Due to replication, there are inconsistencies. Rooms shouldn't be able to possess an equipment type that their corresponding location doesn't allow. However, ZODB's lack of schema or any other constraints makes it hard to enforce that.
- Booking occurrences are calculated on-the-fly. Booking objects keep track of the start and end data of the reservation as a whole, but the actual occurrences (in the case of recurring bookings) are generated in runtime, as are collisions checked.

The opportunity to re-design the DB schema brought the added benefit of being able to address these issues.

Finally, some helper scripts were implemented in order to easily generate a graph of the schema and a *UML* diagram of classes. Understanding what is what from code inspection as a start may be a waste of time. Hopefully, these scripts will save some time in development, as they provide a high level view of the system that can be very useful. Fortunately, not all DB system suffer from the same issue as ZODB in terms of lack of tools: PostgreSQL, for instance, has pgAdmin, a powerful inspection tool.

6.4 ORM - Object Relational Mapper

A common problem that affects many modern web applications is that object orientation is not always easy to couple with relational models. An RDBMS is expecting tabular data. While a custom solution could be possible, fortunately there are libraries that provide this functionality out of the box:

- SQLALchemy
- Django ORM
- pony
- peewee
- Storm

We have chosen SQLAlchemy. Django ORM would be an option, but it would be more appropriate if Indico used the Django web framework. Storm is developed at Canonical and has shown capabilities at the scale of Launchpad but other than that, there aren't many products. pony and peewee have a smaller user community and less features compared to SQLAlchemy.

SQLAlchemy is a very stable library approaching its 1.0 major version and preferred by Mozilla, Nasa, OpenStack. Some of its nice features are:

- Architecture: SQLAlchemy is composed of two layers, core and orm. ORM layer makes it easier to play with objects but more fine-grained control is tuned by the core layer.
- Declarative DDL, Data Definition Language.
- Unit of work: SQLAlchemy prevents excessive communication with the database, instead everything is put into queues and batched in one go.
- Powerful query generation: SQL construction from Python functions and expressions.
- Modular and extensible: Custom types, custom compilation of queries.
- Eager/Lazy loading and caching

Recently, the Flask web micro-framework was integrated into Indico. Before that, Indico had its own WSGI-compatible request handling and dispatching mechanism. Flask is now doing that job behind the scenes (even if there are some features in Flask that Indico is not yet using at their full extent). Flask-SQLAlchemy was chosen to provide tighter integration between Flask and SQLAlchemy.

6.5 MVC - Model View Controller

Many parts of Indico are not as modular as they should be. Every function related to one layer is put into the same file. Even though a lot of work is being done in order to improve that, some older files still have as much as ∼15000 lines. Navigating, understanding and editing it are difficult. Even text editor may sometimes freeze while editing such a big file.

Since we're refactoring, why don't we make our lives easier for the future? MVC (Model-View-Controller) pattern is a well-known software pattern that splits applications into interconnected parts. MVC enables separation between how data is represented within the system, how requests are handled and how data is shown to users.

Adapting the MVC pattern extensively in the new database-enabled code, in order to create better and manageable software modules.

6.6 Models

We use SQLAlchemy and Flask-SQLAlchemy and their declarative DDL.

Firstly, a base SQLAlchemy object that provides an interface with the database is put into `indico.core.db`. This object makes table and type primitives available.

Secondly, Models, which represent business objects, are put into their own separate files. Each class attribute is set to a column object of a specific type which is retrieved from the base *DB* object.

At runtime, after the Flask application is created, the *DB* object is imported and configured. Flask-SQLAlchemy comes with reasonable defaults. However, *DB* is configured with values retrieved from the main Indico *config* object, which itself contains Indico's default set of parameters for SQLAlchemy.

After getting an application context from the flask application and setting application of *DB* object to this application, each model that is using SQLAlchemy should be imported to resolve interdependence between models. At this time, we can drop existing database and/or create database. Models are ready to use.

6.7 Distributed Queries

ZODB can't be replaced from day to night so the chosen RDBMS and ZODB will have to coexist until full transition is achieved. Therefore, distributed queries must be supported. If one of databases returns an error, the transaction must be aborted and the connection state must be rolled back.

Once again, this problem could be solved with a "homemade" implementation, but, luckily, SQLAlchemy's *session* object, ongoing transaction and local cache, is extensible enough to allow better than that. Therefore, adding /removing objects to/from the *session* may be done by the application developer while commit/roll-back is called by an extension. Moreover, ZODB's transaction module can manage external transactions in addition to its own ZODB transaction with a helper that provides a common ZODB interface. The architectures of these libraries nicely fit into Indico's own structured request handler hierarchy. Every request handler extends the *base* request handler which is the only point where transactions are committed or rolled back.

To make this design possible, we use *zope.sqlalchemy* which is a joint effort of ZODB and SQLAlchemy developers. Since only the base request handler in Indico deals with transaction logic, developers are not exposed to it.

Making *base* request handler only responsible, developers aren't needed to deal with transaction logic any where else, they only manipulate objects, instead and *base* request handler handles the rest because when a request has come, one thread-local *session* is created and one transaction is started. Developer uses this *session* to implement application logic and at the end, *base* request handler checks session for changes. If there are changes, a commit is attempted. Otherwise the transaction is rolled back and an appropriate error message is displayed. Finally, the transaction is finalized and thread-local *session* is closed which means the request has ended.

There is a subtle "nuance" in all of this because *db* object must be created using the *zope.sqlalchemy* extension. However, this extension is not always needed: migration scripts (ZODB → RDBMS) and unit tests do not need it. When it's used with no need, it causes more troubles than benefits so this object has to be created dynamically. Since *db* object is the main building of database layer, it's in use literally everywhere. The only logical place to add this capability is `indico.core.db`. With recourse to some Python magic, this can be done. Caller modules set `__no_session_options__` global to *true*. While *db* is being loaded, *db* module checks that specific global in the call stack frames of parent modules. If global is seen, then pure vanilla SQLAlchemy object is imported without any extension. Otherwise, it is loaded with ZODB transaction extension.

6.8 Queries

In ZODB, accessing objects and their properties are easy but inefficient. Even if a small property of an object is needed, we need to load whole object and then access its property.

There is no possibility of doing projections. As said in discussion of database types, object is seen a big blob from the point of view of database.

Even worse, since we can't load object in batches in ZODB, there must be multiple back and forth connections between server and client. In terms of performance, this connection should be minimized into one.

Relational databases helps us to overcome these problems at the expense of an increase query complexity and portability between different databases.

6.8.1 Complexity

In ZODB, main implementation is to get all objects one by one while doing filtering and computation on client with retrieved object and putting it into result set if it satisfies. Some tricks are already implemented to efficiently get objects such as indexes. If every reservation within one particular day, for example, is put into a Python list, all reservations can be loaded easily for requesting that list. This computation is *pre-map-reduce* era such that data is brought for code. It should have been the other way, moving code to data, since costly one is moving data like favoured by *map-reduce* paradigm.

In RDBMS, we can create a huge filter, query, and send it to server where data lives and get back only what is needed to satisfy the request. Since schema is fully normalized, accessing data requires joins and combining different types of data to make only one request to the database server.

For example, to show booking interface to user, rooms and their capacity are needed but also max capacity should be known to initialize filters. There are two options:

- Getting all rooms and then iterating over them and finding max in client
- Getting all rooms and max capacity at the same time from server

First option is easier but not efficient as much as it could be. Second option is blazingly fast since database has already an index on capacity such that there is no need for computation, it's only one look-up. However, retrieving rooms and one integer in the same result is hard because each record should have same structure which is *not*. To get all into same structure, max capacity should be faked into a room object.

Another example may be showing availabilities of rooms in some time span because a tree-like structure should be retrieved from server in one go such that there may be multiple rooms with multiple reservations within one day. Since it's favourable to return one record for each day, rooms and their reservations must be aggregated somehow. However, this aggregation isn't a simple SQL function like *min* and it's more similar to concatenating records. Making this aggregation without powerful types supported by database such as *array* in *PostgreSQL* to keep queries portable loses type information.

As in examples, there is a trade-off between performance and complexity. As it gets more performant, it gets more complex. However, queries are written once and updated occasionally but they are run frequently. That's why performance is chosen over complexity.

6.8.2 Portability

Making queries better in terms of performance requires us to optimize for one database. This breaks portability because some aggregation function supported in PostgreSQL isn't

available in MySQL, for instance. Using SQLAlchemy abstracts database but if specific extensions and types are utilized, then it's hard plug and play a different database. Likely, SQLAlchemy is extensible and provides a compiler to be extended.

For example, we would like to aggregate one column into an array in PostgreSQL with *array_agg* method. However, this method isn't available in MySQL so it'll cause an error. Compiler provided by SQLAlchemy generates custom SQL for each database. If we write an *array_agg* compiler construct and we can supply with information to call default *array_agg* on PostgreSQL and do custom string concatenation on MySQL to get the similar result.

Main goal is to prevent ourselves from using specific methods and types as much as possible and when it's not possible, one compiler construct for respective method is provided. Even if we try not to diverge from common features, queries are optimized for PostgreSQL since it'll be production environment at the end.

6.9 Testing

A complex system is getting more complex by entering into a transition phase in which many libraries are being integrated. Thus, unit tests are required to validate that same functionality as refactoring has been kept. Flask and its ecosystem are being more utilized in Indico so we chose using *Flask-Testing* to write unit tests.

Flask-Testing automatically populates and drops database for each to decrease coupling between tests. Due to the usage of *Flask-SQLAlchemy*, we should be in *app_context* or *test_request_context* which mean that application must be set for *db* object or one request should be qoing on, respectively. *Flask-SQLAlchemy* provides *test_request_context*, creation and dropping of *session* for us.

6.10 Controllers

Controllers are moved into their new place, *indico.modules.rb.controllers* and divided into small self-contained packages such that:

- admin
  - locations
  - rooms
  - reservations
- user
  - locations
  - rooms
  - reservations
  - blockings
- decorators
- forms

- mixins
- utils

There are two main packages; namely, *admin* and *user*. Under each one of them, specific controllers are laid out. Common functionality is put into top level such as decorators, form classes and utility methods.

With this structure, navigation within code base is faster and since related functionality is clustered in one small file, finding one piece of information is quickier.

6.10.1 Forms

Most complexity of controllers code comes from validation of form fields. No form library is utilized and validation of each field manually which is repetitive and error-prone. Since huge body of controllers is changing, field validation may also be off-loaded to 3rd party form library.

The most popular and stable library for forms is *WTForms* and we used it with Flask and SQLAlchemy extensions to automate validation.

Request handlers in Indico has mainly 3 important methods:

- authorization check: put into package base nicely
- parameter check: used to create a respective form object and validate
- process: generates response by using validated form data

Controllers are light-weight because nice package structure enabled to get authorization check out of picture. Process is implemented as one query in appropriate model. Now, with integration of WTForms, parameter check is delegated. As a result of these transformations, controllers composed of hundreds lines of code is now a simple glue between models and forms.

WTForms provides:

- A declarative language for fields as SQLAlchemy does for models.
- Type casting and many default validators as well as ability to write custom validators
- Custom types
- Automatic *CSRF* protection which isn't used since Indico has already protection in a higher level.

6.10.2 More Flask

Flask is integrated into Indico because

- there was a legacy hard-coded request dispatch mechanism which should be automated and extensible
- custom solution of Indico was like reinventing the wheel which is costly in terms of time and money

- custom solution is hard-coded which is repetitive and error-prone by nature but Flask creates an abstraction layer to easily manage URLs and is more resistant to errors.
- Flask does automatic type validation and conversion for URL params
- Flask provides better compatibility with low-level WSGI server implementation.
- All of the previous enables beautiful URLs which are better for users to remember as well as crawlers for SEO, search engine optimization.
- Flask provides powerful thread-local *flash*, *request* and *session* objects to show messages to users, to access request related information and to access all data for the current user, respectively.

Even if Flask could bring above features, it's just a facade between WSGI and Indico custom request handler logic because it's only used for URL mapping. For instance, request handlers get their request data as parameters from base request handler by copying. This is unnecessary since request handlers are already able to access request data from *flask.request*. While base request handler is being refactored for distributed queries, it's also updated to pass request data conditionally so that new request handlers have a simpler signatures and access their data from *flask.request* while keeping compatibility with old ones.

Parameters are passed back and forth to notify users for action success or fail. Interface shows respective message according to action status. However, in refresh, since parameters are a part of URL, same message will be displayed again which is misleading and ugly. Therefore, new request handlers use *flask.flash* to show messages only one and getting rid of parameter navigation clutter.

6.11 Views and Templates

The same structure of controllers is replicated under views. That enables easy navigation and more understandable code base such that if there is a request handler in *indico.modules.rb.controllers.x.y.z*, its respective view is simply in *indico.modules.rb.views.x.y.z*.

Templates are mainly updated to keep consistency and naming scheme. However, in some interface, there were legacy inefficient JavaScript and old widgets that may be better to change for consistent user interface. Pure JavaScript is mostly rewritten in *underscore.js* to offload browser quirks to library and have a modern, concise code since it's already in use. Old widgets are replaced by jQuery UI widgets such as *DatePicker*.

6.11.1 Unicode

Mako template engine (developed by main SQLAlchemy developer) is in production at Indico and Mako provides an option to disable Unicode explicitly. Until now, fields are Python byte strings and internationalization engine returns UTF-8 encoded strings so currently templates don't support unicode objects. Using deeply Flask and SQLAlchemy makes transition to Python 3 faster by using unicode objects everywhere but until that time, refactored modules, room booking for now, differs from rest. Location, room or

reservation objects can't be passed directly templates. Their properties must be converted into byte string by encoding before generating result.

This problem can be solved in two ways:

- Calling encode on properties whenever it's written into template
- Writing a custom template to encode unicode objects and setting it as first default filter in template engine

First one is verbose and error-prone so second one is chosen not to bother developers with a bad result of transition phase. Second implementation is worse in terms of performance but pros and cons are compared, it weighs heavier such that a new developer that isn't informed about this quirk may not be aware of it.

In overall, ZODB version wasn't supporting Python 3 so getting rid of main dependency of Indico and tight integration of Flask, WTForms and SQLAlchemy are a huge push for the adaptation of Python 3 in Indico.

7

Validation of PostgreSQL Decision

Porting Room Booking module to PostgreSQL via SQLAlchemy has proved that PostgreSQL is a good solution for the problems of Indico.

7.1 Comparison of PostgreSQL to Document-Oriented Databases in Room Booking

As seen in schema of Room Booking, it's a tree from left to right, locations to reservations. Document-oriented databases are tailored this kind of data. If whole data is composed of only room booking, then choosing a document-oriented database would be a better fit since only one entity is modified and document-oriented database such as MongoDB or CouchDB provide ACID-like guarantees in the document level.

In PostgreSQL, schema is fully normalized and accessing objects on the right such as repetitions of reservations usually requires joining of locations and rooms.

Even if access flow gets more involving, writing queries with SQLAlchemy seemed to be comfortable because SQLAlchemy provides high level constructs such as directly mapping joins to lazy loaded collections in Python objects but it also enables literal SQL if needed. Thus, SQLAlchemy is in the both end of abstraction, it can be high and very low level at the same time according to needs.

Joins are the worst point of relational world. Main feature provided by room booking is searching for availability and then choosing a room. This feature roughly requires join of whole database. In ZODB, this big join was being done in client after inefficiently loading many unnecessary objects and generating repetitions. With B-Tree indexes on materialized reservation repetitions, PostgreSQL performs it instantly.

7.2 Database Size

Firstly, there was a packing need in ZODB to remove old versions of objects. With transition to PostgreSQL, this problem is solved for free.

Secondly, our biggest table in room booking is reservation repetitions with nearly half million rows which weren't put into database before. Even if they are materialized and room photos are put into database instead of being served from file system, database size is dramatically reduced compared to ZODB. It's around $\sim$400 megabytes and, now $\sim$130 megabytes, even $\sim$ 60 megabytes without occurrence and photo table. Thus, it

Table	Size
pg_toast_141558	37 MB
reservations	28 MB
reservation_occurrences	27 MB
reservation_edit_logs	14 MB
reservation_occurrences_pkey	14 MB
reservation_edit_logs_pkey	4736 kB
reservations_pkey	3808 kB
pg_toast_141558_index	432 kB
pg_toast_2618	336 kB
pg_toast_2619	136 kB
rooms_attributes_association	72 kB
rooms_attributes_association_pkey	64 kB
rooms	56 kB
rooms_equipments_pkey	40 kB
rooms_equipments	24 kB
room_equipments_name_location_id_key	16 kB
ix_locations_name	16 kB
ix_start	16 kB
ix_end	16 kB
locations_pkey	16 kB
pg_toast_2618_index	16 kB
room_equipments_pkey	16 kB

Table 7.1: Size of Room Booking Module in PostgreSQL

is expected that main storage will occupy ∼7 gigabytes after complete transition which around ∼40 gigabytes, now because room booking showed PostgreSQL saved %560 of space.

Exact size information can be seen in the table. Some rows which don't exist in schema, *pg_toast* rows, are written in PostgreSQL output. PostgreSQL has a fixed page size, generally ∼8 kilobytes which uses to load or flush records in batch. Moreover, PostgreSQL doesn't permit large records to span multiple pages because it introduces complexity and inefficiency such that modification requires twice of time. PostgreSQL transparently divides these big rows to multiple small physical rows and it's called TOAST which also supports simple and fast compression. Therefore, our photos is a nice use-case for TOAST storage and as seen in table, *pg_toast_141558* is our photo table.

7.3 Documentation and Tooling

One of our main differentiating factors in choosing new database was community and room booking porting experiment has confirmed that PostgreSQL and SQLAlchemy have really good documentation and vibrant community. ZODB barely has had documentation and some technical articles are occasionally published but that's all. However, getting questions answered instantly isn't literally false with many contributors on respective IRC channels, dedicated Stackoverflow users and active developers.

7.4 Summary

Expected features of PostgreSQL are validated and seamlessly and efficiently works. ZODB implementation is mirrored to PostgreSQL but PostgreSQL enables features which are impossible in ZODB due to loading strategy.

For example, in ZODB, rooms can be loaded in different ways such that:

- Only one room at a time
- Subset of rooms if they are put into a specific collection, such as all rooms

Since only collections can be retrieved in one connection, would-be-needed rooms should be put into a collection in advance. When multiple different subsets are needed, many different collections must be arranged n database to be able to retrieve all of them in one access. Therefore, easiest way is to load everything and send them to client but this is costly. To overcome this problem, infinite scrolling is implemented. Implementing infinite scrolling in ZODB was very difficult but PostgreSQL now makes it straightforward which may be leveraged by many pages of Indico such as reservation availability listing and hierarchy of categories.

8

Conclusion

Indico has become ubiquitous at CERN and so it started to see exponential increase in usage. Database back-end couldn't cope with high traffic and several improvements are adapted as workarounds such as application level indexing, usage of Redis as caching layer and separating costly query flows into an independent database like room booking module. These improvements were satisfactory but problem wasn't solved and it was just delayed for some time because development of new features efficiently was getting costlier in every passing day. The solution, database change, was obvious but also very involving that prevented it from being pronounced until now.

Database will change but which database should be used requires details analysis because

- Database change is very expensive
- There are many candidates in very different types
- There are slight differences in features of candidates but also drop-in replacement is very rarely possible
- Difficulty in estimation of future which combines features, traffic and services around Indico

In the light of these issues, we made a quite broad comparison within candidates and decided on PostgreSQL. Moreover, it's decided to incrementally extract modules and port into use new database so as to make aging complex code base more modular and extensible. Throughout improvement phase room booking part was already put into a different database since it has the most involving queries. Therefore, it's a good starting point with minimal dependence to main database and a need for a powerful back-end.

Incremental change started with room booking part. Since ZODB and PostgreSQL will be coexist until transition is complete, distributed query machinery is put into place. Code structure is changed to conform MVC pattern. Our custom processing in controllers is delegated to libraries as much as possible which has provided better compatibility with Flask (web), SQLAlchemy (back-end) and Unicode in overall.

As room booking is being ported, PostgreSQL has shown that it's a natural fit for Indico schema. Expected performance and size improvements are achieved. Even if PostgreSQL performs well in itself, Redis is still in front of PostgreSQL to off-load some of its traffic.

To sum up, very important steps are taken so far but there are still huge body of code waiting to be renovated. [18]

References

[1] **Joel on Software - Things You Should Never Do**. 3

[2] **Rewrite Code From Stratch**. 3

[3] Rob Pike Brian W. Kernighan. *The Practice of Programming (Professional Computing), Chapter 5 - Debugging*. Addison-Wesley Professional, 1999. 3

[4] Jeffrey Dean Fay Chang et al. **Bigtable: A Distributed Storage System for Structured Data**. *OSDI*, 2006. 3, 22, 24

[5] Mike Burrows. **The Chubby lock service for loosely-coupled distributed systems**. *OSDI*, 2006. 3

[6] Sanjay Ghemawat Jeffrey Dean. **MapReduce: Simplied Data Processing on Large Clusters**. *OSDI*, 2004. 3

[7] Werner Vogels et al. **Dynamo: Amazon's Highly Available Key-value Store**. *Symposium on Operating Systems Principles*, 2007. 3, 22

[8] **Paying Down Your Technical Debt**. 12

[9] **Technical Debt Quadrant**. 12

[10] Kevlin Henney. *97 Things Every Programmer Should Know: Collective Wisdom from the Experts - Act with Prudence*. O'Reilly Media, 2010. 12

[11] Prashant Malik Avinash Lakshman. **Cassandra — A Decentralized Structured Storage System**. *SIGOPS Operating Systems Review*, 2010. 22

[12] Nancy Lynch Seth Gilbert. **Brewer's Conjeture and the Feasibility of Consistent, Available, Partition-Tolerant Web Servies**. *ACM SIGACT*, 2002. 22

[13] *A Universally Unique IDentifier (UUID) URN Namespace*. 22

[14] Howard Gobioff Sanjay Ghemawat and Shun-Tak Leung. **The Google File System**. *ACM Symposium on Operating Systems Principles*, 2003. 24

[15] Google Inc. **Spanner: Google's Globally-Distributed Database**. *OSDI*, 2012. 24

[16] Jim R. Wilson Eric Raymond. *Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement*. Pragmatic Bookshelf, 2012. 48

[17] Michael Stonebraker and Lawrence A. Rowe (editors). **The Postgres Papers**. 1987. 51

[18] Shashank Tiwari. *Professional NoSQL*. Wrox Press, 2011. 67

Declaration

I herewith declare that I have produced this paper without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such. This paper has not previously been presented in identical or similar form to any other examination board.

The thesis work was conducted from 17 September 2013 to 14 Mar 2014 under the supervision of Pedro Ferreira at CERN and Karl Aberer at EPFL.

GENEVA, Switzerland