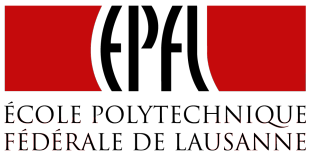
Indico: Behind the Scenes of CERN events



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 for the fast development of many features of Indico 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: SQL, NoSQL, Python, Web Development, CERN

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

Contents

1

Introduction

1.1 Overview - What is Indico?

Indico (Integrated Digital Conferencing) is a web application for event management, initially developed by a joint initiative of CERN, SISSA, University of Udine, Nederlandse Organisatie voor Toegepast Natuurwetenschappelijk Onderzoek (TNO), and University of Amsterdam. Currently, main development is going on at CERN with occasional contributions from outside such as time zone support by Fermi Lab.

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

In addition to events having a type, each event belongs to one category which enable category-driven navigation between events. Since finding interested event in possibly huge number of categories, time-driven (day/week/month) view is also supported. Even time-driven navigation isn't enough because a user, extensively uses Indico, can easily see tens of events in a daily view. To remedy this problem, personal dashboard is being developed in which approaching events are shown in sorted order for 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 manages full conference cycle starting with customization of website. 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 others, for instance. Indico makes setup of access and modification rights easy. Conferences are nothing without registrants and Indico provides a powerful fully customizable wysiwyg registration form. Indico supports creating custom forms in addition to registration form such as to prepare a survey. Call for abstracts, rich text editing in Latex and Markdown, and paper reviewing are enabled by Indico. Indico is a document store as much as being metadata store but saving different kinds of files; pictures, slides, videos, etc. Indico is even capable of converting documents betweeen different formats such as automatic PDF generation from Microsoft Office slides. Everybody has her different flow and tools to manage her calendar. Indico allows different views of same event and exporting events in multiple formats such as HTML, XML, iCal, PDF, etc. Last but not least, Indico is extensible via plug-ins to integrate with other systems or to provide more localised services so that only Indico is sufficient for everything. Room booking to arrange event place or Vidyo for video conference may be given as examples of notable plug-ins.

Extensible architecture of Indico have enabled easy integration with different services. As time advanced, search engine (Invenio) is added to find catgories and events easily. Indico totally embraces mobile and recently mobile check-in application is developed. Video conference (Vidyo and EVO) and instant messaging (XMPP) are easy to use within Indico. Payment processing for registrations in different ways (PayPal, PostFinance, WorldPay, etc.) is possible. Location, room and booking management are made possible with a clear interface for places and schedule of events. As a result, 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.

1.2 Motivation - Why change?

Main driving force in development of Indico was a required service to manage events as following agenda of CDS, CERN Document Server. 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 must have been minimized.

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

ZODB (Zope Object Database) was a viewable option in 2002 for Python, main programming language of Indico. Since ZODB is an object database, it enables directly manipulating and transparently storing objects. This capability of data store eliminates a layer between objects in programming language and data in store which in turn, saves a lot of time, money and complexity in application.

Fast forward to 2014, ZODB is far from having medium sized community. What is saved before by using ZODB is now being paid to solve limitations of ZODB. Finally, net gain turned into negative which signified to replace ZODB with more mainstream, powerful and scalable data store which will support next decade of Indico.

1.3 Approach - How change?

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. It's developed for more than 10 years, leaving ZODB means discarding this accumulated knowledge. Moreover, unless problems are made ridiculously clear, clean solutions can't be found.

We will present these reasons and the ways how we tried to solve these limitations in existing system. After we become certain about that refactoring would be more cost-effective, next step is to define criterias for the replacement technology in terms of where ZODB has failed and also keeping the future of Indico in mind.

In the golden era of data store, started by Google and Amazon to implement their extreme requirements, there are zillions of candidates, even barely knowing each of them is a really difficult task. Goal is to collect and classify candidates that conform to criterias as much as possible.

After first elimination and classification, next round is to get familiar with each one via implementing a very small subset of Indico. Implementation is important because making sense of abstract concepts is difficult to interpolate expectations. Even if feature implemented is small, it makes easier to predict the schedule and complexity.

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

In either case, transition phase is long and painful so it should be well-planned in terms of bugs, needs of community and release cycle. When 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, we will give every tiny detail from the study of limitations of ZODB to new implementation with new polyglot database system. Moreover, database change involves refactoring 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 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 for better scalability and facilitate the development of certain features such as statistics and social features.

2

Data Storage in Indico

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

2.1 The Early Days

Since the early days, Indico has been 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 (which is the number one reason of why PHP is still so popular, isn't it?).

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 (mainly due to the increasing usage of the Zope framework), non-relational DBs 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.

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. 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 is 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 DB
  - 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" (Object-Relational Mapping) and is nowadays a common feature of many web frameworks (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.

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 article, 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 storage. However, it also comes with its own problems; data must be synchronized with main DB which requires extra overhead for writes and maintenance cost of extra code (7 Lua scripts for server-side queries).

2.3 The Need for a New Storage

The evolution of Indico's feature set has been increasingly towards a more personal, user-centered application with an emphasis on the events that matter to each particular user. 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 favorite 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 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 *BTree* and *TreeSet* (implemented using BTrees), two structures aimed at large volumes of data that need to be stored in chunks and accessed in sub-linear time. BTrees 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.

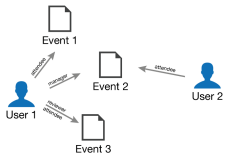
The simplified logic on the server side together with the absence of a feature that can be compared to "stored procedures" leave the developer no other option than implementing everything on the client side. While this can be seen as a way of keeping things simple, it is highly inefficient as it implies 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 implement the full query logic on the client side.

2.3.2 An Example: The Dashboard

The User Dashboard has been 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 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.



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 OO-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 BTree, 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 BTree, 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 subproblem 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

Limitation	Explanation
Ad-hoc queries	...
Indexing	...
Caching on the Server Side	...
Replication	...
Community	...

Table 2.1: Limitations of ZODB

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 BTrees, 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 highly interrelated connections between entities so that they can be accessed from many different points. Therefore, ability of querying arbitrarily and doing projections is vital from performance point of view. Overcoming this problem is possible via caching but it risks consistency of data and brings maintenance burden of extra code which means developers are working in spite of DB, not for new features with the support of DB. Rapid development, the reason of why ZODB is chosen at first isn't true any more. Moreover, its 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 Criterias of New Database

There are mainly 6 categories:

1. *Availability*: Replacement technology must be open source since Indico is an open source application, users can not be forced to use a commercial solution. Moreover, DB license should comply with the licenses of other software packages used.

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

3. *Ease of Use / Development*: DB should have good tool support for Python, main development language of Indico and it should decrease the time to implement new features so it should be transparent as much as possible like ZODB. Complexity of a feature should be come from application logic, not the commanding DB for data.

4. *Transactions / Consistency*: Writes touch many entities at the same time due to devilish dependencies between entities. Thus, transaction capability of DB is deadly important to keep data consistent.

5. *Community / Momentum*: Indico aims solving conference management problem, not the nitty gritty use-case details of a specific DB. When a problem is encountered, respective solution should mostly have been studied before. Successful deployments are an important representative of this requirement.

6. *Cost / Exit Strategy*: Transition costs should be minimized and also dependency to DB should be as low as possible to decrease the exit cost when time to pay technical debt has come.

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

3

Survey of Candidates

In this chapter, we try to look into many databases as much as possible to compare strong and weak places and to find the best possible fit.

3.1 Main Database Types

Database systems are mainly divided into 6 categories:

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

Object databases have been around since object-oriented programming has become popular by Small Talk at 1970s. Although they have a long history, their main showcase started with 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 no very successful products which leaves us with small number of choices to consider. ZODB and WakandaDB are the main viable systems.

3.2.1 WakandaDB 

WakandaDB is very new, only half year passed since first public stable release at the time of writing. It pushes the motto of *JavaScript is at everywhere* which is a new wave pioneered by *Node.js*.

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

- *Wakanda Studio*: IDE with full featured debugger
- *Model Designer*: Classes and schema are easily created
- *GUI Designer*: Makes easier creating visual interfaces
- *Web Application Framework*: Plug and play framework to use WakandaDB with 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 

The Zope Object Database is an object-oriented database to transparently persist Python objects. In the beginning, it was a 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 called *root*. Objects must be attached to *root* or one of its branches to be persisted. Thus, persisted objects can be accessed by starting from root and following links to children of *root*.

ZODB has powerful features in addition to being transparent storage:

1. ACID transactions
2. Version control for objects which causes packing problem unless properly automated. Each version of objects are stored which dramatically increases database size. In advanced usage, old version of objects may be needed but Indico doesn't require so old objects are deleted daily(db is *packed*). This may be a serious problem for a system heavily in use with limited disk space. Unfortunately, Indico fits into this system class.
3. Pluggable storage

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 ACID guarantees.
Community Momentum	New project so it's normal that 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 stack is a huge time-saver and JavaScript is living its golden age so the choice of *JavaScript at Everywhere* makes sense but results aren't obvious yet.
Cost Exit Strategy	Starting to whole tool chain would be huge but if only server component is utilized, cost can be reduced. Even using only server 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	★★★☆☆☆

   (a) File: Only one file on file system and it's in use Indico.

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

   (c) Relational Database: Brings independence from one particular product at the expense of extra complexity and overhead.

4. Client caching: Just a remedy for the lack of caching in server

5. MVCC: Reads aren't blocked while only one write is allowed.

6. Replication via *ZRS* (Zope Replication Services): It is recently open sourced (May 2013). It was totally out-of-picture before due to its ridiculous price. ZRS enables 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 many successful products in which Indico exists.

One interesting drawback of ZODB being very tightly coupled with code base is that each object is stored by their full class name to disk. That prevents moving classes around without a migration step. *Makac*, Make a Conference, name was chosen for Indico at first development stage. As a result, code base contains a lot of code under *Makac* package or prefixed by *Makac*. Indico name is the final which means Indico should replace *Makac*. It wasn't easy like just copying files but database change makes this required adaptation easier. Therefore, what we wished for at start seems possible such as modular Indico.

Table 3.2: ZODB Object-Oriented Database

Criteria	**Feature**
Availability	Zope Public License
Scalability	Although there are solutions like ZEO and ZRS (recently), it's limited for the Indico use case which involves multiple big entities and their relationships such as conferences, avatars, reservations. ZODB is successful for big entities but their relationships is where ZODB performs poorly.
Ease of Use Development	ZODB is more like a library than standalone program which makes setup easy. It's tightly integrated to code which is both an advantage and a disadvantage. ZODB puts or retrieves objects without interfering with you but development should be done as it expects. The result is that ZODB is barely seen while you're developing with it but when you would like to change, you can't because it's in any tiny part of code
Transactions Consistency	ACID
Community Momentum	Even if it's most well-known and mature object database in Python world, it's a niche project by being object database. Documentation and surrounding tool chain are lacking despite its long history. In overall, the company behind of ZODB is in the decline overall.
Cost Exit Strategy	Starting to use ZODB is easy by simple setup and heavy integration with Python. These are main reasons why it's chosen but not interesting any more like exit and exit isn't easy again due to same reasons; namely, tightly-coupled code base.
Score	★★★☆☆☆

3.3 Relational Databases

A relational database is a collection of tables of data items (*row*) in a relational model. Each table has a strict schema which specifies data parts, their types and their relationships to each other. Each row has a *primary key*, composed of one or multiple columns, uniquely identifies itself. Relationships can be established within a table or between tables via *foreign key* which is a pointer to primary key of some table. Foreign keys

provide various level for normalization of tables. Normalization is a methodology to avoid data manipulation anomalies and loss of integrity by preventing redundancy and non-atomic values in table design. After schema is normalized, 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 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. Moreover, normalization requires touching multiple tables at the same time to join data but relational database systems are fully ACID compliant with tunable guarantees and levels such as row or column level. Furthermore, with the push of big data requirements, replication is supported at default. Finally, schema design and normalization level are deadly important for sharding. Having both replication at default and good design, scalability, even huge scale, isn't a problem.

Relational databases category is the one most battle-tested for at least 20 years and well-studied for the last 40 years. As a fruit of it, there are multiple very successful deployments, huge living community with extensive know-how, powerful tools.

3.3.1 PostgreSQL 

General properties are provided here but explanation of in-depth features are given later since PostgreSQL is the winner of our survey.

3.3.2 MySQL 

MySQL is the most popular relational database by far. MySQL owes its popularity to investing into its performance by lacking some features exist in PostgreSQL. This is a long story which has a dramatic end because PostgreSQL increased its performance in recent versions while MySQL has adapted most of the lacking features. Even if 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, one 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 and features, transaction guarantees, of storage layers are highly different. Thus, carefully specifying storage is important for a healthy comparison. In that respect, MySQL supports multiple storage engines; namely, InnoDB, BerkeleyDB, TokuDB and MyISAM, etc. but in terms of ACID requirements of Indico, development activity, and better comparison to PostgreSQL, InnoDB is our main concern.

Table 3.3: PostgreSQL Relational Database

Criteria	Feature
Availability	PostgreSQL License (BSD or MIT-like) and clear roadmap and open development
Scalability	Scalability is 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 library, SQLAlchemy, which is in production at Dropbox, Yelp, Uber, OpenStack.
Transactions Consistency	ACID
Community Momentum	One of two biggest communities in database ecosystem and it's stealing users from MySQL due to unclear future and commercialization since acquisition of Sun Microsystems, primary developer, by Oracle. Although many frameworks and PaaS providers support relational databases in general, they recommend PostgreSQL usage notably as a result of standard compliance.
Cost Exit Strategy	Integration of ORM layer makes code back-end agnostic unless back-end specific extensions are used. However, special features and types make code more efficient which can be even reduced by providing ORM with back-end specific package to transfer queries in the intended way.
Score	★★★★★★

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 Oracle acquisition, MySQL has become open-source product but lost its open source development culture. As a solution, some concerning old MySQL developers forked project but this results in multiple MySQL-like products in the market such as MariaDB, Drizzle, etc. Moreover, with the help of layered architecture, different companies are providing similar but different products with reusing SQL layer and plugging their custom storage layer such as Percona Server, TokuDB, etc.
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 diverging standard and lacking features. However, new owner Oracle, the company also has the biggest share in commercial database world, damages MySQL.

Table 3.5: MySQL Relational Database

Criteria	Feature
Availability	Dual license (GPLv2 and commercial) and what Oracle is trying to do with MySQL isn't clear.
Scalability	Scalability wouldn't be problem since it serves well even in the scale of Facebook, Google and Twitter.
Ease of Use Development	ORM layer is needed but code targeting for PostgreSQL with SQLAlchemy can be adopted for MySQL unless richer types of PostgreSQL are leveraged extensively.
Transactions Consistency	ACID in InnoDB
Community Momentum	According to various rankings, MySQL is the most popular open source database with huge community. Oracle damaged its reputation for openness. Forks emerged and followed closely with upstream until now but after stabilization of long waited 5.6 version, first version in NoSQL storm, forks started to diverging. MariaDB versions were one-to-one mappable with MySQL versions until 5.5 but with 5.6 version, they even chose to change version scheme to start with 10.0, for instance.
Cost Exit Strategy	ORM layer makes a relational database replacement easy so what is said for PostgreSQL is also valid for MySQL. Their difference are getting subtle and unimportant. Moreover, MySQL has forks for replacement without doing any change if they continue following trunk but they started to signalling detachment.
Score	★★★★★☆

3.3.3 SQLite

SQLite is an embedded relational database management system. In lay-man 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 pictures as an independent module within application and can be used as function calls like other methods.

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

Writes lock this file which means writes are executed sequentially but reads don't require that lock 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. With these execution characteristics, it's simpler than PostgreSQL and MySQL.

It's self-contained, server-less (no daemon for waiting client connections), zero-configuration relational database which makes it a universal solution for small needs but it's not capable of serving high load of production. As a result of this ease of use, it is preferred 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 complexity.

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 for only testing since no configuration is required and database can totally be kept in memory for running tests faster.
Ease of Use Development	ORM layer is needed as being a relational database system and it is supported by SQLAlchemy.
Transactions Consistency	ACID
Community Momentum	Due to its popularity (Android, Chrome, Firefox, Opera, Skype, ThunderBird, etc.), community is huge and it is well tested and documented (more than 2 million tests are run for each release).
Cost Exit Strategy	ORM layer provides good abstraction but lacking features raise question marks.
Score	★★★☆☆☆

3.4 Column Family Databases

Column-family databases are designed to handle large amount of data across many commodity servers at the scale of 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 row id as RDBMS, timestamp for conflict resolution and/or versioning, column-family and column name. Column-family is a container for columns and columns can be added and deleted whenever needed without any down time. Moreover, every row doesn't need to have same columns as RDBMS rows.

There are also column-oriented relational database. They see 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 this design difference, their features are more or less same well-known row-oriented RDBMSs. Commercial product, Microsoft SQL Server supports both storage design, for instance. Vocabulary is similar but they are very different designs.

3.4.1 Cassandra 

Apache Cassandra is an open source distributed (mainly) column oriented key-value store achieving high performance and availability. According to CAP theorem, 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 assigning same role to each node in the cluster. In addition, Cassandra supports master-less asynchronous inter-cluster replication for higher availability guarantees. Meantime, third dimension, consistency, can also be adjusted according to the needs of applications. Since there is no difference between nodes, performance linearly scales as much as new nodes are added into cluster without interrupting existing nodes.

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

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

- Single server: unique id is just an auto incrementing counter.
- Multiple servers: application logic (not a solution, escaping solving problem in database), 3-rd party service or ticket servers. Latter options bring extra components and they may also be a single point of failure and write bottlenecks.

As seen, solving this problem in a distributed environment is hard. Relational options aren't usually identical but Cassandra nodes are identical so this problem should be solved while keeping this architecture. Permitting each node generate ids may result in inconsistencies and lock is needed to prevent . However, lock means waiting and performance degradation where the main objective of Cassandra is high throughput.

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

- Generates id from data
- Uses Universal Unique ID (UUID), easily produced by the client

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

In RDBMS, fully normalization is recommended to prevent update anomalies. In Cassandra, there are more higher level abstractions to store 1 to many relations in a column; namely, set, list and map and Cassandra doesn't support joins and sub-queries except Hadoop tasks (Pig and Hive are also supported) so de-normalization is encouraged via manage logical relations by using these abstractions.

3.4.2 HBase

Apache HBase is open source clone of Google's BigTable. It's 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, system is actually scalable because 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 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 meta table stored in master which isn't up now.

HBase runs on top of Hadoop Distributed File System (HDFS). Adaptation phase of HBase becomes more complex due to HDFS setup and configuration but when it's there, replication of data comes free with HDFS. HBase is designed to crunch very very big data, peta-bytes of data. That's why HDFS requirement seems reasonable but if we think about Indico, HBase is much more complex solution than enough. 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 to be universal back-end at Google. Google is working on the next generation of BigTable called Spanner which is storage engine of Drive, GMail, Groups and Maps.

HBase doesn't support transactions but atomic operations are supported in the row level. ACID like guarantees provided by relational databases is achievable via de-normalization. However, nesting data requires more or less to know access patterns. Otherwise, ad-hoc queries would be a necessary but they aren't possible to lack of transactions. 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 middle compromise between key-value store and RDBMS. Key-value stores identify records by 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

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	★★★★☆☆

of both worlds 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.

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 trying databases in a small case, next logical step is to decide on one or poly-glot database system. We decided on running *ZODB* with *PostgreSQL* for main data storage. 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 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.

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. Inside of available document-oriented databases, we decided on *MongoDB* since it's 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*. Mobile version uses *MongoDB* since there is no need for ACID guarantee, ad-hoc queries are supported and lack of schema enables faster development. Due to nice fit between use-case and provided features by MongoDB, Indico mobile has been very successful. Data manipulation flow of new planned use-cases are very similar to mobile version. That's why *MongoDB* is the selected option.

[1] Software as a Service

Socialization of Indico requires graph processing in which graph databases excel but at the time of writing, there is no final decision. Current main focus 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 no successful big deployments with graph databases except Neo4j and custom/proprietary solutions such as Facebook. Waiting until we have to give a decision seemed to be better in that situation.

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 over complex in the scale of Indico.

Key-value stores are overly simplistic. It's very difficult to convert Indico schema to key-value storage and keeping it manageable. References between entities or higher level collections are needed to easily store/retrieve relatedness of entities. However, they have specific cases where managing data is easy such as url and session caching. Since the less main database is used, the better it is; Key-value store, *Redis*, is used and it'll be adapted wherever it makes sense but not for being main database.

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 every entity of complex Indico schema isn't mappable because entities are usually like trees and putting these *contains* relationship into graph database exponentially increases number of edges. Therefore, storing *contains* relationship into document 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 details but as a result, we find it to be in a fast development phase and not production ready.

After exhausting other types, we had two options; namely, document-oriented and relational databases. This is basically a trade-off between ease-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 in document level and what is 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 as prototyping. However, when other parts started to huge documents, we can't work on document level any more, there is a need of cross references between tables.

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

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

5.2 Chosen Database

Since we are looking for a relational database, we would take the lead of the *MySQL* cloud or *PostgreSQL*. The future of *MySQL* cloud is confusing while *PostgreSQL* is gaining popularity steadily and a much more tightly-connected community is coming life around it. In addition to these accessibility issues, getting *PostgreSQL* boxes at CERN is easier via a service provided by database team at CERN [1]. Choosing *MySQL* means diverging from other services at CERN which puts maintenance burden on the Indico team. We're already paying technical debt of choosing *ZODB*. Moreover, *ZODB* even needs little administration by being a Python library, except automated scripts such as daily packer, compared to *MySQL* which is outside of Python world. Finally, *PostgreSQL* has powerful features:

- 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
      - JSON
      - range
  - Table Inheritance
  - Auto Master-Slave Replication
  - Views
    - Materialized
    - Modifiable
    - Recursive
  - Notifications

[1] http://www.cern.ch
[2] ACID, 64 cores
[3] fully 92-compliant
[4] bitmap index can utilize multiple indexes at the same time
[5] content → row

  - 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 via Instagram
  - Heroku
  - Microsoft via Skype
  - Nasa
  - Yahoo

In the light of that study, *PostgreSQL* is chosen to replace *ZODB*.

One final remark is that *PostgreSQL* was an ugly academic project in early 2000s but today, *PostgreSQL* is undergoing a renaissance in features have attracted many small businesses and enterprises. Thus, knowing how to use *PostgreSQL* has become an important skill set and a valuable career path. Having a open source *PostgreSQL* product may have propelled the contributions.

6

Implementation

In this chapter, we share our experience of how implementation is planned and being done.

6.1 Big Picture

Indico has a complex product with many features. As a result, Indico is a big code base which is easily seen table **??** in comparison to some popular services.

Due to complexity, it isn't possible to change database instantly. Since we would like to see the result of our changes all the time, we need a working system. Therefore, main database machinery, firstly, is integrated and then each modules is refactored into using new database. At the same time, tests are being written to validate the behaviour of code. After one module is ready, it will be pushed into production to be tested there, too while other modules are being refactored.

6.2 Scope of Project

When ZODB started to perform badly, incremental optimizations were done. For instance, Room Booking - enables location and room management, and reserving rooms for events, is one of the most complex and involving modules and it's separated into its own database. Since it's not very tightly integrated into main database, it is the first module as a good starting point to get experience in refactoring, frameworks, tool chain and interpolate complexity.

6.3 Room Booking Schema

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

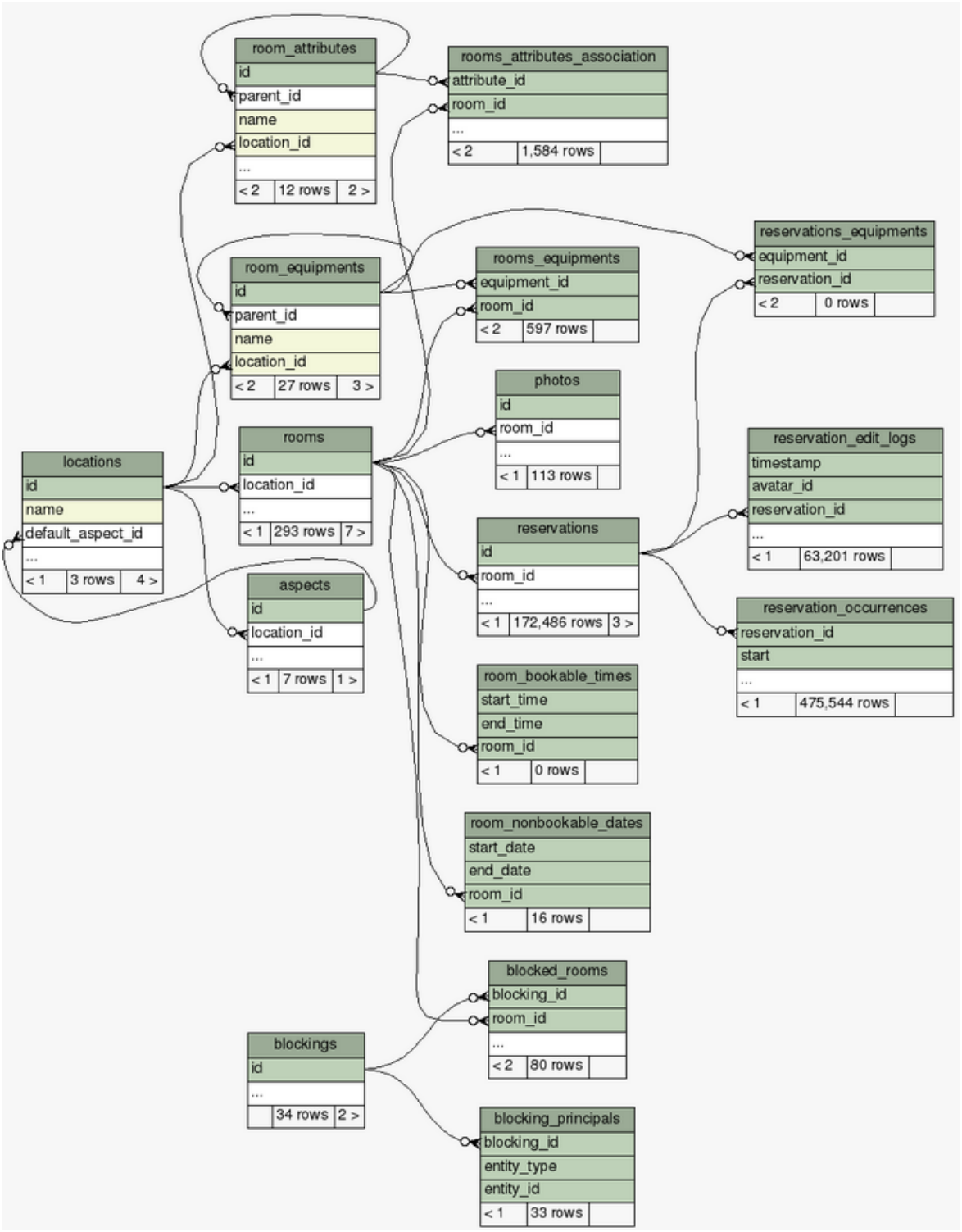
We started to inspect database with external tools to get the object structure. However, ZODB is an object database, inspection database without their mapped run-time classes is much more difficult to relational databases. Even if there are 3 available tools , non of them is stable and well-supported:

- zodbbrowser
- eye

- z3c.zodbbrowser (unfinished GSOC project)

We have tried all of them but *eye* was the most stable to our experience but even *eye* couldn't open our production database which is around 30 GB after one day of interaction, get biggers with every update to objects since storage mechanism of ZODB is to save every version of objects and only to put a reference to the latest version.

Viewer may be patched but responses with developer of module back and forth were slow and since we don't want to lose time with it, we worked with only empty database at first. We loaded data incrementally by testing to find the tipping point where it fails. Next logical step is to analyse the mapped classes which isn't an easy task because we are talking about classes composed of ∼2000 lines.



It's a bit challenge because classes in ZODB uses *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:

- Repetition enumeration for reservations: currently, there are only 6 options so their text representations and dates are calculated explicitly. Moreover, their implemen-

tation doesn't do what their description says. Adding one more repetition type isn't easy so their text and date generation must be humanized and automated.

- 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 source code. Each level of hierarchy is one list in room object, for instance.
- Location and Room attributes and equipments: Due to replication, there are inconsistencies. Room can't have an equipment which its location doesn't have but since there is no check, it's possible.
- Reservation repetitions are materialized. Reservation objects keep track of start and end date with repetition type. Whenever reservation request has come, these repetitions are generated in client and they're checked for overlap with the requested time span.

In addition to refactoring to use new database, these problems are also addressed with new schema.

Finally, some helper scripts are implemented to easily generate a graph of schema and *UML* diagram of classes. Understanding what is what from code inspection as a start may be time wasting. Hopefully, these scripts will same some time later by providing a high level of system. Actually, analysis of ZODB comes from lack of tools. In RDBMS, especially PostgreSQL, pgAdmin makes this kind of inspection much easier.

6.4 ORM - Object Relational Mapper

We are dealing with objects and RDBMS is waiting pure serialized binary data. This may be solved by ourselves but there are stable object relational mapper libraries for Python:

- SQLALchemy
- Django ORM
- pony
- peewee
- Storm

We have chosen SQLAlchemy. Django ORM would be a ridiculous idea without Django web framework. Storm is developed at Canonical and has shown capabilities in Launchpad scale but other than that, there aren't many . 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 easier to play with objects but more fine-grained control is tuned by core layer.
- Declarative DDL, Data Definition Language.
- Unit of work: SQLAlchemy prevents excessive talking to 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 compile of queries.
- Eager/Lazy loading and caching

Morever, Flask, micro-web framework, recently is integrated into Indico. Before Flask, Indico has a custom request handle and dispatch mechanism. Even if it's done by Flask now, Flask isn't fully utilized. We added Flask-SQLAlchemy to our arsenal with SQLAlchemy to more tight integration of Flask into Indico.

6.5 MVC - Model View Controller

Indico isn't as modular as it should be. Every function related to one layer is put into the same file. For example, in Room Booking module, each request handler is in the same file so this file is huge, ∼15000 lines. Navigating, understanding and editing it is 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 to divide applications into interconnected parts. MVC enables separation between how data is represented within system, how requests are handled and how data is shown to users.

We are adapting MVC pattern as refactoring for new database so as to have manageable pieces.

6.6 Models

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

Firstly, one SQLAlchemy object for database interface is put into *indico.core.db*. This object makes table and type primitives available.

Secondly, each model is put into its own file. After importing *db* object, each class member is set to a column object with specific type from *db* object.

At runtime, after Flask application is created, *db* object is imported and configured. Flask-SQLAlchemy comes with reasonable defaults. However, *db* is configured with values retrieved from main Indico config object which contains Flask-SQLAlchemy defaults at default.

After getting an application context from 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 isn't dropped at one night so RDBMS and ZODB should coexist until complete transition is done. Therefore, distributed queries must be supported. If one of databases gives error, transaction must be aborted and state must be roll-backed to previous status.

We may develop this functionality ourselves but luckily, SQLAlchemy provides extensibility for *session* object which is the interface to the *db* object and local cache. Therefore, adding /removing objects to/from *session* may be done by application developer while commit/roll-back is called by extension. Moreover, ZODB transaction module can manage external transactions in addition to its own ZODB transaction with a helper that provides a common ZODB interface. Architectures of these libraries is nicely fits into structured Indico request handler hierarchy. Every request handler extends *base* request handler which is the only point where transactions are committed or roll-backed.

To make this design possible, we use *zope.sqlalchemy* which is a joint effort of ZODB and SQLAlchemy developers. 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, they have been tried to be committed . If everything goes well, results are show, otherwise transaction is roll-backed and appropriate error message is displayed. Finally, transaction is finalized and thread-local *session* is closed which means request has ended.

There is a subtle detail because *db* object must be created with that *zope.sqlalchemy* extension. However, we don't always need that extension such as migration scripts (ZODB $\rightarrow$ RDBMS) and unit tests. When it's used unless needed, it causes more troubles than benefits so dynamic loading of *db* object is needed. Since *db* object is the main building of database, it's in use literally everywhere. The only logical place to add this capability is *indico.core.db*. It's implemented by malleability of Python. Caller modules set *__no_session_options__* global to *true*. While *db* is being loaded, *db* module checks that specific global in frames, call stack, 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. [1]

References

[1] Shashank Tiwari. *Professional NoSQL*. Wrox Press, 2011. 64

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