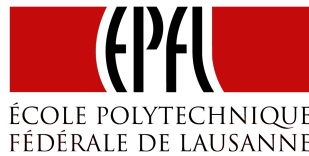


# Indico: Behind the Scenes of CERN events



**Ferhat Elmas**

CERN Supervisor: **Pedro Macedo Alves 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 to schedule and organise events, from simple lectures to complex meetings, workshops and conferences with sessions and contributions. The tool also includes advanced user delegation mechanism, paper reviewing, archiving of event materials. More tailored features such as room booking and collaboration are provided via plug-ins.

Indico is heavily used at CERN and in more than 10 years, very different features are added according to the needs of increasing number of users. However, data store(ZODB) have never changed. ZODB has been very important for Indico to have been Indico but it's a bottleneck now in terms of scalability and elapsed time in feature development.

Indico demands more scalable backend to serve increasing number of events with fancy features. The aim of the project is to replace ZODB with a highly scalable data layer which will enable global Indico, bringing  $\sim 120$  different Indico servers around the world together. First part of project involves the survey of different technologies according to the needs of Indico and second part is composed of preparation of transition environment with a prototype.

**Keywords:** SQL, NoSQL, Python, Web Development

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Overview - What is Indico? . . . . .	1
1.2	Motivation - Why change? . . . . .	2
1.3	Approach - How change? . . . . .	2
1.4	Outline - What is Done and Next? . . . . .	3
<b>2</b>	<b>Data Storage in Indico</b>	<b>4</b>
2.1	The Early Days . . . . .	4
2.2	The Decade of ZODB . . . . .	5
2.2.1	The Age of Caching . . . . .	6
2.3	The Need for a New Storage . . . . .	6
2.3.1	Indexing . . . . .	7
2.3.2	An Example: The Dashboard . . . . .	7
2.4	Selection Criterias of New Database . . . . .	9
<b>3</b>	<b>Survey of Candidates</b>	<b>11</b>
3.1	Main Database Types . . . . .	11
3.2	Object-Oriented Databases . . . . .	11
3.2.1	WakandaDB . . . . .	12
3.2.2	ZODB . . . . .	12
3.3	Relational Databases . . . . .	14
3.3.1	PostgreSQL . . . . .	15
3.3.2	MySQL . . . . .	15
3.3.3	SQLite . . . . .	17
3.4	Column Oriented Databases . . . . .	18
3.4.1	Cassandra . . . . .	18
3.4.2	HBase . . . . .	19
3.5	Document Oriented Databases . . . . .	21
3.5.1	MongoDB . . . . .	21
3.5.2	CouchDB . . . . .	21
3.6	Key-Value Stores . . . . .	22
3.6.1	Redis . . . . .	22
3.6.2	Riak . . . . .	23
3.7	Graph Databases . . . . .	23
3.7.1	OrientDB . . . . .	24
3.7.2	Neo4j . . . . .	24
3.7.3	Titan . . . . .	24

# 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 between 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 categories 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:

```
SELECT last_name , first_name
FROM conference_attendants
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.

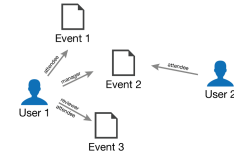
The simplified logic on the server side together with the absence of a feature that can be compared to "stored procedures" leave the developer no other option than implementing everything on the client side. While this can be seen as a way of keeping things simple, it is highly inefficient as it implies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

## 2.4 Selection Criterias of New Database

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.

## 2.4 Selection Criterias of New Database

---

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 <i>JavaScript at Everywhere</i> 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

#### 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 Oriented Databases

### 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.

**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 <i>set</i> 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, <i>CQL</i> , 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	☆☆★★★★

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 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. These 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 querying 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 use 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.

### 3.7.2 Neo4j

### 3.7.3 Titan

**Table 3.8:** HBase Column-Oriented Database

Criteria	Feature
<b>Availability</b>	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.
<b>Scalability</b>	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.
<b>Ease of Use Development</b>	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.
<b>Transactions Consistency</b>	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.
<b>Community Momentum</b>	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.
<b>Cost Exit Strategy</b>	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.
<b>Score</b>	☆☆★★★★

**Table 3.9:** MongoDB Document-Oriented Database

Criteria	Feature
Availability	AGPLv3
Scalability	<p>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, <i>mongos</i>. 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 <i>mongod</i> instance but it is recently improved to be for a collection which should be improved for document level.</p>
Ease of Use Development	<p>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.</p>
Transactions Consistency	<p>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.</p>
Community Momentum	<p>Without any doubt, MongoDB is the most popular NoSQL database in the wild and it is very well documented.</p>
Cost Exit Strategy	<p>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.</p>
Score	☆★★★★★

**Table 3.10:** CouchDB Document-Oriented Database

Criteria	Feature
<b>Availability</b>	CouchDB, licensed under Apache License 2.0, is one of the high priority projects in Apache Foundation and more mature compared to other NoSQL products because it has longer history and is developed by more inter-company developers.
<b>Scalability</b>	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.
<b>Ease of Use Development</b>	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.
<b>Transactions Consistency</b>	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.
<b>Community Momentum</b>	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.
<b>Cost Exit Strategy</b>	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.
<b>Score</b>	☆★★★★★

**Table 3.11:** Redis Key-Value Store

Criteria	Feature
<b>Availability</b>	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.
<b>Scalability</b>	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, <i>journaling</i> . 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.
<b>Ease of Use Development</b>	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.
<b>Transactions Consistency</b>	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.
<b>Community Momentum</b>	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 craigslist, flickr, stackoverflow.
<b>Cost Exit Strategy</b>	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.
<b>Score</b>	☆☆★★★★

**Table 3.12:** Riak Key-Value Store

Criteria	Feature
<b>Availability</b>	It has Apache License which is suitable for Indico and it is one of most popular key-value stores with Redis.
<b>Scalability</b>	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.
<b>Ease of Use Development</b>	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.
<b>Transactions Consistency</b>	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.
<b>Community Momentum</b>	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.
<b>Cost Exit Strategy</b>	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.
<b>Score</b>	☆☆★★★★

**Table 3.13:** Architecture of Graph Databases

Graph DB	Storage	Engine
<b>Twitter FlockDB</b>	non-native	non-native
<b>neo4j</b>	native	native
<b>OrientDB</b>	native	native
<b>Titan</b>	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.
Ease of Use Development	Python is supported by drivers provided from community (binary or HTTP).
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.
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.
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 <i>Gremlin</i> , 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 <i>SQLAlchemy</i> 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 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. If GitHub popularity of graph databases is ranked (all three are hosted on GitHub), then Titan seems to be the most popular one.
Cost Exit Strategy	Start phase to use Titan may be probably difficult since required storage engine. Exit is easier comparably since Titan is also 100% compliant with TinkerPop stack. If application is using TinkerPop stack, when graph databases is changed, there is nothing to be changed.
Score	☆☆★★★★