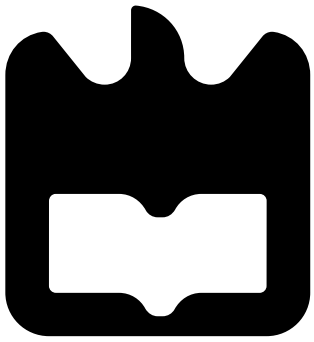
Universidade de Aveiro
Departamento de Electrónica, Telecomunicações e Informática,
2011

Nuno Filipe
Vieira dos Santos

Armazenamento de Contexto com NoSQL

Context Storage Using NoSQL



Nuno Filipe Vieira dos Santos

Armazenamento de Contexto com NoSQL

Context Storage Using NoSQL

dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Dr. Diogo Nuno Pereira Gomes, Assistente Convidado do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Mestre Óscar Narciso Mortágua Pereira, Assistente Convidado do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

o júri / the jury

presidente / president

Prof. Dr. Rui Luis Andrade Aguiar
Professor Associado da Universidade de Aveiro

vogais / examiners committee

Prof. Dra. Maribel Yasmina Campos Alves Santos
Professora Auxiliar do Dep. de Sistemas de Informação da
Escola de Engenharia da Universidade do Minho (Arguente Principal)

Dr. Diogo Nuno Pereira Gomes
Assistente Convidado da Universidade de Aveiro (Orientador)

Mestre Óscar Narciso Mortágua Pereira
Assistente Convidado da Universidade de Aveiro (Co-Orientador)

palavras-chave contexto, xmpp, bases de dados, nosql, couchdb

resumo O presente trabalho propõe o uso de uma nova geração de bases de dados, denominadas NoSQL, para o armazenamento e gestão de informação de contexto, assente numa arquitectura de gestão contexto baseada em XCoA (XMPP-based Context Architecture). Nesta arquitectura de gestão de contexto toda a informação de contexto é publicada num componente central, denominado Context Broker, e armazenada para posterior requisição, processamento e análise. Dado o elevado volume de informação de contexto publicada, as bases de dados tradicionais rapidamente apresentam limitações de desempenho, que por sua vez poem em causa a eficiência do funcionamento do protocolo XMPP; pelo contrário, as bases de dados NoSQL apresentam serias melhorias de desempenho na gestão de bases de dados de grande dimensão. Este trabalho estuda várias soluções NoSQL existentes, e apresenta as vantagens e desvantagens do uso de uma solução específica neste tipo de problema.

keywords context, xmpp, databases, nosql, couchdb

abstract This thesis proposes the usage of an upcoming new breed of databases, called NoSQL, for the storage and management of context information, in a context management architecture based on XCoA (XMPP-based Context Architecture). In this context management architecture all context information is published on a central component called Context Broker, and stored for further requisition, processing and analysis. Due to the high number of published context data, traditional databases quickly develop serious performance limitations, which in turn may jeopardize the XMPP protocol's efficiency; alternatively, NoSQL databases show serious performance improvements in the storage and management of large-scale data sets. This thesis presents a study of the several available NoSQL solutions, and presents the advantages and disadvantages of the usage of a specific solution in this type of problem.

Contents

List of Figures

List of Tables

List of Acronyms

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
BSON	Binary Javascript Object Notation
CB	Context Broker
CC	Context Consumer
CP	Context Provider
CPU	Central Processing Unit
DB	Database
DNS	Domain Name System
GPL	GNU General Public License
GPS	Global Positioning System
HTTP	Hypertext Transfer Protocol
IBM	International Business Machines
JID	Jabber ID
JSON	Javascript Object Notation
LZW	Lempel–Ziv–Welch
MVCC	Multiversion Concurrency Control
OTP	Open Telecom Platform
POSIX	Portable Operating System Interface for Unix
RDBMS	Relational Database Management System
REST	Representational State Transfer
RFID	Radio-Frequency Identification

SQL	Structured Query Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XCoA	XMPP-based Context Architecture
XEP	XMPP Extension Protocol
XMPP	eXtensible Messaging and Presence Protocol

Chapter 1

Introduction

With the ubiquity and pervasiveness of mobile computing, together with the increasing number of social networks, end-users have learned to live and share all kinds of information about themselves. For example, Facebook reports that it has currently 500 million active users, 200 million of which access its services on mobile systems; moreover, users that access Facebook through mobile applications are twice as active as non-mobile users, and it is used by 200 mobile operators in 60 countries [1]. More specific mobile platforms such as Foursquare, which unlike Facebook only collects location information, reports 6.5 million users worldwide, and also has a mobile presence (both with a web application and iPhone / Android applications) [2]. Context-aware architectures intend to explore this increasing number of context information sources and provide richer, targeted services to end-users, while also taking into account arising privacy issues.

While multiple context management platform architectures have been devised [3], this thesis focuses primarily on Context-Broker-based architectures, such as the ones proposed in the EU-funded projects Mobilife [4] and C-Cast [5]. More specifically, it focuses on the context management platform XCoA [6]. This platform uses XMPP as its main communication protocol, and publishes context information in a Context Broker. This context information is provided by Context-Agents, such as mobile terminals and social networks. Due to the nature of the XMPP protocol, the context information is provided in XML form.

This thesis proposes the usage of a NoSQL storage system in an XMPP broker-based context platform such as XCoA, together with a full-text searching engine. The usage of a NoSQL storage systems allows for better performance, availability and reliability. Moreover, integration with a full-text searching engine would allow extensive searching and data processing on the context information.

Chapter 2

State of the Art

2.1 Context

Before describing existing Context Management Platforms, we should define exactly what is "Context".

One of the first (if not the first) definition of Context-Aware applications was given by Schilit and Theimer [7] as being the ability of a mobile user's applications to discover and adapt to the environment it is situated in. Hull [8] also describes context-awareness (or situation-awareness) as the ability of computing devices to detect, adapt and respond to changes in the user's local environment, in a setting of wearable devices.

Perhaps a more generic definition would be the one given by Dey and Abowd [9]:

> any information that can be used to characterize the situation of an entity (a person, place, or object) that is considered relevant to the interaction between a user and an application, including the user and applications themselves.

There can be several types of context. For example, a user's present location can be considered Location Context, while a user's Facebook friends can be considered Social Context.

Some types of context have an associated timeframe. If we have an information about a user's Location Context that is one day old, we can assume it may already be outdated and incorrect; meanwhile, the user's Social Context may have not been changed, and can be assumed to be correct.

Why is context information important? Context-aware social networks are becoming increasingly more important. Facebook, Gowalla and Foursquare, which collect social preferences and location context information, have proved end-users are willing to share information. Moreover, context-aware information has the potential to enable targeted services to be delivered, according to the user's context, provided such information is properly collected and stored, while taking into account privacy issues arising from such scenarios.

The concept of Internet-of-Things, or IoT, also refers to the importance of context information, although with different names and different approaches. In the IoT concept, every-day objects have a virtual representation, and are addressable, using for example RFID tags. These objects collect context information, can communicate with each other

and provide reasoning on the collected context information. The IoT concept focuses more on every-day objects and real-world scenarios, unlike the Context Management Platforms discussed in this dissertation, which although also deal with sensors, focus more on web context sources such as social networks. However, the IoT presents to a much larger scale, possibly city-wide or country-wide scale. At this scale, scalability becomes a much greater concern, and scalability approaches discussed in this dissertation also apply to it.

In Context Management Platforms it becomes, then, essential the existence of a centralized and heterogenous platform that enables the collection of context information, the storage of that same context information, and provide mechanisms to allow the proper reasoning and interpretation of context information, providing users with richer services. This heterogeneity refers to the multitude of existing context sources, all with their particular data access models, all of which must coexist in the same repository. As this information will be accessed by actuators, which will use it to provide better services to users, the centralization of the information brings obvious benefits to these actuators, as all context information is accessed in a central location.

The following section presents a brief state-of-the-art in Context Management Platforms, as well as large-scale storage systems, required to handle and store large volumes of context information.

2.2 Context Management Platforms

This thesis is based on the XCoA, or XMPP-based Context Architecture, an XMPP broker-based context platform, first devised by D. Gomes et al. [6]. This platform represents an iterative evolution from previous broker-based architectures, trying to better understand their limitations and solve some of the problems found. The following two European Projects implemented such broker-based architectures, first MobiLife, with a "no-persistent-context" approach, then followed by C-Cast. Finally the XCoA platform tries to build upon these approaches and evolve them to a more efficient solution.

2.2.1 MobiLife

The MobiLife project aimed to "bring advances in mobile applications and services within the reach of users in their everyday life" [4], providing context-aware services. It devised a new Context Management Framework to handle the context information, and allow context acquisition and reasoning. This Context Management Platform or Framework is broker-based, and based on the producer-consumer principle. The architecture is shown in figure 2.1.

In MobiLife the Context Provider is the main component; it is responsible for producing context information, either from internal or external context sources. It exposes interfaces for communication with the Context Consumers, and to other Context Providers. The Providers also publish the context information in a Context Broker, which the Consumers can then inspect.

The Context Broker is the component responsible for publishing the location and interfaces of all Context Providers, which the Consumers can then use to interact with the desired Providers. This Broker is not responsible for mediating access between the

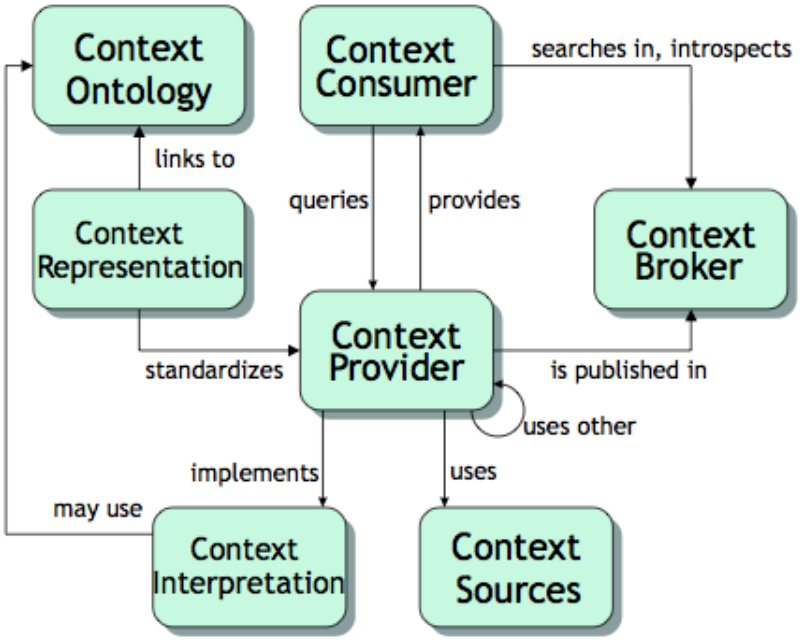


Figure 2.1: KeyFunctions of the Context Management Framework [4, p. 2]

Providers and the Consumers; instead, it publishes the Context Providers' location and interface information, which the Consumers then use to communicate directly with the Providers. MobiLife's architecture allows only for synchronous request / response context queries, not supporting asynchronous request models.

The Broker in this architecture assumes, then, more of a passive role, being mainly a registration and lookup service. This is unlike other architectures, where the Broker assumes an active, central role.

Context representation is achieved through the usage of the XML format, as it allows for easy human inspection, extension, validation and integration with existing development facilities and tools.

One important characteristic of this architecture, is that it does not persistently store all context information, with the rationale that "that would be almost the same as caching all the data that is routed over a server in the Internet" [10]. Some data may be stored or cached, but only in the Context Providers, as decided by them, but never stored in the Context Broker.

2.2.2 C-Cast

C-Cast was an European Project, whose main objective was to "evolve mobile multicasting to exploit the increasing integration of mobile devices with out everyday physical world and environment" [11]. It was based on two main competence areas: development of context awareness and multicasting technologies.

It devised a possibly-distributed broker-based architecture, based on the producer-consumer model. The architecture is shown in figure 2.2.

This architecture is mainly composed by the Context Broker (CB) , Context Providers

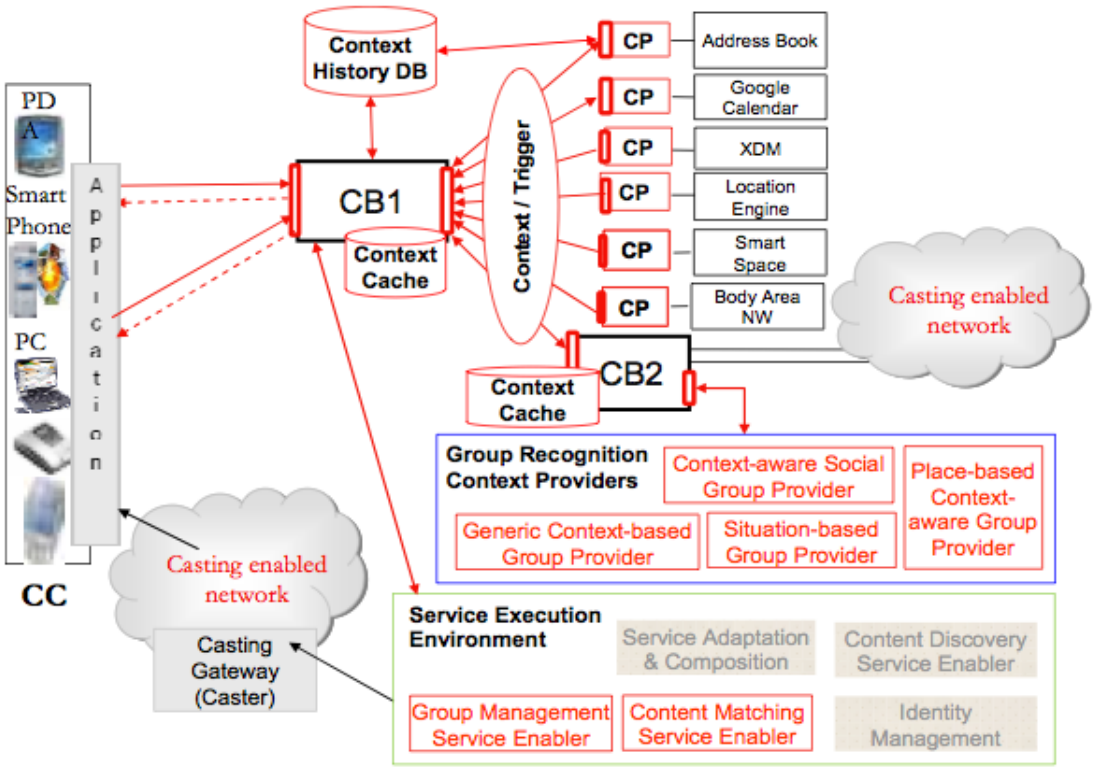


Figure 2.2: C-Cast Context Management Architecture & Functional Entities [5, p. 3]

(CP) , and Context Consumers (CC) [12, p. 13].

The central piece of this architecture is the Context Broker, which is responsible for managing the relationship between the Providers and the Consumers. It is the main handler of context information, and performs context information lookup and discoverability operations, keeps a persistent log of context information, which is basically a log of all context information exchanged between the Broker and the Providers, and a persistent context Cache, with the most up-to-date context information. This is unlike MobiLife's approach, which stored no context information (although some information might be cached in the Providers, at its discretion).

The Context Providers are the components responsible for collecting context information from the various context sources. They may also provide data aggregation and inference, based on their internal logic. They also provide methods for on-demand querying and subscription mechanisms, to allow for subscription-based notifications. Context reasoning and interpretation is also embedded in the Providers, deriving high-level context information abstractions, which requires inter-CP interaction.

Context Consumers are, as the name implies, the consumers of the context information, as provided and aggregated by the CPs. They are also the actuators, which interpret the context information and perform actions according to this information. The context information can be retrieved on an event-base (with subscriptions), or on an as-needed query base, with request / response messages. Some architecture components, such as Providers or Service Enablers, can also assume the role of Context Consumers.

This architecture developed a new document format for representing context: ContextML. ContextML is based on the XML format, and allows the representation of the type, scope, metadata and validity of the context information.

Persistent storage of context information was achieved with an Oracle relational database, with the translation of the context structures to the relational model, where every new context type would generate a new database table. This made the creation of new context types difficult, as they required a change (a new table) in the data model.

2.2.3 XCoA: XMPP-based Context Architecture

XCoA, or XMPP-based Context Architecture, builds upon the previous successes (and overcoming some deficiencies) of previous broker-based context architectures, such as the ones implemented in the MobiLife 2.2.1 and C-Cast 2.2.2 European Projects. Because of this they share some similarities, but XCoA tries to iteratively build a better solution than the previous attempts.

XMPP, or eXtensible Messaging and Presence Protocol, is an application-layer open-standard communication protocol, based on XML [13]. It was originally designed for near-real-time instant messaging, and presence information. It is used by Facebook for its chat feature [14], and by Google for its Google Talk instant messaging service [15]. However, due to its open-standard and extensible nature, several extensions were developed, which extended the use of XMPP for areas other than instant messaging and presence information. One of these extensions is the XMPP Publish-Subscribe Extension [16]. This extensions implements a simple Publish-Subscribe service on top of the XMPP communications protocol, using XML as its format. The XMPP Publish-Subscribe extensions allows entities (subscribers) to subscribe to specific types of information (through the subscription of specific XMPP PubSub Nodes); other entities called publishers are allowed to publish information on specific Nodes, which will then generate notifications to all subscribers of that particular Node.

XCoa is a broker-based context architecture which uses XMPP and XMPP Publish-Subscribe capabilities as its main communication protocol. It was first devised by D. Gomes, et al. in "XMPP based Context Management Architecture" [6]. The XMPP Extension XEP-0060 "Publish-Subscribe", or PubSub [16], and XEP-0248 "PubSub Collection Nodes" [17], are central building blocks of this architecture. The XMPP PubSub framework makes it possible to receive context information in near-real-time, without polling the server for changes. PubSub Collection Nodes allows PubSub nodes to be organized as a tree, where collection nodes are nodes which can only contain other collection nodes or leaf nodes, but no items can be published to it, and leaf nodes are nodes contained either inside collection nodes or as root nodes, and items are published to them. The XMPP PubSub component can be contained in an XMPP Server, or can be implemented in an external XMPP PubSub component, and then integrated in an XMPP server. The XMPP PubSub component is responsible for managing entities (or publishers), nodes, subscribers and items published.

In XCoA there are 4 main components:

- Context Agents
- Context Providers

- Context Broker
- Context Consumers

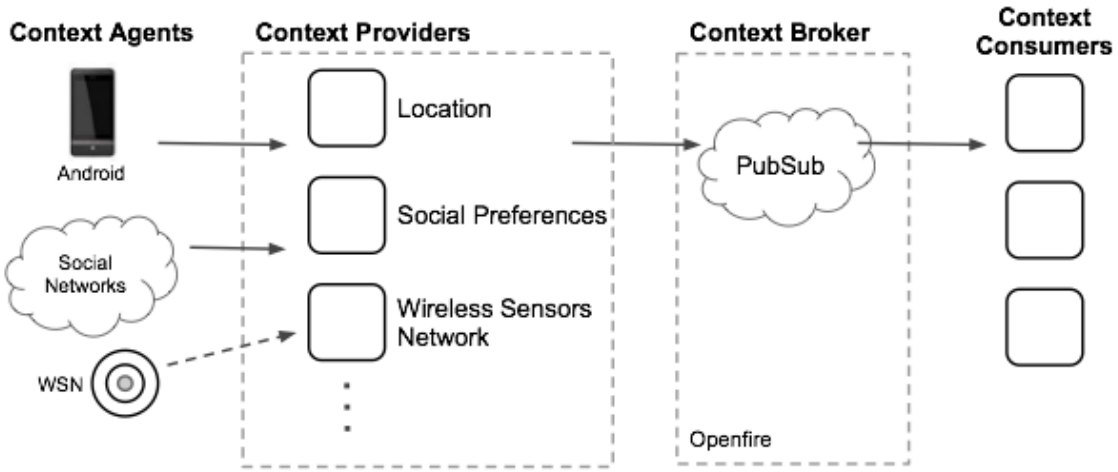


Figure 2.3: XCoA Architecture [6]

As we can see from figure 2.3, Context Agents collect information from context sources. For example, a mobile terminal can provide a multitude of context data, such as Location data, GPS coordinates, etc. Social Networks are also another source of context data; they can provide a list of contacts, a user's profile, etc. Finally, Wireless Sensor Networks are also good context sources, as they can provide detailed location information. There is no enforced communications protocol for transmitting this context data to Providers, and any Context Agent can provide the information as they see fit.

This context information is then sent and validated by Context Providers. Context Providers are specific entities, which usually only accept one type of context data, be it Location, Social, etc. They are in charge of receiving the context data from Context Agents, parsing and validating it, and then publishing it in the Context Broker. Each Provider is associated to one or more XMPP PubSub nodes, typically one node for each type of context, although more nodes can exist. Moreover, nodes can be arranged as a tree, so nodes can contain any number of nodes themselves, and a Provider can be responsible for one node and all their child nodes. The Provider, after receiving context data from Agents, publishes the context data, already properly parsed and validated, in the Context Broker, using the XMPP PubSub protocol. The context data is published inside an XMPP PubSub Item, in XML form.

Context Consumers can be subscribed to XMPP PubSub nodes. They can be subscribed to specific leaf nodes (on which context data will be published), or to collection nodes. When items are published to PubSub leaf nodes, all subscribers of that node will receive notifications with the contents of the data published. Subscribers of collection nodes receive notifications when items are published to any of its descendent leaf nodes. Polling operations, although limited, are also possible. Context Consumers can request context data by requesting data published on specific nodes. It is possible, for example, to ask for the last 10 items published on node X. This should be seen as an extra operation,

and not the main focus of the XCoA architecture, where publish-subscribe plays a much bigger part.

The Context Broker, the central part of the architecture, is the XMPP PubSub component (either a XMPP Server, or an external component, deployed in an XMPP Server). It handles all the subscription and node management, and is in charge of notifying subscribers on each item published (if any). It is also responsible for storing the context information in a persistent way. The implementation described in D. Gomes' paper [6] uses Openfire, which can store item publications in an relational database such as MySQL or PostgreSQL. In this case, item contents (in XML form) are published in a single table column, as an XML string.

Although this architecture is similar to the ones already described before, such as C-Cast's (2.2.2) and MobiLife's (2.2.1), there are differences.

In MobiLife context is exchanged directly between Providers and Consumers, with the Broker being in charge of only publishing the Providers' location and interfaces; in XCoA, however, the Broker assumes a more active approach, and all interactions are mediated by the Broker, both in synchronous and asynchronous modes of operation, through XMPP PubSub protocol functionalities. Mobilife also does not store all context information; instead, a small cache of the most recent context information is stored. This cache is however present in the Context Providers, and not in the Context Broker.

The C-Cast architecture is very similar to this XCoA architecture, where a full context history exists, along with a context cache; subscription-based notifications are also allowed, and context is represented in XML format. XCoA improves upon this architecture by using XMPP Publish-Subscribe to allow for easier subscription-based notifications, as well as synchronous queries.

Moreover, in XCoA the context information does not require prior data modeling in the database for storage, as was the case in C-Cast, where each context type required a new data table to be created, and a conversion from the internal context representation to a database table; instead, context information is stored as an XML string in a single table field, meaning that new context types do not require a change in the data model, or the creation of a new database table.

2.3 Large-Scale Data Management Systems

With a context platform's ability to persist context information comes the question of how and where to store it. Context platforms handle very large numbers of context data, and it becomes necessary to evaluate what solutions exist, what are the most efficient ones and what features are needed for this particular type of problem. Existing solutions fall into two broad categories: Relational Databases, traditional SQL-based storage systems based on the relational model, usually grouped by rows of data into tables; and the upcoming NoSQL storage systems, specifically designed to efficiently handle large amounts of data, at the expense of richer features offered by relational databases.

2.3.1 Relational Database Management Systems (RDBMS)

Relational databases are the most common solution for persistently storing structured data. They are based on the relational model, first introduced by E. F. Codd in the paper "A Relational Model of Data for Large Shared Data Banks" [18]. The relational model allows data to be modeled with its natural representation only (with properties and relationships between them), without imposing any additional structure for machine representation purposes, unlike other models such as the hierarchical and the network model. This provides a basis for a high-level data language representation, protecting it from disruptive changes on its underlying low-level representation.

These relational databases keep and manage data as a set of relations. In the relational model, a relation is a data structure composed of tuples, all of which share the same type. In the context of relational databases, these relations are usually a table: a set of columns and rows. Each column has an associated data type, and different rows of a column may have different values, but the same data type. These systems must also provide relational operators to manipulate data, such as operators to combine and filter data, according to rules defined by the users.

Most relational databases use SQL (Structured Query Language) as its high-level language querying format. Although it does not adhere 100% to Codd's relational data model, it became the most widely used database language, being deployed in virtually all relational database management systems.

Relational databases have a fixed structure (or schema), requiring prior database modeling. This modeling consists in identifying and creating the database relations (or tables), and their respective composition in terms of columns, data types and constraints. This modeling is then enforced throughout the database usage, and every data item must obey to its schema. Although fixed, this model can be altered after the table's creation, taking into account the table's restrictions.

One important advantage of these solutions is the possibility of organizing data to minimize redundancy, also known as normalization. Normalization allows tables to be decomposed in order to produce better, smaller, well-structured relations. This isolates redundant data in separate tables, allowing it to be altered and changed without changing other related data.

These features all contribute to make relational databases a very structured and disciplined data model, allowing for several features such as enforcing data integrity and consistency, all described in sections 2.3.1.1 through 2.3.1.4. A list of the most popular Relational Database Management Systems (RDBMS), both open-source and commercial ones, is listed bellow:

- Microsoft SQL Server
- Oracle
- MySQL
- PostgreSQL
- SQLite

- IBM DB2

Following is a list of common features offered by traditional relational databases, some of whom important for efficient context information handling in context platforms, and some of them less important, and may even be abandoned and traded for other more important features.

2.3.1.1 Replication

Replication (or as V. Gupta [19] refers to it, "scaling by duplication"), in a database context, means duplicating data through more than one database node, increasing read performance, as it distributes all read operations among all nodes that contain the needed data. It also has the additional benefit of providing higher availability, as data is distributed among several nodes, and one node failure will not result in database downtime, as other nodes will service the request. However, write operations become complex, as all operations must be synchronized between all nodes. This leads to multiple models of replication, such as Master-Slave and Multi-Master replication. [19] [20]

Replication may play an important role in scaling a context storage management system, if it means the distribution of load between different machines (horizontal scalability).

2.3.1.1.1 Master-Slave Replication Master-Slave database replication is the simplest form of replication. In this configuration there is only one master, and one or more slaves. The master is regarded as the owner of the objects, where all changes are first sent to. The Master is the only node allowed to change the database data. It logs the updates, and then propagates them to all the slaves, who confirm the success of the operation back to the Master, allowing it to re-send the failed updates to the slaves as needed. This always requires a connection between the Slave and the Master, making it less scalable but also less prone to failures [20, p.178].

This type of replication is useless to horizontal scalability and reliability, as it only guarantees the existence of backup copies of the data, but in case the Master fails all the databases become inaccessible, and it is impossible to continue normal operations until the Master comes back online.

2.3.1.1.2 Multi-Master Replication In Multi-Master replication, any node can be responsible for updating the data, and propagating the changes further to all the other nodes. It is also able to resolve conflicts arising from updating the same data in different nodes. Transactions can be propagated synchronously, or asynchronously. In the synchronous mode of operation, transactions are propagated immediately; in asynchronous mode, the changes are buffered temporarily, and only propagated periodically. This has the advantage of requiring less bandwidth and providing better performance. It also provides higher availability, so if any node in the set becomes unavailable, all the changes will be buffered for that node, while all other nodes receive the changes as if no node had failed; when the failing node gets back online, the buffered changes will be propagated to it. With synchronous replication, if one node becomes unavailable, no changes will be propagated to any node until all nodes are available.

Although Multi-Master replication makes the node set more error prone, as the failing of one database node does not compromise the availability of the data, it also has some drawbacks. Due to the buffer-and-send nature of asynchronous Multi-Master replication, the possibility of conflicts between buffered changes arises, requiring a complex conflict-detection strategy. [21]

Multi-Master replication in relational databases is a means to achieve better availability and prevent downtime, not as a scaling mechanism. Most relational databases provide very few or even none clustering capabilities, which is a central feature to achieve horizontal scalability, although some third-party solutions exist [22].

2.3.1.2 Sharding

Sharding means distributing the contents of a database between a number of nodes; this can be done by applying a hash function to a field of the data, to determine the right shard node. This means that a database table is not stored only on a single machine, but on a cluster of nodes. This has some advantages and disadvantages.

Partitioning the data through several nodes has the advantage of requiring less storage space requirements, while also distributing the load through all the nodes. How even this distribution of load is depends on how even the data is distributed through the cluster. Besides balancing data operations through the cluster, availability is also increased, as the failure of a single node does not compromise the whole database, but only the single data partition which was assigned to it.

However, sharding makes access operations much more complex; most sharded databases do not support join operations, even though these are one of the most important operations in relational databases. Join operations are used to produce data from two different sets of data, a left and a right one, connecting them by a pair of attributes. This means retrieving all data items from the right set associated with each item in the left set. In a sharded database, this would mean requests to all nodes that contain data from the two data sets, which would be impractical. Sharding can be done by using a consistent hash function, applied to primary keys of data items, to determine the correct associated shard node [23].

2.3.1.3 ACID Properties

ACID (Atomicity, Consistency, Isolation, Durability) are a set of properties designed to ensure the reliable process of database transactions. The term was coined in 1983 by Andreas Reuter and Theo Haerder, in "Principles of transaction-oriented database recovery" [24].

Transaction, in a database context, consists in a set of operations that are treated as a single unit of work, and are treated independently of other transactions. The transaction can either be committed to the database, or aborted.

Atomicity means that all operations associated with a transaction are either successfully executed, or the whole transaction is aborted. In this case, the database is left in the same state it was before the transaction was attempted;

Consistency means that all operations of a transaction must leave the database in a consistent state; if any of the operations in a transaction violate the database consistency,

the whole transaction must be aborted, and any changes rollbacked;

Isolation means any uncommitted changes are invisible to the database, making transactions unaware of other currently running transactions;

Durability guarantees that committed transactions are never lost, such that in case of a system failure, the data is available in a correct state. This is usually achieved in RDBMSs by using a write-ahead log file. This log file allows a database to redo transactions that were committed but never applied to the database.

In RDBMSs, ACID properties are usually achieved through a locking mechanism. This allows transactions to have a lock on the database, perform all operations, and release the lock after all is done. If another transaction wants to access data already locked by another transaction, it has to wait for the locking transaction to finish, and release the lock. While this is generally not a problem with most relational databases, it can present problems and become a bottleneck for distributed databases. [23]

These are all necessary features for a context storage management platform.

2.3.1.4 Strong Consistency

Strong consistency, a particular type of consistency usually used in traditional relational databases, consists in ensuring that all entities and processes always see the same version of a database. After a database update operation, every subsequent read operation will always return the same value. [23] [25]

Although this is a useful feature, it may not be strictly necessary in this particular scenario of a context management platform. Context management platforms handle real-time context data from multiple sources, sometimes with very little time intervals and very little change between them. As in a context management platform notifications are not dependent on the stored information and are generated in real-time only with the received information, strong consistency may be abandoned for other more useful features. The only operations that retrieve context information from the database are specific queries for specific types of context, and usually retrieve ranges of context information and not a single item; for this reason, the exclusion of a single context item in a range of 10 is acceptable.

It is sufficient, for example, to think of a case of retrieval of the last 20 locations a user has visited. If a user is publishing location context information at the exact same time the retrieval of the last 20 locations is issued, the exclusion of the item the user is currently publishing can be accepted, as we are more interested in its location history, not necessarily the present location. The notification, however, is completely unaffected, as the notification is issued on-the-fly without ever touching the database (although the context item is eventually persistently stored).

2.3.2 NoSQL

In this section we describe what are NoSQL databases, and what are their main characteristics and advantages, followed by the types of NoSQL databases available, and some examples of each.

"NoSQL" is a name given to databases that share one common characteristic: they are not relational; however, they have different characteristics and follow different approaches

to storing data.

Leavitt [26] refers this:

> What they [NoSQL databases] have in common is that they're not relational. Their primary advantage is that, unlike relational databases, they handle unstructured data such as word-processing files, e-mail, multimedia, and social media efficiently.

This lack of relations makes them automatically unsuited for a large set of applications. Businesses that have large, complex business rules, with multiple tables all related to each other, would find it hard to migrate to a solution with a simpler data-model, limited query support and lack of strong consistency.

Unlike relational databases, these NoSQL databases generally lack any enforced structure, being ideal candidates for storing random unstructured data.

Usually with a relational database, one has to define the structure of the data beforehand, creating the database schema (tables and columns), depending on the structure of the data. This gives them very little flexibility with regards to the data it can store. With a schema-free NoSQL database (some NoSQL databases are not 100% schema-free), any data can be stored. Besides the flexibility gains, this may also give them better storage efficiency, as no empty columns are wasted.

The need for the development of NoSQL databases arose primarily out of scalability concerns issues with relational databases, such as its lack of support for being distributed, which some argue is "key to write scalability" [19]. Relational databases provide a rich set of features (such as strong consistency, ACID properties, rich data query model) that, for some use-cases, can be traded for higher scalability and availability and performance.

While not all NoSQL databases are distributable, they all try to address scalability concerns, "by being distributed, by providing a simpler data / query model, by relaxing consistency requirements" [19].

The greatest advantage this type of databases have is, then, the increased performance and scaling capabilities they offer. This performance advantage is obtained thanks to two important factors: the non-relational and schema-free nature of this databases means data models are much simpler, increasing the database (DB) access speeds significantly; Leavitt [26] quotes Kyle Banker, engineer at 10gen, makers of MongoDB, a document-oriented NoSQL solution:

> There's a bit of a trade-off between speed and model complexity, [...] but it's frequently a tradeoff worth making.

The other factor is the fact that most of NoSQL databases are able to be distributed. V. Gupta [19] notes this "is key to write scalability", and offers a list of Distributed / Not Distributed NoSQL solutions, shown in table 2.1.

A database, whether relational or NoSQL, can be scalable in three essential ways: with the amount of read operations, the number of write operations, and the size of the database. This is usually achieved through Replication and/or Sharding (described in the following section 2.3.2.1) [23].

With the ability to distribute a database through several nodes comes several problems, such as maintaining data consistency. One way to guarantee data consistency in distributed databases is "Multiversion Concurrency Control", described in section 2.3.2.3.

Distributed	Not Distributed
Amazon Dynamo	Redis
Amazon S3	Tokyo Tyrant
Scalaris	MemcacheDb
Voldemort	Amazon SimpleDb
CouchDb	
Riak	
MongoDb	
BigTable	
Cassandra	
HyperTable	
HBase	

Table 2.1: List of distributable and not-distributable NoSQL databases

Even when NoSQL databases are a good fit for a given problem, it bears mentioning that they are relatively recent and, in some cases, largely untested in real-world scenarios. Leavitt [26] cites Google's BigTable (2006) [27] and Amazon's Dynamo (2007) [28] as the forerunners in the development of NoSQL solutions, "which have inspired many of today's NoSQL applications".

There are essentially three types of NoSQL databases that should be considered for a context management storage system, described in detail in the coming sections:

- Key-Value Store (section 2.3.2.6)
- Wide Column Store / Column Oriented (section 2.3.2.7)
- Document Oriented (section 2.3.2.8)

It should be noted that a fourth type of NoSQL databases exist: Graph Databases, such as Neo4j [29], based on graph structures, that use graph nodes, edges and properties to represent information. They were not considered because context information, as well as the XMPP Publish-Subscribe model, do not map to graph structures at all.

First, a list of features usually available in NoSQL solutions is presented bellow, as well as its need and importance in relation to a context management platform.

2.3.2.1 Replication and Sharding

Replication, described in section 2.3.1.1 for RDBMSs, is also used in NoSQL databases, perhaps even more so than in traditional relational databases. As scalability in NoSQL databases is addressed by distributing the database through several nodes, replication becomes very important. While in traditional databases replication becomes a complex task, due to the complexity of the data models involved, and the need to provide strong consistency, NoSQL solutions usually don't provide strong consistency, and as such replicating data between nodes no longer require the operation to be atomic, and changes can be propagated to all nodes without disrupting read operations (NoSQL databases generally only provide eventual consistency, which is further described in section

2.3.2.2). Replication in NoSQL solutions usually follow more of a peer-to-peer architecture, very similar to Master-Master replication, where all nodes are equal and operations can be executed on any replica, with the changes being replicated through all nodes afterwards.

Sharding also shares the same advantages and disadvantages with RDBMSs 2.3.1.2: they provide higher availability and balance the load between all nodes, but also make access operations much more complex.

Replication and sharding are both very important features in a large-scale context management system, essential to achieve horizontal scalability and increase reliability and availability. Ideally both would be present; sharding distributes data evenly across different nodes, allowing greater load-balancing, and replication guarantees the existence of backup data replicas through one or more nodes, providing reliability and availability.

In the absence of sharding, replication can also provide some form of horizontal scalability, provided a load-balancing application is used to evenly distribute operations through all existing replica nodes. On the other hand, this means that all nodes contain a full snapshot of the database, which may or may not be desired, and may lead to higher storage requirements.

2.3.2.2 Eventual Consistency

Eventual Consistency, as opposed to relational databases' "strong consistency" (section 2.3.1.4), means that there are no guarantees given that different processes accessing the database will see the same version of the data. It is a specific form of weak consistency, where the storage system simply guarantees that, if no new updates are made to an object, eventually all accesses return the same, last updated version of the object. As updates to an object are made, updates to all replicas are propagated asynchronously. While no guarantees are made as to when all replicas will be updated, usually the maximum size of the inconsistency window can be determined based on known factors, such as number of replicas and communication delays [25].

One of the most popular distributed systems that implements eventual consistency is the Domain Name System, or DNS. Updates to a name are propagated upwards through the hierarchy tree, and eventually all clients will see the update [25].

While eventual consistency is inherently a feature commonly found in NoSQL databases (due to its requirements), some modern traditional RDBMSs that provide primary-backup mechanisms implement a form of asynchronous backup techniques, where the backups arrive in a delayed manner, similar to the Eventual Consistency model [25].

As explained previously in Relational Databases' Strong Consistency (2.3.1.4), eventual consistency is an acceptable tradeoff to make in exchange for better performance, horizontal scalability and reliability guarantees. Some NoSQL solutions use Multiversion Concurrency Control for ensuring eventual consistency, which is described in the next section.

2.3.2.3 Multiversion Concurrency Control

Multiversion Concurrency Control, or MVCC , was first described in 1978 by D. Reed [30], and is a mechanism that provides concurrent access to databases, mostly distributed

ones.

It implements a system where no data in the database is overwritten; instead, every change to the database creates a new version of the data, marking the old version obsolete. This guarantees that database-read locking mechanisms don't interfere with database-writing locks, meaning that no read operations are every delayed by a write operation. Each read operation sees only a snapshot of the data, as it was some time ago, regardless of the current state of the data [31]. However, a read operation can read an outdated version of the document, if a new updated version is being written simultaneously. MVCC is also mention in the P. Bernstein, N. Goodman paper "Concurrency Control in Distributed Database Systems" [32].

Instead of locking access to the database and providing exclusive access to processes that execute write operations, MVCC allows the data to be read in parallel, even if the data is currently being updated. All data items maintain some form of timestamp or version id, to guarantee consistency and detect and resolve conflicts. When a document update occurs, it is first necessary to read the document and get the timestamp or version id of the document. This timestamp is then provided in the document update request. Upon successful update of the document, this timestamp or version id is then incremented; however, if the current timestamp of the document (as currently present in the database) is different than the timestamp given in the update request, then it is possible the document has already been changed between the document read and write operations, generating a conflict. This conflict must then be resolved, with different storage systems offering different solutions to these conflicts.

The following diagram 2.4 (from Figure 2.8 in Orend [23, p.14]) shows the creation of a conflict in a database with MVCC. Process A and B are both trying to write a new value to the data item x. Both read at first item x including its current revision number. Process B is the first to write its new version to the database. Process A tries to write a new version of x based on an older version than the current version in the database and thereby generates the conflict.

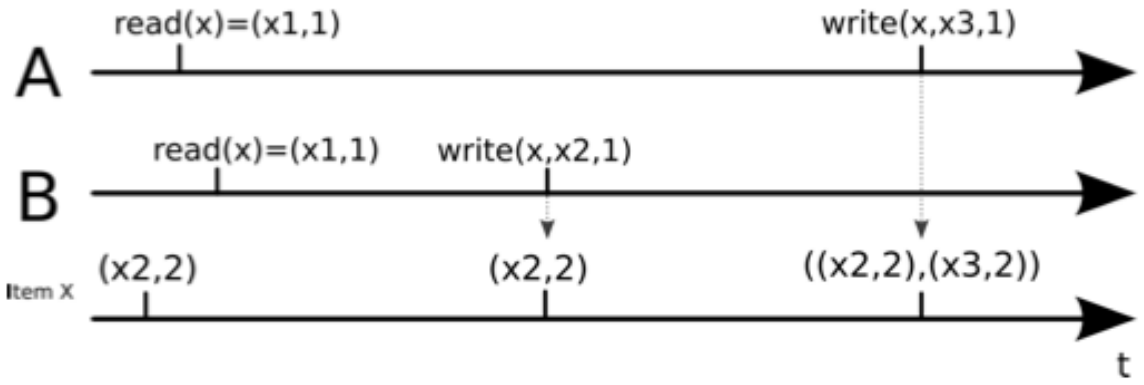


Figure 2.4: Creation of a conflict in a database with MVCC [23]

This conflict can be generated in two cases: two clients trying to write the same data on the same node, where they both retrieve the same document revision, and try to write the data, when only one update is valid; this conflict is easily detected by the database

during the write operation. In the other case, two clients are trying to write data to different nodes, where one node's data is not yet in a consistent state. In a distributed, asynchronous database, the conflict can not be detected during the write operation. The node must first be synchronized to a consistent state before the conflict can be handled. [23]

It is also present in some traditional RDBMSs, such as Oracle or PostgreSQL [31], although in a more transparent manner (CouchDB provides explicit document versions). In NoSQL solutions, MVCC enables concurrent database accesses, provides higher performance, and ensures weaker forms of data consistency.

2.3.2.4 ACID Properties

ACID properties (Atomicity, Consistency, Isolation, Durability) was already explained in section 2.3.1.3; However, due to its distributed nature and simplistic data model, NoSQL storage systems either use different approaches to achieve the same guarantees, or choose to only offer limited support for ACID. A central locking mechanism, used by most RDBMSs to guarantee ACID properties, is infeasible in distributed databases, as most NoSQL databases are. Instead, most NoSQL storage systems only offer limited support for ACID; for example, consistency may be guaranteed by using Multiversion Concurrency Control (section 2.3.2.3) [23]

Most of these features are required for a context management platform, although they may not necessarily be guaranteed by a single-server configuration. Data durability is essential, as it guarantees that no data loss occurs, but although a storage system in a single-node deployment scenario may not guarantee it, a cluster of multiple nodes can guarantee data durability on the cluster as a whole, but not on a single node.

2.3.2.5 Map/Reduce

Map/Reduce is a concept largely used by distributed storage systems, usually applied on large data sets. It was first develped and patented by Google [33], although it is based on a concept originally found in functional programming languages.

Map/Reduce consists basically in performing two operations on a dataset: applying a Map function, which partitions the data into a smaller dataset, based on filtering criteria; followed by an optional Reduce function, which can combine the resulting partitioned data into an even smaller dataset. These operations' workload is easily distributable, making its use ideal in distributable storage systems.

Some storage systems such as CouchDB provide internal Map/Reduce facilities, while external Map/Reduce frameworks such as Hadoop [34] provide stand-alone Map/Reduce features. A more detailed description of Map/Reduce is given in subsequent chapters, when describing specific NoSQL solutions.

2.3.2.6 Key-Value Store Storage Systems

Key-Value store databases are very similar to Hash Tables, with low complexity, that allow storing of data indexed by a key. The data stored can be of any type, and can be structured or unstructured; however, they don't allow the retrieving of data by using

anything other than its key, making them very high performant, but very little flexible.

Given these limitations, they are usually used alongside traditional relational databases, allowing fast access to a limited set of data, with full (and slower) access to all the data provided by more powerful solutions such as RDBMSs. A good example would be the usage of a data cache alongside a full RDBMS: storing the most frequently accessed data items in a separate NoSQL Key/Value storage system would provide faster access to most accessed data, while also preserving the full data model in the RDBMS, ready to be accessed as needed. MemcacheDB, a persistent version of memcached [1], follows this usage pattern.

A few examples of Key-Value Store databases:

- Amazon Dynamo
- MemcacheDB [35]
- Amazon SimpleDB [36]
- Redis [37]
- Riak [38]

Key-Value Store systems were deemed very high-performant, but very basic, not offering enough features required for a large-scale context management system, such as data retrieval by any field other than its key. Moreover, most are mainly oriented towards in-memory storage deployments. Although they can also persist data items asynchronously to disk as a fallback (in case of a crash), they should be regarded as mainly in-memory storage systems, as the data is read from disk to memory when the database restarts [39].

2.3.2.6.1 Amazon Dynamo Dynamo is a proprietary, highly-available, distributed key-value storage system developed and used internally at Amazon for its own services. It focuses on high reliability and scalability, reliability being "one of the most important requirements [at Amazon] because even the slightest outage has significant financial consequences and impacts customer trust" [28].

It was developed in detriment of traditional RDBMSs because most internal Amazon services only need to "store and retrieve data by primary key and do not require the complex querying and management functionality offered by an RDBMS. This excess functionality requires expensive hardware and highly skilled personnel for its operation, making it a very inefficient solution". Further concerns of scalability arose, as "available replication technologies are limited and typically choose consistency over availability" [28].

Amazon Dynamo has a very simple key/value interface, with objects being stored in binary blobs. Queries are executed giving only a key. Multiple-data-item operations are not allowed, and no relational schema is supported. It targets "applications that need to store objects that are relatively small (usually less than 1 MB)" [28]. It only provides Weak Consistency, resulting in very high Availability. It does not provide any isolation guarantees.

[1] http://memcached.org/

It is designed to operate in a network of nodes built from commodity hardware, assuming that any node can fail at any given time. Nodes can be added at any given time, without requiring further data partitioning, and with workload being distributed among all available nodes automatically.

It uses partitioning (or sharding) through consistent hashing (see section 2.3.2.1), and replication, to achieve high availability and "incremental stability".

2.3.2.6.2 Riak Riak is a Dynamo-inspired open-source Key-Value store storage system [38]. Its development was inspired by the Dynamo paper "Dynamo: Amazon's Highly Available Key-value Store" [28]. It is primarily written in Erlang and C, and used by Mozilla and Comcast [38].

Its local storage is organized in Buckets, and each bucket has Key/Value pairs. Each data entry is then identified by the combination of its Bucket and Key information. It offers an HTTP REST API, making client development easier and language-agnostic.

Riak has a distributed architecture, where all nodes are equal and there is no Master node. It supports sharding, where data is partitioned through multiple nodes, and each data partition can also be replicated to several nodes, increasing data availability and reliability.

Data is partitioned in a 160-bit keyspace, divided into equal-size partitions. The cluster is composed by physical nodes, and each physical node is further divided into several virtual nodes, called vnodes. Each physical node is responsible for 1/(total number of nodes) of the cluster, and contains an equal number of vnodes. Vnodes are then arranged in a ring, as detailed in the following figure 2.5.

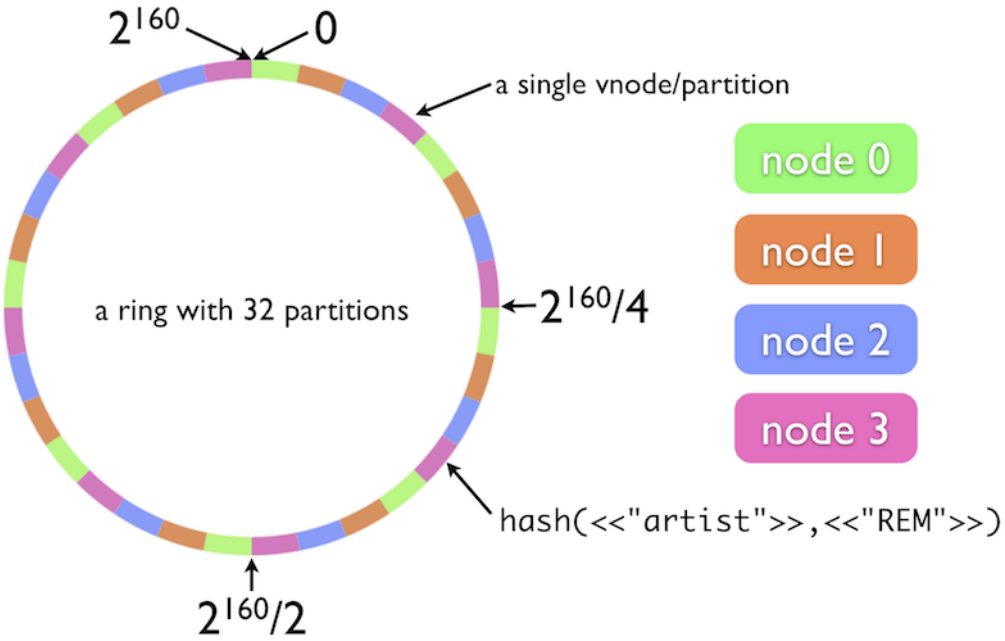


Figure 2.5: Riak Cluster Architecture

Riak offers querying functionalities through Map/Reduce, which are submitted as an

HTTP POST to a predefined address. These Map/Reduce functions can be written in either Erlang, or Javascript [40].

2.3.2.7 Wide Column Store / Column Oriented Storage Systems

In a very simplified view, the *Column-Store Database* concept consists on storing information grouped by columns, instead of the traditional RDBMSs grouping by lines. This allows faster data processing over similar data items, as they are all physically close in disk; it allows also several optimizations, such as compression (the LZW algorithm, for example, benefits the similarity of adjacent data, which translates in efficient compression). Its emphasis are, thus, on performance and disk efficiency.

This architecture has most advantages when dealing with small sets of columns, but large sets of rows; let's consider the following example relation:

ID	Name	Salary
1	Alice	30000
2	Bob	40000
3	Claire	45000

A traditional relational database would serialize this data by row; however, a column-store database would serialize this data by column, grouping similar data together.

```
1;2;3;
Alice;Bob;Claire
30000;40000;45000
```

When adding or accessing data, if all columns are supplied / required, there would be little advantage of using a column-store database, as this could mean only one disk-write. If, however, only sets of columns are supplied (for example, only the Name column), there would need to be one or more disk seeks to move to the right column on each row, on a traditional database; this would be a case where using a column-store database would be advantageous, as all data would be written adjacently, possibly with only one disk-write.

Some examples of Column-Store databases include:

- Cassandra
- Apache HBase [41]

Column-Store storage systems were also deemed unfit for an XMPP-based context management platform. Although they are also distributable, scalable and high performant, Document-Oriented systems were preferred, as context information is transmitted in document form (XML); moreover, most column-store solutions lack full-text searching capabilities, or any type of integration with a full-text searching engine such as Apache Lucene. A project exists to integrate Cassandra and Solr, named Solandra [42] (Solr is a a document indexer based on Apache Lucene). However, instead of enabling searching on an existing Cassandra dataset, it works the other way around, where it indexes documents by itself but stores its internal index in Cassandra. Cassandra is only used internally by Solr, so the main application of this configuration is then Solr and not Cassandra.

2.3.2.7.1 Cassandra Cassandra is a NoSQL decentralized eventual-consistent Column-Store database, initially developed by Facebook to power their Inbox Search facility [43] [44], promoted to Apache Incubator Project in March 2009 and to Top-Level Project in February 2010. Its main focus are Distribution, Replication and Fault Tolerance, with emphasis on Performance.

Cassandra has a distributed architecture and is decentralized, where every node has an equal role and there is no Master node. It supports sharding, or data partitioning, and data is partition using a consistent hashing function. Two partitioners exist: one which distributes data randomly across the cluster, or an order-preserving partitioner, which preserves the order of the data and allows range scans to be applied later on the data. Moreover, data is replicated to several nodes for fault-tolerance.

Quoting Avinash Lakshman on its Data Model [43]:

> Every row is identified by a unique key. The key is a string and there is no limit on its size.
>
> An instance of Cassandra has one table which is made up of one or more column families as defined by the user.
>
> The number of column families and the name of each of the above must be fixed at the time the cluster is started. There is no limitation the number of column families but it is expected that there would be a few of these.
>
> Each column family can contain one of two structures: supercolumns or columns. Both of these are dynamically created and there is no limit on the number of these that can be stored in a column family.
>
> Columns are constructs that have a name, a value and a user-defined timestamp associated with them. The number of columns that can be contained in a column family is very large. Columns could be of variable number per key. For instance key K1 could have 1024 columns/super columns while key K2 could have 64 columns/super columns.
>
> "Supercolumns" are a construct that have a name, and an infinite number of columns associated with them. The number of "Supercolumns" associated with any column family could be infinite and of a variable number per key. They exhibit the same characteristics as columns.

Figure 2.6 shows a single Column Family in Twitter's data model [45]:

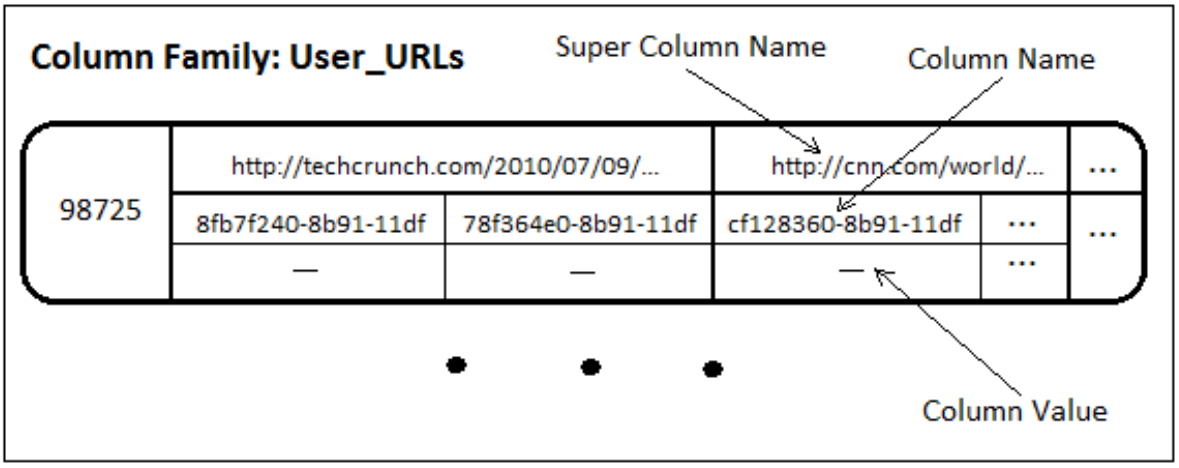


Figure 2.6: Twitter's data model: User_URLs [45]

This figure shows a single column family, UserURLs , with a single row identified by the key "98725". This column family has two super columns ("http://techcrunch.com/..." and "http://cnn.com/..."), and each super column has exactly two columns.

2.3.2.7.2 Apache HBase HBase is an open-source, distributed, versioned column-oriented database, modeled after Google's paper by Chang, et al. "Bigtable : A Distributed Storage System for Structured Data" [27].

HBase's datamodel is, then, very similar to that of BigTable. The data rows are stored in labeled tables, and each data row has a sortable key and multiple columns. Columns are identified by <family>:<label>, where family and label can be arbitary byte arrays. Column families fixed and set administratively, while labels can be added any time, even during a write operation, without previous configuration. Columns may not have values for all data rows. All column families are stored physically close on disk, so items in the same column family have roughly similar read / write characteristics and access times [41] [46].

The next example, taken from HBase's Architecture wiki page, shows a typical HBase data row, three column families: "contents", "anchor" and "mime". The "anchor" column family then has two columns: "anchor:cnnsi.com" and "anchor:my.look.ca".

Row Key	Time Stamp	Column *"contents:"*	Column *"anchor:"*		Column *"mime:"*
"com.cnn.www"	t9		"anchor:cnnsi.com"	"CNN"	
	t8		"anchor:my.look.ca"	"CNN.com"	
	t6	"<html>..."			"text/html"
	t5	"<html>..."			
	t3	"<html>..."			

Figure 2.7: HBase Conceptual View [46]

2.3.2.8 Document-Oriented Storage Systems

Document-Oriented Databases are an evolution on Key/Value Store databases; instead of storing any type of data, as most Key/Value Stores, Document-Oriented databases store only properly formatted documents, offering a rich feature-set on top of simple Key/Value Stores. Most Document-Oriented databases offer the possibility of searching documents according to its semantic value (searching for particular fields in documents), or the possibility of filtering documents also based on its semantic value (for example, display only documents that have start with tag X and value Y).

Most Document-Oriented databases support JSON (Javascript Object Notation) and XML documents.

These storage systems are usually schema-free, or schemaless, not enforcing any type of structure on the data it stores (provided all are valid documents). They also don't support relational schemas, although most solutions offer some mechanisms that allow retrieval of documents related to other documents, simulating a primitive type of relations (CouchDB offers this feature; see section 2.3.2.8.2). This is not, however, the main focus of these type of solutions.

Due to the schema-free nature of these solutions, and unlike traditional RDBMSs, any type of data can be added, without having to change the database schema to accomodate the new data. This simplicity means any user, even ones with low or even no previous database knowledge, can use and develop applications that use these type of databases. Conversely, this also means that every solution develops its own query mechanism and syntax, which is a drawback of RDBMSs which use a commonly known query standard, SQL.

Even though there are many XML document-oriented databases (such as Apache XIndice and MonetDB), most recent advances and implementations support only JSON documents. This is due to the high verbosity of XML documents, leading to higher storage requirements. While using JSON, the documents are smaller, while being able to convey the same semantic meaning as XML documents, which translates in a higher disk efficiency. However important, a discussion on JSON vs. XML-based documents is outside the scope of this dissertation.

The most popular document-oriented databases are:

- Apache CouchDB (2.3.2.8.1)
- MongoDB (2.3.2.8.2)

Due to the XML-format of the context information transmitted in the XCoA architecture, document-oriented databases were deemed ideal for storing context data. The document paradigm of these solutions mapped perfectly to the context information model used; even though they were in different document formats, being that context is in XML and these solutions mostly store JSON documents, it still seemed appropriate, provided a proper conversion was made. Because of this, a more in-depth description of the most popular document-oriented databases is shown bellow, as well as a comparison of both databases.

2.3.2.8.1 MongoDB MongoDB is a scalable, high-performance, open-source document-oriented database, implemented in C++ by 10gen and an open-source community, with emphasis on performance. The name Mongo comes from "huMONGOus". Besides storing documents, it also supports binary data, such as videos and images. It has support for indexes, replication, map-reduce and commercial support [47].

It stores BSON (Binary-JSON) documents, a JSON-style binary serialization of JSON documents. BSON supports embedded or nested objects, and reference objects. MongoDB supports in-place updates, meaning only the changed attributes must be sent to the database, unlike CouchDB, where updates provoke an insert of the same document, with a diffferent revision (see section 2.3.2.8.2).

MongoDB supports a schema (MongoDB is not schema-less), where each top-level object, or document, is represented by a "collection". Nested objects can either be embedded in the same collection, or can be in a new "collection", with a reference inside the first collection. [48] The MongoDB schema design documentation shows an example of two "collections": one representing a student, and one representing courses.

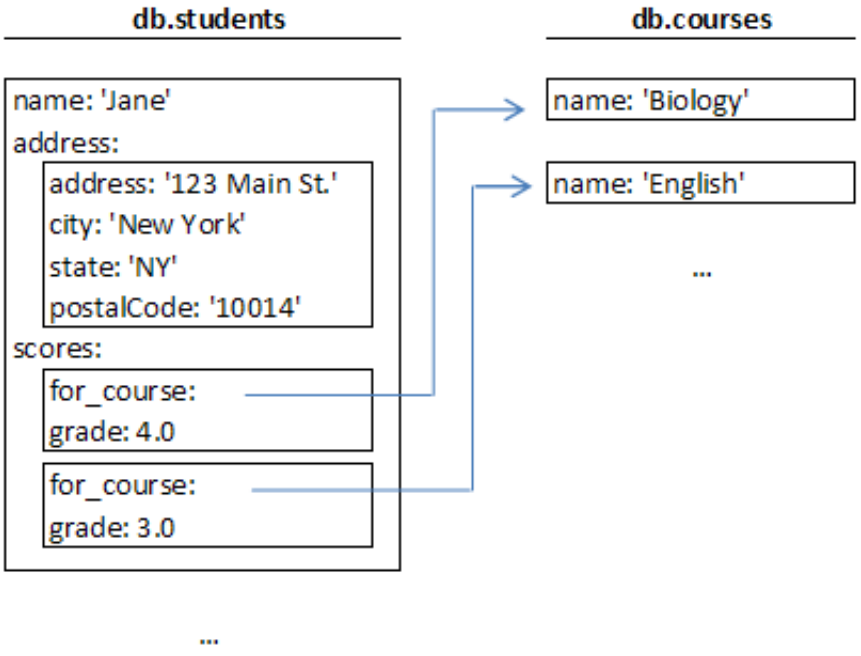


Figure 2.8: MongoDB Schema Example [48]

As mentioned in the documentation, in a traditional RDBMS, the scores information would lead to the creation of a new table, with a foreign-key relationship back to the student. However, only the course leads to a new MongoDB collection, with a reference in the student document to the proper course collection. This means that "with Mongo, you do less normalization than you would perform designing a relational schema because there are no server-side joins" [48].

Each document is associated with an id, used as primary key, and index, although the developer can create indexes on any fields he finds fit, which allows queries to be made against that particular field.

MongoDB has a rich query model (in document-oriented database standards), allowing queries using comparators, logical operators, conditional operators and aggregation; this means that developers coming from relational databases find that many SQL queries are possible and can translate to MongoDB queries [49].

Additionally, MongoDB also allows more complex aggregation with a variation of MapReduce. As Orend notes [23]:

> The MongoDB version of MapReduce is a bit different to the original one. Instead of having a Map, Reduce and a Combiner function, it uses a Map, Reduce and a Finalize function and requires that the Reduce function is associative and idempotent, because MongoDB calculates the reduction iteratively. So the Reduce function in MongoDB is more like the Combiner function in Google's MapReduce model.

MongoDB, unlike CouchDB, sees replication as "a way to gain reliability / failover rather than scalability" [50]. It does not use Multiversion Concurrency Control (MVCC, see section 2.3.2.3), and thus does not support optimistic replication, requiring only a single node (Primary, or Master) active for writes at any given time. This allows for Strong Consistency, unlike most NoSQL solutions. Replication is implemented with the usage of a log file on the master node, containing all write operations performed on the database. The slaves then ask the master for this log file, and reproduce the changes on the local database. All operations can be performed repeatedly without compromising database consistency, even if the slave database is recovering from a failure.

Moreover, it supports two replication setups: Master-Slave, where one Master writes a transaction log, which is then forwarded to all slaves; or Replica Pairs, which are "an elaboration on the existing master/slave replication, adding automatic failover and automatic recovery of member nodes" [51], where nodes can negotiate which one is the Master. When the current master fails, the slave becomes the new Master [23, p.40,41].

Regarding durability, early versions of MongoDB didn't support single-server durability, meaning that in single-server configurations there was a probability of data-loss occurring. Starting with version 1.7.5, MongoDB "supports durability (via write ahead logging) in the storage engine as an option" [52]. However, when not using said option, a machine or application crash will corrupt the database, requiring a database-repair command. This command "will check all data for corruption, remove any corruption found" [52]. Durability is best achieved, then, when used in multiple server configurations (Master-Slave or Replica Pairs).

However, the absence of durability translates in significant performance gains, compared to CouchDB [53][54].

The API access is made through language-specific drivers, being necessary to use a library to access MongoDB programatically.

2.3.2.8.2 CouchDB CouchDB is an open-source, cross-platform document-oriented database-management system written in Erlang. It was created by Damien Katz, former Lotus Notes developer, at the time employed by IBM, and initially released in 2005. Couch means "Cluster Of Unreliable Cluster Hardware", reflecting his goal of high availability and reliability, while at the same time being used on largely unreliable hardware

[55]. CouchDB supports most POSIX systems, including Linux and Mac OS X; although Windows isn't officially supported, "work is under way on an unofficial binary installer for the Windows platform" [55]. It supports JSON objects, where each document is identified by an id, or key.

Although it was initially implemented in C++, it was later moved to Erlang OTP (Open Telecom Platform) "because of its world-class reliability and concurrency" [56]. It was originally released under the GNU General Public License (GPL), becoming an Apache Incubator project in February 2008, with the license changed to version 2.0 of the Apache License. In November 2008 it graduated to a Top-Level Apache project, alongside the likes of Apache HTTP Server, Tomcat and Ant. In 2009 Damien Katz, J. Chris Anderson and Jan Lehnardt founded CouchOne, a company that "provide[s] support, development toolkits, training and hosting" [56].

CouchDB features an HTTP RESTful JSON API that allows applications to view, insert, modify and delete documents. This makes the database access language-agnostic, with the only requirement being an HTTP library (although a JSON library may be necessary for requesting and manipulating documents).

Its documentation describes itself as being "ad-hoc and schema-free with a flat address space" [57], meaning that, unlike MongoDB (which has limited but effective schema support), supports no schema, making no meaningful distinction between the documents it stores; the only requirement being that the documents be JSON documents. This means that no previous database modeling or normalization needs to take place, reducing significantly its data model complexity, and greatly simplifying "the development of document oriented applications" [57].

CouchDB documents are JSON documents identified by a unique DocumentID (_id). Besides the DocumentID, all documents also have a RevisionID (_rev), used to identify different versions of the same document.

It features replication (described in 2.3.2.8.2.1), supports all ACID properties 2.3.2.8.2.2 and, unlike MongoDB, doesn't support *in-place updates*; instead makes use of MVCC [50] (described in 2.3.2.8.2.2). All these features are described in the following sections.

2.3.2.8.2.1 CouchDB: Replication CouchDB features "distributed, featuring robust, incremental replication with bi-directional conflict detection and management" [57]; it features a peer-based distributed architecture, where all CouchDB hosts (online or offline) can have completely independent "replica copies" of the same database, and applications have full access (read/write) to the hosts, whether they are online or offline; upon them being back online, all changes are replicated bi-directionally. It also features built-in conflict detection and management. Its replication process is "incremental and fast, copying only documents and individual fields changed since the previous replication", which translates in efficient replication and backup facilities [57].

Unlike MongoDB, which tends to "think of replication as a way to gain reliability/-failover rather than scalability" , CouchDB sees replication more as a way to scale [50]. As such, and also unlike MongoDB, CouchDB uses a "crash-only" design, where it is assumed that the only way to terminate the application is by crashing it, even if intentionally ("does not go through a shut down process, it's simply terminated" [58]); the application is then prepared to handle crashes, and consistency is always guaranteed,

even in the absence of replication (this is further explained in 2.3.2.8.2.2).

2.3.2.8.2.2 CouchDB: ACID / MVCC CouchDB supports all the ACID properties (properties explained in sections 2.3.1.3 and 2.3.2.4): Atomicity, Consistency, Isolation, Durability. As its documentation states [58]:

> On-disk, CouchDB never overwrites committed data or associated structures, ensuring the database file is always in a consistent state. This is a "crash-only" design where the CouchDB server does not go through a shut down process, it's simply terminated.

Instead of supporting *in-place* document updates, like MongoDB [50], CouchDB uses a different approach: every document update means a new version of the same document being written to disk, with a new RevisionID (_rev). This RevisionID identifies the version of the document, and is used to resolve conflicts. The conflict management in CouchDB is non-destructive, meaning any number of revisions of the same document can exist in the database, with each database instance "deterministically deciding which document is the winner and which are conflicts. Only the winning document can appear in views, while losing conflicts are still accessible and remain in the database until deleted or purged during database compaction" [58]. Database Compaction consists in eliminating old revisions of documents, reclaiming disk space.

Using MVCC 2.3.2.3, every read operation returns a snapshot of the database. As such, database read operations are never locked out of the database and never have to wait for other read/write operations to finish.

Documents are indexed in B-Trees, by their DocumentID and SequenceID. Each update to the database generates a new SequenceID, which are later used for "incrementally finding changes in a database" [58]. These B-Tree indexes are updated simultaneously when documents are saved or deleted.

2.3.2.8.2.3 CouchDB: Views CouchDB views are one of the most important features of CouchDB. It allows the indexing and querying of documents by fields other than their key. They are the result of Map/Reduce functions 2.3.2.5, which are called incrementally on each access.

There are two types of views: permanent views, and ad-hoc or temporary views.

Permanent views are stored inside special documents called "design documents", and can be accessed with an HTTP GET to the URI "/_design/views/<viewname>". Temporary views are not stored in the database, and are created on-demand, with an HTTP POST, with the map function in the HTTP body. As they are not stored, they are extremely inefficient, and should only be used during development.

Map/Reduce functions are implemented in JavaScript. A Map function filters unwanted documents and maps specific document fields to document keys, which can then be used to access the document. Reduce functions are used to combine Map function results into a single key. Following is an example, taken from CouchDB's wiki [59].

```
Doc1:
{
    'id': 'doc1',
    'title': 'title1',
    'content': 'content1',
    'category': 'category1'
}

Doc2:
{
    'id': 'doc2',
    'title': 'title2',
    'content': 'content2',
    'category': 'category1'
}

Doc3:
{
    'id': 'doc3',
    'title': 'title3',
    'content': 'content3',
    'category': 'category2'
}
```

A Map function to map documents to their "category" field, would be:

```
function(doc) {
   emit(doc.category, doc);
}
```

This Map function would emit as keys the field "category", and as values the full document. The result would be (with <doc1>, <doc2> and <doc3> being the full documents):

```
{
    "category1": [
    <doc1>,
     <doc2>
    ],
    "category2": [
    <doc3>,
    ]
}
```

If we want to go even further and combine these results, to display the document count instead of the whole document, we could use a Reduce function as follows:

```
function(key, values, rereduce) {
```

```
    return sum(values);
}
```

This would return the following JSON:

```
{
    "category1": 2,
    "category2": 1
}
```

2.3.2.8.3 MongoDB / CouchDB Comparison CouchDB and MongoDB are two apparently very similar storage systems, but conceptually very different and focused on different problems. They are both based on JSON documents, although MongoDB stores documents internally as BSON, or Binary JSON, to achieve better performance; they are both also schemaless, but the similarities end there.

MongoDB focus itself mostly on performance, and most architectural decisions have this in mind. CouchDB on the other hand, also focuses on performance, but does not sacrifice durability. MongoDB does not offer durability guarantees on single-node configurations; instead, replication is supposed to increase this durability. CouchDB, on the other hand, offers single-node durability guarantees, and sees replication more as a way to provide horizontal scalability. However, unlike MongoDB, CouchDB does not offer data-partitioning facilities, which is the basis of MongoDB's horizontal scalability.

Regarding data-modeling, both are schemaless in the sense that documents with different schemas can be mixed in the same database. MongoDB however requires some form of prior schema design, translating each data object to a MongoDB "class", which translates to a single document schema; CouchDB requires no such modeling. Moreover, MongoDB allows document references, where a JSON document field element can "embed" or reference another document, where CouchDB provides no such functionality. The only way to achieve anything close to this in CouchDB is to have a document field in one document being another document's document ID, and request the required documents accordingly, but there are no internal CouchDB functionalities to optimize this.

Thanks to MongoDB's richer data model, simple queries are also allowed, and it is possible to fetch documents according to a pattern matching any field. CouchDB has no such "richer" query facilities ("richer" in relation to regular NoSQL storage systems, not related to relational databases' rich SQL query facilities), and uses Map/Reduce functions to allow queries that don't match the document's ID, which translates in larger disk resource requirements.

CouchDB also offers a REST HTTP API as its sole API access, whereas MongoDB offers language-specific native APIs, which also shows MongoDB's focus on performance versus CouchDB more simple approach. Both are open-source, and CouchDB is maintained by The Apache Software Foundation, and provides easy integration with other Apache projects such as the Apache Lucene full-text searching engine; MongoDB is maintained the company 10gen [60].

This makes these two apparently very similar storage systems oriented to very different use-case scenarios, with MongoDB more oriented towards performance and richer data query use cases, and CouchDB more oriented towards simplicity, lack of data modeling,

universal (obviously language-agnostic) HTTP API access and single-server durability guarantees, while also offering integration with other Apache projects.

Chapter 3

Context Management and Storage

This dissertation focus on the XCoA context management platform, which is based on XMPP. It proposes a new storage engine that must be prepared to accommodate large numbers of XMPP Items, retrieval and searching of context inside this Items. As persistent context storage, in XCoA, is handled by the Context Broker in an XMPP server, this new storage engine is implemented in an external XMPP Publish-Subscribe component, to be integrated with a regular XMPP server, which is the central piece of the XCoA platform.

In the XCoA platform, context information is published using the Publish-Subscribe paradigm, through the XMPP Publish-Subscribe protocol, where notifications are generated when XMPP Items with context information are published; this operations persists items in a storage system, but its performance is not very relevant, as the operation is concluded when the notifications are generated, and these notifications are not dependent on the insertion of the XMPP Item in the database. It is even possible to delay the persisting of these XMPP Items to the database without impacting the generation of notifications.

However, the XMPP Publish-Subscribe protocol also provides query functionalities, where XMPP Items can be retrieved on-demand, outside the usual publish-notification mode of operation. This operation can retrieve one or more Items, and it is possible to query for Items from a specific Node, or from all Nodes. For this functionalities, performance is of greater importance, as the operation is dependent of the retrieval of persisted Items from the storage system.

For this reasons, searching and accessing Items require much more performance guarantees than the insertion of Items.

Moreover, a context management platform that persistently stores context information to disk must be prepared to handle very large volumes of data, as every new publication of context information will have to be persisted to disk.

This chapter discusses the reasons why a NoSQL storage system was chosen in detriment of a relational database, as well as the reasons that led to the choice of CouchDB.

3.1 Why NoSQL?

In the context of the XCoA platform, and especially the XMPP PubSub protocol, scalability and availability are key requisites for a storage system. Although most traditional relational databases offer very high performance guarantees and vertical scalability (upgrading a machine's specs, such as memory and CPU, as opposed to horizontal scalability where new similar machines are added), they require high processing power and are geared towards datacenter-type mainframe clusters, whereas current hardware pricing trends tend to favor the use of large clusters made of commodity hardware instead. This requires a new approach to storage systems, with ones that adapt to highly distributable environments, and scale horizontally. NoSQL solutions are geared exactly for those scenarios. They emphasize simpler data models, and focus on horizontal scalability and high availability.

A very important requisite of the XCoA platform is the ability of storing all published context for further analysis. Depending on the type of context and the choices made by context publishers (Context Providers in the XCoA platform's case), there can be a large number of items published. Let's take, for example, the case of a GPS Context Provider. Taking a conservative approach, let's say that for every user, there are an average of 5 context publications each day. Considering the platform would only have 10000 users (low estimate), this would give an average of 50000 publications per day, or 18250000 per year, over 18 million Items. And this would be only for one type of context, GPS context. With the rise of the number of context types, number of publications would quickly rise to unsustainable levels.

NoSQL storage systems, as detailed in 2.3.2, are focused towards very high performance and reliability. They make certain trade-offs, regarding traditional relational databases, to achieve better scalability. One of these trade-offs, and arguably the most important one, is strong consistency.

Strong consistency means that all clients of a database always see the same version of the database, irregardless of current or pending operations. This is usually achieved using a locking mechanism, meaning that only one client can access the database at any given time, while others are locked out, waiting for their turn. As most NoSQL solutions are distributable, a central locking mechanism is impractical, and these solutions often trade this mechanism for some kind of weaker consistency. CouchDB, for example, uses Multi-Version Concurrency Control, or MVCC 2.3.2.8.2.2. MVCC guarantees that multiple write operations can occur simultaneously, and no write operations ever lock read operations out. Using a revision system, every document write generates a new version of the document, eliminating the need for a locking mechanism. This also means that a read operation can return an outdated version of a document, in case of a simultaneous write / read operation. As the XCoA platform makes heavy use of the Publish-Subscribe functionalities of XMPP PubSub, where an Item publication invokes an immediate notification, without the need for retrieving the Item from the database, and context retrieval operations to the database are very rare, this is a very acceptable trade-off. Even when context retrieval operations are needed, they are not high-priority, meaning that the exclusion of a few-minutes-old context publication is not important.

Another very important characteristic to achieve better scalability is **distributability**, the ability to distribute the database through several instances. This allows the

distribution and balancing of operations between all instances, increasing not only performance but also availability. When the database is distributed through several separate instances, there is no single point of failure, and the failing of one instance can be mitigated by other instances.

Traditional relational databases also have rigid data models, as required by the relational model. This model maps complex information to tables of rows and columns, and allow some advanced operations such as table JOINs (which join two database tables in one database query operation). In NoSQL databases, there is usually no enforced model, and when there is it is a very relaxed model, with very few fixed items. Key-value store and Document-oriented databases, such as CouchDB 2.3.2.8.2, Amazon's SimpleDB [36] and Redis [37] are usually **schemaless** or *schema-free*, meaning there is no enforced schema or data model, and any data can be stored and mixed (although Document-oriented databases such as CouchDB required that data must be structured JSON documents). Other solutions such as MongoDB 2.3.2.8.1 and Cassandra 2.3.2.7.1 require some minimal database modeling; Cassandra, for example, requires the modeling of Column Families, but the Super Columns or Columns contained inside them are dynamically created, and no modeling is needed. When storing context information as structured Documents or Columns, this schemaless design allows the dynamic creation of more context types without altering the database; if a relational database were used, new tables would have to be created to accomodate the new context types. In the end, however, context information was not stored as structured document, as described in the following sections.

Additionally, there needs to be a mechanism to allow out-of-band (in regards to the XMPP PubSub protocol) searching of stored context information. This is best achieved with a full-text searching engine, such as Apache Lucene [61]. Apache Lucene is a searching engine that allows high-performance indexing of data, ranked searching, field searching, sorting and supports several query types ("phrase queries, wildcard queries, proximity queries, range queries and more"). The storing engine should also allow seamless integration with this searching engine.

3.2 Why CouchDB?

First of all, the choice of a particular NoSQL solution should take into consideration the maturity of the solution. Storage systems that are actively developed and have a company behind them are good choices. The most popular NoSQL solutions are:

- Cassandra, initially developed by Facebook, now an Apache project, used by Facebook (no longer, as of late 2010), Digg, Reddit and Twitter
- MongoDB, developed by 10gen, used by foursquare, New York Times, SourceForge
- Apache CouchDB, developed by Apache, see [62] for deployments

As Lucene is developed by Apache, Apache NoSQL solutions are easily integrated with Apache Lucene. Apache CouchDB in particular has several projects to allow integration with Lucene, such as couchdb-lucene [63] and elasticsearch [64], which also uses Apache Lucene as an engine but is a stand-alone distributable searching application.

Cassandra is a column-oriented database, with a somewhat complex data model, and focuses on performance. MongoDB and CouchDB are both document-oriented databases, with MongoDB storing documents as BSON (Binary JSON), making integration with other applications difficult. CouchDB stores documents as JSON documents, in text format. CouchDB, unlike MongoDB, is fully schemaless, meaning that documents with different structures can be stored in the same database. MongoDB, on the other hand, requires minimal schema design, with every object mapping to a document "class", and supports references to other documents.

Seeing as context information is in XML form, a document-oriented database is the obvious choice. Even though most document-oriented databases support only JSON and not XML, the fact that most don't enforce a document structure means that the database is oblivious to the existence of different context types and their structures. Nevertheless, a conversion from XML to JSON is required to store context information. This will be discussed in the following section.

One other difference between MongoDB and CouchDB is their view of replication. CouchDB views replication as a way to scale, where MongoDB views it as a way to gain reliability rather than scalability. CouchDB uses a crash-only design, where it is assumed that the only way to shutdown the application is by crashing, even intentionally; the database can terminate at any given time and still remain consistent. MongoDB, however, requires a repair operation, where data-loss might occur, which is mitigated by the replicated instances.

CouchDB also offers a REST HTTP API, unlike MongoDB which requires language-specific drivers. This makes CouchDB application development easier and language-independent.

CouchDB is also particularly good when dealing with batches of data. As its API is an HTTP REST API, it is always limited by HTTP Request / Response latency times, so it tries to minimize this by providing an API for handling large numbers of documents in a single request. This fits well within the XCoA architecture, where context information is not exchanged directly between Context Agents (the source) and the Context Broker (the XMPP PubSub server); instead, they are aggregated by Context Providers, and then transmitted to the Context Broker. The Providers can, instead of transmitting context Items one by one, send them in batches, taking full advantage of CouchDB's batch-oriented API features.

To summarize, context information maps extremely well to a document-oriented paradigm, and CouchDB offers several advantages such as horizontal scalability, replication, REST API, handles batches of data very efficiently and provides very easy integration with Apache Lucene either through couchdb-lucene or through elasticsearch. Moreover it is an open-source project and actively developed by Apache. All these reasons make CouchDB a good fit.

3.3 Full-Text Searching

For full-text searching, Apache Lucene [61] provides a very good and widely deployed [65] searching engine, developed by The Apache Software Foundation in Java. As CouchDB is also developed by The Apache Software Foundation, a project to integrate

CouchDB with Lucene exists, "couchdb-lucene" [63], developed by Robert Newson, also a CouchDB contributor. This project tightly integrates with CouchDB, which provides direct hooks to integrate with an external full-text document indexing engine, through an external CouchDB HTTP API. Couchdb-lucene provides an HTTP-based searching JSON API, accessible directly in CouchDB. Despite tight integration, couchdb-lucene runs in a single, standalone Java Virtual Machine, and can be located in a different machine to CouchDB, although its deployment on the same machine reduces latency times and improves performance.

Another searching application exists, "Elastic Search" [64], which also provides integration with Apache CouchDB. This application also uses Apache Lucene as its searching engine, but is a standalone fully-distributed application itself. As such, it is relatively more complex than couchdb-lucene, and as couchdb-lucene is actively maintained by a core CouchDB contributor, it was chosen in detriment of Elastic Search. Also, distributability and the ability do deploy Elastic Search as a cluster was considered overkill for the needs of the XCoA platform, and the deployment of a single CouchDB / couchdb-lucene node was considered more than enough.

Although it is possible to deploy couchdb-lucene in a separate machine to CouchDB, it was chosen to always deploy it in the same machine, to reduce network latencies. Moreover, as explained in the following possible architectures (3.4.1, 3.4.2, 3.4.3, 3.4.4 and 3.4.5), in some cases the full database snapshot is not present in all CouchDB nodes, and instead is partitioned through several nodes, which would make couchdb-lucene's integration impossible, as it must hook itself with a single CouchDB node. For this reason, although not mandatory in all architectures, there is always a single CouchDB node with a full database snapshot responsible for the couchdb-lucene document indexing facilities, separate from the XMPP PubSub CouchDB cluster (although separate, its data is fetched through CouchDB's replication mechanisms). However, Lucene's document index is not distributed or replicated, and does not provide strong availability guarantees. In case of machine failure, searching facilities would be unavailable until the restoration of the machine; moreover, a network failure between couchdb-lucene's node and the XMPP PubSub CouchDB cluster would prevent Lucene's CouchDB instance from receiving replicated updates generated by the XMP PubSub protocol, and searching operations would retrieve possibly outdated information. It should also be noted that after restoration of the faulty apache-lucene machine, or the faulty network connection, the document index would be quickly rebuilt and searching facilities quickly re-established, with absolutely no data loss. As searching is an important feature of the XCoA platform, but not the highest priority one, and as its unavailability does not interfere with the XMPP PubSub protocol whatsoever, this deployment option seems like a good tradeoff.

For a higher-availability deployment option, multiple CouchDB / couchdb-lucene nodes can be deployed in parallel, serving as backups. The searching application (not provided by couchdb-lucene, as it merely exposes an HTTP searching JSON API, but implemented in this thesis as a proof-of-concept web interface) must be aware of the multiple couchdb-lucene deployments, or alternatively an HTTP load-balancing proxy can be deployed in front of the couchdb-lucene nodes, which would redirect search requests to an available node.

3.4 Architecture

As explained in 3.2, CouchDB was the NoSQL storage system chosen for storing XMPP PubSub Items, while all other XMPP PubSub data was kept in a relational PostgreSQL database.

CouchDB, like many other NoSQL solutions, provides horizontal scalability and higher availability and reliability through its distributability features. By distributing the Items database through a cluster of more than one node, we guarantee that the failing of one node does not impact the availability or reliability of the data, and scaling is achieved by adding more nodes to the cluster. Even when distributing the database through a cluster of nodes there are several possible ways to do it. This creates a few possible architectures, each with each advantages and disadvantages, which are described in this section.

Requests from the Context Providers are always sent to the XMPP PubSub server. The XMPP PubSub server and the PostgreSQL database (which stores the XMPP PubSub Nodes, Entities, Subscribers and entity-node Affiliations) can (and should, for performance reasons) be co-located in the same machine or node; however, CouchDB (which stores XMPP PubSub Items) can be deployed in several ways. For example, there can be one single CouchDB node, although this doesn't provide great availability guarantees; there can be multiple CouchDB nodes, all containing the full snapshot of the database (as CouchDB does not natively support data-partitioning, or sharding), where the XMPP PubSub server communicates with a single node and all the data is replicated throughout all remaining nodes; there can be multiple replicated nodes and a single proxy which routes REST HTTP request from XMPP PubSub server to any available CouchDB node, and CouchDB would then take care of maintaining consistency between all nodes; there can be multiple partitioned CouchDB nodes, using a third-party sharding solution to partition the data between all CouchDB nodes.

The full-text searching engine couchdb-lucene can also be deployed in any CouchDB node, so we can either integrated it with a single CouchDB node in case there is only one node, or we can integrate it with a replicated CouchDB node, responsible only for providing the searching facilities offered by Apache Lucene, separating the XMPP PubSub and the searching functionalities entirely.

The main focus of these architecture options are the Items storage in CouchDB, and not the XMPP PubSub component or the PostgreSQL database. The PostgreSQL database can be deployed in a Master-Slave replication scheme for increased availability and reliability, with a main Master node used by XMPP PubSub server and Slave read-only nodes as backups. This is possible due to the nature of the XCoA architecture and the XMPP PubSub protocol, as XMPP PubSub Nodes and Subscriptions are mostly created only once, and remain constant throughout the lifetime of the XCoA deployment; as such, a read-only Slave backup node with the XMPP PubSub Nodes, Affiliations, Entities and Subscriptions is sufficient for the correct functioning of the XMPP PubSub protocol, provided this data does not require change until the Master PostgreSQL node comes back online (for example, the creation of a new Context Consumer would trigger new XMPP Subscriptions, which would require a full read/write PostgreSQL Master node).

Moreover, a backup XMPP PubSub server node could be deployed, to be used in case the main PubSub server node fails, provided it used the same PostgreSQL database as the main node. It can either use the Master PostgreSQL database, or alternatively a

replicated Slave read-only database, only providing continuous XMPP Item publishing and notification facilities but not allowing creation of new Nodes or Subscriptions until the main PubSub node is back online.

After the creation of the XCoA deployment, the PostgreSQL data should then remain constant and therefore read-only backup facilities should be enough. To simplify the following architectural layouts, these options were left out, and the single XMPP PubSub server and PostgreSQL database nodes shown can also represent these read-only backup options. The architectures described bellow show, instead, different architectural deployment options for the Items' CouchDB database, which is the central piece of the XMPP PubSub protocol and XCoA architecture, containing the stored context information and being continuously updated throughout the entire lifetime of the XCoA deployment, and is therefore more susceptible to failures.

3.4.1 Architecture 1: Single PubSub + CouchDB + Lucene node

The simplest architecture is shown in the following diagram. We will call it "architecture 1", and it is shown in figure 3.1.

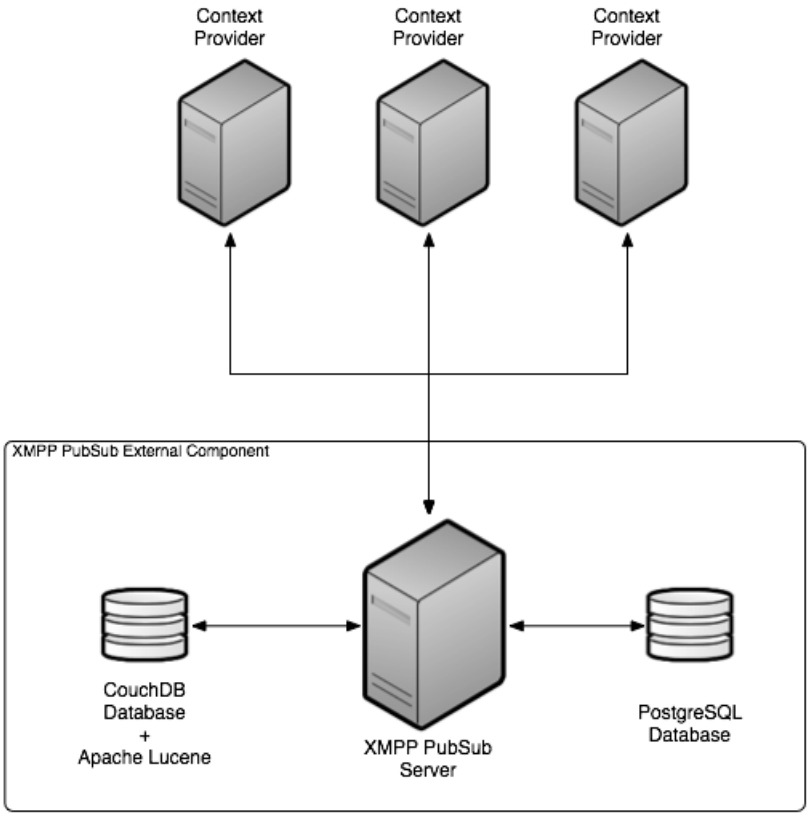


Figure 3.1: Architecture 1 – Single XMPP PubSub + CouchDB + Lucene node architecture

In this architecture there is only one CouchDB instance, in the same node as the XMPP PubSub server and the PostgreSQL database. There are very few advantages from the use of a NoSQL solution instead of a relational database in this case, besides

the performance advantages and the integration with Apache Lucene. There is no greater availability guaranteed, nor is horizontal scalability possible, but requires fewer hardware resources. There is a single-point-of-failure problem however, as all entities are in the same machine, including the couchdb-lucene document indexer.

3.4.2 Architecture 2: Separate CouchDB + Lucene node

A variation of this architecture is presented in figure 3.2, which we will call "architecture 2". This architecture separates the CouchDB and the couchdb-lucene instances from the XMPP PubSub server, although the same problems present in "architecture 1" are also present.

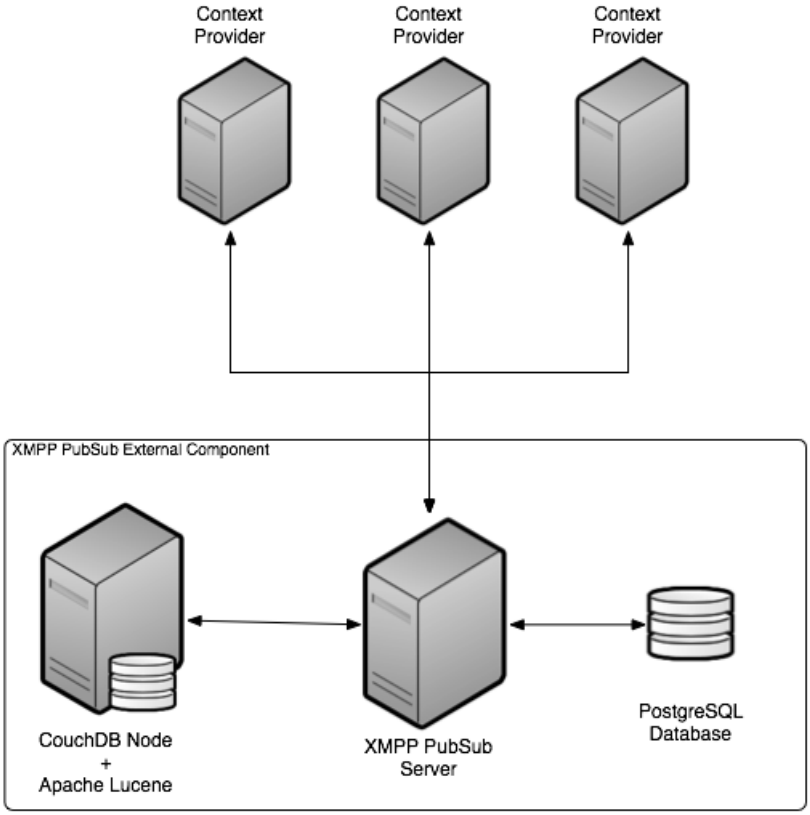


Figure 3.2: Architecture 2 – Single CouchDB + Lucene node architecture

Separating the CouchDB and couchdb-lucene instances from the XMPP PubSub node separates the document-indexing and searching functionalities into a new node, away from the XMPP PubSub / PostgreSQL. This way, heavy document-indexing processing is guaranteed to not interfere with the XMPP PubSub Server functionalities that don't require Item accessing (such as Node manipulation, subscription of nodes, etc). Although computation power is distributed by these two nodes, the failing of the single CouchDB node can still compromise the XMPP PubSub protocol's availability, and horizontal scalability is still impossible.

A better architecture would be to further separate and distribute CouchDB through two nodes, using one as the responsible for document-indexing and searching, and the

other for the XMPP PubSub protocol (3.4.3).

3.4.3 Architecture 3: Single replicated CouchDB + Lucene node

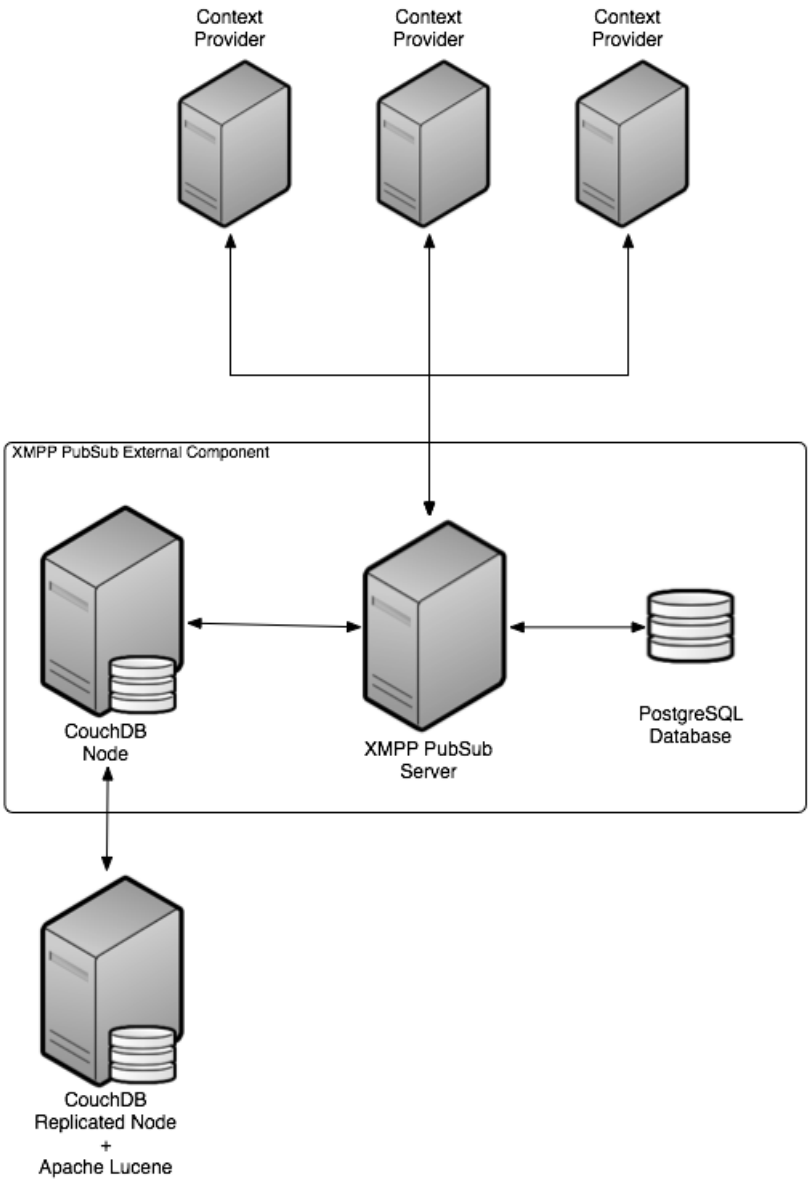


Figure 3.3: Architecture 3 – separate CouchDB replicated node with Lucene

This architecture, shown in figure 3.3, uses two replicated CouchDB nodes, one responsible for the XMPP PubSub protocol functionalities, and a separate node responsible for indexing CouchDB documents and fulfilling the full-text search requests. This is the first architecture that provides greater availability, where if one CouchDB node fails, the other can take its place and fulfill the requests made by the XMPP PubSub server, without degrading the XMPP PubSub service availability. Like "architecture 3", it also separates the document-indexing and searching mechanism from the regular XMPP Pub-

Sub functionalities, and a degradation in the searching node (e.g. caused by a slow search query) does not affect the XMPP PubSub protocol.

The replicated CouchDB node acts as the main node for Apache Lucene, but as a kind of "backup node" for the XMPP PubSub server, which it will use in case the main CouchDB fails. This also provides a low form of horizontal scalability, as the storage system is distributed and performance is not limited by the performance of a single node; however, increasing the number of replicas in this case won't help scalability, as there is one main CouchDB node that is always used, and there is no form of "load balancing" or "data partitioning" to evenly distribute the data through several nodes.

3.4.4 Architecture 4: CouchDB cluster with a Load-Balancer

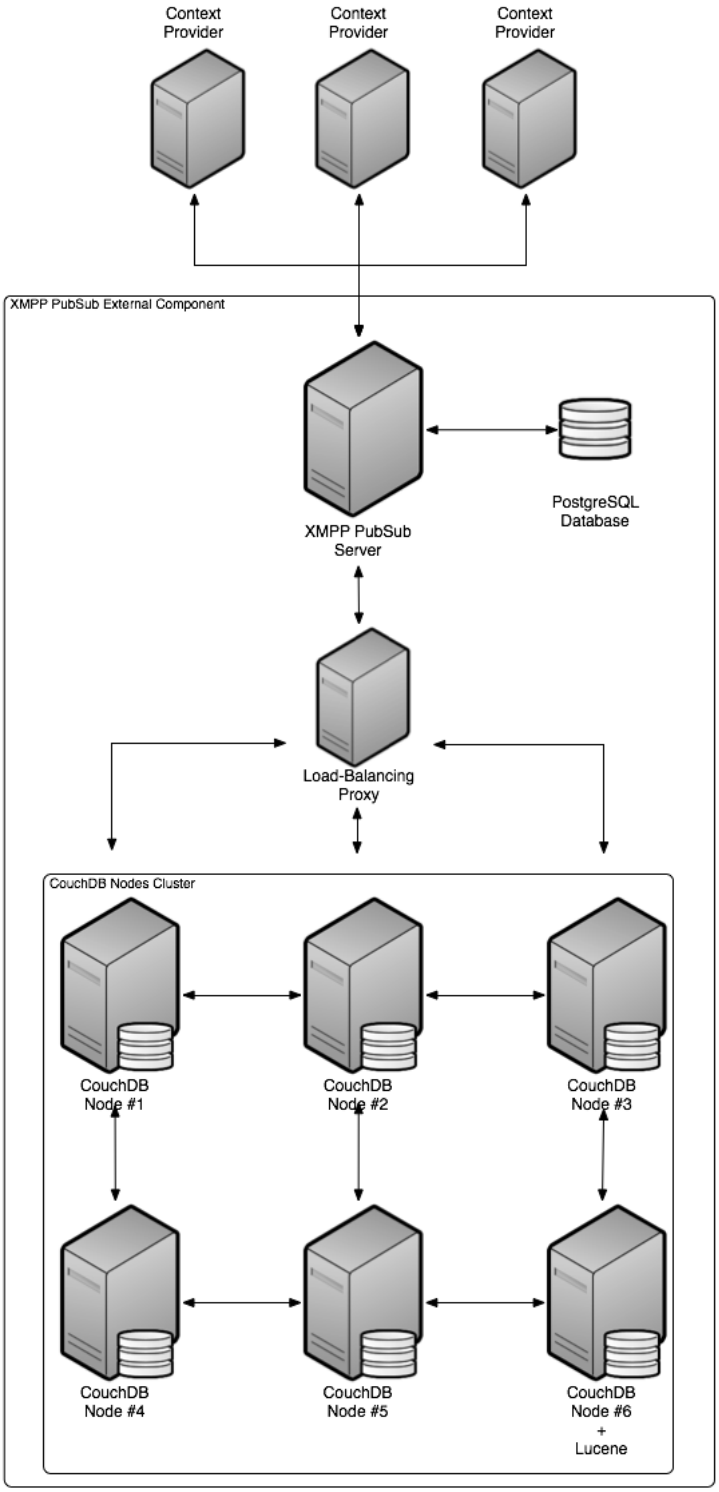


Figure 3.4: Architecture 4 – CouchDB replicated cluster

This architecture, shown in figure 3.4, provides a cluster of CouchDB nodes, where each node contains a full database snapshot, as CouchDB does not natively support

data-partitioning, and any node can be chosen to host the couchdb-lucene indexing and searching engine. This provides horizontal scalability, as the data is distributed through several nodes, and the application can access the node which has the lowest load. This requires a Load-Balancer proxy, to evenly distribute the load through the several nodes of the cluster. CouchDB provides no such mechanism, so there must be a stand-alone proxy module to provide such load balancing between the cluster nodes. CouchDB provides a REST HTTP API, so this load-balancing proxy can theoretically be any HTTP load-balancing proxy, such as Nginx. Also note that this cluster is not sharded, meaning that there is no data-partitioning between the cluster nodes, and every node contains a full snapshot of the database, and with every update to the data, the updates are replicated between all nodes through CouchDB's internal replication and conflict-resolution mechanisms.

Besides horizontal scalability, this architecture also provides much higher availability guarantees, as all nodes contain the full database snapshot, and one failing node does not compromise the cluster availability, and data requests will only be routed to a different node. The document indexing engine couchdb-lucene, however, cannot be distributed and must be located on a single node. Thus it does not offer the same reliability and availability guarantees that the CouchDB cluster does; however, document searching takes a lower priority in this architecture, and is not required for the XMPP PubSub protocol, so it is an acceptable compromise.

Although this architecture offers both horizontal scalability and high reliability / availability guarantees, there is no data partitioning between the nodes, and an increase in cluster nodes has significant storage space costs, which in turn may also degrade performance. For a database snapshot with size X, every node replica needs to accommodate the full snapshot of size X. This means that in a cluster of N nodes, the disk-space requirements are, at least, of size $X * N$. Although not critical, it is a disadvantage, compared to data-partitioning-aware storage systems. Moreover, availability increases greatly with the number of added nodes. In a cluster of N nodes, every single document is replicated exactly N times, which may be excessive in large node clusters.

3.4.5 Architecture 5: CouchDB sharded cluster

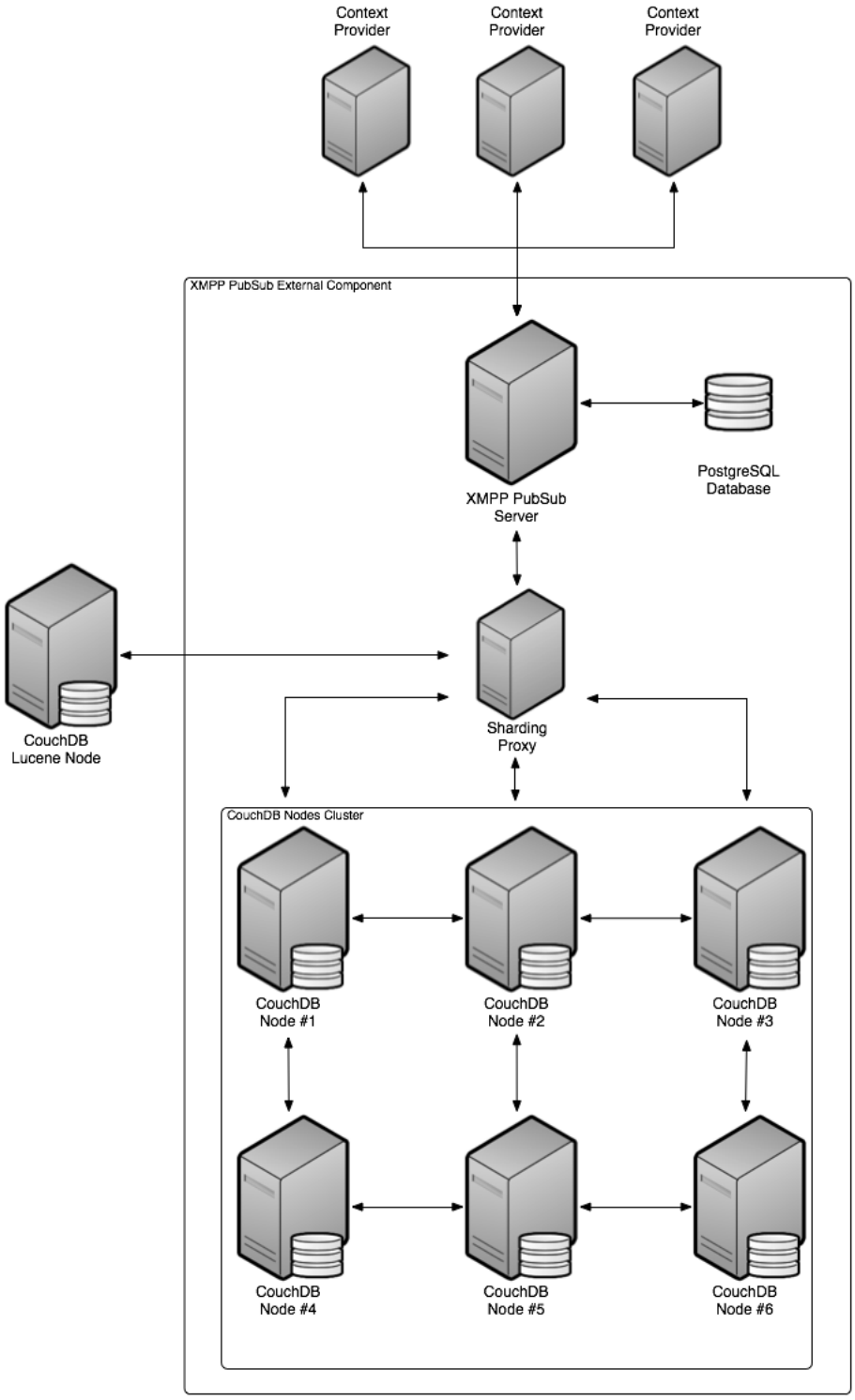


Figure 3.5: Architecture 5 – CouchDB sharded cluster

An architecture based on a sharded-CouchDB-cluster (shown in figure 3.5) appears to be very similar to the regular CouchDB cluster with a Load-Balancer (non-sharded),

but conceptually it is quite different. In this case, the data is partitioned and evenly distributed through all the nodes. This means that no cluster node holds a full snapshot anymore. As CouchDB does not provide data partitioning facilities, a sharding proxy must be used, to perform the data distribution between the cluster nodes. This distribution can be performed by applying a consistent hashing function, breaking up the data evenly across partitions. To provide greater reliability, data partitions are also distributed to more than one node, guaranteeing that a single node failure does not compromise any data. Although data is partitioned between nodes, all CouchDB Views, which are the output of map/reduce functions, are not partitioned and are contained in all nodes. The partitioning proxy takes care of distributing view requests to all nodes. Couchdb-lounge ([66]), developed by meebo.com, provides such functionalities [67].

As no cluster node contains a full database snapshot, it becomes necessary to maintain a separate CouchDB instance to contain a full snapshot, to integrate with couchdb-lucene and provide document indexing and searching functionalities. Integration with a partitioned node is not possible, as only that partition would be indexed, and it is necessary to index all partitions across all nodes. The partitioning proxy, besides partitioning the data through the cluster nodes, should then also be responsible for replicating all data to a separate CouchDB / couchdb-lucene node, which sits outside the XMPP PubSub protocol. Every change committed to the proxy must also be replicated to this separate node, and all searches will be performed on it.

Like architecture 4 (3.4.4), this architecture provides horizontal scalability and high reliability / availability guarantees, while requiring much less disk-space. With a database snapshot of size X, evenly distributed through N nodes, every node contains only a partition of size $\frac{X}{N}$, without accounting for the redundant data. If every partition is also redundantly stored across 3 nodes, the size of a single cluster node is $3 * \frac{X}{N}$, and the total size (without accounting the separate lucene node) is $N * (3 * \frac{X}{N})$, or $3 * X$, significantly lower than the size required in architecture 3 ($X * N$).

Chapter 4

Prototype

The implementation of the solution went through several iterations. Although several scenarios were possible, the existence of an XMPP Publish-Subscribe server or external component was essential, as it is the module responsible for receiving and persisting publications. This XMPP PubSub server or component would obviously have to support both the XMPP PubSub specification (XEP-0060 [16]) and the PubSub Collection Nodes specification (XEP-0248 [17]), both of which were already present in XCoA's chosen XMPP server, Openfire. To allow maximum modularity, the choice to implement the needed features on an XMPP PubSub external component was an obvious one. This way, the solution presented in this dissertation can be deployed on any XMPP Server, as long as external components are allowed.

The XMPP PubSub component chosen was Idavoll [68], an open-source publish-subscribe service component implemented in Python using the Twisted networking framework. Idavoll's original author, Ralph Meijer, is one of the co-authors of the XMPP Publish-Subscribe specification (XEP-0060 [16]). The Idavoll application, as well as modifications made to it which are common to all implementation iterations, are described in detail in section 4.3.

4.1 Prototype considerations and objectives

The implemented prototype should satisfy the following list of objectives:

- Allow the storage of a "history" of context publications
- Allow out-of-band searching of context information
- Be prepared to accommodate tens or hundreds of millions of context publications
- Maximize performance and availability

The existence of a "history" of published context is dependent on the way publications are made to the XMPP Publish-Subscribe component. If every publication is independent, or in other words, if for every new context information triggering event a new XMPP PubSub Item is generated, the "history" functionality would be implicitly present, as long as only the most recent information is retrieved. If, on the other

hand, context information changes generate existing context data updates, separate data storages of up-to-date context information and "history" would have to exist.

As the existing Context Providers in the XCoA platform generate new data publications, no separate storage was needed, and all context publications are made in the same storage system.

4.2 Libraries and Frameworks

Before discussing Idavoll and its internals, a brief enumeration of all libraries and frameworks used by it follows.

4.2.1 Twisted Networking Framework

At its core, Idavoll is implemented on top of the Twisted Networking Framework. Twisted is an open-source, event-driven networking engine developed in Python, supporting multiple protocols. It is based on an event-driven architecture, where operations are asynchronously deferred for later execution. Its central concept is called a "deferred" object, which is a value that has not yet been computed, but can be passed around like a regular object. Each one of these "deferred" objects are associated with a specific event, and are fired when certain event conditions are met (for example, when receiving a specific type of network packet). These "deferred" objects support callback chains. When a "deferred" objects computes its value, execution is passed down through the callback chain, and the output of the "deferred" object's function becomes an input for its first callback function; when this callback executes, execution is further passed down to the next callback in the chain, with the output of the first being an input in the second, and so on. This makes it possible to create an asynchronous execution chain without knowing the values of a "deferred" object.

Twisted also supports threads and concurrency, making it an ideal fit for networking servers and clients. [69]

4.2.2 Wokkel

Wokkel is a python library with a collection of Twisted Framework enhancements. It has many modules for experimental features that should eventually be included in Twisted, but have not made the transition yet. Its author is Ralph Meijer, the same author of Idavoll, and co-author of the XMPP XEP-0060 Publish-Subscribe specification [16].

Specifically in regards to Idavoll, it contains the necessary modules for implementing an XMPP Publish-Subscribe external component which adheres to the XEP-0060 specification. It supports the messaging layer of the protocol, handles the message parsing and creation, and provides interfaces to which XMPP PubSub components must adhere to implement the XEP-0060. While Idavoll deals with the actual handling of the operations, handling the storage and querying of the data, basically the "how" of the protocol, Wokkel handles the "what". When adding new features, it is necessary to implement the messaging layer in Wokkel, and the respective handling in Idavoll.

Wokkel supports also the creation of XMPP PubSub clients and proxies; in regards to Idavoll, only the XMPP PubSub external component features are necessary, as well as all associated message parsing / handling facilities.

4.3 Idavoll

Idavoll is a generic implementation of the XMPP Publish-Subscribe [16] protocol. It is written in Python, and based on the Twisted Networking Framework 4.2.1 and on the Wokkel library 4.2.2. Like the Wokkel library, its author is also Ralph Meijer, co-author of the XMPP XEP-0060 Publish-Subscribe specification.

Where Wokkel implements only the parsing and messaging facilities needed for the XMPP PubSub protocol, Idavoll implements the actual functioning of the protocol. It has several storing engines for persistently storing XMPP PubSub information, such as Nodes, Subscriptions and Item publications. Initially there were storing engines for Memory (which stored information only in memory, being lost in case of crashes or a restart), and for the PostgreSQL database.

Although already very feature complete, the application lacked several features, such as support for the XEP-0248 Collection Nodes [17], support for XEP-0030 Service Discovery of Nodes [70]; it also lacked several minor features present in XEP-0060: Publish Subscribe, described in detail further.

The Idavoll application can be divided in two main components: backend and storing-engines.

The backend contains generic code and business rules for the functioning of the XMPP Publish-Subscribe protocol. It contains what happens when a node is created, what happens when a subscription is added, etc. This generic code then invokes storing-engine specific functions, depending on the storing-engine currently in use. It is, then, completely independent of the storing-engine.

On the other hand, the storing-engines contain specific code for each storing solution used. This code is invoked by the backend, according to the storing-engine currently in use, and is in charge of managing the data needed by the protocol, either in memory or persistently.

Table 4.1 lists the data model of the XMPP PubSub protocol, as implemented in Idavoll. It mainly represents the data model used in the PostgreSQL storing engine, although the Memory data model is very similar. In the PostgreSQL storing engine, each data item corresponds to a PostgreSQL table. Figure 4.1 shows the relationships between the data items.

Nodes	Entities	Affiliations	Subscriptions	Items
id	id	id	id	id
name	jabber id	entity	entity	node
type		node	node	item id
collection		affiliation	state	publisher
persist items			subscription type	content
deliver payloads			subscription depth	date
send last published item				

Table 4.1: XMPP Publish-Subscribe Data Model, as implemented in Idavoll

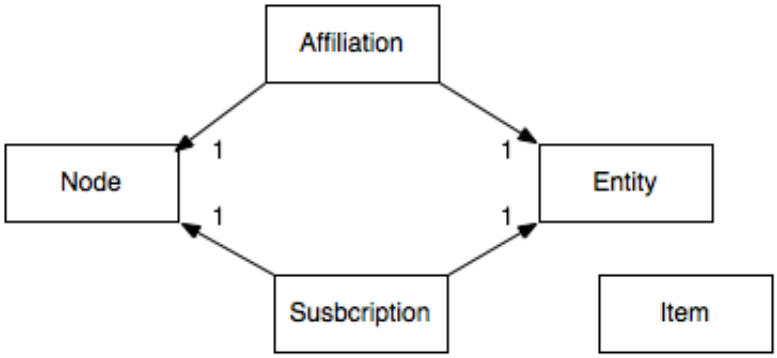


Figure 4.1: XMPP Publish-Susbcribe Data Model Relations, as implemented in Idavoll

Although this implementation went through several iterations, most features were implemented independently of the iteration. Mostly there were two new storing-engines implemented: a pure NoSQL CouchDB storing engine, which uses only CouchDB, and a PostgreSQL / CouchDB hybrid storing engine, described in detail in the iteration sections; several features were also implemented, both on the backend and all storing engines (Memory, PostgreSQL, CouchDB and PostgreSQL/CouchDB hybrid).

In the following sections we describe the details of the implemented solution. Sections 4.4 through 4.7 describe the iterations the solution went through and what was implemented in each iteration; section 4.8 provides a detailed summary of all implemented features after all iterations.

4.4 Iteration 1: Pure NoSQL Solution

The first prototype that was tested was a pure NoSQL solution, using CouchDB. CouchDB is an open-source, document-oriented database, and is one of the most popular NoSQL storage systems. It is also schema-less, so all data could be stored without previously modeling the database. The data would be stored as JSON documents, as required by CouchDB.

For a pure NoSQL solution, and taking into count the Idavoll's PubSub data model in 4.1, the following JSON documents would be created.

```
{
    "_id": "node:<node name>"
    "doc_type": "node",
    "node": <node name>,
    "node_type": <node type>,
    "collection": <collection>,
    "date": <date and time of creation>,
    "persist_items": <persist_items>,
    "deliver_payloads": <deliver_payloads>,
    "send_last_published_item": <send_last_published_item>
}
```

Figure 4.2: Pure NoSQL: JSON Node

```
{
    "_id": "entity:<Jabber ID>"
    "doc_type": "entity",
    "jid": <Jabber ID>,
}
```

Figure 4.3: Pure NoSQL: JSON Entity

```
{
    "_id": "affiliation:<node name>:<entity JID>"
    "doc_type": "affiliation",
    "entity": <Jabber ID>,
    "node": <node name>,
    "affiliation": <title of the affiliation>,
}
```

Figure 4.4: Pure NoSQL: JSON Affiliation

```
{
    "_id": "subscription:<node name>:<subscriber JID>:<subscriber
       resource>"
    "doc_type": "subscription",
    "entity": <subscriber Jabber ID>,
    "resource": <subscriber Jabber resource>,
    "node": <node name>,
    "node_type": <node type>,
    "subscription_type": <subscription type>,
    "subscription_depth": <subscription depth>
}
```

Figure 4.5: Pure NoSQL: JSON Subscription

```
{
    "_id": "item:<node name>:<item ID>"
    "doc_type": "item",
    "item_id": <item ID>,
    "node": <node name>,
    "publisher": <publisher Jabber ID>,
    "date": <date and time>,
    "data": <item contents>
}
```

Figure 4.6: Pure NoSQL: JSON Item

In addition, to satisfy the XEP-0248 "PubSub Collection Nodes" requisites, a single document was created, with the id (_id field) "nodecollection", to represent the entire collection node tree hierarchy. This document would be updated with every new collection node created. As the XEP-0060 specification nodes, in section 12.8 "Node ID and Item ID uniqueness", all Node IDs must be unique [16, 12.8]. This means that all nodes can be represented in a flat representation, with the tree hierarchy being represented in the "nodecollection" document; when the full hierarchy of a given node is needed, the "collection" field of the Node document is consulted, and the full tree path is obtained from the "nodecollection" document.

The node collection tree is represented in the following document:

```
{
    "_id": "nodecollection"
    "doc_type": "collection_node_tree",
    "collection": <node tree hierarchy>
}
```

Figure 4.7: Pure NoSQL: JSON Collection Node Tree

As an example of a node collection tree "collection" field, the existence of a root node "root", with 3 child nodes named "child1", "child2" and "child3", and a single collection node "child11" child of the node "child1", would generate the following "nodecollection" document.

```
{
    "_id": "nodecollection"
    "doc_type": "collection_node_tree",
    "collection":
    {
        "root":
        {
            "child1":
            {
                "child11"
            },
            "child2": {},
            "child3": {},
        }
    }
}
```

Figure 4.8: Pure NoSQL: Example of a node collection tree

CouchDB, as described in section 2.3.2.8.2, stores documents in JSON format. Context information, however, being transmitted through the XMPP protocol, is in XML format. It becomes, then, a challenge to decide in which format to store context data. Several options are possible: we can convert XML data to JSON, while taking care of storing all semantic data, both XML elements and XML attributes; we store the XML data as a CouchDB attachment; or we can, simply, store context data as an XML string in a JSON field. Each option has its advantages and disadvantages, and all are discussed in subsequent implementations. For this first implementation, context data was stored after being converted from XML to JSON.

Due to the differences between XML and JSON, a direct conversion is not possible. The most obvious difference is the existence of XML elements and XML attributes, which don't exist in JSON. Let's take, for example, the following XML for GPS context information:

```
<gps xmlns="http://c3s.av.it.pt/gps">
  <latitude>41.22256815655704</latitude>
  <accuracy>40</accuracy>
  <published>Wed, 08 Mar 2011 16:30:36 +0000</published>
  <altitude/>
  <bearing>12</bearing>
  <speed>0</speed>
  <longitude>-8.612594604492188</longitude>
</gps>
```

Figure 4.9: XML for GPS context information

The obvious way to store this kind of information in JSON form would be as follows:

```
{
  "gps":
  {
    "latitude": "41.22256815655704",
    "accuracy": "40",
    "published": "Wed, 08 Mar 2011 16:30:36 +0000",
    "altitude": null,
    "bearing": 12,
    "speed": 0,
    "longitude": "-8.612594604492188"
  }
}
```

Figure 4.10: Possible JSON for GPS context information

However, with this representation, we would lose attribute information, such as the "xmlns" attribute present in the XML representation. We could store attributes the same way we store elements, but then we would be unable to re-convert back from JSON to XML, as we wouldn't know which fields were attributes and which were elements. A conversion system was then devised, where for each XML element there would be a corresponding JSON pair of "attributes" and "value". XML attributes would be stored in the "attributes" field of the JSON pair, while XML elements (including nested elements) would be included in the "value" field. The previous XML context information 4.9 would, then, generate the following JSON:

```
{
  "gps":
  {
    "attributes":
    {
      "xmlns": "http://c3s.av.it.pt/gps"
    },
    "value":
    {
      "latitude":
      {
          "attributes": null,
          "value": "41.22256815655704"
      },
      "accuracy":
      {
          "attributes": null,
          "value": "40"
      },
      "published":
      {
          "attributes": null,
          "value": "Wed, 08 Mar 2011 16:30:36 +0000"
      },
      "altitude":
      {
        "attributes": null,
        "value": null
      },
      "bearing":
      {
        "attributes": null,
        "value": 12
      },
      "speed":
      {
        "attributes": null,
        "value": 0
      },
      "longitude":
      {
        "attributes": null,
        "value": "-8.612594604492188"
      }
    }
  }
}
```


Figure 4.11: JSON for GPS context information, without losing attributes

Although this context structure would preserve the semantic data, as CouchDB understood and properly handled JSON data, allowing rich manipulation features within CouchDB, such as Map/Reduce operations in CouchDB Views, integration with searching engines proved difficult, as detailed in the description of the subsequent iteration 4.5.

CouchDB, as most NoSQL storage systems, does not support relations, and as such, all documents are independent of each other. This means that documents contain no reference to each other. However, some operations require the retrieval of several documents.

For example, to create a new collection node, there are several new documents created (or accessed). First, the Node document is created, with the correct configuration. Next, the "nodecollection" document is updated with the new hierarchy. Finally, two new documents must be created (or at the very least, accessed): an Entity document, representing the entity which is creating the node (if the Entity document already exists, nothing is changed; however, if there is a conflict and the document already exists, no changes are made); and an Affiliation document, representing the owner affiliation between the created node and the entity that created it. This translates in, at most, 3 new documents created (Node, Entity, Affiliation) and 1 document update (nodecollection); at a minimum, this would translate into 2 new documents (Node, Affiliation), 1 document accessed (Entity), and 1 document update (nodecollection).

Although the previous node creation scenario is the worst existing case, several other operations need two document accesses, such as adding a new node subscription, where the existence of the node must be checked (1 document access), followed by the creation of the corresponding Subscription document. As seen by these use-cases, the XMPP Publish-Subscribe protocol requires a relational model, where several relations between their data members exist (between Nodes, Entities, Subscriptions, etc).

This is not, however, the only problem with a pure NoSQL solution.

The XMPP Publish-Subscribe protocol requires that several data items (documents in this case) are retrieved based on different fields. For example, for retrieving an entity's affiliation list, Affiliation documents must be retrieved based on their "entity" field. However, for retrieving a node's affiliation list, Affiliation documents must be retrieved based on their "node" field (see figure 4.4). This makes necessary the retrieval of documents based on properties other than their respective IDs, or keys. For each of these retrieval operations, a new CouchDB View must be created.

A CouchDB View allows us to map documents to keys other than their original ones. For example, for retrieving Affiliation documents by "node", we would create a CouchDB View, for example named "affiliations_by_node", which would map Affiliation documents to the following key:

```
[Affiliation.node]
```

When requesting documents through the "affiliations_by_node" View, giving a specific node name as key would retrieve a list of all Affiliation documents for that node.

Moreover, CouchDB allows keys to be in any format, including tuples or lists. As such, we could create keys based on multiple fields, and retrieve them based on those fields. The fields, however, need to be in specific order, where only the last fields of a key list are optional. Once we omit a parameter, all following parameters must also be

omitted. Let's take as an example a View with the following key:

```
[Affiliation.node, Affiliation.entity, Affiliation.affiliation]
```

In this View, we could retrieve all Affiliation documents based only on their "node" field, based on both their "node" and "entity" fields, or on all three fields. It is impossible to retrieve Affiliation documents based on both the "node" and "affiliation" fields, or only the "entity", or based on "entity" and "affiliation".

These key lists also have the advantage of allowing sorting. All returned documents are ordered by their keys, so supposing we would want to retrieve all Affiliation documents only by "node", sorted by the "entity" field, we could use a View with the following key:

```
[Affiliation.node, Affiliation.entity]
```

When retrieving documents, we would give a parameter range, as allowed by CouchDB Views. This parameter range would be between the keys "[node]" and "[node,]", which would give all documents based on the given "node" parameter, and sorted by all followed omitted keys, in this case only the "entity" field.

With all these document retrieval requirements the number of CouchDB Views needed is relatively large. The Views created to facilitate the XMPP Publish-Subscribe correct functioning were the following:

- affiliations_by_entity (Affiliations only, key: [Affiliation.entity])
- affiliations_by_node (Affiliations only, key: [Affiliation.node])
- affiliations_by_node_entity (Affiliations only, key: [Affiliation.node, Affiliation.entity])
- items_by_id (Items only, key: [Item.item_id])
- items_by_node (Items only, key: [Item.node])
- items_by_node_date (Items only, key: [Item.node, Item.date])
- nodes_by_collection (Nodes only, key: [Node.collection])
- nodes_by_name (Nodes only, key: [Node.name])
- subscriptions_by_entity (Subscriptions only, key: [Subscription.entity])
- subscriptions_by_node (Subscription only, key: [Subscription.node])
- subscriptions_by_node_state (Subscription only, key: [Subscription.node, Subscription.state])

As CouchDB Views are generated and stored on demand, their regeneration is relatively fast; however, a disk-space tradeoff is made, and when the database grows, every View grows proportionally. When this pure NoSQL solution was deployed, it was rapidly discovered that space-wise it would not scale. At one point the database was approximately 2GB in size, and Views accounted for over 10GB in disk. Although some optimizations could be made on an individual-View level, the number of required Views would hardly diminish.

One mistake was made on this implementation: CouchDB Views are the result of map/reduce functions, and as such are composed by a Key and a Document. This Document can be any field, the original Document itself, or null (empty) field. With these created Views, as the documents are always needed when retrieving results, the Document part of the view always contained the associated the document. It was later understood that no document is needed in the map function result: Views also store

the associated document ID, and when requesting View results, it is possible to request both the View result and the corresponding document(s) in only one request, making it redundant to store the original Document in the View itself. This would trim Views size considerably, as only the View keys would be stored, and no documents.

However, given the relational nature of the XMPP Publish-Subscribe data, and the HTTP API of CouchDB, each XMPP PubSub operation could translate in multiple HTTP Requests, seriously compromising the XMPP PubSub component's performance.

In conclusion, a pure NoSQL solution presents essentially two problems: XMPP Publish-Subscribe handles largely with relational data, making a pure NoSQL solution inviable or inefficient; and a large number of data items must be retrieved by a multitude of data fields, requiring multiple CouchDB Views, which translates in additional disk-space requirements and high latency response times, as multiple HTTP Requests must be made for each XMPP PubSub operation.

At first glance, all these problems would indicate that a NoSQL storage system would not be a good fit for the system being implemented. However, such is not completely true.

4.5 Iteration 2: NoSQL Items-only with JSON context data

As seen in the previous section 4.4, a first prototype was developed using document-oriented NoSQL solution CouchDB to store all necessary data. This included context-unrelated data such as Nodes, Entities and Subscriptions. With the relational-nature of the XMPP Publish-Subscribe data, which required a large number of CouchDB Views, it was quickly realized this was a model which would not scale.

Considering that context information is only present in XMPP PubSub Items, and the objective was to allow full-text searching capabilities on context information, it became obvious that, while storing all XMPP PubSub data in a NoSQL solution was impractical, it was very important to store XMPP PubSub Item publications in a NoSQL solution, while other PubSub data, which is mostly used to coordinate the XMPP Publish-Subscribe protocol but contains no context information, could be separated into another storage solution. It was also rather obvious that most XMPP PubSub data would also be in PubSub Items, while other data such as Nodes and Subscriptions would mostly be in a small number. Looking at a subsequently deployed instance of Idavoll, this was confirmed, as the following table shows:

Nodes	Entities	Affiliations	Subscriptions	Items
166	23	219	91	>1,000,000

Table 4.2: Example of a deployed Idavoll data structure

After analyzing Idavoll's Publish-Subscribe data model (see 4.1 and 4.1), it was determined that, even though the XMPP Publish-Subscribe data model fits in a relational data model, PubSub Items data model has absolutely no relations to other data. Despite the fact that "publisher" is indeed an entity, and the "node" to which the item is pub-

lished should exist in the Nodes table, these are fixed and cannot be changed after Item creation, and therefore are not relations to other tables' entries.

Taking all this into account, Items are the ideal candidate for deployment in a NoSQL storage system, while maintaining other PubSub data in a relational database. With this we should gain the performance benefits of storing the most-occurring data entries in a NoSQL solution, while still maintaining the relational-nature of other PubSub data in a relational database. Context information would be stored in the NoSQL solution, enabling integration with searching engines and allowing searches to be made out-of-band against all context data.

All data described in the table 4.1 was then stored in PostgreSQL, with the exception of Items. Items would then be stored in CouchDB, with the format described in 4.6. The only difference between the Item documents in the previous iteration and this one would be the document id; as only Items are stored in CouchDB, the "_id" field could be the ID of the Item, and the "item_id" field would disappear. As explained in detail in the previous iteration (4.4), there were several options for storing context information, which is received in XML format; first implementation used was a conversion from XML to JSON. This second iteration also used this XML to JSON conversion.

This SQL+NoSQL hybrid solution presented a much better architecture, with relational data in a relational database, and the context information in a NoSQL solution. As the bulk of the messages used in the XMPP PubSub protocol are XMPP PubSub Items, performance gains would be obtained only in with these Items, which makes more sense than sacrificing relational data for dubious performance gains.

This also reduced significantly the number of required CouchDB Views. As seen in the previous iteration, there were 11 CouchDB Views, of which only 3 were related to Items. As now only Items are stored, this reduces the number of Views from 11 to 3; moreover, one of the views allowed the retrieval of Items by their Item ID, but as now the ID of the document is the ID of the Item, this view can be eliminated, leaving only 2 required CouchDB Views:

- items_by_node_item (key: [Item.node, Item._id])
- items_by_node_date (key: [Item.node, Item.date])

The first View groups and sorts all Items by the combination of Node and ID; the second one sorts by Date instead of ID. The first View is required when we want to retrieve a specific Item from a Node, given the Item's ID; the second is required to retrieve one or more Items given a Node, sorting the resulting Items by date. These are the two essential Views for the correct functioning of the XMPP Publish-Subscribe protocol.

Another objective of this NoSQL storage would be to enable integration with a full-text searching engine. Although there are at least two searching platforms for achieving this, both use the same searching engine: Apache Lucene [61]. Integration with CouchDB could then be achieved by two ways: either using couchdb-lucene [63], an open-source project by Robert Newson, an Apache contributor, which uses Apache Lucene as a searching engine and tightly integrates with CouchDB; or using Elastic Search [64], also an open-source project which uses the Apache Lucene engine, but is a distributed and separate module, which listens on the CouchDB API for changes and indexes the documents as they are created.

While both approaches are different, indexing is performed by Lucene on both cases. Lucene searches occur on a specific field, so fields on which searching is to be permitted must be indexed. This presents a problem on this type of context storage format. As each XML field is associated with an "attribute" and a "value" field, it would be impossible to index specific fields, as for every field name corresponds an "attributes" and "value" pair. The only option would be to index "value" fields recursively, which would have to be made manually for every different type of context. Also, this would make it impossible to search for any string inside the context data. We would be unable, for example, to search for occurrences of the name of a field, for example "gps", because it is a field name and not field content.

Although using a SQL+NoSQL hybrid makes sense, the question of how to store XML context information remains an issue. With a direct XML to JSON conversion, document indexing would prove difficult and inefficient, the resulting JSON-formatted context representation would be very verbose, and its direct conversion from XML to JSON wouldn't provide much advantages; moreover, conversion from XML to JSON and back to XML would also prove inefficient, especially for operations on large numbers of Items. The next iteration tried to resolve this problem.

4.6 Iteration 3: NoSQL Items-only with XML document attachments

As seen in section 4.5, the SQL+NoSQL hybrid makes perfect sense, as we don't give up on relational data, and the most common data messages, which are also the ones which contain context data, are stored in CouchDB, with performance benefits, and which also should allow integration with a text searching engine. However, the format of the stored context data is still an unresolved issue.

The first implementations tried to store the context information as JSON objects. As the context information was received in XML format, this required a conversion from XML to JSON, and JSON to XML when XMPP PubSub Items were requested. Besides this making the documents much more verbose, it would present problems with text searching. Apache Lucene, the most popular full-text searching engine, requires searches to provide both a field and a search string, which will then be searched against the given field. As it was seen in the last section 4.5, each XML element or field is associated with a pair of "attributes" and "value" items, represented as a nested JSON object, making searching inside it impossible.

For full-text searching to be permitted, JSON documents would have to be in this format, and the "latitude" field should be indexed in Apache Lucene:

```
{
  "gps":
  {
    "latitude": "41.22256815655704"
  }
}
```

Then, we could perform searches giving "latitude" as the field, and for example "41"

as the search string which will be searched inside the "latitude" field. However, the format of JSON documents after being converted from XML is as follows:

```
{
  "gps":
  {
    "latitude":
    {
      "attributes": null,
      "value": "41.22256815655704"
    }
  }
}
```

As the content of the "latitude" field is itself a nested JSON object, no searching can be made against that field. Moreover, as context information can have any structure, which is also allowed by CouchDB, Apache Lucene requires fields to be indexed; as the exact structure of the context data is unknown, it becomes impossible to tell what fields should be indexed.

Another option allowed by CouchDB would be, instead of converting context data from XML to JSON, storing it as an XML attachment to the document. In this case, instead of the structure presented in figure 4.6, the "data" field, which is the actual context information present in the XMPP PubSub Item, would not be present (at least as a JSON field); it would, instead, be the XML attachment to the document. This attachment would be identified by an ID, for example, "data". For accessing these attachments, a separate CouchDB API request would have to be made, as the document retrieved does not retrieve its associated attachments. We can also, for example, retrieve the attachment without retrieving the document, making only one API request, as long as we know the exact ID of the document we want.

With this option, the structure of the Item document would be as follows:

```
{
    "_id": "<item ID>"
    "doc_type": "item",
    "node": <node name>,
    "publisher": <publisher Jabber ID>,
    "date": <date and time>,
    "_attachments":
    {
      data:
      {
        content_type: "text/xml",
        length: <length of XML context data>,
        stub: true,
      }
    }
}
```

Figure 4.12: SQL+NoSQL Hybrid: JSON Item with Attachment

As we can see, the XML attachment is not retrieved with the document; instead, it should be requested in a separate API request, giving the attachment name (in this case, "data") as parameter. The response would be the context data associated with that Item, in XML format.

This option has two big advantages: on one hand, we no longer would have to worry about manually converting XML to JSON, and back to XML. The context data would be stored exactly as received, without further processing; on the other hand, the Apache Lucene searching engine (which is used both on the couchdb-lucene [63] plugin and the elasticsearch [64] application) allows attachments to be indexed. This would seem to make XML attachments the obvious solution to the problem.

After implementing and deploying this solution, one big problem emerged: for every CouchDB API request, only attachments from a single document can be retrieved. This would jeopardize the XMPP PubSub protocol, as it is possible to ask for the most recent 20 Items, which would then translate in 20 CouchDB API requests. As the CouchDB API is HTTP based, this would mean 20 HTTP requests and 20 HTTP responses; even if 20 HTTP requests plus 20 responses wouldn't seem as much impediment, it is also possible to as for 100, or even 1000 items. It is then obvious that, although conceptually it was the best option, this architecture would not scale with a large number of Item requests.

4.7 Iteration 4: NoSQL Items-only with XML document string

Although the solution implemented in the previous iteration (4.6), which stores context information as XML attachments, provides the advantage of storing context data in a structured way, in XML format, the inability to retrieve multiple XMPP PubSub Items in a single request makes this implementation inefficient. Even though the largest

part of the XMPP PubSub protocol consists in generating notifications when Items are published, which doesn't need to access previously stored Items, the ability to retrieve multiple stored Items on-demand must not be overlooked.

With the options of storing context as structured JSON documents and as XML attachments out of the question, one option remains: storing context data as an XML string in a JSON document field. At first glance this would seem counter-productive; after all, we are using a document-oriented storage systems, which allows us to store JSON documents with any structure. However, after deployment, this proved to be the best option, and fulfills all requisites.

With this solution, we overcome all problems posed by Iterations 2 4.5 and 3 4.6 (Iteration 1's problem was the usage of CouchDB to store all data, which proved inefficient, and was corrected from Iteration 2 onwards; see 4.4). In Iteration 2 context data was converted from XML to JSON and stored as a nested JSON field, which although possible, made indexing by Apache Lucene difficult, by requiring the configuration of the indexing function for each different context type; furthermore, Lucene requires that JSON fields be indexed individually, meaning that for each context type, it would be required to indicate all fields which should be indexed. With Iteration 3, context storage was stored in an XML attachment; however, it is currently impossible to retrieve attachments from multiple documents in CouchDB, as required by the XMPP PubSub protocol.

On the other hand, when storing context data as an XML string in JSON, it is possible to retrieve context data from multiple documents, as they are included in the documents themselves, which overcomes the problem with Iteration 3. As context data is stored as a single XML string in a JSON document, this field can be indexed in Apache Lucene, and that makes all context data automatically searchable, overcoming the problem with Iteration 2.

The resulting document would look like the following:

```
{
    "_id": "3bbba1c1-2cff-4a5c-824a-91e8e51663ec",
    "_rev": "1-6ccd0747a2780ca866240df3e725ed56",
    "node": "socialprofile:tiago@c3s.av.it.pt/facebook",
    "doc_type": "item",
    "date": "2011/04/29 18:23:34.305383",
    "publisher": "socialprofile.c3s.av.it.pt",
    "data": "<item id='3bbba1c1-2cff-4a5c-824a-91e8e51663ec'><person
       xmlns='http://jabber.org/protocol/pubsub' jid='tiago@c3s.av.it
       .pt'> <name> <given_name>Tiago</given_name> <family_name>
       Ribeiro</family_name> </name> <gender>male</gender> <
       date_of_birth>09/18/1988</date_of_birth> <current_location>
       Aveiro, Portugal</current_location> <musics> <music>Led
       Zeppelin Official</music> <music>K.A.R.M.A.</music> <music>
       Ornatos Violeta</music> <music>Green Day</music> <music>Muse</
       music> <music>Foo Fighters</music> <music>The Prodigy</music>
       <music>Coldplay</music> <music>Pink Floyd</music> <music>HOME
       DA LUTA</music> <music>Pearl Jam</music> <music>Slash</music>
       <music>Bob Marley</music> <music>Guns N' Roses</music> </
       musics> <tv_shows> <tv_show>UTAD TV</tv_show> <tv_show>24</
       tv_show> <tv_show>South Park</tv_show> <tv_show>The Big Bang
       Theory</tv_show> <tv_show>5 para a meia noite</tv_show> <
       tv_show>How I Met Your Mother</tv_show> </tv_shows> </person
       ></item>"
}
```


Figure 4.13: JSON document with context data as an XML string

The next step is integrating CouchDB with a searching engine, either Apache Lucene itself or elastic search (which uses Apache Lucene internally). Although elastic search is a distributable application in itself, which provides bindings to CouchDB, a plugin exists to integrate CouchDB with Apache Lucene directly in CouchDB, couchdb-lucene, made by Robert Newson, an Apache CouchDB commiter [63]. As CouchDB provides replication features, a single replicated node could be setup to provide searching capabilities independently of the XMPP PubSub component. Items would be published to the XMPP PubSub component's CouchDB node, which would then be replicated to the lower-priority Apache Lucene CouchDB node.

Couchdb-lucene requires a design document, similar to CouchDB Views, with a function (in javascript) responsible for indexing JSON document fields. This fields would then be searchable directly in CouchDB. The indexing function used was the following:

```
function(doc)
{
    var ret = new Document();
    ret.add(doc.node, {'field': 'node'});
    ret.add(doc.publisher, {'field': 'publisher'});
    ret.add(new Date(doc.date.substring(0, 19)), {'type': 'date', '
       field': 'date'});
    ret.add(doc.data, {'field': 'context'});
    return ret;
}
```

Figure 4.14: couchdb-lucene indexing javascript function

This function creates a new document, adds the fields to be indexed and exposed to the searching engine, as well as the field name with which it will be identified. In this case we maintain all field names except the "data" JSON field, which we will call "context" in the search index. It is now possible to search the context database by any of these fields: "node", "publisher", "date" or "context". The indexing happens couchdb-lucene, but the searching API is provided by the CouchDB node itself. The query API address is the following:

```
http://<CouchDB URI>/<DB>/_fti/_design/<Design Doc>/<Indexing Func>?q=<Query>
```

"CouchDB URI" corresponds to CouchDB's URI, usually "localhost:5984"; "Indexing Function Name" corresponds to the name given to the indexing function created in 4.14; the "Query" is the searching query we want to perform, and it can be a search string, or a combination of <Field>:<Query String>. Using only a query string searches for the string in all fields, where giving the field/search string combination searches only in that particular field.

Some optional arguments can be appended at the end of the query, such as "include_docs=true" to include documents in the results, "sort" to sort the results by fields, or "limit" to limit the number of results. The full parameters description can be found in the github project page [63].

We can then, for example, make the following query using an HTTP GET:

```
http://localhost:5984/pubsub_items/_fti/_design/pubsub/context?q=context:24
```

This address corresponds to the "pubsub_items" CouchDB database, and to the "pubsub" design documents. The indexing function shown in 4.14 is named "pubsub", and the query is "context:24". This query means that we want to search in the field "context" for the string "24".

The result would be returned as a JSON document, with the following format:

```
{
    limit: <The maximum number of results that can appear>
    etag: <An opaque token that reflects the current version of the
       index>
    fetch_duration: <The number of milliseconds spent retrieving the
       documents>
    q: <The query string>
    search_duration: <The number of milliseconds spent performing the
        search>
    total_rows: <Number of matches>
    skip: <Number of initial matches that were skipped>
    rows: <Rows of results>
}
```

Figure 4.15: couchdb-lucene query result

The result rows have the following format:

```
{
    id: <ID of the CouchDB document>
    score: <The normalized score (0.0-1.0, inclusive) for this match>
    doc: <CouchDB document, in case the "include\_docs" parameter is
       true>
}
```

Figure 4.16: couchdb-lucene query result row

The result of the query of the previous example would be (with number of rows trimmed):

```
{
    "limit": 25,
    "etag": "3f6b7850b421d",
    "fetch_duration": 0,
    "q": "context:24",
    "search_duration": 2,
    "total_rows": 40,
    "skip": 0,
    "rows": [
        {
            "id":"lzqgLAkJIj71iGP",
            "score":0.614368200302124
        },
        {
            "id":"Oxw9PPYYEuO1HVw",
            "score":0.614368200302124
        },
        {
```

```
            "id":"XPd6F5955e9EEAD",
            "score":0.614368200302124
        },
        {
            "id":"9Jo0mexlw5EPPVT",
            "score":0.614368200302124
        },
        {
            "id":"21TLXq1lsE02xl6",
            "score":0.614368200302124
        },
        {
            "id":"sfnf445tOPgpuMb",
            "score":0.614368200302124
        },
        ...
    ]
}
```

4.8 Implemented Features

As seen in the previous sections, after all iterations the following features were implemented:

- Full CouchDB storage-engine
- PostgreSQL / CouchDB hybrid storage-engine
- XEP-0248 (PubSub Collection Nodes) spec, for all storage-engines
- XEP-0030 (Service Discovery) spec, also for all storage-engines
- XEP-0060: Retrieving all subscriptions for a given node
- XEP-0060: Managing node configuration
- XEP-0060: Managing subscription options
- XEP-0060: Managing node affiliations
- Minor bugfixes

4.8.1 CouchDB Storage-Engine

As detailed in Iteration 1 4.4, a full CouchDB storage-engine was developed. This storage engine would store all data required by the XMPP Publish-Subscribe protocol, as described in Idavoll's data model 4.1, on CouchDB, instead of a traditional SQL solution that was Idavoll's main storage-engine, and also opposed to the approach used by other XMPP servers such as Openfire. The reasons that led to its implementation, and its

subsequent advantages and disadvantages are best described in the Iteration 1 section 4.4.

CouchDB is a document-oriented storage system, which means it stores documents, in this case in JSON format. These documents can be of any structure, and can perfectly be mixed in the same database. To accomplish this, five types of documents were created in Idavoll, all detailed in section 4.4: Node, Entity, Subscription, Affiliation and Item. These documents would then be created as demanded, and stored in CouchDB, using its HTTP API. Document retrievals would also be handled through this HTTP API.

The implementation structure for this storage-engine was very similar to the one already implemented, which used PostgreSQL. It mostly followed the same principles and control structures, while the data was adapted to fit into these documents, instead of database tables. As CouchDB does not support relations, some database operations that required only one access to the database in PostgreSQL (due to SQL JOIN operations) require several CouchDB API requests, with every request translating in an HTTP request / response pair.

4.8.2 PostgreSQL / CouchDB Hybrid Storage-Engine

The PostgreSQL / CouchDB hybrid storage-engine was developed in the second iteration 4.5, and used throughout the remaining iterations (3 4.6 and 4 4.7), although with minor adjustments between them.

The objective was to store only XMPP PubSub Items in CouchDB, while all other data being stored in PostgreSQL, as was the case with the existing PostgreSQL storage engine. A new storage-engine was then created, to implement this hybrid approach; however, not everything was implemented from the ground up, even though all features needed by Idavoll were provided. This was accomplished using the following approach.

It was understood that only XMPP PubSub Items were to be stored in CouchDB. This translates in a very short number of operations that need interaction with CouchDB: namely the operations to store Items, and retrieve Items (either by Node, or by their respective Item ids). As such, all the remaining operations would use PostgreSQL, as was the case with the PostgreSQL storage-engine. Thanks to the polymorphism features available in Python, together with the way storage-engines were implemented in Idavoll, it was possible to create a new storage-engine which derived from the PostgreSQL storage-engine. This hybrid storage engine implemented the CouchDB operations (the ones which handled Items), but not the PostgreSQL operations (Nodes, Subscribers, etc), instead delegating these to the PostgreSQL storage-engine. This allowed for a much cleaner development of most features, as when implementing features of fixing bugs in the PostgreSQL storage-engine, both this and the PostgreSQL/CouchDB hybrid would benefit.

The document structure used was identical to the one used in the full CouchDB storage-engine, shown in 4.6. Several different approaches were used in storing the actual context information inside this document, such as storing it as JSON object, XML document attachment, or XML string in a JSON field, all described in sections 2 through 4, with the XML string being the final implemented solution.

4.8.3 XMPP PubSub Collection Nodes (XEP-0248)

The XMPP PubSub Collection Nodes extension allows for the existence of two types of nodes: leaf nodes, into which Items are published; and collection nodes, which can contain other collection nodes or leaf nodes, and no Items are published to it. This allows the existence of a hierarchy of nodes. When a subscription is made to a collection node, notifications for all its child nodes are also received.

It is, then, necessary to represent hierarchies both in PostgreSQL and CouchDB.

Independently of each storage-engine implementation details, the following functionalities were altered / implemented, to support the existence of collection nodes:

- Creating a node
- Listing all descendants of a node
- Configuring a node
- Listing subscribers of a node

For creating a node (_createNode function in all storage-engines) the existence of 2 new configuration parameters is searched: "pubsub#node_type", indicating if we are creating a collection node or a leaf node; and "pubsub#collection", indication what is (if any) the parent collection of the node. It also checks for the existence of the collection node in the database.

To list all descendants of a node (getNodeIds function in all storage-engines), only nodes which are descendent of the parent node indicated as a parameter are returned; in case there is no parameter given, only root nodes are returned. This change simultaneously implemented the Service Discovery extension (XEP-0030).

For configuring a node (_setConfiguration function in all storage-engines), as was the case with creating a node, the 2 new configuration parameters "pubsub#node_type" and "pubsub#collection" are also processed and stored, and the existence of the collection node is also checked.

The listing of subscribers of a node is a more complex case. Let's consider the following node hierarchy:

- collection-1
- **collection-2**
  - **collection-2-1**
    - collection-2-1-1
    - **leaf**
  - collection-2-2
    - collection-2-2-1
- collection-3
  - collection-3-1

In this example, only a single leaf node exists, "leaf", with ancestors "collection-2-1" and "collection-2" (this dependency hierarchy is emphasized in bold). In this case, whenever an item is published on the node "leaf", it is not enough to send notifications to its subscribers; it is also necessary to identify all its ancestor nodes' subscribers, and notify them also. This means identifying all the subscribers of the nodes "leaf", "collection-2-1" and "collection-2". This subscribers are then grouped by Jabber ID and Resource, thus avoiding repeated notifications (for example, an entity could be subscribed to both the "collection-2-1" and "collection-2", which would generate two notifications).

4.8.3.1 Collection Nodes: PostgreSQL

Representing hierarchies in SQL presents a known problem, with several possible solutions: these include representing the hierarchy model with an Adjacency List, or as Nested Sets.

4.8.3.1.1 Adjacency List The implementation of hierarchies using an adjacency list consists in, for every item, storing a reference to its parent item. It is well described in Joe Celko's book "Trees and Hierarchies in SQL for Smarties" [71, Chapter 2.1]. The following figure, taken from the book, and the corresponding table of Item / Parent relationship, are an example of the Adjacency List model:

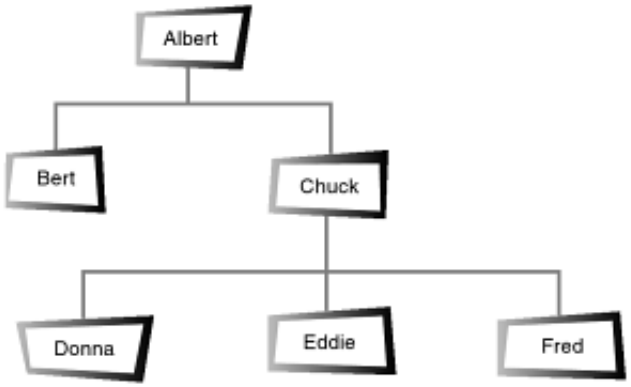


Figure 4.17: "Joe Celko's Trees and hierarchies in SQL for smarties": Adjacency List [71, p. 18]

Item	Parent
Albert	-
Bert	Albert
Chuck	Albert
Donna	Chuck
Eddie	Chuck
Fred	Chuck

This is the solution with simpler implementation, as only a reference to the parent item is needed; however, it needs recursive queries. For example, to find the direct descendants

of "Albert", we only need to query all items which contain as parent "Albert"; however, to find all descendants of Albert, besides querying all items with parent "Albert", we then need to query all items which contain the result from the previous query as parent, and so on. For implementing this solution in SQL, optimized recursive queries are an absolute requirement.

4.8.3.1.2 Nested Sets The Nested Sets model is another model for representing hierarchical data in SQL systems.

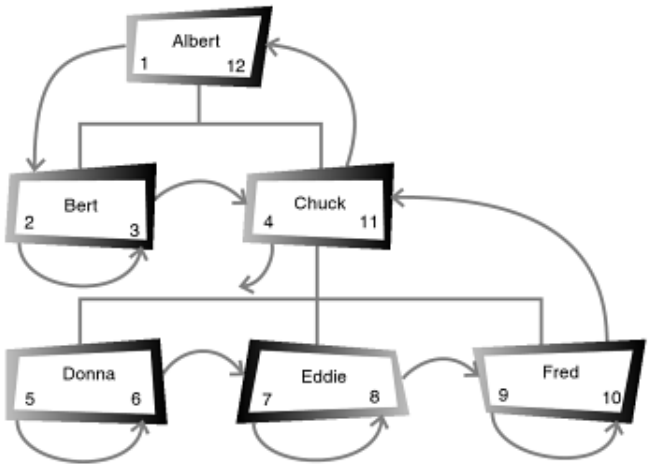


Figure 4.18: "Joe Celko's Trees and hierarchies in SQL for smarties": Nested Sets [71, Chapter 4.1]

This model consists in enumerating all nodes, according to the tree traversal, visiting each node twice, assigning numbers in the order of visiting. In the example of the previous figure 4.18, this would generate the following data:

Item	Left	Right
Albert	1	12
Bert	2	3
Chuck	4	11
Donna	5	6
Eddie	7	8
Fred	9	10

To obtain a list of parents of a node, we have to search for nodes which have, simultaneously, a smaller Left value and a larger Right value, or in other words, nodes which its Left/Right ranges contain the initial node's Left value (in properly build hierarchies, these also automatically contain the initial node's Right value). For example, for the node Eddie (7, 8), the only nodes whose Left/Right ranges contain the first's Left value are Chuck ($4 < \mathbf{7} < 11$) and Albert ($1 < \mathbf{7} < 12$).

To obtain a list of direct descendants, we just have to make the inverse operation, which is, look for nodes whose Left value is inside the Left/Right value range of the first

node. For example, for the Chuck node (4, 11), the only nodes whose Left value are inside the initial node's ranges are Donna ($4 < \mathbf{5} < 11$), Eddie ($4 < \mathbf{7} < 11$) and Fred ($4 < \mathbf{9} < 11$).

This model allows the building of complete hierarchies (descendants, parents) without recursive queries. Although this model is slightly more complex than Adjacency List, it is the only one available on platforms without recursive query optimization.

4.8.3.1.3 Selected Model Performance tests between the two models are presented in the subsequent section 5.1.1. The database used was PostgreSQL 8.4, as it contains optimizations for recursive queries; moreover it was released in 2009, so it is quite stable.

As referred in 5.1.1.5, the selected model as the **Adjacency List** model, as it provides very good performance, and benefits greatly from PostgreSQL 8.4's recursive query optimizations.

For the XMPP PubSub Collection Nodes specification it is only necessary to obtain a list of direct descendants (also for Service Discovery) and a list of all ancestors of a node (to generate notifications to subscribers of all ancestor nodes).

For this, a new field was added to the Nodes data model (see 4.1), "collection", representing the id of the collection node to which the node is associated with. For root nodes, which are not associated with any node, this value is "0"; there is obviously no node with id "0", all nodes have an id > 0.

4.8.3.2 Collection Nodes: CouchDB

Representation of collection nodes in CouchDB was only necessary in the first iteration (4.4). On subsequent iterations, nodes were stored in SQL solutions, described in the previous section 4.8.3.1. Nevertheless, the approach used is here described.

As CouchDB store JSON documents, a simple way to represent the node tree hierarchy would be to represent it exactly as a JSON object. This was the chosen solution as implemented in the first iteration. Let's take, as an example, the node collection tree presented in figure 4.8, here repeated:

```
{
    "_id": "nodecollection"
    "doc_type": "collection_node_tree",
    "collection":
    {
        "root":
        {
            "child1":
            {
                "child11"
            },
            "child2": {},
            "child3": {},
        }
    }
}
```

Figure 4.19: Example of a node collection tree in JSON

If an item is published to a leaf node which has the node "child11" as a collection, it is necessary to first find all ancestors of "child11". The first step is, then, to find the correct node in the JSON object.

In Python, this JSON structure is directly translated to a Python dictionary, with the exact same representation as the example JSON object. The process of finding the necessary collection node is the most computationally intensive one, as one has to search first on the root nodes, then on every node for each root node, and so on. A list of the tree traversal is maintained, and when the correct node is found, all its ancestors are also retrieved. After finding all ancestors, all subscribers of these nodes are retrieved, and notifications for all subscribers are sent.

For small numbers of nodes, with a maximum of 3 levels of depth, as is the case observed in the deployed instances of this solution, this exhaustive search does not present a problem. As a generic solution, though, this is far from ideal.

As the idea of storing and representing nodes and node hierarchies in CouchDB was abandoned after the first iteration, this became a non-problem; otherwise, better algorithms and representations for node hierarchies would have to be investigated.

4.8.4 XMPP Service Discovery (XEP-0030)

The XMPP Service Discovery specification allows for the interactive discovery of Nodes. Although initially there was no support for this specification, the Wokkel library already had support for the required messages; all that was required was to implement the functionality to allow the retrieving of all nodes with a specific collection node.

For the retrieval, initially the collection node parameter would be empty, thus returning only the root nodes. After selecting one of the root nodes, all nodes which have this selected node as collection would be retrieved, and so on.

With the implementation of the XEP-0248 XMPP PubSub Collection Nodes (4.8.3),

and more specifically the implementation of the retrieval of descendants of a given node, the Service Discovery was also completed, as for every node selected, all nodes which have the selected node as collection are retrieved, and so on.

4.8.5 XMPP Publish-Susbcribe (XEP-0060): New Use Cases

Besides modifications and minor additions to use cases that were already implemented in Idavoll, the following use-cases were fully implemented from the ground up, as no previous implementation existed:

4.8.5.1 Configure Node

XMPP PubSub 8.2: Configure a Node

`http://xmpp.org/extensions/xep-0060.html#owner-configure`

Allows the changing of a node's configuration after its creation. Supports the same fields as in the node creation: "pubsub#node_type", "pubsub#collection", "pubsub#persist_items", "pubsub#deliver_payloads" and "pubsub#send_last_published_item". Only Idavoll needed to be altered for this use-case, both on the Backend layer and all storage-engines.

4.8.5.2 Retrieve Subscriptions

XMPP PubSub 5.6: Retrieve Subscriptions

`http://xmpp.org/extensions/xep-0060.html#entity-subscriptions`

Allows the listing of an entity's own subscriptions, either to a specific node, or to all nodes. The Wokkel library was already prepared for the XMPP messages, and the only change was made to Idavoll, on the Backend layer and all storage-engines.

4.8.5.3 Modify Affiliation

XMPP PubSub 8.9.2: Modify Affiliations

`http://xmpp.org/extensions/xep-0060.html#owner-affiliations-modify`

This use case enables a node's owner to modify the node's entity affiliations. The owner can change one or more affiliations, for example adding publishers ("publisher", "publish-only") or blacklisting users ("outcast"). The owner can even change its own affiliation to the node.

The Wokkel library was already prepared for the needed XMPP messages, being only necessary to implement this feature in Idavoll, both on the Backend layer and all storage-engines.

4.8.5.4 Retrieve Affiliations List

XMPP PubSub 8.9.1: Retrieve Affiliations List

`http://xmpp.org/extensions/xep-0060.html#owner-affiliations-retrieve`

Refers to the possibility of retrieving a list of a node's affiliations, by the node's owner.parte do seu dono. As before, only Idavoll was changed for this use-case, also on both the Backend and all storage-engines.

4.8.5.5 Configure Subscription Options

XMPP PubSub 6.3: Configure Subscription Options
`http://xmpp.org/extensions/xep-0060.html#subscriber-configure`

Support for configuring subscription options was added, essential to allow subscription of collection nodes. This feature only needed to be implemented in Idavoll, both on the Backend layer and all storage-engines, and allows the following options:

- "pubsub#subscription_type"
- "pubsub#subscription_depth"

The parameter "subscription_type" refers to the type of subscription we want to make. It can either be "items", where notifications of item publications in the node's child leaf nodes are sent; or "nodes", where notifications are sent when nodes are created inside the subscribed node.

The "susbcription_depth" parameter refers to the collection node depth of the subscription. A subscription depth of 1 subscribes to changes only to direct child nodes, 2 subscribes to both direct child nodes and childs of these direct childs, and so on; a value of "all" subscribes to all the collection node tree depth.

The project requirements only required the implementation of the "subscription_type" of "nodes", and the "subscription_depth" of "all".

4.9 Idavoll Web Interface

To facilitate the management of the XMPP Publish-Subscribe protocol, a web interface was created. This web interface enables read-only access to the XMPP PubSub database, and allows us to check the existing Nodes, Entities, Affiliations and Subscriptions present in the system, as well as searching the existing Items using Apache Lucene.

As Idavoll was developed in Python, it was decided to implement the web interface in Django [1], which is a Python Web Framework. Using the same programming language for both Idavoll and the web interface allows the usage of the same libraries for accessing CouchDB and PostgreSQL, greatly simplifying the development.

A list of features offered by the web interface is presented bellow:

- Nodes
  - List all
  - Details
- Items
  - Number of items, size of CouchDB Items database
  - Full-text search of item contents, using Apache Lucene
- Entities

[1] https://www.djangoproject.com/

  - List all
  - Details
- Subscriptions
  - List all
  - Filter by Entity
  - Filter by Node
- Affiliations
  - List all
  - Filter by Entity
  - Filter by Node

Some screenshots of the implemented web interface are shown bellow.

This first screenshot in figure 4.20 shows the start page of the web interface. It shows the number of Nodes, Items, Entities, Subscriptions and Affiliations.

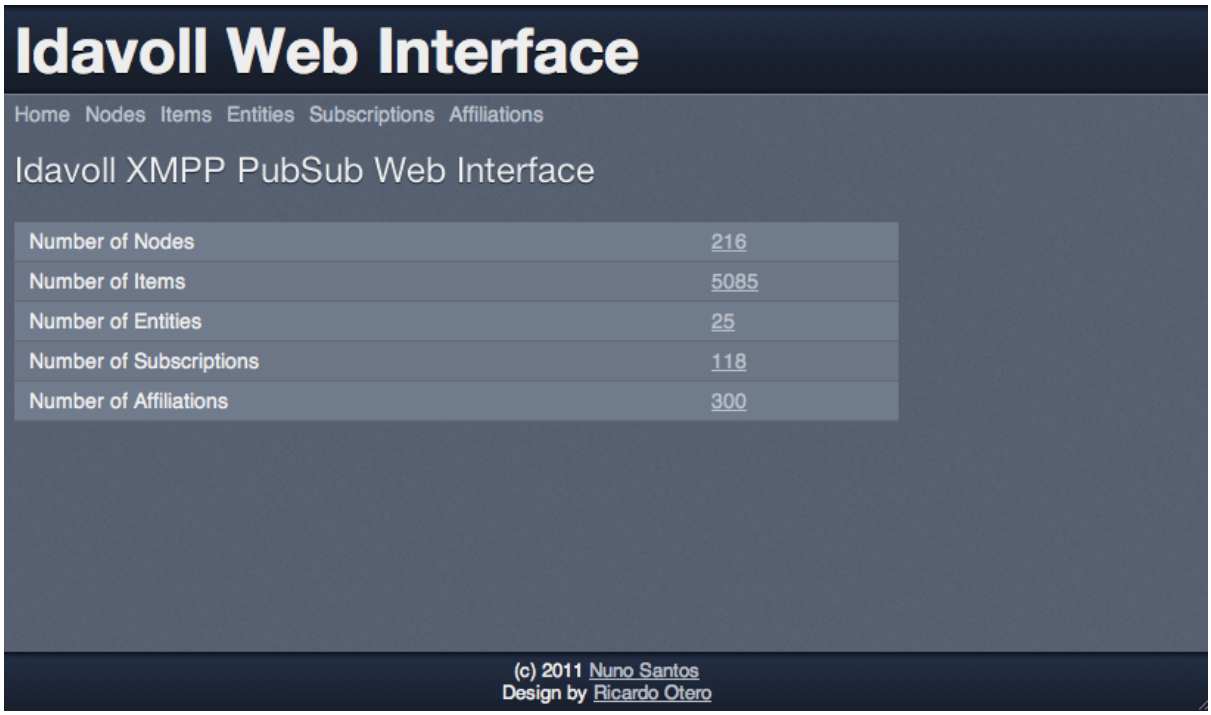


Figure 4.20: Idavoll Web Interface: Start Page

The screenshot in figure 4.21 shows the interface that allows Item searching, using Apache Lucene; it shows also the searching result statistics, such as search duration and document fetch duration (as obtained by Lucene).

The screenshot in figure 4.22 shows the details of an Item, as returned by an Item search query. It shows the Node on which it was published on, its Publisher and the Date of publishing, as well as the context information in XML format.

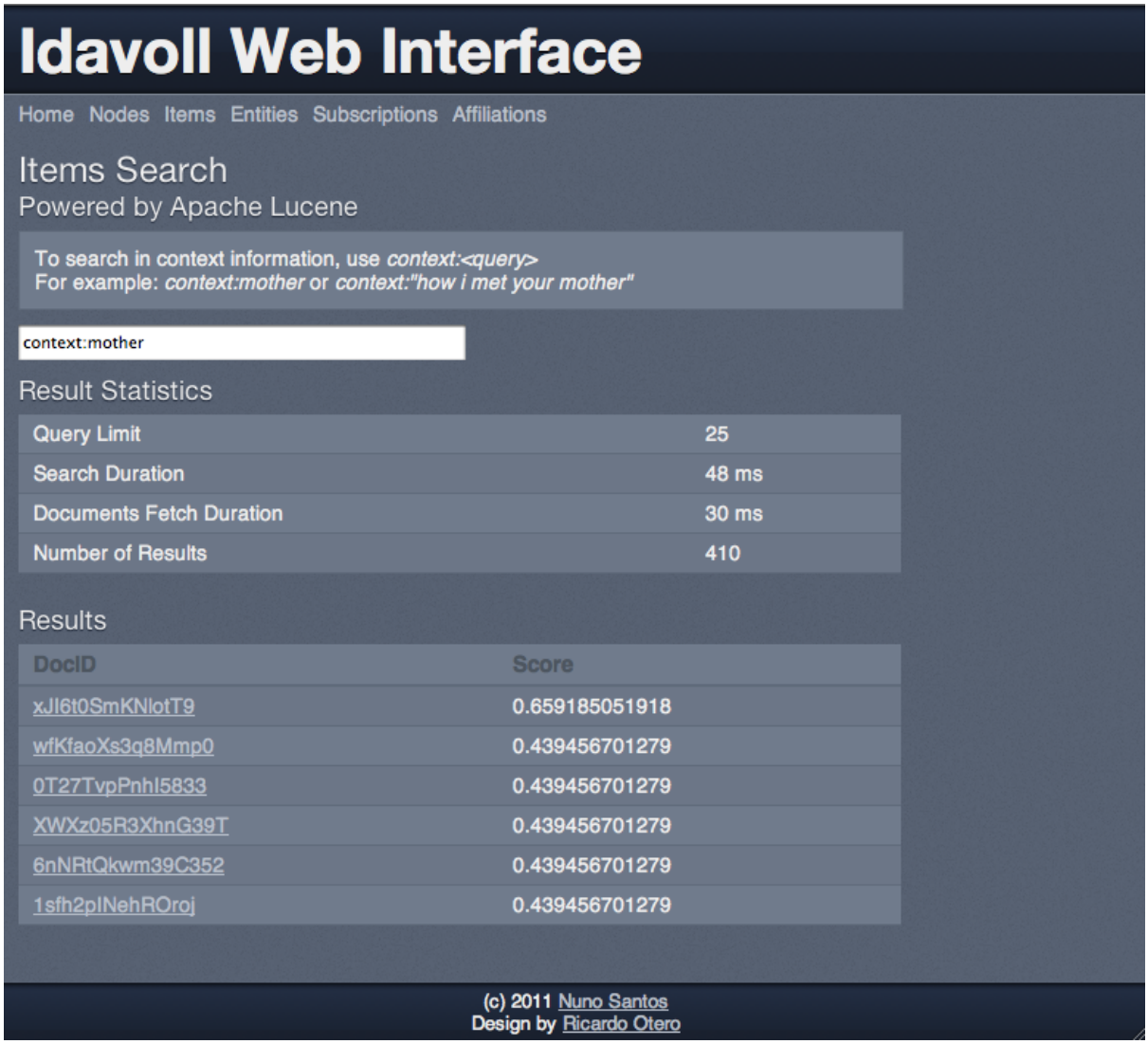


Figure 4.21: Idavoll Web Interface: Item Search (using Apache Lucene)

The screenshot in figure 4.23 shows a list of subscriptions for a specific Entity. It is possible to remove the Entity filter (and return all subscriptions), or further filter by Node.

Figure 4.22: Idavoll Web Interface: Item Details

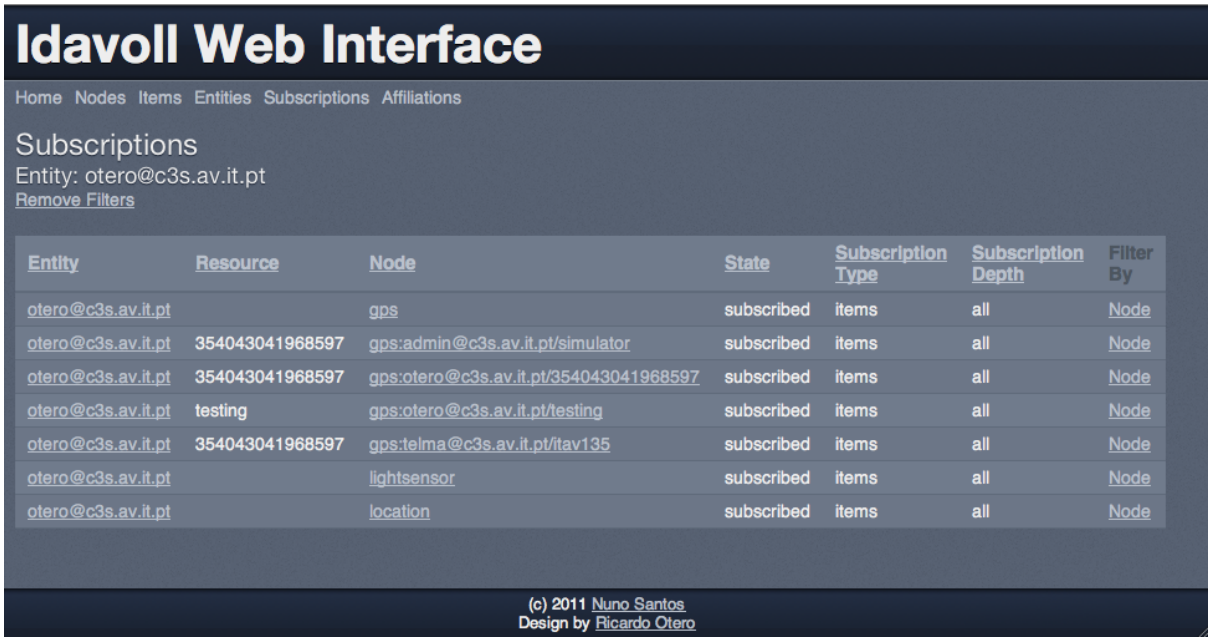


Figure 4.23: Idavoll Web Interface: Subscriptions, filtered by Entity

Chapter 5

Results

5.1 Performance Tests

5.1.1 PostgreSQL: Nested Sets vs Adjacency List Model

As explained in 4.8.3.1, for representing hierarchical data in SQL there are essentially two possibilities: using Nested Sets, or the Adjacency List model. These performance tests evaluate the performance implications of using these two models, and help explain the choice that was made in section 4.8.3.1.3.

As the database used by the Idavoll application is PostgreSQL, and PostgreSQL supports recursive queries only since version 8.4 [72], both the application and these performance tests require version 8.4 or above. These tests were performed inside a VMWare Virtual Machine, using Ubuntu 10.04 LTS.

For the XMPP Publish-Subscribe Collection Nodes implementation, the only necessary operations are the listing of direct descendants, and the listing of all ancestors of a node. These were the use cases tested (5.1.1.4.1 and 5.1.1.4.2).

5.1.1.1 Nodes Database Structure

For the performance tests, a database structure for representing nodes was created, similar to the structure used in the Idavoll application. For the Nested Set model, only two fields are needed: "lft" and "rgt" representing, respectively, the IDs of the nodes for the left and right fields needed by the Nested Sets. For the Adjacency List, only the "parent" field is needed, representing the node ID of the parent in the hierarchy. Additionally, indexes were created for these three fields.

The structure of the nodes table created was as follows:

Field	Type	Extras
node_id	int	index, primary key
node	text	not null, unique
node_type	text	not null
persist_items	boolean	-
deliver_payloads	boolean	not null, default true
send_last_published_item	text	not null, default "on_sub"
parent	int	index
lft	int	index
rgt	int	index

Table 5.1: PostgreSQL Nested Sets vs Adjacency List: Nodes Data Structure

Indexes were created on the primary key field "node_id", and on the "parent", "lft" and "rgt" fields.

A "dummy" node with node_id = 0 was created, to represent the "fake" root of the node hierarchy. This is only a "fake" node because it has no name, and is only used to allow the representation of a forest (node hierarchy with multiple root nodes), with a single root node represented in the database (the "fake" root node).

5.1.1.2 Test Environment

CPU	Intel(R) Core(TM)2 Duo CPU E8335 @ 2.66GHz (limited to 1 core)
Memory	512 MB
Operating System	Ubuntu 10.04 LTS VMware virtual machine
Database	PostgreSQL 8.4.4

Table 5.2: PostgreSQL Nested Sets vs Adjacency List: Test Environment

5.1.1.3 Dataset

For these tests it is essential that the data-set tested is identical to a real world use-case. For this, the following node hierarchy was created:

- First level: 10 nodes
- Second level: 20,000 nodes
- Third level: 3 nodes
- Total: 10 + 10*20000 + 10*20000*3 = 800,010 nodes

The node hierarchy in an example deployment of the XCoA platform follows a simple hierarchy based on the JabberIDs (JIDs) of users. The first level represents all possible context types (gps, location, social profile, etc). The 10 nodes created represent 10 different types of context stored in the database. In the second level, and for each first level item, there are the base JIDs of users (gps:user@example.com, location:user@example.com, gps:user2@example.com, etc). Each second level node represents

a user, while third level nodes represent JID resources, i.e. different equipments used by each user. 20,000 users with three resources each represents an acceptable real-world test-case. A dataset of 200,000 users, would better emulate a good real-world scenario; however, with these numbers some tests would take a very long time to complete, so a choice o 20,000 users was a good compromise, as it already shows tendencies of which solution is better performing.

5.1.1.4 Tests

5.1.1.4.1 Retrieving Direct Descendants Given that our node tree has 3 levels, it only makes sense to retrieve a list of direct descendants in the first and second levels (as the third level nodes have no descendants). Theoretically the retrieving of first level descendants should take longer than second level, as each first level node has exactly 20000 descendants, and each second level only 3, so only the retrieval of direct descendants of first level nodes was tested.

The SQL statements were taken from Quassnoi's "Explain Extended" comparison between Adjacency List and Nested Sets in PostgreSQL [73].

The following tests retrieved direct descendants of first-level nodes, which should all retrieve exactly 20001 nodes (20000 child nodes plus the parent node itself).

5.1.1.4.1.1 Adjacency List Using the adjacency list, the SQL used to retrieve the descendants was the following:

```
SELECT nodes.node FROM nodes
INNER JOIN nodes AS n ON (nodes.parent = n.node_id)
WHERE n.node = <NODE NAME>
```

This is a recursive query, as it performs a single JOIN to itself.
100 runs were executed, giving the following results (in milliseconds):

- Mean: 31.065 ms
- Standard Deviation: 4.645 ms

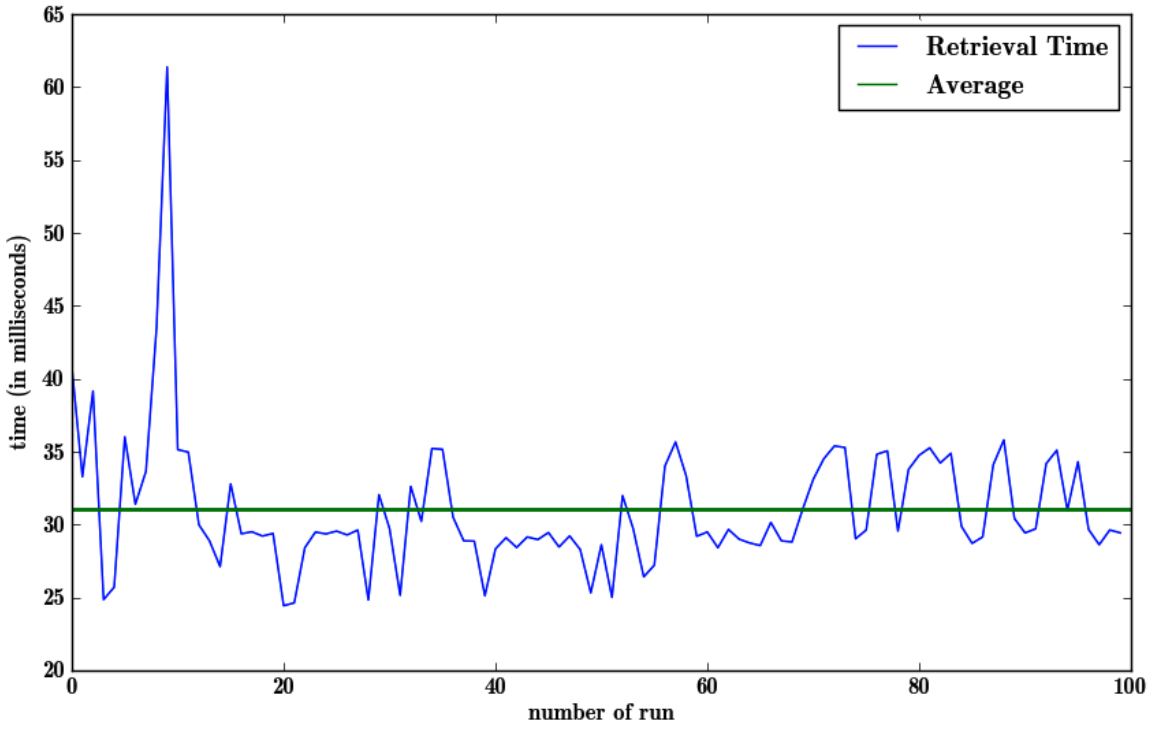


Figure 5.1: Retrieving direct descendants using an adjacency list

5.1.1.4.1.2 Nested Sets The SQL used for retrieving direct descendants of a node was the following:

```
SELECT  nc.node
FROM    nodes np
JOIN    nodes nc
ON      nc.lft BETWEEN np.lft AND np.rgt
WHERE   np.node = <NODE NAME>
AND
(
SELECT  COUNT(*)
FROM    nodes nn
WHERE   nc.lft BETWEEN nn.lft AND nn.rgt
        AND nn.lft BETWEEN np.lft AND np.rgt
) <= 2
```

This is clearly not a recursive query, as opposed to the adjacency list SQL code. As in nested sets all descendants (through all levels) are within the parent's left-right range, it is not enough to just retrieve all nodes between this range; it becomes necessary to differentiate between those who are direct descendants, and the others. The "≤ 2" validates the level within the node, in this case 2, which is both the node itself and its direct descendants.

Upon executing the previous SQL statement, it was noted that it had very poor performance, measured in tens of minutes instead of milliseconds, so only 10 runs were

executed. The results were the following:

- Mean: 1509071.255 ms (25.151 minutes)
- Standard Deviation: 8120.113 ms (0.135 minutes)

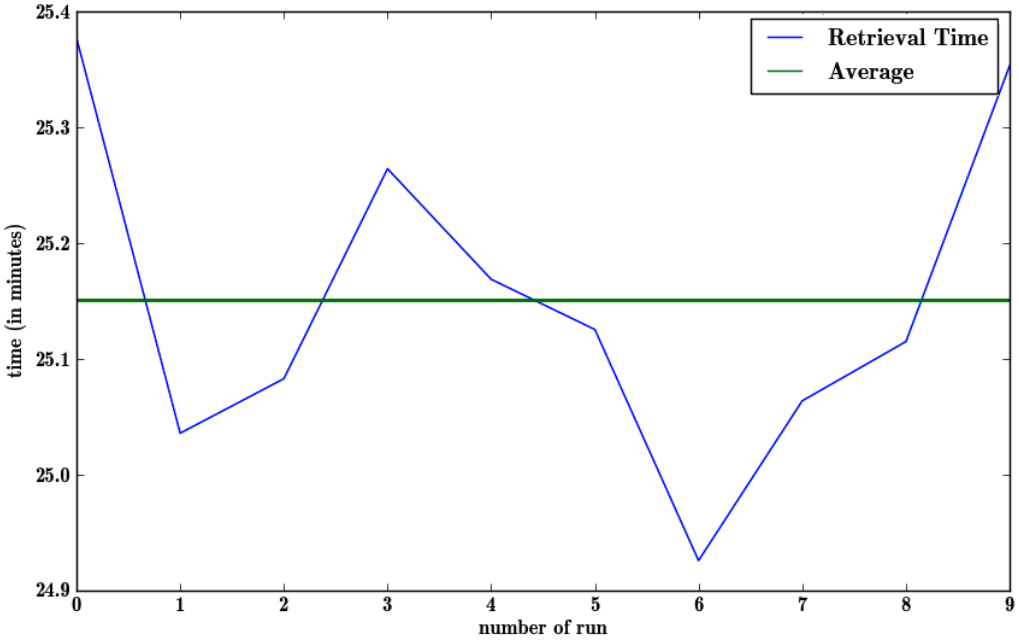


Figure 5.2: Retrieving direct descendants using nested sets

This low performance can be explained by the large number of nodes retrieved (20001 in this case). To assess exactly how the SQL statement deteriorates, the same test was executed, retrieving all descending nodes for nodes which have 500, 1000, 2500, 5000, 7500 and 10000 child nodes.

- 500 nodes: 957.823 ± 33.834 ms (approx. <1 s)
- 1000 nodes: 3777.456 ± 213.354 ms (approx. 3.7 s)
- 2500 nodes: 23621.023 ± 609.109 ms (approx. 23 s)
- 5000 nodes: 93519.411 ± 3221.499 ms (approx. 93 s)
- 7500 nodes: 211035.669 ± 4130.448 ms (approx. 3.5 m)
- 10000 nodes: 371131.703 ± 6936.825 ms (approx. 6 m)

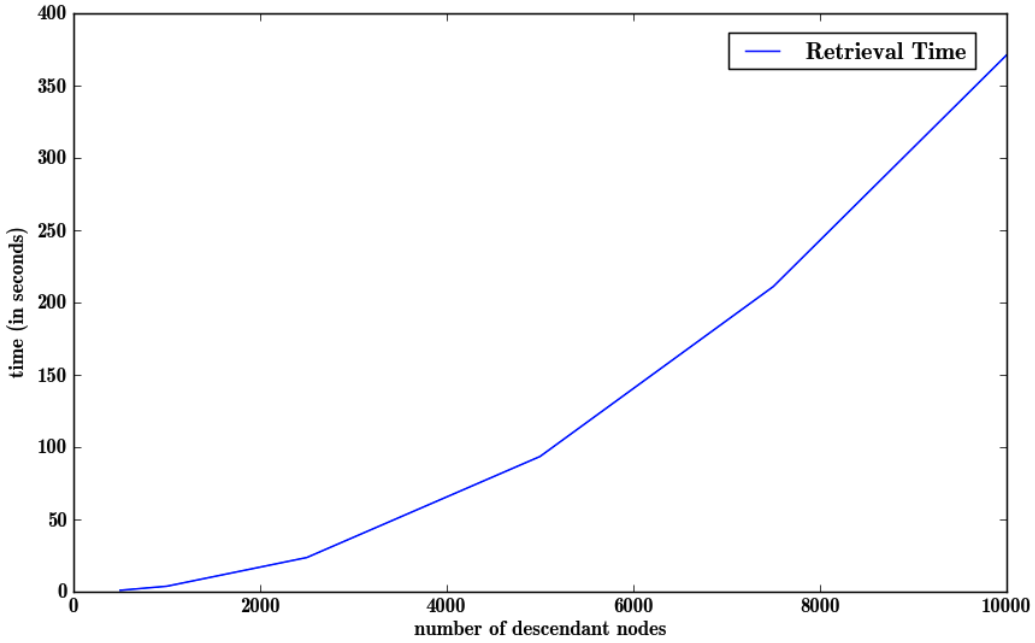


Figure 5.3: Retrieving direct descendants using nested sets: complexity

As we can see in figure 5.3, the performance when retrieving direct descendant nodes rises non-linearly with the increase of the number of child nodes.

5.1.1.4.1.3 Retrieving Direct Descendants: Conclusion

- Adjacency List: 0.031065 ± 0.004645 seconds
- Nested Sets: 1509.06 ± 8.1 seconds

With the extremely poor performance exhibited by the Nested Sets model when retrieving direct descendants, which is a crucial part of the XMPP Publish-Subscribe protocol, the obvious choice falls to the Adjacency List model. This is a much simpler model, and exhibits very good performance even for very large numbers of nodes such as 20000, with a mean access time of approximately 31ms (versus 25 minutes).

5.1.1.4.2 Retrieving All Ancestors Retrieving all ancestors of a node is an essential operation of the XMPP Publish-Subscribe protocol. When publishing items to a node, all subscribers of the target node and all subscribers of its ancestor nodes must be notified, making it necessary to retrieve a nodes' ancestor list. Both SQL statements retrieve a list of all ancestors, including the node itself. However, the SQL statement for the Adjacency List returns also the database hierarchy's "fake" root, which must not be retrieved in the protocol. It is, however, trivial to remove this "fake" node from the list of retrieved nodes in the application.

As the dataset has a hierarchy with only 3 levels, the tests focused on retrieving ancestors of third level nodes, to maximize the number of nodes retrieved (with second level nodes only one extra node is retrieved, and first level nodes have no ancestors).

The SQL statements were also taken from Quassnoi's "Explain Extended" comparison between Adjacency List and Nested Sets in PostgreSQL [73].

5.1.1.4.2.1 Adjacency List With the adjacency list, the SQL to retrieve a list of a node's ancestors is the following:

```
WITH RECURSIVE
q AS
(
SELECT  n.node, n.parent, 1 AS level
FROM    nodes n
WHERE   node = <NODE NAME>
UNION ALL
SELECT  np.node, np.parent, level + 1
FROM    q
JOIN    nodes np
ON      np.node_id = q.parent
)
SELECT  node
FROM    q
ORDER BY
       level DESC
```

Like the SQL for retrieving direct descendants using an adjacency list, this SQL statement is also recursive, taking advantage of PostgreSQL's recursive query optimizations. The retrieved nodes are ordered by level, with the higher level nodes first in the list, followed by the lower level ones.

100 runs were executed, giving the following results:

- Mean: 11.385 ms

- Standard Deviation: 8.029 ms

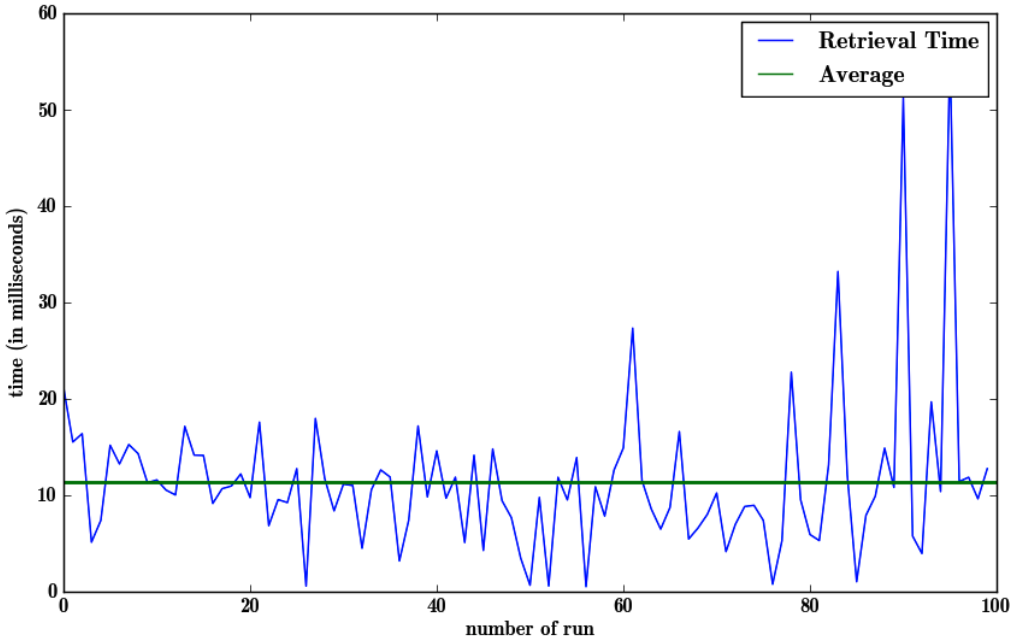


Figure 5.4: Retrieving all ancestors using an adjacency list

As we can see, even though it is a recursive query, the query is almost immediate, with a mean of approx. 11 ms. The high value of the standard deviation can be explained by the inability to measure performance times with very high accuracy. With such small numbers, getting an error margin of less than 8 ms becomes infeasible, and somewhat unnecessary.

5.1.1.4.2.2 Nested Sets The SQL for retrieving all ancestors using a Nested Sets model is the following:

```
SELECT  np.node
FROM    nodes nc
JOIN    nodes np
ON      nc.lft BETWEEN np.lft AND np.rgt
WHERE   nc.node = %s
```

As we can see, all that is needed is to retrieve all nodes whose left-right ranges contain the original node. The results obtained after 100 runs were the following:

- Mean: 4.117 ms
- Standard Deviation: 7.272 ms

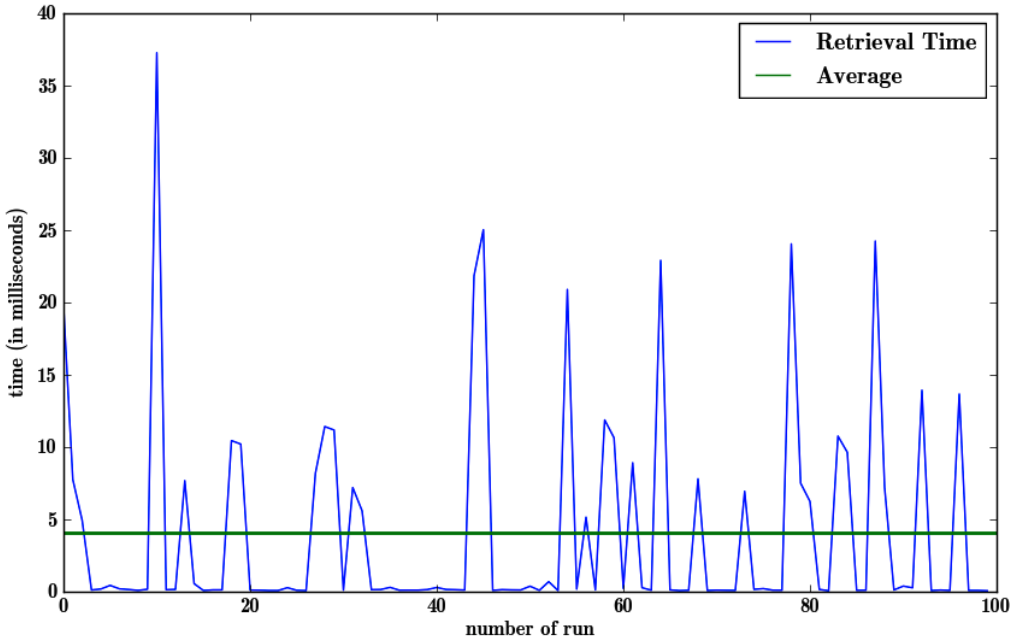


Figure 5.5: Retrieving all ancestors using nested sets

Unlike the retrieval of direct descendants, this operation is also very fast with the Nested Sets model. Like the Adjacency List operation, the standard deviation is also very high, higher than the mean, for the very same reasons as the Adjacency List's standard deviation: the operations are very fast, sometimes on the order of less than 10 ms, and it is impossible to take measurements with a higher precision. It was attempted to have more than 100 runs, but the standard deviation measured was virtually identical.

5.1.1.4.2.3 Retrieving All Ancestors: Conclusion The mean retrieval times for all ancestors of third-level nodes, where each contain exactly two ancestors, a second-level one (there are 10 + 10*20,000 = 200,010 second-level nodes total) and a first-level one (there are 10 first-level nodes total) are the following:

- Adjacency List: 11.386 ± 5.029 milliseconds
- Nested Sets: 4.117 ± 7.272 milliseconds

This operation, given the small number of nodes returned (at most N, with a hierarchy of N levels), has very high performance using both models. It could be argued that the Nested Sets model offers higher performance than the Adjacency List model; however, given that both models have such low latency numbers, the performance is virtually identical.

The high margins of error shown can be explained by the operating system's inability to accurately measure running times of such low dimensions. However, even if we assume a worst-case scenario of 4.117 + 7.272 = 11.389 for nested sets, the latencies are so considerably low that we can assume them as acceptable.

5.1.1.5 Nested Sets vs Adjacency List: Conclusions

A summary of the mean retrieval times for both retrieval of direct descendants and all ancestors, with equal datasets, is displayed below:

-	Nested Sets	Adjacency List
Direct Descendants	25.151 ± 0.135 **m**	31.065 ± 4.645 ms
All Ancestors	4.117 ± 7.272 ms	11.386 ± 8.029 ms

Table 5.3: PostgreSQL Nested Sets vs Adjacency List: Test Results

In conclusion, we can see that the Nested Sets model has a very competitive performance, beating the Adjacency List model by a mere 8ms, although when accounting with their error margins, they are both virtually identical; on the other hand, Nested sets has very poor performance when retrieving a list of direct descendants, an operation that is crucial for the XMPP PubSub protocol, with a retrieval time of approx. 25 minutes, against 31 ms of the Adjacency List.

The choice of a model falls obviously with the **Adjacency List** model. It is a much simpler model, and has the advantage of PostgreSQL 8.4 being optimized for recursive queries, a requirement of the Adjacency List; Nested Sets on the other hand is geared for when recursive queries are not an option, which is fortunately not the case with PostgreSQL 8.4.

5.1.2 PostgreSQL vs CouchDB: Item Insertion (Publishing)

This set of tests measure the performance of both PostgreSQL and CouchDB when handling large numbers of inserts, both sequentially and in batches.

Tests were made using a threaded model, resembling Idavoll's model, and a pool of 10 threads was used. Insertion performance was tested for an Item count of 5000, 10,000, 100,000 and 1,000,000 Items.

The libraries and functions used were the same as the ones used in Idavoll, to more closely emulate the application in these tests.

5.1.2.1 Items Database Structure

For this tests, a PostgreSQL table representing the Items was created, with the same structure as the one used by the Idavoll application; for the CouchDB tests, no database modeling is needed, but the structure of the documents used is also described below. It should also be noted that, as explained in 4.5, two CouchDB Views are required for the XMPP PubSub protocol, and although they are not necessary for these tests, they were also created, to emulate closely the behavior of the real XMPP PubSub application. For the PostgreSQL table, a single index was created in the primary key "item_id" field.

Field	Type	Extras
item_id	int	primary key, index
node_id	int	not null
item	text	not null
publisher	text	not null
data	text	not null
date	timestamp	not null

Table 5.4: PostgreSQL vs CouchDB Item Insertion: Items Data Structure

```
{
    "_id": "<item ID>"
    "doc_type": "item",
    "node": <node name>,
    "publisher": <publisher Jabber ID>,
    "date": <date and time>,
    "data": <context xml data>
}
```

Figure 5.6: PostgreSQL vs CouchDB Item Insertion: CouchDB Item document structure

5.1.2.2 Test Environment

CPU	Intel(R) Core(TM)2 Duo CPU E8335 @ 2.66GHz (2 cores)
Memory	512 MB
Operating System	Ubuntu 10.04 LTS VMware virtual machine
Databases	CouchDB 1.0.1, PostgreSQL 8.4.4
Disk	30GB Hard Drive

Table 5.5: PostgreSQL vs CouchDB Item Insertion: Test Environment

5.1.2.3 Dataset

To test insert operations, the Node to which the Items are published, as well as the publisher and the Item context content, is irrelevant. Although irrelevant, the same were used throughout all the tests and for both databases. The "node_id" used was always 1 on PostgreSQL, and on CouchDB the node used was always "node", and the publisher "publisher"; the context information's XML representation is the following:

```
<item id='0df3ecaa-3af9-4e38-ba82-37243e92f882'>
  <accelerometer xmlns='http://c3s.av.it.pt/accelerometer'>
    <published>Mon, 02 May 11 18:28:55 +0000</published>
    <movement>12.124095916748047</movement>
  </accelerometer>
</item>
```

5.1.2.4 Tests

5.1.2.4.1 Sequential Inserts These tests measure the performance of sequentially inserting items in both storage systems. Tests were made for insertions of 5000, 10,000, 100,000 and 1,000,000 Items. In all tests the database was in a clean state (with no items).

The following table presents the throughput of both storage systems, in terms of insertions per second. It is not an average / standard deviation measurement; instead, it shows the time it took to complete all the insertions (for each number of Items), and divides the number of inserted Items by that time to give the Items per second measure.

Number of Items	PostgreSQL	CouchDB
5000	1890.585	228.231
10,000	1317.965	230.049
100,000	1070.199	227.179
1,000,000	893.9464	217.512

Table 5.6: PostgreSQL vs CouchDB: Sequential Insertion Throughput (in items per second)

As we can see, PostgreSQL has a much better throughput than CouchDB. The poor performance of CouchDB when compared to PostgreSQL can be explained by its REST HTTP API, where each API access is always limited by the creation of an HTTP request, its sending and the waiting for a response. PostgreSQL has native access to a multitude of languages through native libraries, including Python (the one used by Idavoll and these tests), making the API access much faster.

However, we can already see that PostgreSQL's throughput degrades considerably with the increase of number of items already inserted in the database, where CouchDB's throughput is much more consistent. This further confirms that the throughput limitations of CouchDB may be caused by the API access, and not so much by its internal architecture.

The following two graphs (5.7 and 5.8) show the degradation of average insertion duration times in both storage systems for the insertion of 1,000,000 items. Note that there are 10 threads, and different threads are represented in the graphs by different colors, so the maximum number of inserted items in each thread is 1,000,000 / 10, or 100,000.

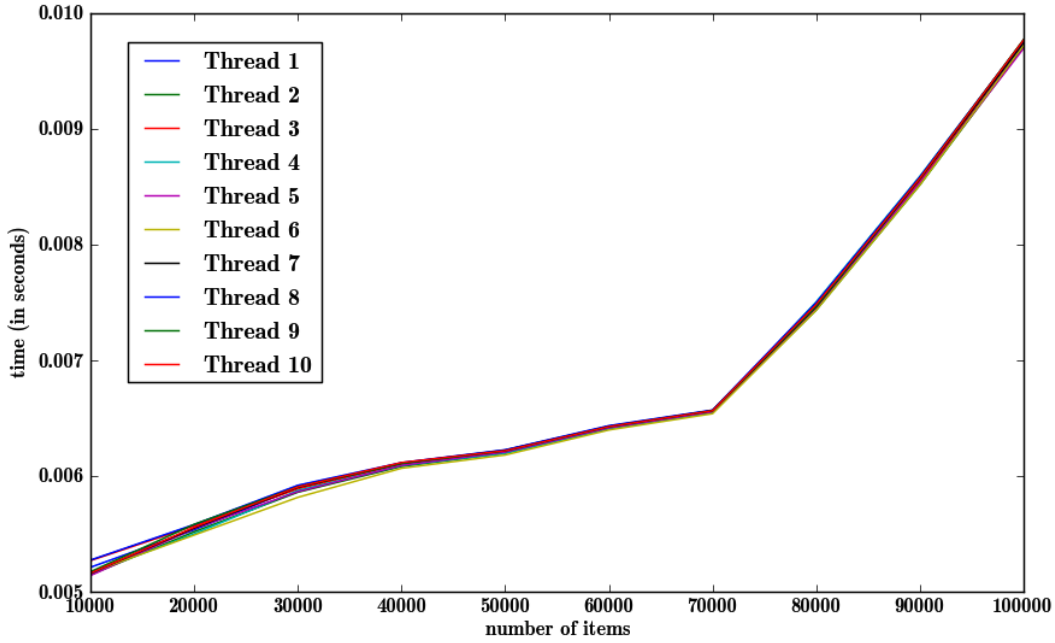


Figure 5.7: Sequential Inserts: Average Insertion Time, PostgreSQL

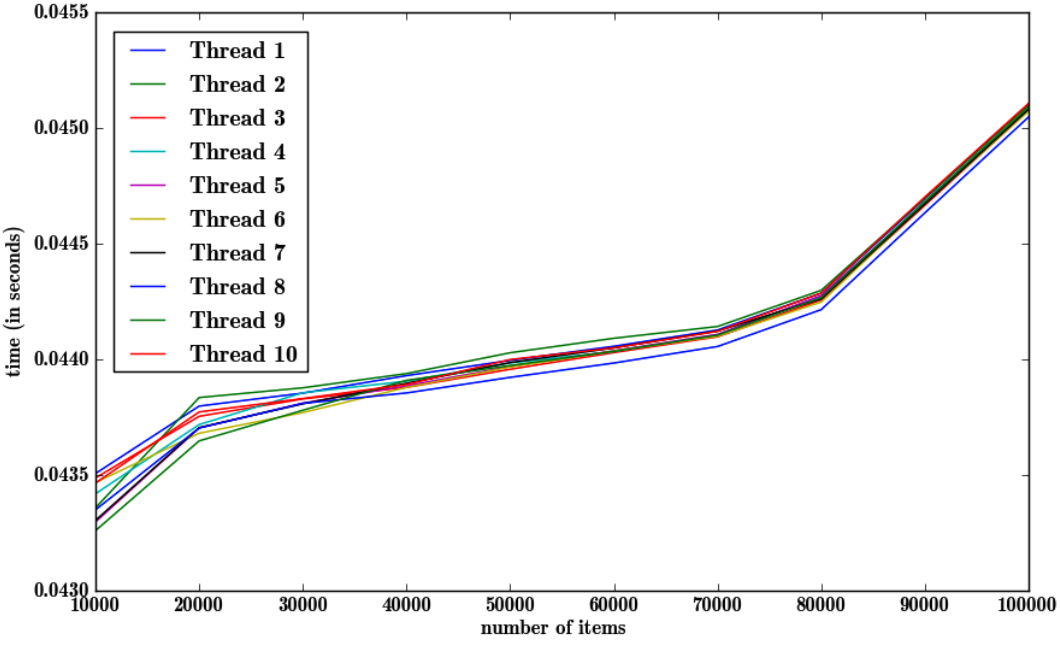


Figure 5.8: Sequential Inserts: Average Insertion Time, CouchDB

From these graphs we can further confirm that, even though both storage systems degrade, PostgreSQL's average insertion time degrades much quicker than CouchDB. For

example, in PostgreSQL the average time for inserting 100,000 items (10,000 in each thread) is about 0.0052 seconds (5.2 ms). For 1,000,000 items, the average is about 0.0098 seconds (9.8 ms), an **88%** increase. For CouchDB, the difference is between 0.0434 seconds (43.4 ms) and about 0.0451 seconds (45.1 ms), a mere **3.9%** increase.

5.1.2.4.2 Batch Inserts These tests measure the throughput (in number of items) of both storage systems when inserting multiple items at once. This is a facility offered by the XMPP PubSub protocol, and that must be supported by both storage systems. In the case of PostgreSQL, each item is inserted in a new Insert statement (it was attempted to insert Items in bulk, using PostgreSQL's features to insert multiple Items in a single statement; however, it proved to be much slower than using a single insert statement per Item); CouchDB provides bulk insertion facilities, and all the documents are included in the HTTP Post request.

Ten independent runs were performed for each number of items to be inserted, and the databases were in a clean state initially (no items were present); the mean of the duration of the insert operations was determined, and the throughput was calculated dividing the number of items by the mean duration.

Number of Items	PostgreSQL	CouchDB
10	2559.234	243.018
25	3144.118	3836.075
50	3422.016	4337.609
100	3868.056	4293.057
250	3966.683	4526.502
500	4164.119	4546.418
1000	4619.965	4172.164
2500	4609.997	4052.886
5000	4591.912	3317.449
10,000	4504.505	3304.525
25,000	4288.102	2975.982
50,000	4208.922	2803.661
100,000	4301.277	2349.015

Table 5.7: PostgreSQL vs CouchDB: Batch Insertion Throughput (in items per second)

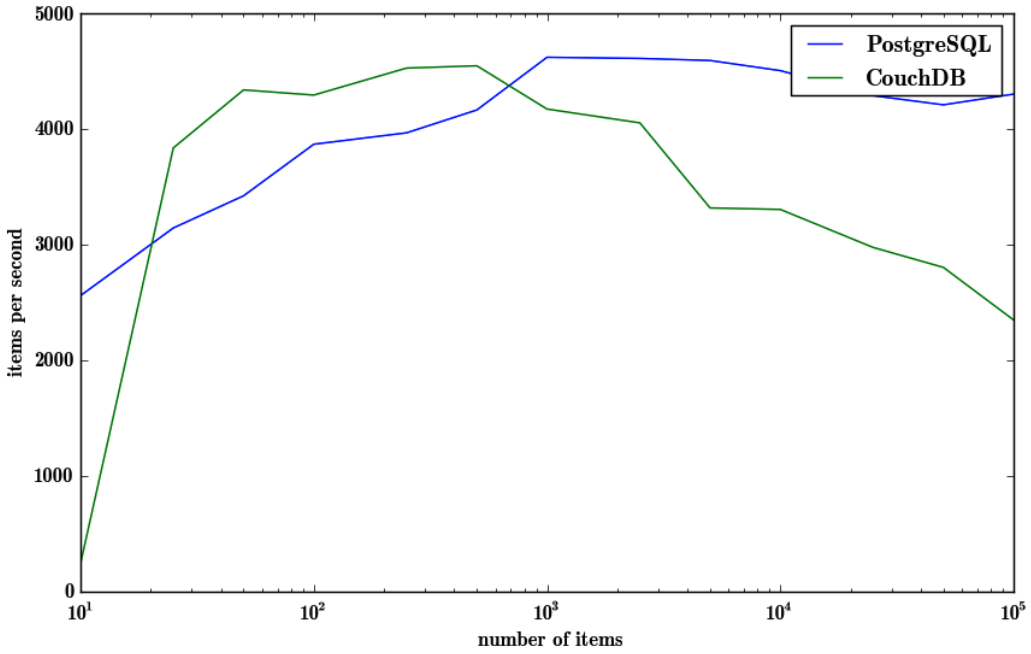


Figure 5.9: PostgreSQL vs CouchDB: Average Batch Insertion Speeds (in items per second)

The first thing we can observe by these numbers is that initially CouchDB is limited by the throughput limits also observed in the sequential inserts test (5.1.2.4.1), with a minimum throughput of approximately 243 items per second, caused by the REST HTTP API request / response delays.

CouchDB has a greater throughput than PostgreSQL for numbers of items somewhere between 25 and 500, but performance degrades rather quickly after 1000 items. PostgreSQL however shows a consistent increase in performance, with more throughput for larger numbers of items inserted. CouchDB has a peak performance of about 4500 Items per second for numbers of Items between 250 and 500. PostgreSQL however shows a peak of about 4,600 Items per second somewhere between 1,000 and 2,500 Items.

5.1.2.5 Conclusions

As observed in both the sequential insertion (5.1.2.4.1) and the batch insertion tests (5.1.2.4.2), database insertion is not a particular strong point of CouchDB, and PostgreSQL wins in virtually all measurements. For 1,000,000 sequential Item insertion, PostgreSQL wins with an average of 893.946 Items per second, against CouchDB's 217.512 Items per second; for batch insertions between 25 and around 1,000 Items CouchDB has greater throughput (above 4,000 Items per second) than PostgreSQL, with PostgreSQL winning for batches of 1,000 Items or more.

It should be noted, however, that even though PostgreSQL shows a much higher throughput than CouchDB in sequential inserts (**311%** higher in the case of 1,000,000 items), CouchDB shows a much smoother performance degradation with the increase

of numbers of items inserted than PostgreSQL (**3.9%** performance degradation between 100,000 and 1,000,000 items, compared to PostgreSQL's **88%**).

5.1.3 PostgreSQL vs CouchDB: Item Retrieval

Item retrieval is a very important part of the XMPP PubSub protocol. Although it is not needed for the regular publish-subscribe use cases, it may be necessary to retrieve context information on an on-demand basis. This context information retrieval must indicate the node on which the context information was published. These tests measure access times of both storage systems (PostgreSQL and CouchDB), as well as what impact the number of existent Items and Nodes has on these access times.

For this, 10 Nodes were created, and Items were distributed evenly between the Nodes; the tests measure the retrieval of all Items belonging to a single node. In every retrieval operation, the number of retrieved Items was always limited to 100, simulating a retrieval of the most recent 100 Items; this test focuses on Item retrieval performance degradation when the number of Items in the database increases, not on which storage system can retrieve the most data, as most Item retrieval operations usually request a small number of data Items.

5.1.3.1 Items Database Structure

Both PostgreSQL's database structure and CouchDB's Item document format are identical to the ones in the previous test 5.1.2.1; also like the previous test, a single index was created in the primary key "item_id" field:

Field	**Type**	**Extras**
item_id	int	primary key, index
node_id	int	not null
item	text	not null
publisher	text	not null
data	text	not null
date	timestamp	not null

Table 5.8: PostgreSQL vs CouchDB Item Retrieval: Items Data Structure

```
{
    "_id": "<item ID>"
    "doc_type": "item",
    "node": <node name>,
    "publisher": <publisher Jabber ID>,
    "date": <date and time>,
    "data": <context xml data>
}
```

Figure 5.10: PostgreSQL vs CouchDB Item Retrieval: CouchDB Item document structure

5.1.3.2 Test Environment

CPU	Intel(R) Core(TM)2 Duo CPU E8335 @ 2.66GHz (2 cores)
Memory	512 MB
Operating System	Ubuntu 10.04 LTS VMware virtual machine
Databases	CouchDB 1.0.1, PostgreSQL 8.4.4
Disk	30GB Hard Drive

Table 5.9: PostgreSQL vs CouchDB Item Retrieval: Test Environment

5.1.3.3 Dataset

These tests measure the access times of Item retrievals for several different numbers of pre-existing Items and Nodes. Although the number of inserted Items vary, the number of Nodes remains constant throughout these tests: 10 Nodes. Items are then evenly distributed through these 10 Nodes, so each Node contains exactly $\frac{N}{10}$ Items, being N the total number of Items. So for every retrieval operation, $\frac{N}{10}$

Test were made for the following Node / Item configurations:

Nodes	Items	Items per Node
10	10,000	1,000
10	50,000	5,000
10	100,000	10,000
10	200,000	20,000
10	500,000	50,000
10	1,000,000	100,000

Table 5.10: PostgreSQL vs CouchDB Item Retrieval: Dataset

5.1.3.4 Tests

The following table presents the mean access times for retrieving a list of all Items on a Node. As previously noted, there are always 10 Nodes, and the Items are evenly distributed, so for a number of N Items, each Node always has $\frac{N}{10}$ Items.

For each number of Items, 100 retrieval operations were measured, each to a random Node. The number of Items retrieved was *always* limited to 100 Items, as this test focuses on access times according to the size of the database, not on the size of the returned Items.

For CouchDB, a View was used, "items_by_node_date" (as described in Iteration 2 4.5), to return all Items for a given node, ordered by date. This is one of the required operations by the XMPP PubSub protocol, which is the reason that this test focuses on it.

The mean retrieval times are presented bellow.

Items	Items per Node	PostgreSQL	CouchDB
10,000	1,000	30.363 ± 0.897 ms	59.596 ± 17.500 ms
50,000	5,000	75.026 ± 2.786 ms	64.499 ± 17.093 ms
100,000	10,000	131.956 ± 5.523 ms	67.849 ± 23.113 ms
200,000	20,000	237.061 ± 6.702 ms	65.457 ± 16.829 ms
500,000	50,000	572.548 ± 14.324 ms	75.283 ± 27.178 ms
1,000,000	100,000	1133.610 ± 21.374 ms	78.465 ± 28.916 ms

Table 5.11: PostgreSQL vs CouchDB Item Retrieval: Test Results

And the evolution of the degradation of the access times of both storage systems (using a logarithmic scale):

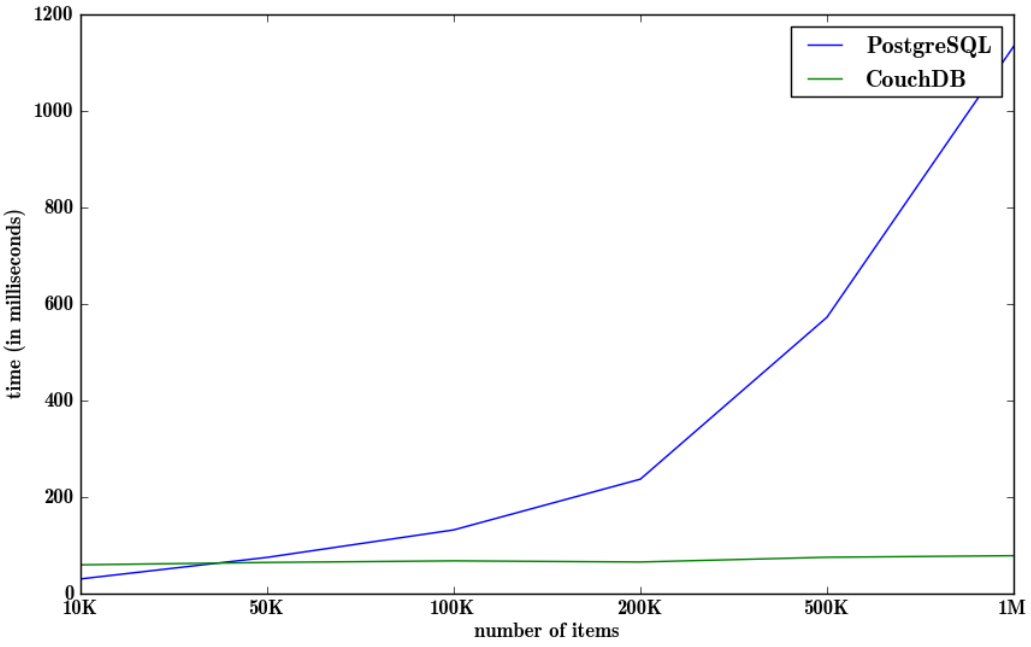


Figure 5.11: PostgreSQL vs CouchDB Item Retrieval: Average Access Times (in milliseconds)

This graph clearly indicates that even though PostgreSQL starts with a lower latency than CouchDB (for low numbers of Items, lower than 50,000), PostgreSQL's access times degrade considerably with the increase of the number of Items present in the database. It should be noted, however, that although CouchDB's graph line appears to be completely horizontal, its access times are not constant, and access times for 1,000,000 Items are **24%** slower than for 10,000 Items, although the latencies are still very low (less than 80 ms); PostgreSQL, however, sees a **97%** slowdown for the same increase of Items, increasing to more than 1 second.

It is also visible that CouchDB, as noted in the previous performance tests' conclusions (5.1.2.4.1 and 5.1.2.4.2), has a lower-bound performance limitation, possibly also due to

its HTTP REST API; its API access needs the creation and exchange of an HTTP request / response pair, which may keep it from performing better for low numbers of Items.

5.1.3.5 Conclusions

For Item retrieval operations, CouchDB provides much performance advantages when compared to PostgreSQL, as was expected from a NoSQL storage system, and is a natural choice for the XCoA platform, and the XMPP Publish-Subscribe component in particular. With a dataset of 1,000,000 Items, CouchDB shows an average retrieval time of about 78.465 ms, against PostgreSQL's average of over one second. Also, with an increase from 10,000 to 1,000,000 Items, CouchDB showed a mere **24%** degradation of access times to less than 80 ms, versus a **97%** degradation observed in PostgreSQL to more than one second.

5.1.4 PostgreSQL vs CouchDB: Item Search

This set of tests measure the performance of both storage systems (PostgreSQL and CouchDB) when searching for specific search strings inside context information, for different numbers of published Items. The stored Items have a common XML structure, but each one has a unique string inside; the time it takes to retrieve the Items that match the particular search query is then measured.

Tests were made for 10,000, 100,000, 200,000, 500,000 and 1,000,000 Items. For every test 100 iterations were executed, and the mean and standard deviation times were measured.

5.1.4.1 Items Database Structure

Both PostgreSQL's database structure and CouchDB's Item document format are identical to the ones in both the previous tests (5.1.2.1 and 5.1.3.1):

Field	**Type**	**Extras**
item_id	int	primary key, index
node_id	int	not null
item	text	not null
publisher	text	not null
data	text	not null
date	timestamp	not null

Table 5.12: PostgreSQL vs CouchDB Item Search: Items Table Structure

```
{
    "_id": "<item ID>"
    "doc_type": "item",
    "node": <node name>,
    "publisher": <publisher Jabber ID>,
    "date": <date and time>,
    "data": <context xml data>
}
```

Figure 5.12: PostgreSQL vs CouchDB Item Search: CouchDB Item document structure

Full-text searching in CouchDB is supported through Apache Lucene [61] and the couchdb-lucene plugin [63]. The index javascript function, which indicates which field to be indexed by Lucene, is identical to the one used by Idavoll It indexes the "data" field, as well as the "publisher", "node" and "date" fields; only the "data" field is relevant for this test, as it contains the XML-formatted context information. The indexing function is the following:

```
function(doc)
{
    var ret=new Document();
    ret.add(doc.node, {'field': 'node'});
    ret.add(doc.publisher, {'field': 'publisher'});
    ret.add(new Date(doc.date.substring(0, 19)), {'type': 'date', '
       field': 'date'});
    ret.add(doc.data, {'field': 'context'});
    return ret;
}
```

Figure 5.13: PostgreSQL vs CouchDB Item Search: Apache Lucene indexing function

A single index was created in the primary key "item_id" field on the PostgreSQL table. Furthermore, and since the requirements for Idavoll include at least version 8.4 of PostgreSQL, a GIN (Generalized Inverted Index) Index was created on PostgreSQL's Items table, on the "data" field, to optimize search queries [74] [75]. The full-text searching feature was introduced in PostgreSQL 8.3 [76]. Tests were made using both an indexed and non-indexed PostgreSQL table, as well as with CouchDB / Apache Lucene.

5.1.4.2 Test Environment

CPU	Intel(R) Core(TM)2 Duo CPU E8335 @ 2.66GHz (2 cores)
Memory	512 MB
Operating System	Ubuntu 10.04 LTS VMware virtual machine
Databases	CouchDB 1.0.1, PostgreSQL 8.4.4
Disk	30GB Hard Drive

Table 5.13: PostgreSQL vs CouchDB Item Search: Test Environment

5.1.4.3 Dataset

The XML context information structure of the published Items is very small and simple. Every inserted Item is numbered from 1 to N, being N the number of Items published, and the resulting XML is the following:

```
<context>
    <item>item_(*)</item>
</context>
```

The "(*)" corresponds to the number of the Item.

5.1.4.4 Tests

As previously said, tests were made for 10,000, 50,000, 100,000, 200,000, 500,000 and 1,000,000 Items, and 100 iterations for each. As every Item is numbered and the search query to be made is the Item's assigned number, on every iteration the search query is randomized, to make sure every query iteration searches for a different Item.

Searching times were measured for CouchDB and PostgreSQL, both with and without the context data field indexed. For CouchDB, Apache Lucene was used, through the couchdb-lucene plugin [63]; the time results are the ones returned in the resulting response sent by couchdb-lucene. The timings are split into "search_time", which is the time it took to search the documents index, and "fetch_time", which is the time it took to fetch the resulting documents. The timings represented in CouchDB are the sum of these two values (search and document fetch times), as in a real deployed scenario the documents resulting from the search query are also required, and should be fetched.

The results are shown below (all times are in milliseconds):

Items	PostgreSQL without index	PostgreSQL with index	CouchDB / Lucene
10,000	189.316 ± 4.388 ms	2.109 ± 0.114 ms	4.050 ± 1.935 ms
50,000	744.296 ± 14.586 ms	9.608 ± 0.541 ms	4.570 ± 1.779 ms
100,000	1472.714 ± 103.841 ms	19.132 ± 1.111 ms	4.940 ± 2.444 ms
200,000	2955.029 ± 286.949 ms	37.181 ± 2.149 ms	6.050 ± 3.314 ms
500,000	7064.893 ± 111.082 ms	92.139 ± 5.597 ms	5.870 ± 2.759 ms
1,000,000	14175.919 ± 416.168 ms	181.267 ± 11.141 ms	12.260 ± 7.882 ms

Table 5.14: PostgreSQL vs CouchDB Item Search: Test Results

From this table we can see that unindexed PostgreSQL has prohibitive performance, as searching a dataset with 1,000,000 Items takes on average 14.175 seconds. The following graph shows the values of the previous table, with both indexed and unindexed PostgreSQL, and CouchDB / Lucene:

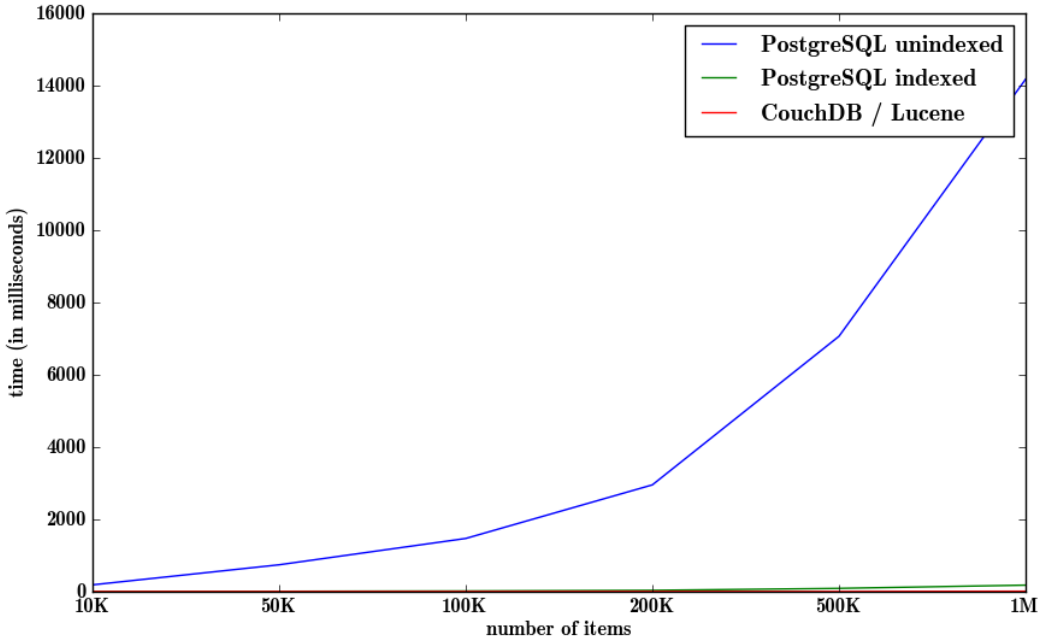


Figure 5.14: PostgreSQL vs CouchDB Item Search: Average Searching Times (indexed, unindexed PostgreSQL and CouchDB / Lucene)

As the values for indexed PostgreSQL and CouchDB / Lucene are insignificant when compared with unindexed PostgreSQL, the following graph compares only indexed PostgreSQL and CouchDB / Lucene:

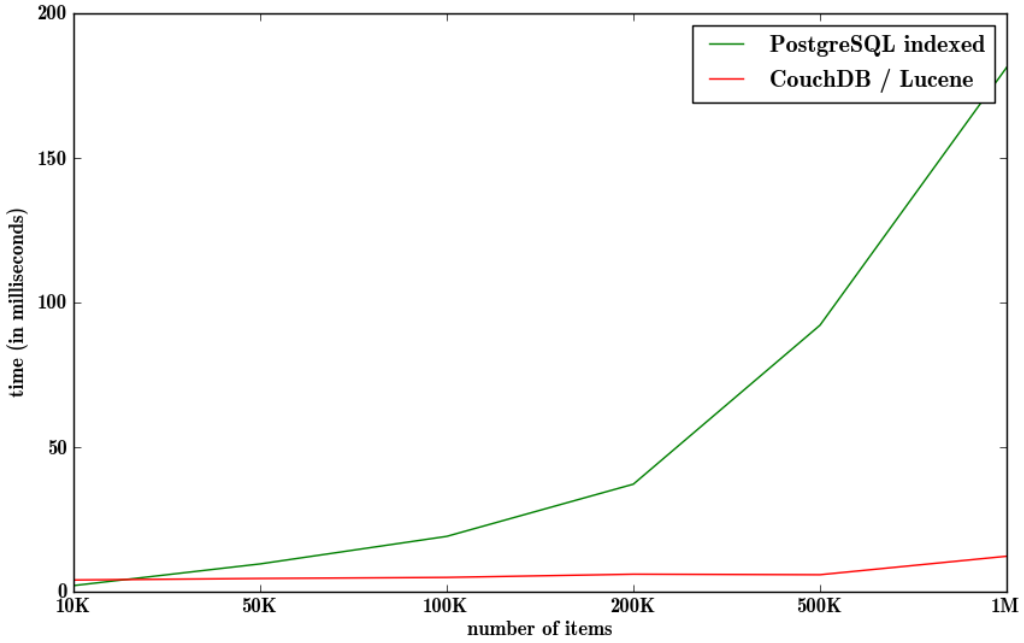


Figure 5.15: PostgreSQL vs CouchDB Item Search: Average Searching Times (indexed PostgreSQL and CouchDB / Lucene)

This last graph shows that even though indexed PostgreSQL shows acceptable performance degradation, Apache Lucene integrated with CouchDB shows much better performance, and very little performance degradation with the increase of Items in the database. It should also be noted that PostgreSQL's performance for 10,000 Items is somewhat better than CouchDB / Lucene, even if the difference is very small; however, its performance quickly degrades, contrary to CouchDB / Lucene.

5.1.4.5 Conclusions

Performance for an unindexed PostgreSQL database is very poor and even prohibitive, with a search in a 1,000,000-Item database taking over 14 seconds on average. With the context-information column indexed in PostgreSQL, performance is greatly increased, but it's still no match for the CouchDB / Apache Lucene combination. Performance between searching a database with 10,000 Items and 1,000,000 Items degrades by over 8494% in indexed PostgreSQL, with a maximum of 181 ms on average, well within reasonable bounds; however, in CouchDB this degradation is only 202%, with a maximum of over 12 ms on average, a clearly better solution. Apache Lucene, in integration with CouchDB, is the better and most scalable solution.

5.2 Final Notes

This section detailed the performance considerations for the most important operations in the chosen storage system, for use in a context management platform. These

operations are Item insertion 5.1.2, Item retrieval 5.1.3 and Item searching 5.1.4. Item deletion is not supported, as context information is published but never deleted; Item updating is not supported as well, as context information always refers to a specific timeframe, and subsequent updates of context generate new Items instead of updating existing ones.

For most operations, CouchDB showed much better performance and scalability than PostgreSQL. PostgreSQL showed a small performance advantage on Item insertion, both with sequential insert operations and batch insert operations.

For 1,000,000 sequential inserts, PostgreSQL wins with an average of about 893.946 Items inserted per second, against CouchDB's throughput of 217.512 Items per second.

For batch inserts, CouchDB wins with batches of 25 to about 1,000 Items, with an average of about 4,000 Items per second; PostgreSQL shows better performance for batches of 1,000 Items or more, with a worst-case measurement of 2349.015 Items per second for batches of 100,000 Items.

However, despite showing lower performance on Insertions, CouchDB shows much smoother degradation, with only a 3.9% degradation between inserting 100,000 and 1,000,000 Items; PostgreSQL, for the same increase in Items, showed an 88% performance degradation.

On Item retrievals, CouchDB is a clear winner, with an average of about 78.465 ms retrieval time for a database with 1,000,000 Items; on the same dataset, PostgreSQL shows an average of over a second. With the increase from 10,000 to 1,000,000 Items, CouchDB shows a mere 24% performance degradation, versus PostgreSQL's 97% degradation.

On Item searching CouchDB's performance advantages are even clearer, both against unindexed and indexed PostgreSQL databases. For a 1,000,000 Items dataset, CouchDB shows an average searching time of about 12.260 ms, against indexed PostgreSQL's searching times of about 181.267 ms and unindexed PostgreSQL's 14175.919 ms (over 14 seconds).

CouchDB is the clear performance winner, losing only marginally in Item insertion, although still showing acceptable performance. Large sequential insertion operations are also pretty rare, considering that Item insertions happen through XMPP's Publish-Subscribe publishing mechanisms, with the unavoidable message exchanging delays, so the storage system should not be the bottleneck. In extreme and unlikely cases that it is indeed the bottleneck (1,000,000 simultaneous Item insertions), it still shows an average insertion time bellow 45 ms, that although is worse than PostgreSQL, is still well within reasonable bounds.

Chapter 6

Conclusion

The use of a NoSQL solution, particularly CouchDB, as a storage system in an XMPP-based Context Management Platform achieved all the proposed objectives. First of all, it didn't present any disadvantages regarding a traditional relational database, which is usually the storage system of choice of most XMPP servers, and is the case with the XMPP-based Context Architecture (XCoA) proposed by D. Gomes et. al [6]. Instead of using the proposed XMPP server (OpenFire), this proposal required a new XMPP Publish-Subscribe external component (Idavoll 4.3), which fully integrates with OpenFire. This external component allows the choice of either a PostgreSQL storage system, similar to the one present in OpenFire, or a newly-developed PostgreSQL + CouchDB hybrid 4.8.2. This exchange of storage systems is transparent to the XMPP server, and requires only a restart of the XMPP PubSub component, provided the correct configuration files are present.

As mentioned, all required functionalities of XCoA are guaranteed by this proposal, and were implemented in Idavoll when not present. Besides guaranteeing all required functionalities, several new functionalities are now present, each with its advantages and disadvantages.

6.1 Scalability, Availability and Reliability

Through the replication and sharding functionalities provided by CouchDB (and third-parties in the case of sharding), the context information database, composed of XMPP PubSub Items, is now fully distributable, and offers horizontal scalability, higher availability and reliability guarantees, depending on the deployment options (described in the Architectures sections 3.4.1, 3.4.2, 3.4.3, 3.4.4 and 3.4.5).

Architectures which use only one CouchDB node (Architectures 1 3.4.1 and 2 3.4.2), or a single CouchDB node with a replicated node for searching and backup (Architecture 3 3.4.3), offer no horizontal scalability, although a replicated CouchDB node offers higher reliability and availability, as the replicated node is used in case the main node fails.

Architectures 4 (3.4.4) and 5 (3.4.5), which use CouchDB clusters of full-snapshots replicas and partitioned data respectively, offer both horizontal scalability, higher availability and reliability, and provide load-balancing between the several nodes. When using full-snapshot replicas, only internal CouchDB facilities are needed, as CouchDB replicates

data to all its nodes automatically, and as its API is a REST HTTP one, a single HTTP load-balancing proxy is enough to balance the load through all nodes. However, all nodes contain a full database snapshot, which requires additional disk-space guarantees. When using partitioned data (Architecture 5), data is evenly partitioned through all nodes, and each partition is replicated through one or more nodes, which requires less disk-space; however, CouchDB does not currently provide data partitioning facilities, and a third-party proxy solution must be used to partition the data and store it in the correct node.

This last architecture, Architecture 5, with data-partitioning, is the optimal solution, as load balancing is provided as a side-effect of the data partitioning, provided that data is evenly partitioned through all nodes. These solutions usually partition data by applying a consistent hash function to documents' IDs, evenly distributing documents through all nodes. Moreover, with the exception of the Lucene Node, no single CouchDB node contains a full snapshot of the database, so less disk space is required, and high availability is still guaranteed.

6.2 Searching

Full-text searching functionalities are guaranteed through the use of the Apache Lucene searching engine. An existing project exists to integrate CouchDB and Lucene, couchdb-lucene, developed by an Apache CouchDB contributor, while equivalent projects for relational databases, as well as for other popular NoSQL solutions such as Cassandra were not found (even though Cassandra is also an Apache project, no project exists that allow indexing of a Cassandra dataset). The existence of an integration plugin between CouchDB and Lucene, as well as they both being Apache projects, proved that the choice of CouchDB was the right one.

Searching is provided outside of the scope of the XMPP Publish-Subscribe protocol, directly in CouchDB's REST API. However, a web interface was also created to show information about the XMPP PubSub component and provide a searching interface to CouchDB 4.9.

It is recommended the usage of a separate replicated CouchDB node for the CouchDB / Lucene integration, so as to separate completely the XMPP PubSub protocol and the searching functionalities, considering that the XMPP PubSub is the main protocol of XCoA, and searching is instead a bonus functionality built on top of it. Moreover, some architectures even demand it, such as Architecture 5 with data partitioning. As no CouchDB node has a full database snapshot, it becomes necessary to have a separate node with a full snapshot of the database, to provide searching on the complete database.

6.3 Performance

The usage of a NoSQL storage system provided strong performance advantages, as previously detailed in the performance tests (5.1.2, 5.1.3 and 5.1.4), and properly summarized in 5.2. For large numbers of XMPP PubSub Items, CouchDB handles data much more efficiently than PostgreSQL, which is one of the storage systems used by OpenFire,

and the main storage system in Idavoll.

6.4 Common Pitfalls

Although CouchDB and NoSQL solutions provide several advantages over traditional relational database for certain use-cases, there are a number of common pitfalls that come with its use.

For example, the tendency to try and fit all data into a NoSQl solution is usually a bad idea. NoSQL mostly means Not-only-SQL [77], meaning that its meant to be used in conjunction with other relational "SQL" solutions, and not to provide a complete replacement. This thesis implementation suffered from this, as the first implementation 4.4 tried to fit all the XMPP PubSub data model into CouchDB. As its data model fit well in a relational model, this quickly proved to be a mistake, and the idea was abandoned with the subsequent iterations. A better approach is to, instead, find data items which may not fit into a relational model, which may not have relationships to other items, and which are in large numbers when compared to the remaining data items, and separate them in a NoSQL solution. This way, relational data remains in a relational database, and non-relational large-datasets are moved to special-purpose storage systems, depending on the requirements.

Web startup Linkfluence [78] used CouchDB internally, but moved away to Key/Value Store Riak [38], based on 3 problems they encountered: Scalability, internal single-file storage mechanism, and stability [79].

For the first problem, "scalability", they meant disk-scalability. They found that they update documents very often, which considering that CouchDB uses MVCC 2.3.2.8.2.2, generates a new version of a document on every update. Without proper occasional compacting operations, this makes CouchDB's disk-space requirements go up very quickly. However, in the context of this thesis, no document updates ever occur, so this shouldn't present a problem; if it does, a frequent database-compacting operation can be scheduled to run in low-usage periods of the day.

The second problem, the internal was CouchDB stores a database, which is a single-file, caused some problems when making backups. With a 2TB dataset, backing up the database on a daily basis proves a very difficult challenge indeed. However, CouchDB already provides mechanisms for incrementally backing up data, through database replication, and file-backup is not supported. Using replication as a backup mechanism, only the new inserted data relative to the last backup is replicated, which provides a richer and more efficient backup strategy.

The third problem, stability, is a very case-specific problem, and can't easily be confirmed nor denied. For this thesis, the XMPP PubSub component with the implemented PostgreSQL + CouchDB hybrid storage system was deployed and used extensively in the C3S testbed, without any stability problems observed.

6.5 NoSQL Ecosystem and Future Direction

The whole NoSQL ecosystem is an ever-evolving field, with new storage systems being developed every day. For example, near the completion of this theses a new document-oriented NoSQL solution was discovered, ThriftDB [80]. This NoSQL solution offers document-oriented storage with integrated search built-in, unlike current document-oriented solutions. However, presently only an API to a cloud-based hosting storage solution is offered, with a local storage option being developed, but not yet available.

Also, the NoSQL ecosystem, unlike relational databases, focuses on specialization, so this solutions are increasingly growing in different, very focused directions, leaving the door open for new players to emerge in the field, providing different features and solving different problems. This makes the NoSQL ecosystem an exciting and ever-evolving field, being difficult to assess in which direction the field is moving and how the existing and new, unforeseen problems will be tackled by these solutions.

Bibliography

[1] Facebook. Facebook Press Room: Statistics. `http://www.facebook.com/press/info.php?statistics`.

[2] Foursquare. Foursquare: About. `http://foursquare.com/about`.

[3] Terry Winograd. Architectures for Context. *Human-Computer Interaction*, 16(2):401–419, 2001.

[4] P Floreen, M Przybilski, P Nurmi, J Koolwaaij, A Tarlano, M Wagner, M Luther, F Bataille, M Boussard, B Mrohs, and Sianlun Lau. Towards a Context Management Framework for MobiLife. *Management*, pages 120–131, 2005.

[5] M Zafar, N Baker, B Moltchanov, J M Gonçalves, S Liaquat, and M Knappmeyer. Context Management Architecture for Future Internet Services. *Applied Sciences*, 2009.

[6] Diogo Gomes, João Gonçalves, Ricardo Santos, and Rui L Aguiar. XMPP based Context Management Architecture. In *IEEE Globecom*, 2010.

[7] B N Schilit and M M Theimer. Disseminating active map information to mobile hosts. *Ieee Network*, 8(5):22–32, 1994.

[8] Richard Hull, Philip Neaves, and James Bedford-Roberts. Towards Situated Computing. In *ISWC 97 Proceedings of the 1st IEEE International Symposium on Wearable Computers*, page 146. IEEE Computer Society, 1997.

[9] Anind K Dey and Gregory D Abowd. Towards a Better Understanding of Context and. In *Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, pages 304 – 307. Springer-Verlag London, UK, 1999.

[10] R Kernchen, David Bonnefoy, A Battestini, B Mrohs, M Wagner, and M Klemettinen. Context-Awareness in MobiLife. *Provider*, 2006.

[11] C-Cast. `http://www.ict-ccast.eu/`.

[12] Nigel Baker and et al. Madiha Zafa. C-Cast D12: Specification of Context Casting Service Enablers, Context Management Context Brokering. Technical report.

[13] The XMPP Standards Foundation. XMPP. `http://xmpp.org/`.

[14] David Reiss. Facebook Blog: Facebook Chat Now Available Everywhere. http://blog.facebook.com/blog.php?post=297991732130.

[15] Google. Google Talk for Developers. http://code.google.com/apis/talk/open_communications.html.

[16] Peter Millard, Peter Saint-Andre, and Ralph Meijer. XMPP XEP-0060: Publish-Subscribe. http://xmpp.org/extensions/xep-0060.html.

[17] Peter Saint-Andre, Ralph Meijer, and Brian Cully. XMPP XEP-0248: PubSub Collection Nodes. http://xmpp.org/extensions/xep-0248.html.

[18] E F Codd. Derivability, Redundancy and Consistency of Relations Stored in Large Data Banks, 1969.

[19] Vineet Gupta. NoSQL Databases Part 1 - Landscape. http://www.vineetgupta.com/2010/01/nosql-databases-part-1-landscape, 2010.

[20] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. *ACM SIGMOD Record*, 25(2):173–182, 1996.

[21] Brian Keating. Challenges Involved in Multimaster Replication. http://www.dbspecialists.com/files/presentations/mm_replication.html, 2001.

[22] PostgreSQL Global Development Group. PostgreSQL Wiki: Clustering. http://wiki.postgresql.org/wiki/Replication,_Clustering,_and_Connection_Pooling#Clustering.

[23] Kai Orend. *Analysis and Classification of NoSQL Databases and Evaluation of their Ability to Replace an Object-relational Persistence.* PhD thesis, Technische Univesität München, 2010.

[24] T Haerder and A Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, 1983.

[25] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40, 2009.

[26] Neal Leavitt. Will NoSQL Databases Live Up to Their Promise? *Computer*, 43(2):12–14, 2010.

[27] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable : A Distributed Storage System for Structured Data. *Sports Illustrated*, 26(2):205–218, 2006.

[28] G Decandia, D Hastorun, M Jampani, G Kakulapati, A Lakshman, A Pilchin, S Sivasubramanian, P Vosshall, and W Vogels. Dynamo: Amazon's Highly Available Key-value Store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007.

[29] Neo Technology. Neo4j. http://neo4j.org/.

[30] D P Reed. *Naming and Synchronization in a Decentralized Computer System*. PhD thesis, Massachusetts Institute of Technology, 1978.

[31] PostgreSQL Global Development Group. PostgreSQL: MVCC. `http://www.postgresql.org/docs/current/static/mvcc-intro.html`.

[32] Philip A Bernstein and Nathan Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, 13(2):185–221, 1981.

[33] Jeffrey Dean and Sanjay Ghemawat. Simplified Data Processing on Large Clusters. In *Proc Symposium on Operating System Design and Implementation*, 2004.

[34] The Apache Software Foundation. Hadoop: MapReduce. `http://hadoop.apache.org/mapreduce/`.

[35] Steve Chu. MemcacheDB. `http://memcachedb.org/`.

[36] Amazon. Amazon's SimpleDB. `http://aws.amazon.com/simpledb/`.

[37] Citrusbyte. Redis: an open source, BSD licensed, advanced key-value store. `http://redis.io`.

[38] Basho Technologies Inc. Riak. `http://wiki.basho.com/`.

[39] Salvatore Sanfilippo. Redis DB: Redis in the place of Tyrant. `http://groups.google.com/group/redis-db/msg/9f494d12275054a7`, 2011.

[40] Basho Technologies Inc. An introduction to Riak. `http://wiki.basho.com/An-Introduction-to-Riak.html`.

[41] The Apache Software Foundation. HBase. `http://hbase.apache.org/`.

[42] Jake Luciani. Solandra. `https://github.com/tjake/Solandra`.

[43] Avinash Lakshman. Cassandra – A structured storage system on a P2P Network. `http://www.facebook.com/note.php?note_id=24413138919&id=9445547199&index=9`, 2008.

[44] Avinash Lakshman and Prashant Malik. Cassandra - A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[45] Maxim Grinev. A Quick Introduction to the Cassandra Data Model. `http://maxgrinev.com/2010/07/09/a-quick-introduction-to-the-cassandra-data-model/`.

[46] The Apache Software Foundation. HBase Wiki: HBase Architecture. `http://wiki.apache.org/hadoop/Hbase/HbaseArchitecture`.

[47] 10gen. MongoDB. `http://www.mongodb.org/`.

[48] 10gen. MongoDB: Schema Design. `http://www.mongodb.org/display/DOCS/Schema+Design`.

[49] 10gen. MongoDB: Querying. http://www.mongodb.org/display/DOCS/Querying.

[50] 10gen. MongoDB vs CouchDB. http://www.mongodb.org/display/DOCS/Comparing+Mongo+DB+and+Couch+DB.

[51] 10gen. MongoDB: Replica Sets. http://www.mongodb.org/display/DOCS/Replica+Sets.

[52] 10gen. MongoDB: Durability and Repair. http://www.mongodb.org/display/DOCS/Durability+and+Repair.

[53] Ben Brown. How'd that Mongo get so fast?????? http://www.idiotsabound.com/howd-that-mongo-get-so-fast.

[54] Ben Brown. Did I mention #MongoDb is fast?!?! Way to go @mongodb. http://www.idiotsabound.com/did-i-mention-mongodb-is-fast-way-to-go-mongo.

[55] Joe Lennon. IBM developerWorks: Exploring CouchDB. http://www.ibm.com/developerworks/opensource/library/os-couchdb/index.html, 2009.

[56] CouchOne. CouchOne: About. http://www.couch.io/page/about.

[57] The Apache Software Foundation. CouchDB Documentation: Introduction. http://couchdb.apache.org/docs/intro.html.

[58] The Apache Software Foundation. CouchDB Documentation: Overview. http://couchdb.apache.org/docs/overview.html.

[59] The Apache Software Foundation. CouchDB: Introduction to CouchDB Views. http://wiki.apache.org/couchdb/Introduction_to_CouchDB_views.

[60] 10gen. 10gen. http://www.10gen.com/.

[61] The Apache Software Foundation. Apache Lucene. http://lucene.apache.org/java/docs/index.html.

[62] The Apache Software Foundation. CouchDB Wiki: CouchDB In The Wild. http://wiki.apache.org/couchdb/CouchDB_in_the_wild.

[63] Robert Newson. Robert Newson's couchdb-lucene. https://github.com/rnewson/couchdb-lucene.

[64] Elasticsearch. Elastic Search. http://www.elasticsearch.org/.

[65] The Apache Software Foundation. Apache Lucene: Powered By. http://wiki.apache.org/lucene-java/PoweredBy.

[66] Meebo.com. couchdb-lounge. http://tilgovi.github.com/couchdb-lounge/.

[67] J. Chris Anderson, Jan Lehnardt, and Noah Slater. CouchDB The Definitive Guide: Clustering. http://guide.couchdb.org/draft/clustering.html.

[68] Ralph Meijer. Ralph Meijer's Idavoll. https://github.com/ralphm/idavoll.

[69] Twisted. Twisted Wiki. http://twistedmatrix.com/trac/wiki.

[70] Joe Hildebrand, Peter Millard, Ryan Eatmon, and Peter Saint-Andre. XMPP XEP-0030: Service Discovery. http://xmpp.org/extensions/xep-0030.html.

[71] Joe Celko. *Trees and Hierarchies in SQL for Smarties*. Morgan Kaufmann, 2004.

[72] PostgreSQL Global Development Group. PostgreSQL 8.4: Release Notes. http://www.postgresql.org/docs/8.4/static/release-8-4.html.

[73] Quassnoi. Explain Extended - Adjacency list vs. nested sets: PostgreSQL. http://explainextended.com/2009/09/24/adjacency-list-vs-nested-sets-postgresql/.

[74] PostgreSQL Global Development Group. PostgreSQL 8.3 - Full Text Search: GiST and GIN Index Types. http://www.postgresql.org/docs/8.3/static/textsearch-indexes.html.

[75] PostgreSQL Global Development Group. PostgreSQL 8.3 - Full Text Search: Tables and Indexes. http://www.postgresql.org/docs/8.3/static/textsearch-tables.html.

[76] PostgreSQL Global Development Group. PostgreSQL 8.3: Release Notes. http://www.postgresql.org/docs/8.3/static/release-8-3.html.

[77] Jonathan Ellis. *Rackspace Blog: NoSQL Ecosystem*. PhD thesis.

[78] Linkfluence. Linkfluence. http://us.linkfluence.net/.

[79] Franck Cuny. Linkfluence: On moving from CouchDB to Riak. http://labs.linkfluence.net/nosql/2011/03/07/moving_from_couchdb_to_riak.html.

[80] ThriftDB. ThriftDB. http://www.thriftdb.com/.

Outros anexos só estão disponíveis para consulta através do CD-ROM.
Queira por favor dirigir-se ao balcão de atendimento da Biblioteca.

Serviços de Biblioteca, Informação Documental e Museologia
Universidade de Aveiro