FAKULTÄT FÜR !NFORMATIK

Data mining strategies in large–scale agent–based models with applications in econophysics

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Werner Bayer
Matrikelnummer 0225629

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: Univ.–Prof.Mag.DDr. Stefan Thurner, Medizinische Universität Wien

Wien, 16. 12. 2009

(Unterschrift Verfasser/in)

(Unterschrift Betreuer/in)

Werner Bayer
Ludwig Hinnerthstrasse 5
3021 Pressbaum

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit — einschließlich Tabellen, Karten und Abbildungen —, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Pressbaum, 16. Dezember 2009 _____

(Unterschrift)

Kurzfassung

Agentenbasierte Modelle haben sich als unschätzbares Werkzeug in einer breiten Palette an Forschungsfeldern erwiesen — trotz der einhergehenden Nachteile von oft komplexer und modellabhängiger Programmierung der Agenten. Die Verwendung eines Massive Multiplayer Online Games (MMOG) mit Menschen als Teilnehmern anstelle von programmierten Agenten beseitigt viele dieser Nachteile und gibt Forschern zusätzlich die Gewissheit, dass sie echtes menschliches Verhalten studieren.

Das MMOG *Pardus* generiert täglich Millionen an Datensätzen und zeichnet dabei jede relevante Aktion eines jeden Spielers auf. Diese Aufzeichnungen sind in Form von täglichen Datenbank Sicherungen verfügbar. Es muss ein System konzipiert und implementiert werden, das alle Daten anonymisiert und vereint. Forscher müssen umfassende Abfragen auf den gesamten Zeitraum der gesammelten Daten tätigen können.

Data Mining Strategien werden evaluiert und in mehreren Bereichen wie Speicher Effizienz, Sicherheit, Geschwindigkeit und Fähigkeit zur simultanen Benutzung verglichen. Schließlich wird ein relationales Datenbank Management System (RDBMS) aufgrund seiner Überlegenheit in allen Gebieten als Daten Back–End verwendet. Ein Datenbank Design wird so entworfen, dass es den enormen Fluss an Daten aufnehmen kann. Eine Applikation bestehend aus einem Kommandozeilen Programm und einem Web Front–End wird implementiert. Das Kommandozeilen Programm enthält Funktionen zur Extrahierung relevanter Daten der Sicherungen, zur Anonymisierung und zur Integration in das neue System. Das Web Front–End kann sowohl von Forschern für den Zugriff zu den Daten im System, als auch von Administratoren verwendet werden um neue Sicherungen zu importieren, Sicherungen vom System selbst anzulegen oder zur automatischen Ausführung zu planen.

Eine erste Verwendung der Applikation zeigt vielversprechende Resultate, die namhafte sozialwissenschaftliche Hypothesen erfüllen, und folglich auch die Realisierbarkeit von Data Mining in menschlichen agentenbasierten Modellen in großem Umfang wie in MMOGs untermauern.

Abstract

Agent–based models have proven to be a unique and valuable tool in a wide range of research areas, despite the inherent drawbacks of complex, large–scale, and often model–specific programming of agents. The use of a massive multiplayer online game (MMOG), with real human participants in place of programmed agents, eliminates many of those drawbacks and allows researchers to confidently read data as accurately reflecting genuine human behavior.

The MMOG *Pardus* generates millions of data sets each day, logging each and every relevant user action including various social and economic engagements. These data sets are available in form of daily database backups. A system must be designed and implemented to combine and anonymize all data while keeping all correlations intact. Researchers need extensive querying capabilities on any and all timeframes of the data.

Data mining strategies are evaluated and compared in regards to storage size, practicality, security, speed and ability for concurrent usage. Eventually a relational database management system (RDBMS) is used as data storage back–end due to its superiority in all areas. Subsequently a database layout is designed to accommodate the huge input of data. An application consisting of a command–line tool and a web front–end is implemented. The command–line tool contains functions for extracting the data from backups, anonymizing it and integrating it into the new system. The web front–end is both used by researchers accessing the data, and by administrators creating schedules for or manually exporting and importing data.

First usage of the application shows promising results, supporting famous social science hypotheses and thus confirming the feasibility of data mining in large–scale human agent–based models like MMOGs.

Contents

Glossary

Notation	Description
ACID	Atomicity Consistency Isolation Durability. Ensures transactions in a database are atomic, consistent, isolated and durable. 27, 28, 30, 31, 36
ANSI	American National Standards Institute. Oversees standardization in the U.S. and is a member of the ISO. 29, 30
class diagram	Visualizes a system's classes, their attributes and methods as well as the relations between them. 13
CLI	Command–Line Interface. In contrast to a graphical user interface, commands are given by textual input only in a command–line interface. 37, 57, 62, 64
Crow's Foot	This notation is used in Entity-Relationship Models. The ''to–many'' end of a relation resembles a crow's foot. 32
cryptographic hash function	Outputs a string of fixed size from an input of variable length. To classify as cryptographic hash function among others there must be no easy way to find the input data from the returned string nor to find multiple input values resulting in the same output. 48
CSV	Comma Separated Values. A text file format storing data values separated by commas. 24, 55, 62
deployment diagram	Displays physical pieces of a model, ''Artifacts'', attached to hardware ''nodes''. 11

Notation	Description
ERD	Entity–Relationship Diagram. A diagram visualizing database structures created through entity-relationship modeling, a software engineering. 6, 32, 53
FCGI	Fast Common Gateway Interface. An interface usually connecting a web server with external programs. Contrary to CGI, FastCGI handles several requests over a single process eliminating the overhead for creating new processes. 37, 52
GPL	GNU General Public License. License dedicated to promoting free software. 30
GUI	Graphical User Interface. A graphical user interface offers visual methods to control a program. 37
HTML	Hyper Text Markup Language. The standard markup language for web documents. 11, 64
index	Used in database systems to store the byte offset at which to find a record. 26–29
ISO	International Organization for Standardization. A non-governmental organization setting international standards. 29
MD5	Message-Digest algorithm 5. A cryptographic hash function that returns a 128–bit string. 36, 48, 53
MVC	Model View Controller. A pattern in software engineering to isolate the presentation layer from business logic. 52
OODBMS	Object–Oriented Database Management System. An OODBMS stores objects persistently. 28, 29
OQL	Object Query Language. A language to query object database systems. 28

Notation	Description
package diagram	Gives an overview of packages and their dependencies. 37
PHP	Hypertext Preprocessor. A scripting language supporting the object-oriented programming paradigm. 11, 37, 46
RAID 1	A redundant array of inexpensive disks mirroring all content in real time. 21
RDBMS	Relational Database Management System. An RDBMS stores data in form of tables consisting of rows and columns. ii, 29–31, 69
salt	In cryptography a salt is a value that is used as additional input for a cryptographic hash function. This measure substantially complicates dictionary attacks. 48
serializable	A serializable mode of transaction isolation ensures that transactions cannot affect each other, as if they were executed after each other. This is done by range–locking queries, even SELECTs, on the WHERE clause. 52
SQL	Structured Query Language. A language to query relational database systems. 24, 27–31, 45–47, 49–53, 55, 60, 62
SQL injection	Malicious SQL queries can be executed when user input is not correctly escaped. 41, 55
SQL/PSM	SQL/Persistent Stored Modules. Standard to add procedural programming features to SQL. 29
surrogate key	A primary key in a database table that is not related to any user data — visible or invisible to the user. 36
UML	Unified Modeling Language. A graphical modeling language to create visual software engineering models. 11

Notation	Description
use case diagram	Shows how a task is executed in terms of actors controlling processes. 15
XML	Extensible Markup Language. A markup language defining rules of how to encode information in text files. 24, 26–28, 55, 62

List of Figures

List of Tables

Listings

Chapter 1.

Introduction

Agent–based Models

The use of agent–based models has been growing at a phenomenal rate since the 1990s in nearly every field of study. The potential offered by agent–based models for both learning and extracting data with practical, real–world use has already been demonstrated by researchers around the globe [Buc09], in everything from studying the spread of pathogens to social sciences to stock market analysis.

Concepts

An agent–based model is an evolving system which contains a number of autonomous agents in a virtual environment. The model's outcome depends on the decisions, interactions and other activities of the agents within. Though they share some similarities, agent–based models differ from the more traditional particle systems in that agents may dynamically learn and modify their behavior through interactions with their environment and each other.

In an agent–based model, an agent is an entity that behaves in a certain way based on a set of assumptions or rules the agent has been given. The rules dictating an agent's behavior are typically defined with the creation of the model and may range from the simplistic to the very complex depending on purpose of the model. In models focusing on sentient entities, such as human beings, agents may be designed to act with self–interest in mind; for example the need for food or evasion of danger, or the desire for socialization or the accumulation of wealth [AAS].

Agent–based models offer a means to simulate large–scale, complex phenomena, such as financial markets, while agents within the model make only minimal assumptions or decisions on a small, local scale. This process of emergence demonstrates how very small, relatively simplistic decisions on the part of individual agents can generate a widespread, complex system [Bon02].

Unlike traditional analytical methods, which only characterize elements of a system or scenario — such as equilibrium — agent–based models can actually produce those elements, or fail to do so if the system does not compel the agents to act in a way that produces them [Art05]. Agent interactions and decisions can be tracked so it is possible to see when, how and why changes or events took place. Modelers can then explore alternative outcomes by introducing new or different elements to the scenario.

Difficulties

Despite their usefulness in many applications, standard agent–based models do present some obstacles.

Agents rely on the pre–defined set of behaviors granted to the agents to produce output. If the behavior is modeled to mimic human behavior, for example, the results of an agent–based model scenario will only accurately reflect similar circumstances in the real world if the agents act with similar variety, complexities and idiosyncrasies [dSGL09].

Another difficulty is the actual programming of the agents; whether agent behavior is being modified or is being written from scratch, this process can be lengthy and arduous depending on the complexity of the agents [Ric05]. In most situations in which using agent–based models is beneficial, be it for business usage or pure research, time and technical difficulties often add undesirable, inefficient and costly elements to any project.

A related issue is in programming itself; as with any human endeavor, it is prone to error. Programming sophisticated agent behavior quickly becomes complex and ''bugs'' are almost a certainty, some of which may be nearly undetectable but can considerably bias the evolution of events within the model [Ehr02]. Of course bugs are a possibility in any program, simple or not, but as the complexity grows so does the likelihood of encountering bugs and the difficulty of removing them.

Replication of human behavior, such as one might need in a large–scale economic model, requires a wide variety of very sophisticated agents. Humans make decisions based on a

large number of factors; education, access to and understanding of available information and social networking just to name a few [KN01]. While this has been done with some success, the difficulties in replicating human behavior are an ongoing issue [FF09].

Agent–based Models as Socio–economic Systems

A practical solution for the previously mentioned drawbacks of the agent–based model is replacing the computerized agents with real human beings. Instead of making assumptions about likely responses in agents, the variety, complexity and occasional irrationality present in real-world human decision making would automatically be represented [Bon02]. Results of the model could be viewed confidently as accurately reflecting the behavior of real human beings.

Real humans in place of agents would also considerably reduce time spent working on the model and, as a likely result, cost of the project. Both creating and modifying the the model would require attention only to the environment, completely eliminating a considerable amount effort and time put into the project.

A significant reduction of time spent as well as quantity of actual programming also logically leads to a reduction in the number of bugs present in the overall model, and time spent rectifying any such errors.

Using real human beings in place of agents presents particular benefits to the study of socio–economic systems, such as financial markets. A controlled model environment using real human agents allows the ability to track both decisions and the factors that influence them, such as social and economic status within the model.

An obvious difficulty in using human beings in place of computerized agents is acquiring cooperative human beings to participate. A solution presents itself in the use of Massively Multiplayer Online Games (MMOGs), in which people from all parts of the world can participate. In addition to providing a pool of participants, the use of a game provides compensation to users in that it brings them enjoyment, a condition which both invokes a willingness to participate and eliminates any need for financial compensation of a participant's time.

Massive Multiplayer Online Games (MMOGs)

Remarkably, one of the largest collective human activities on the planet is the playing of online games [IBM07, Cas05], which are currently played by more than a hundred million people worldwide.

In online games *all* information about *all* actions taken by *all* players can be easily recorded and stored in log files at practically no cost[1]. The quantities of the available data is highly impressive and has previously been unthinkable in the traditional social sciences. There sample sizes often do not exceed several dozens of questionnaires, school classes or social science students in behavioral experiments. In MMOGs not only the actions of the individual players are known, but — most importantly — also the surroundings (of the given players) under which a particular action (or decision) was taken, are available. This offers the unique opportunity to study a complex social system: conditions under which individuals make decisions can in principle be controlled, the outcomes of decisions can be measured. In this respect social science is on the verge of becoming a fully experimental science, which should increasingly become capable of making repeatable and falsifiable statements of collective human behavior, be it in a social or economical context.

Another advantage over traditional methods of data acquisition in the social sciences is that in MMOGs players do not consciously notice their actions are monitored and recorded; thereby avoiding aberrent behaviors resulting from ''observer effect'' or ''reactivity'', longstanding complications in the classic natural sciences [Hol97]. These games do not only offer a way to explore sociological questions, but — if economic aspects are part of the game (as it is in modern complex games) — also to study economical behavior of groups. Here again it is of great importance to mention that not only economical actions (decisions) can be monitored for a huge number of individual players but also the (social and economical) circumstances affecting the players and their decision making are known.

This means that MMOGs offer an environment to perform *behavioral economics* experiments, which have been of great interest recently in a number of small-scale experiments, e.g. [GF99, HBB+05]. These experiments have demonstrated that man — as an economic being — does not behave rationally and is definitely not a *homo oeconomicus*. These experiments and their underlying concepts have led to the Nobel prizes of D. Kahneman and V.L. Smith in 2002.

[1] given the consent of the player, which happens usually at the beginning of her participation in the game.

Another intriguing feature of online game data is that — again only for several complex games — there is data available on both, social *and* economic behavior at the same time, within one coherent setup. This finally presents an opportunity to systematically study economic and social behavior and its interconnections, which are practically unexplored: it becomes possible to study the socio–economic unit of a human online game society.

Problem Statement

In order to further explore the intruiging possibilities of using humans in place of agents, log files of the massive multiplayer online game *Pardus* have been made available. *Pardus* offers its more than 300,000 registered players a wide variety of virtual economic possibilities, as well as an array of social engagements and networking opportunities. All player data has been recorded since 2005 and now comprises about 3,300 database backups for a total amount of over 1,000 GB (see section 2.1).

However, a system must be designed and implemented to enable research of the existing game data. All personal information of players has to be anonymized. Researchers need instant access to any and all timeframes of the data. A client supporting routines for the following purposes has to be developed (see chapter 3):

- Calculate price formation and wealth distribution to compare with real world data and establish further evidence that online game communities serve as a good model for certain real world communities.
- Quantitatively and experimentally explore a series of long standing scientific questions regarding social network dynamics, group formation and dynamics, including the explicit ''experimental'' testing of several famous social science hypotheses.
- Quantify economic ''indices'' such as productivity, trading, price formation for goods and services in the game.
- Relate network structures with economic performance of players and groups of them, including exploring gender and country specific differences in social network dynamics.

The minimum data subsets that need to be prepared for the listed purposes are described in subsection 2.3.1.

The author of this thesis and the research institution are the sole owners of mentioned data and are not limited by data transfer restrictions, see section 2.5 on legal issues.

Overview

Chapter 2

describes the existing game *Pardus* and gives an overview of its data structure. This includes Entity–Relationship Diagrams (ERDs), class diagrams, use case diagrams and descriptions. Various difficulties regarding the data set and data mining are examined.

Chapter 3

explicitly lists the aims of the thesis.

Chapter 4

evaluates data mining strategies. Next the used data structure is described in detail. The process of compiling data for this structure is documented by giving information about the code, data structures, and algorithms used to solve the stated problems. A client for scientific queries is developed and seamless updates to the data are enabled.

Chapter 5

explains the general usage of the program and shows the information gained by selected routines.

Chapter 6

summarizes the work and its results, and gives an outlook for future research.

Chapter 2.

Description of the Data Set

2.1. The Pardus Game

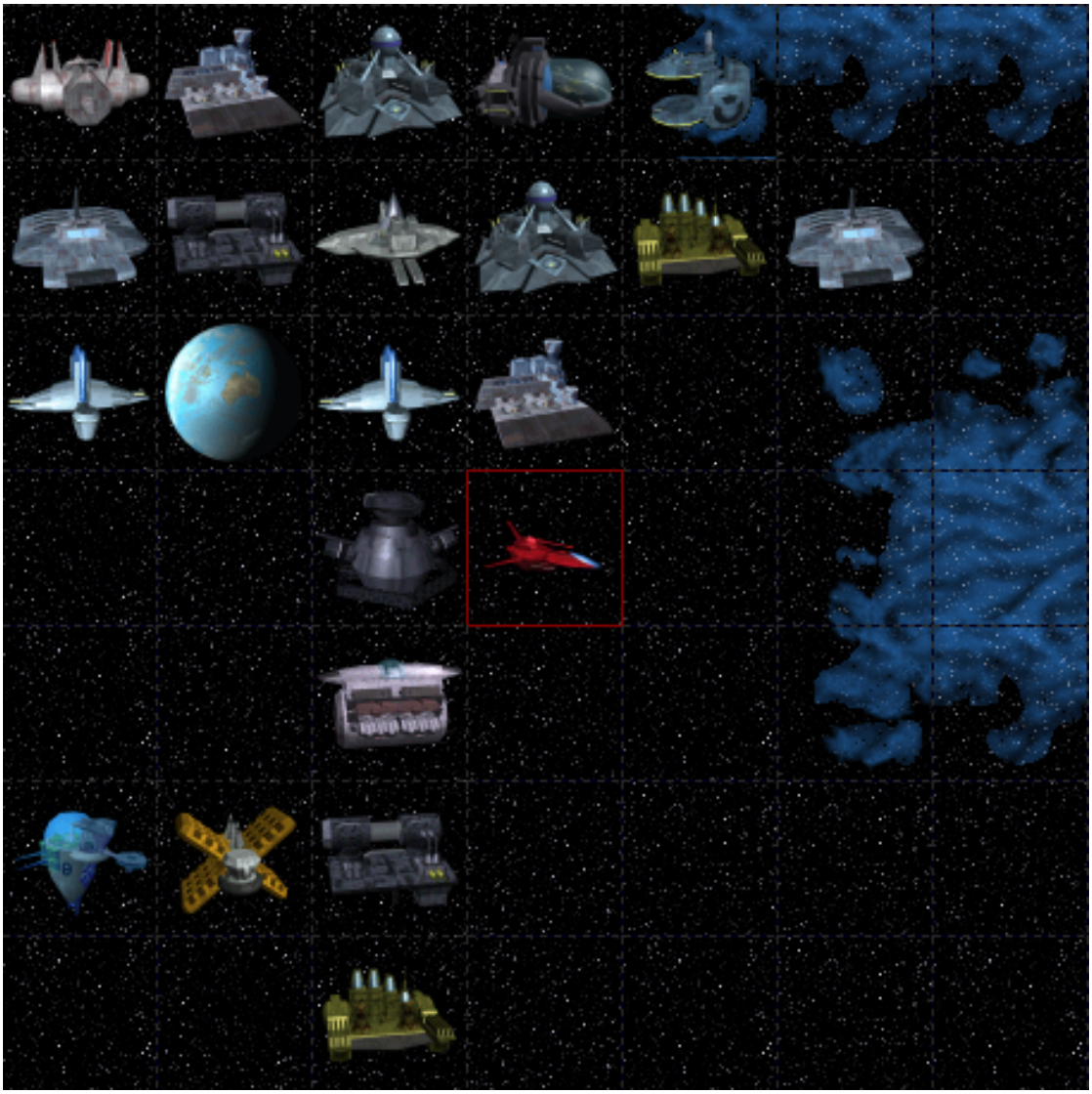
Computer games can be categorized into single-player and multi–player games. Online games of the latter category are often referred to as ''massive'', meaning a large number of users play and interact together in the same virtual environment. *Pardus* [OEG04], like the well-known game *World of Warcraft*, is a massive multiplayer online game (MMOG). However, *Pardus* is browser-based, allowing easy accessibility due to platform independence; whereas games like *World of Warcraft* require software installation in order to play.

Pardus is an open–ended massive multiplayer online game with a worldwide player base of more than 300,000 people. The virtual game setting is a futuristic universe, populated with planets, varying topology, resources, game controlled enemies, player-created structures and of course players themselves. Each player is represented by a graphical spacecraft; depending on the wealth and skills a player accumulates in the game the spacecraft may be upgraded. The game's environmental topology is basically fixed but can be manipulated by the players to some extent, such as constructing buildings that prevent other players from entering an area of the universe.

Figure 2.1 displays the virtual world of *Pardus*; a player's spacecraft in the center surround by various player–owned buildings, a resource field and a planet.

Since the game is open-ended players set their own goals; typically goals are oriented towards accumulation of in-game wealth, power, or fame — positive or negative — among their peers. *Pardus* does have a few fixed game rules, primarily concerning behavioral

Figure 2.1.: The virtual environment of *Pardus*.



etiquette. Most players invent and develop their virtual social lives without any constraints or guidance by the game system.

Observation of players in *Pardus* has shown an astonishing amount of self-organization emerging within the player community, without any influence by game mechanics. Players demonstrate consistent tendencies to self-organize into groups and subgroups; groups of players may work cooperatively to benefit the entire group, such as claiming territories or attempting to improve their local economy. Frequently hostilities will develop between groups, sometimes leading to full–scale war.

Because *Pardus* implements an ''Action Point'' system which restricts the amount of actions a player may perform in a day, many group activities in the game require considerable planning and cooperation. Though it is possible for a player to develop an advanced, powerful character without support from other players, it is generally easier and faster to accomplish goals with teamwork — as is often the case in real–world situations. Some large–scale actions — such as building spacecraft or keeping a starbase operational — require an extensive network of commodities, more than one player could reasonably supply single–handedly. Since no game mechanics exist to replicate this chain, it is entirely up to players to determine need and cooperate to produce supply in a way that all parties involved will generate some degree of profit. Figure 2.2 presents the production–tree of *Pardus*.

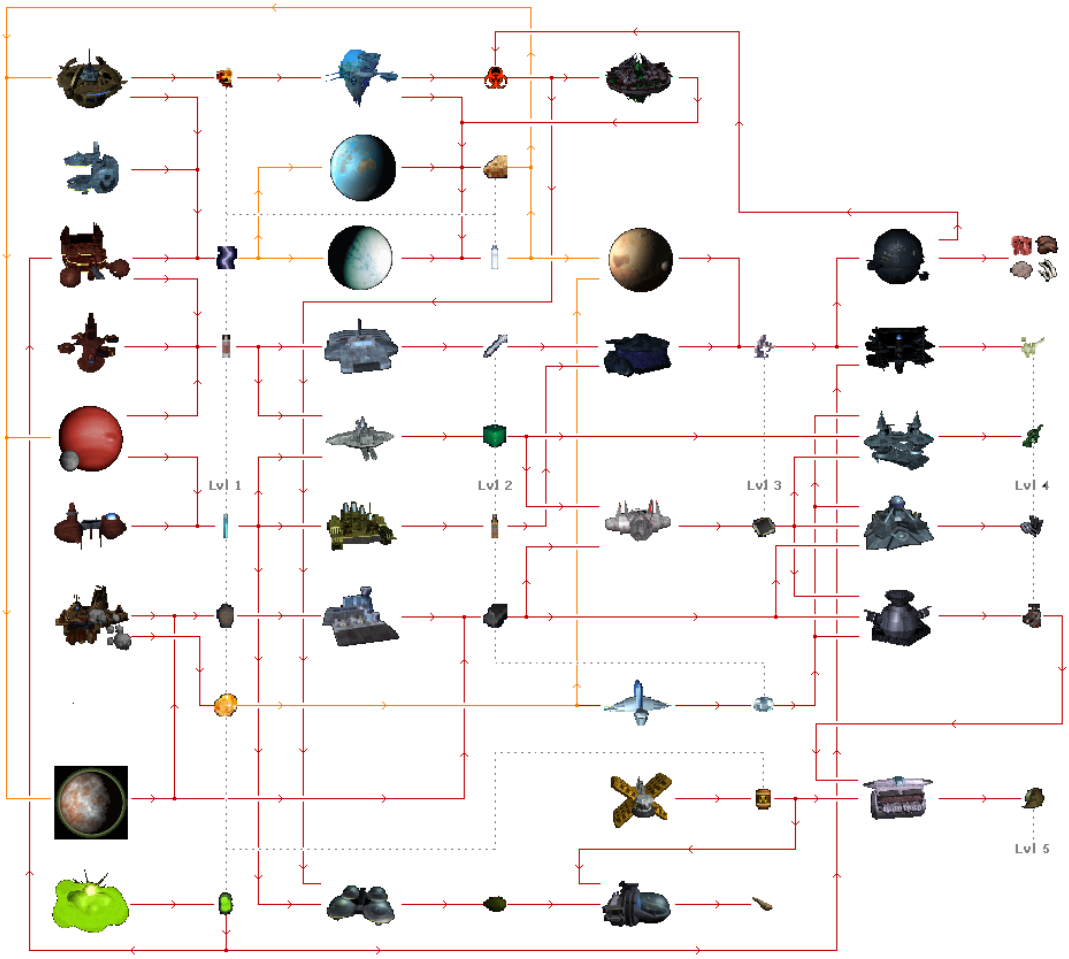
2.2. Hardware

At present *Pardus* operates three separate universes; the universes are carbon copies of each other and differ only in ways players have influenced the environments. The original universe, Orion, was founded on September 14th, 2004. The other two, Artemis and Pegasus, were opened June 10th 2007.

Each player is permitted to have one character in each universe, though some players choose to devote all their energy to only one or two universes. At the peak of every day's activities, there are over 1,000 players online at the same time. Players are automatically deleted after an inactivity period of 120 days.

Daily database backups are available starting 2005-09-09, in total making up about 3,300 databases, each containing data of size about 300 MB. Data sets within each database are organized in about 200 tables of varying size holding up to 60 fields. The game is

Figure 2.2.: A sampling of the *Pardus* production–tree.



programmed in Hyper Text Markup Language (HTML), Hypertext Preprocessor (PHP) and C/C++, using MySQL-databases. Several tables exist, some created for logging purposes only, which trace almost all of the hundreds of possible player actions. Most are provided with a timestamp (the temporal resolution is one second).

2.2.1. Hardware Deployment

The data structure can be split into three conceptual work units, making it easy to distribute the work load among several connected servers (see Figure 2.3 for an Unified Modeling Language (UML) deployment diagram).

A game server handles all game data and workflow of a specific universe. There are three game servers in total, one for each universe. They make up the biggest part of the data size and contain most of the information of importance for socio–economic research.

There is one community server hosting all chats and forums, which are seamlessly integrated into the game areas.

The account server is responsible for logins and account-wide settings. Data from this area is sometimes private or personal and will not be used in any type or research or for any other purpose.

For the scope of this thesis only data of the game servers is relevant. Further work could also include activity logs in forums and chats.

2.3. Data Structure

2.3.1. Relevant Data Subsets

Income and Wealth Distributions

In *Pardus* each player must pursue an economic activity in order to advance their character. Money in the game is represented by ''credits'' and may be acquired in a variety of ways. Daily income is known for all players and can be aggregated to weekly, monthly or yearly timescales. This data can then be straightforwardly compared to income distributions for real world economies.

Figure 2.3.: Deployment Diagram

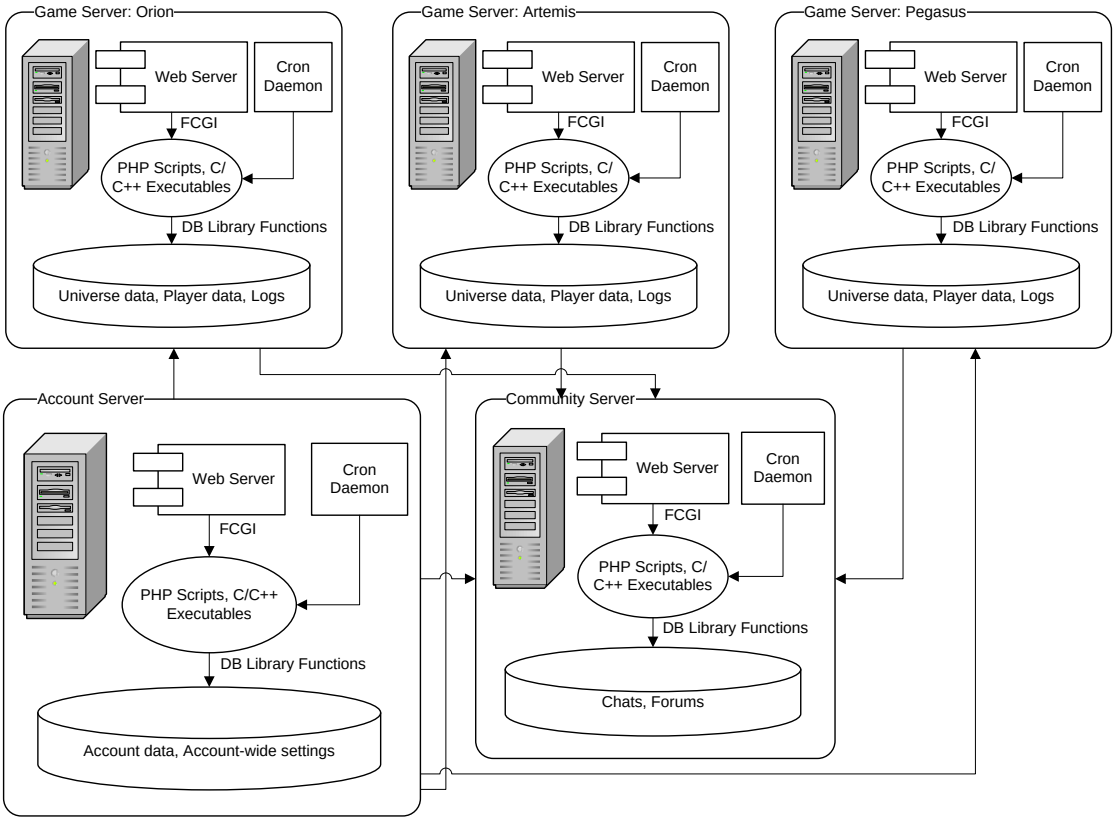
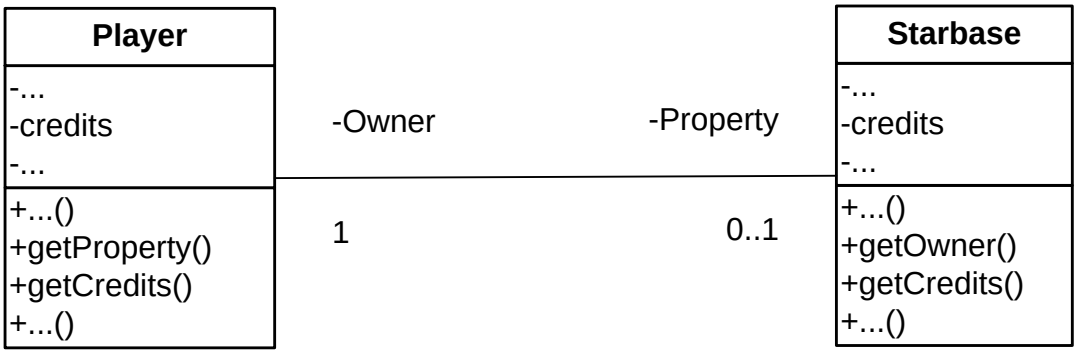


Figure 2.4.: Wealth Class Diagram



Daily backups of classes as in the class diagram in Figure 2.4[1] allow calculation of wealth as well as daily income.

As previously mentioned, *Pardus* has a production–tree for commodities from the simple to the complex. The simplest commodities are produced directly from harvestable resources, making buildings which produce simple commodities ideal for newer players with little financial resources. More advanced buildings require simple commodities in order to produce more complex goods. At the highest level, advanced commodities are required to produce spacecraft, building and ship defenses, and various enhancing equipment. In addition to upgrading spacecraft, equipment and buildings, players need credits and resources to repair their ships and structures damaged in combat or deteriorating over time.

The *Pardus* virtual environment is quite large. Commodities produced, bought and sold in one area will have little or no effect on the economy of distant areas. As a whole, the *Pardus* economy is comprised of many local micro–economies. With the dynamic, player–driven market structure, local economies are constantly changing and evolving. Thriving communities may draw in players until the market is saturated and the economy begins to stagnate; enterprising players may try to establish new markets in low populated areas or revive languishing economies. Player approaches to economic development are often creative and cooperative.

[1] Non-relevant attributes and operations are not displayed.

Figure 2.5.: Trading Class Diagram

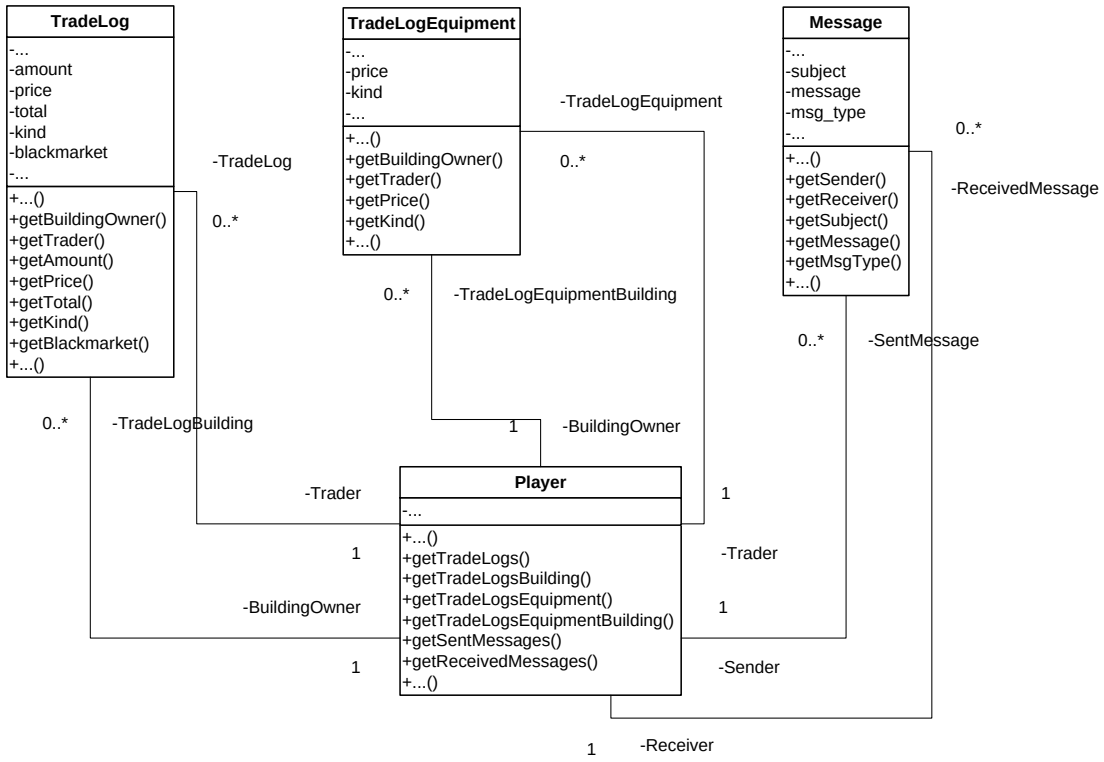
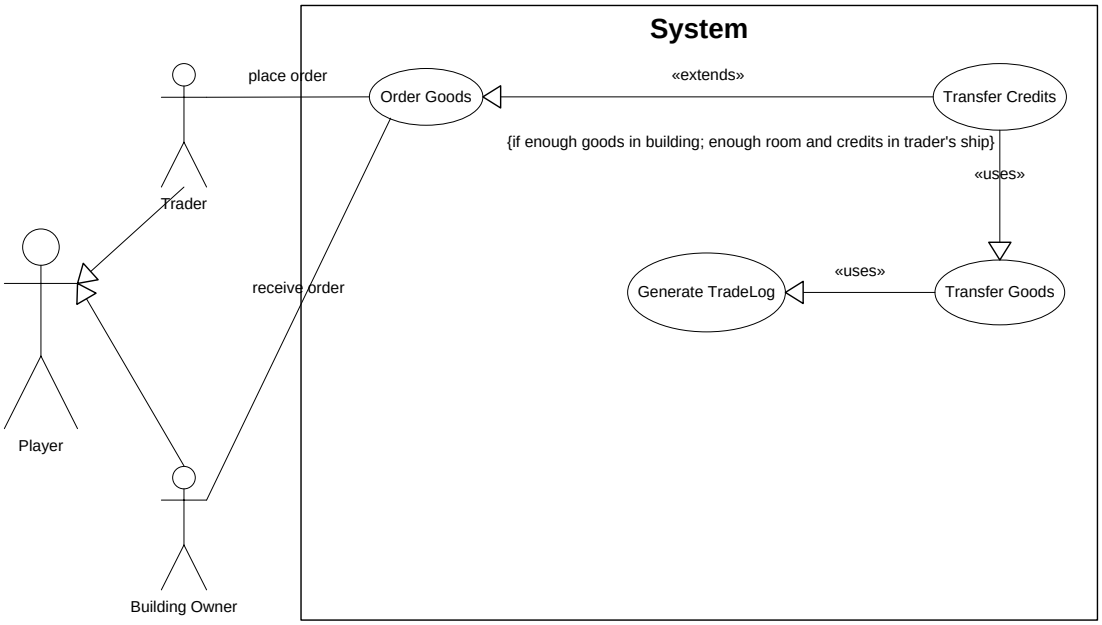


Figure 2.6.: Trading Use Case Diagram



Price Formation

An important component of economic activity in the game is the trade of raw materials (commodities) and manufactured goods. For these products there exists a market among players, leading to a dynamic price for goods. The price formation follows a double auction mechanism, see e.g. [SFGK03], which is a standard trading technique in many real-world stock exchanges. Therefore, it allows direct comparisons between online data and data obtained from large real–world groups.

Figure 2.5 visualizes the classes responsible for saving information about any and all trade transactions. Upon trading with a building, for example an *Asteroid Mine* or a *Smelting Facility*, both buyer and seller receive a *TradeLog* entry for each traded good. The same happens with *TradeLogEquipment* entries when equipment is bought or sold at a starbase. Credit transactions are logged as *Message* object with *msg_type* ''system''. A typical use case is displayed in the use case diagram in Figure 2.6.

Private trade is also frequent between players, in which players communicate privately and arrange to trade directly without using the building, starbase or planet interface. In these cases the value of the commodities may deviate widely from the general market price. The Direct transfer of credits and/or commodities is also logged and easily traced.

Exchanges of commodities or credits between individual players carries with it the risk that one party will not uphold his end of the agreement and may effectively steal from the other player; such conduct has particularly interesting ramifications in future social and economic interactions with the offending player and his recognized friends or alliance.

A market also exists in the game for information and services. It is not uncommon for a player with a spacecraft ill–equipped for battle to pay another player a considerable amount of credits to seek out and destroy an enemy. Likewise players will sometimes pay a large sum for information as to the whereabouts of an enemy or an enemy's buildings. Prices for services such as these are typically negotiated on a case–by–case basis and vary widely.

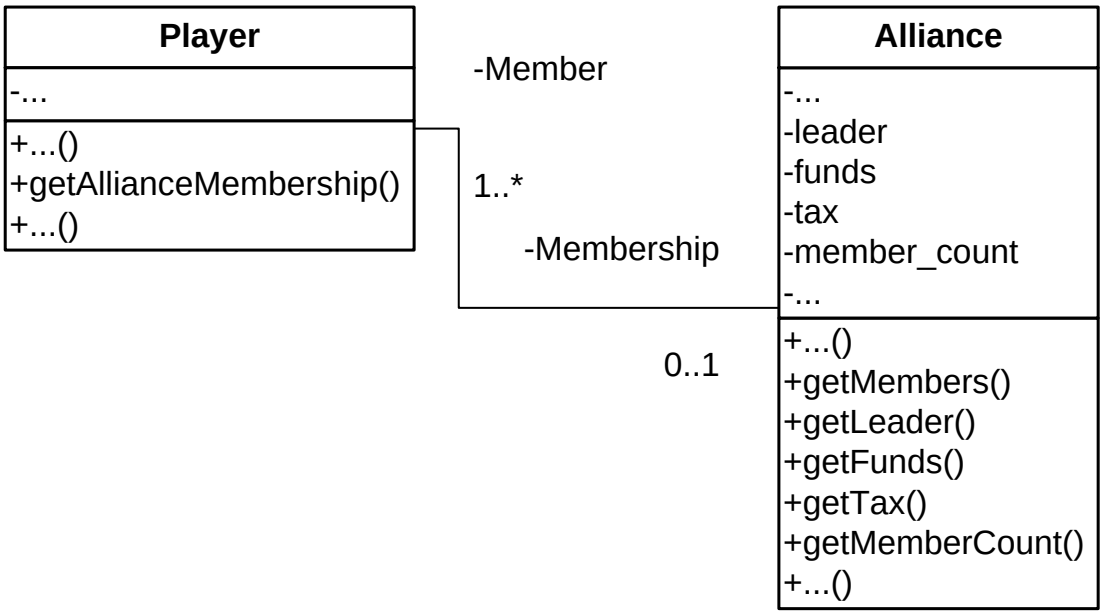
Communication Networks

Socializing is an important part of *Pardus*. Players may communicate with each other via public outlets such as *Pardus* chat rooms or forums, or they may send each other private messages. Most communication between players in the game happens through the exchange of private messages through a system–wide email service that is specifically provided by the *Pardus* game. Every message sent from one player to another is recorded with a time–stamp, which allows the mapping of communication networks of players with a precision of seconds. These networks can be characterized by a number of network measures which can be directly compared with real world communication networks.

As in many real–world social settings, players place a lot of importance on rising in social status. The acquisition and display of status symbols is not only frequent but also an important psychological driving force for many players; a great deal of energy is devoted to gaining recognition of status by peers.

Pardus offers players the ability to publicly display some types of status symbols; such as in–game news entries for certain feats and medals won for defeating enemies or war efforts. A player's ship is the most universal status symbol in the game — high-end ships represent a large investment of time and credits by the player; it therefore naturally follows that the player has considerable experience and knowledge about the game, as well as a firmly established network of friends and supporters, which itself elevates the player's status in the eyes of others.

Figure 2.7.: Alliance Class Diagram



Group Formation

Players have the option to create or join ''alliances'' in the game. Alliances are entities with the purpose of combining groups of individual players, allowing them to operate as a team with ease. In general players form or join alliances with a group of players who share similar goals; for example one alliance may focus on developing local economies and expanding wealth, while another alliance may put its collective energy into building a fighting force to destroy enemies.

Alliances vary from just a few individuals to hundreds of players. Belonging to an alliance can be a status symbol in and of itself; players belonging to a very large and powerful alliance may have less concern for the consequences of their actions, knowing that their alliance will support them if needed — be it financially or with firepower. Conversely, most alliances have a general ideology of how its members should behave and individuals not conforming to that ideology risk being rejected and removed from the alliance. Being forcibly removed from a well–known and respected alliance can have longstanding social consequences for a player.

The *Alliance* class is shown in Figure 2.7.

Table	Size (MB)
player	2
alliance	1.5
diplomacy	1.5
message	65
starbase	0.1
trade_log	68.5
trade_log_eq	11.5
7 tables	**150.1MB**

Table 2.1.: Size of relevant tables

Social Networks

On a smaller scale than alliances, players have the ability to mark other individual players as friends or foes, links which can be changed or removed at any time. These networks can be analyzed on a daily basis, or accurate to the second[2], enabling the research of friend and enemy evolution over time.

Interestingly some players choose to socially ostracize themselves by consistently engaging in undesirable conduct; such as repeatedly destroying other players' ships or buildings, sometimes causing considerable financial losses. These players, typically referred to as ''pirates'' in the game, often unwittingly stimulate local economies. An otherwise stagnant economy will see markedly increased activity if one or more pirates are in the area, as victims purchase additional commodities to rebuild, repair, or increase defenses in the area. Social stimulation is also a frequent side effect in such situations, as players with buildings or other concerns in the area that previously displayed no interest in each other may unite to face the common enemy of the pirate.

2.3.2. Database Layout

Seven tables of the overall amount of 230[3] are relevant for above–mentioned research. The combined size of data and indices of these tables makes up about 50% of the original single database size — 150MB of 300MB. The resulting database is visualized in Figure 2.9.

[2]The time–stamp can be found in the message (*datetime* field) informing about the friend or foe setting (see Figure 2.8).
[3]as of December 16, 2009

Figure 2.8.: Diplomacy / Message Class Diagram

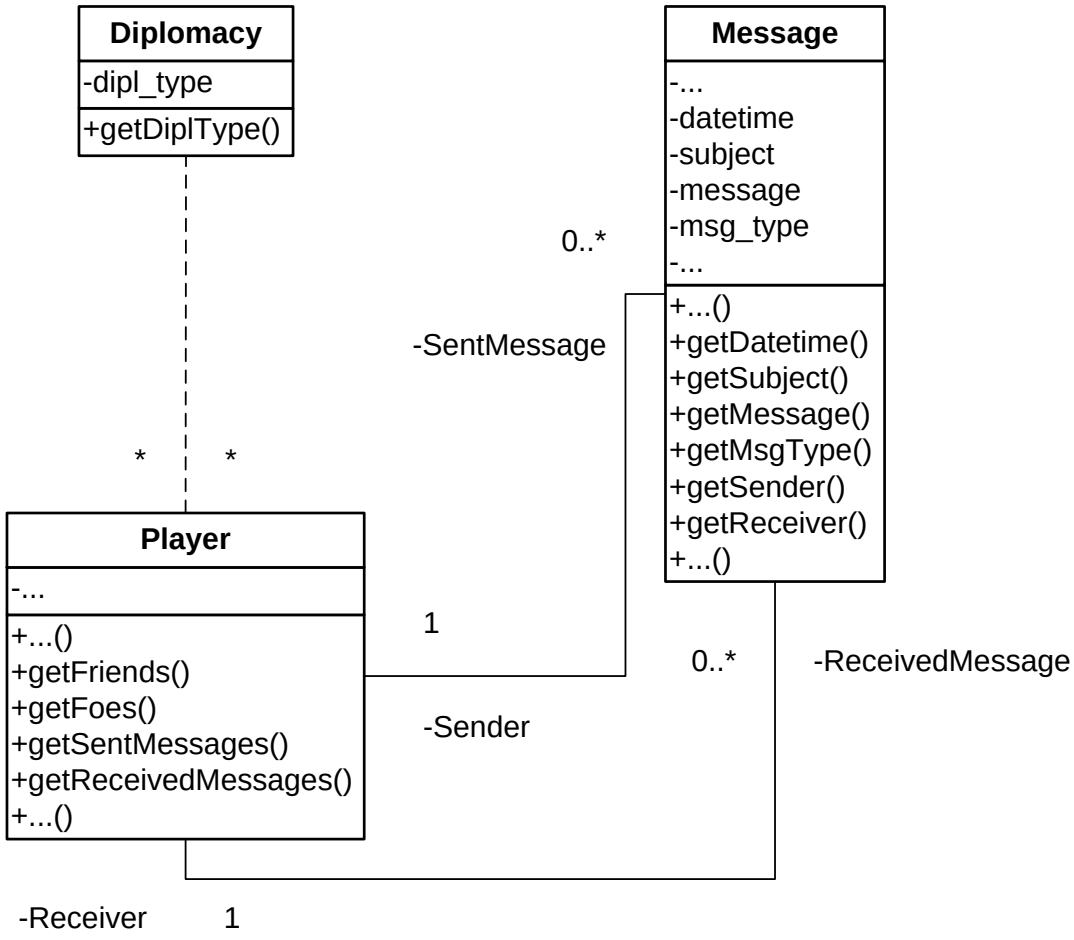
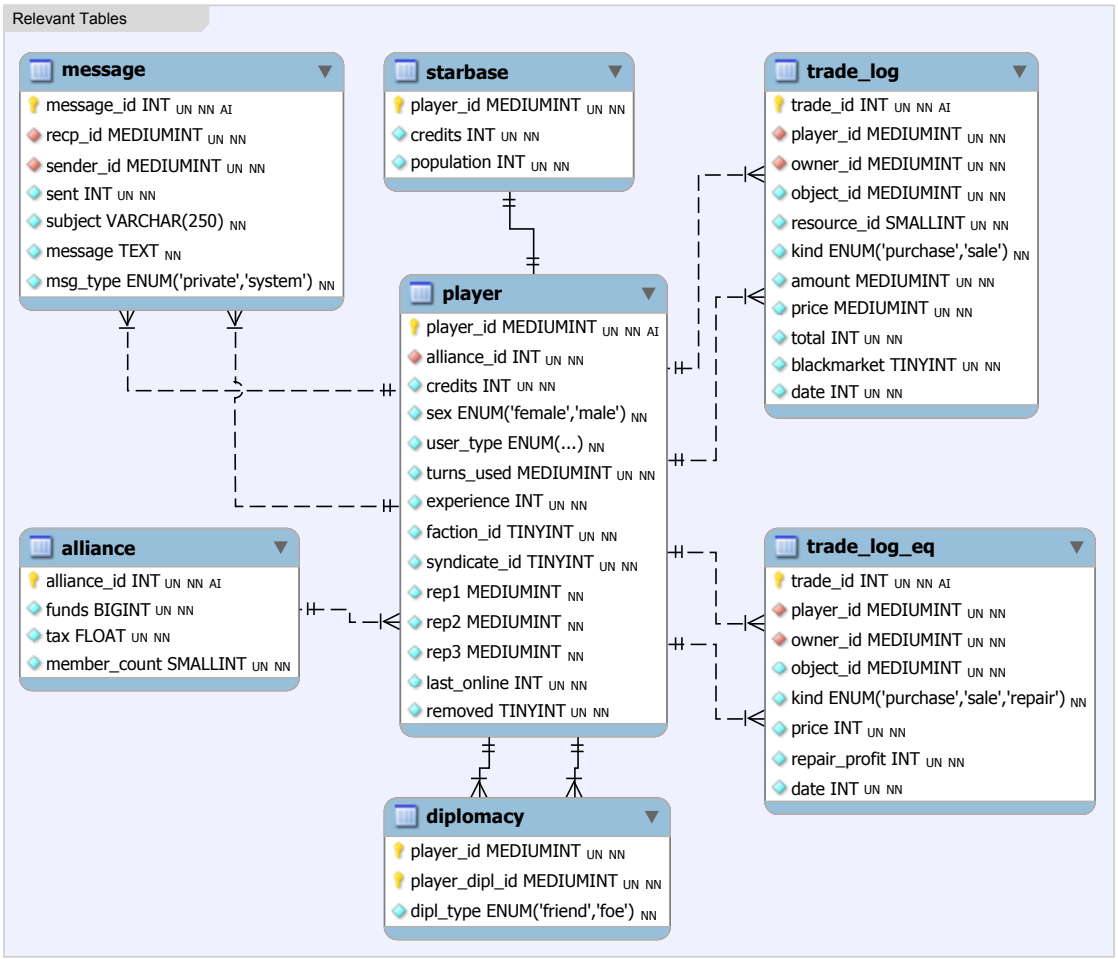


Figure 2.9.: Relevant Tables Entity-Relationship Diagram



2.4. Intermittent Changes and Outages

2.4.1. Past Changes

Database backups are available since 2005-09-09, a time when *Pardus* was still in an *alpha* stage and under heavy development. For example, the *Payment Log*, a log recording various expenses that were previously stored as messages, was only introduced on 2008-09-14. A holiday charity was held in December 2008, allowing players to donate for the poor, in–game. Until 2007-06-09 only one game universe existed, with the account server being integrated into the game server on a single machine. Aside from ramifications for extracting information, additional/missing tables have to be considered when pruning old data. The data structure and client to be developed have to take all of these matters into account.

2.4.2. Future Changes

Even though the game is considered stable, minor changes and readjustments still occur. The new data structure must be designed with keeping future modifications in mind, allowing easy adaption and expansion.

2.4.3. Outages

In the past several server and backup outages have happened, due to hard disk failures, broken connections and failed server migrations. When analyzing the data, the client must allow excluding and interpolating certain dates (see Table 2.2). Future outages are possible but less probable — backups are triple–saved and all servers including all services and hard disk RAID 1 arrays are constantly monitored.

2.5. Legal Issues

All research data will be fully anonymized. Players' identities must not be known by the researchers. All players are aware and agree that their actions are recorded and may be used for scientific work in an anonymized form. The Medical University is informed about the study and has stated in a document that there are no conflicts with the federal

Date	Server	Amount of days
2006-03-24	Orion	31
2006-04-28	Orion	3
2006-10-24	Orion	3
2007-03-20	Orion	1
2007-05-10	Orion	1
2007-09-21	All	1
2008-02-09	Orion	1
2008-03-25	Pegasus	1
2008-06-09	Orion	1
		43

Table 2.2.: Missing database backups

act concerning the protection of personal data, Austrian Federal Law Gazette part I No 165/1999.

Chapter 3.

Aim

The data set described in the previous chapter in its current form of about 3,300 single database backups is the basis for socio-economic research. A system must be designed to provide researchers with instant access and the capability for complex queries on any and all data. In particular, the following points need to be addressed:

1. **Data Mining Strategies**
    a) **Evaluation**
    Data mining strategies have to be evaluated and compared with regard to storage size and practicality, security, speed and ability for concurrent usage.
    b) **Data Structure**
    A system capable of containing all relevant data must be designed in a way to make efficient research possible.
2. **Technical Implementation**
    a) **Data Extraction**
    Relevant information of the data set must be extracted and prepared for anonymization and integration.
    b) **Anonymization**
    Before any data is made available, all personal information has to be anonymized while keeping all correlations and network structures intact.
    c) **Data Integration**
    The anonymized data set must be shaped to integrate with the new data structure.
3. **Client**

a) **Routines**
The client must support scientific routines as specified in the problem statement in chapter 1.

b) **Export / Backups**
Researchers need to be able to export the complete data set in Extensible Markup Language (XML), Structured Query Language (SQL) or Comma Separated Values (CSV) syntax, manually and by setting an automated schedule.

c) **Import / Updates**
Pardus steadily generates information in form of database backups. It must be possible to manually and automatically import and integrate these backups (updates).

Chapter 4.

Data Mining

4.1. Strategies

The existing data set in the form of several thousand daily database backup files must be aggregated to form a unique system providing efficient data mining capabilities. In particular, the following key aspects have to be taken into account:

Completeness

The data structure to be developed must contain all relevant data, allowing any data that exists in the source to be found in the new data set.

Querying

Sorting, ordering and filtering the data set will be a main task of researchers and must be a built–in feature as such.

Speed

While updating plays only a minor role, queries must be handled in an efficient way guaranteeing fast response times.

Storage size

As new data is generated on a daily basis the size of the data set will regularly increase. Care must be taken to avoid any overhead or redundancy; compression to some extent may be desirable.

Safety

The data must be safe from any unwanted modification, anomalies and hardware failures.

Security
: The data needs to be appropriately secured against intruders.

Concurrency
: Multiple reading as well as writing requests need to proceed in a parallel fashion while not interfering with each other.

Consistency
: Failed updates must not leave the data in an inconsistent state.

In the following sections the quality of the different storage systems — flat files, XML storage, object-oriented database and relational database — is measured for the described aspects.

4.1.1. Flat Files

The simplest form of storage is saving information in text files. The most common method of structuring information in a file is using spreadsheets — usually comma or tab delimited values as certain attributes, one row representing one object. Many standard UNIX files, like */etc/passwd* containing user data delimited by colons, are stored using this format.

There exist many tools in the Single UNIX Specification [IEE02] such as sh, grep, awk and perl that can be used to retrieve rows matching certain patterns, although many tasks that play a major role in data mining, like sorting the data set, require programming skills. There also exists a query language that tries to simplify such tasks by providing C style query expressions [Fow94]. This approach still requires knowledge of C for the definition of the attributes and lacks support for updates.

The speed is highest for read operations that do not need to sort, order or filter any data. Operations needing to look for specific records must make use of indices to be efficient. Indices could be created dynamically and be held completely in memory, or stored and updated in external files. External tools like *cql* can add indexing and thus support fast filter and sort operations. As this tool does not support updating of records, this speed improvement is again lost in write operations.

The use of storage size in flat text files is not optimal. Since no typing is possible, any character including the delimiter takes up one byte or more depending on the used

character set. For example, boolean values needing only one bit in typed environments require at least one byte in text files.

Data safety and security can easily be established by traditional means of securing the files and file system.

Concurrency and consistency can only be maintained by the use of an interface through which the files are accessed. The interface must make use of file byte range locking to allow simultaneous access.

As shown, the advantage of using plain files as storage method lies in simplicity. To make this method an option for our purposes, the use of external tools would be mandatory, critically dampening the reason to use flat files in the first place.

As a final note it should be mentioned that there is a relational database which can use flat files as storage while providing SQL syntax for queries [Sch06]. However, this kind of storage engine does not support indexing and lacks Atomicity Consistency Isolation Durability (ACID) compliance.

4.1.2. XML Storage

Files in the XML format contain both data and metadata describing it. Due to its flexibility and simplicity it is frequently used by programs as a method of storing or transmitting data.

There is a wide range of software products that support the XQuery language, a recommendation (''standard'') of the World Wide Web Consortium (W3C). It is a funtional programming language with extensive querying abilities. Lacking support for modifying the XML source file several extensions have been developed [SHS04] to also provide update functionality.

While XML query tools can make use of indices, their speed is dramatically decreased by the additional need to parse each file's structure [NJ03].

XML storage size is considerably bigger than in all other discussed methods. Most importantly, all structure is defined within the XML file leading to a large amount of redundant metadata. Also, since XML files are normal text files, a value of any data type cannot be smaller than the size of one character of the used character set the file is encoded in. Help is provided through queryable compression engines which can compress both tags

and textual content [TH02] and even eliminate the redundant structure [ABMP07] while improving performance under certain conditions.

As with flat files, safety and security can simply be established on the file system level by backups, failsafe hard drive systems and restrictive file permissions.

There are multiple native XML – ACID compliant – databases providing highly configurable modes of locking and isolation [HH04]. Performance penalties vary from 25% to >300% depending on the selected isolation level [HH04].

Interoperability and flexibility are the key advantages of XML storage. The XML format is often used as transmission medium, thus the use of native XML as database saves unnecessary data conversions at servers and clients. On the other hand, the price for flexibility comes in form of a considerable overhead in both storage size and performance.

4.1.3. Object-oriented Database

Object–oriented database management systems (OODBMSs) store objects without the need for a mapping layer as in other methods. The OODBMS generally uses the same model as the programming language. Subclassing, inheritance and other complex constructs are possible without additional effort.

OODBMSs do not necessarily need a query language, they rather use the means the object-oriented programming language has to offer. However, several products also offer Object Query Language (OQL) and even SQL syntax [HLW94].

Querying complex objects in OODBMSs is efficient as connected objects are stored by pointers that can easily be followed. This kind of path is called navigational query. Opposed to that, the broad search for any objects with specified attributes is cumbersome. Searching without indices would require creation and destruction of all objects that are being compared. Creation and use of an index becomes complex because properties of objects can not only be primitive values but objects as well [Ber94]. Several strategies exist for indices to parse through hierarchies of nested classes [CC04].

Persisting objects can differ from one OODBMS to another. For example, an object can be serialized into XML format or binary format. In each case not only the raw data but also all methods and class metadata is stored as well, leading to an unnecessary overhead compared to saving just the data.

Several commercial and free object-oriented database products exist, offering configuration of access security, persistence strategy and different levels of transaction isolation. While speed is also based on the level of isolation or locking, OODBMSs generally scale worse than relational databases under high load [Lea00].

An OODBMS can save the effort of mapping between raw data and objects and can even be of superior performance when used in connection with highly complex classes. In most data mining scenarios however, simple but millions of data sets are dominant. An actual object representation is not needed but instead the support of complex and fast queries, making an OODBMS not a promising choice.

4.1.4. Relational Database

In RDBMSs data is divided into relations, usually called *tables*, consisting of rows and columns. Columns define attributes and their data types – one row is made up of the attributes' values. Tables can be further divided into separate namespaces, *schemas*, and databases. Many RDBMSs however avoid the use of explicit namespaces and imply databases with schemas.

A query language, SQL, exists which is an International Organization for Standardization (ISO) and American National Standards Institute (ANSI) standard and implemented in all major RDBMS products. It allows complex sorting, ordering, filtering as well as modifying of data sets. While SQL is a declarative language, there are several procedural extensions, even object-oriented ones [Kul94]. SQL/Persistent Stored Modules (SQL/PSM) is a standardized extension adding features for defining procedure calls.

RDBMSs have been in use for over 30 years and are highly developed in terms of indexing and general performance. Several indexing algorithms are in use, the most popular being B+ Trees. The same kind of tree is used for organizing metadata in many file system. Various adaptions and enhancements exist, reducing cache misses [HP03], improving update performance [DR01] and speeding up the creation of indices [Kim02].

Although RDBMSs perform best when row sizes are static, they also support dynamic size for character type columns and the *NULL* type for no data. Storage of indices is the real overhead which does not seem to be avoidable with a system that needs to support fast query operations. Transparent compression is implemented for several RDBMSs products, in some areas even contributing to performance by decreasing disk seeks [TLGXYJX06].

Most modern relational database engines are transactional and write logs to record all queries that could potentially modify data. In case of a hardware failure the log can be used to restore the database to a valid backup point – all considering the file system is also properly configured, i.e. *write barriers* must be enabled. The same holds true for data security, which is a matter of correct configuration. All major RDBMSs offer a user account system, different sets of permissions as well as selective access locations, for example only local access.

Transactional RDBMSs support row–level locking ensuring good performance even under a high number of concurrent users. The most popular relational databases are ACID compliant but give the possibility to loosen the ACID conformance in exchange for less locks and thus higher speed.

In comparison to other discussed methods, RDBMSs are capable of all kinds of high–speed read and write queries and offer the most efficient data storage with compression capabilities. RDBMSs also fulfill the requirements of fully isolated simultaneous access, a secure access model, recovery from hardware failures and consistency through ACID compliance.

4.2. Data Structure

As a consequence of the strategies evaluation a relational database is used as the data storage back–end. *MySQL* is a GNU General Public License (GPL) and commercially dual-licensed RDBMS. It adheres to ANSI standard SQL and in large part to *SQL3*.

Among its features are in particular:

- Replication over several systems running *MySQL* server
- User access and permission matrix
- Stored procedures for imperative programming support
- Cursors — enable walking through record sets in an iterative way
- Triggers — actions that are automatically executed when events happen
- Nested queries — queries that use data from included queries
- Views — dynamic tables based on queries

- Query optimization
- Query caching
- Customizable partitioning to group rows and split tables
- Hot backup — saving a consistent backup while the server is running
- Several types of indices
- Data and network–protocol level compression
- Architecture to allow plugins
- Different database engines selectable per table
- *InnoDB* engine providing row–level locking and fully ACID compliant transaction management

Unlike other RDBMSs like *PostgreSQL* or the *MySQL Archive* engine, the *InnoDB* plugin offers compression of not only user data but also indices. As this project relies heavily on indices they can make a big part of the overall storage size. Compression and decompression are done on–the–fly using the *zlib* library implementing the *LZ77* algorithm. Aside from a multitude of server variables, compressed tables can be further tuned by setting table–specific page sizes.

4.2.1. Database Design

Since the game's three universes are completely seperated and a query always only concerns a single one, each of them receives its own data structure. In the course of this thesis a ''single'' database always refers to one for each of the three universes.

The original data source is a set of several thousand database backups. Their layout is shown in Figure 2.9. To enable efficient querying over all data of all backups they are merged into a single database. This method also allows the removal of redundant information. Much of the information is going to be doubled or tripled in subsequent backups, for example messages and trade logs that are not yet deleted. This kind of information is static, as opposed to player information that is changing over time.

Care is taken to ensure referential integrity. One *DELETE* or *UPDATE* SQL query leads to the deletion or update of all fields referencing it (''cascading foreign keys''), or fails.

While researchers will need special compilations of data, the system also has to remain flexible for changes to the game or future research. The solution is to fully store the master data and create *views* to extract data compilations as needed. *Views* behave like normal tables – only queries will dynamically pull information from other tables as specified with the creation of the view. The advantages are:

- an easy interface to get data that normally requires a complex query
- always up–to–date information
- no waste of storage space
- easy to create new *views* for new tasks
- easy to modify *views* after there have been changes to the source tables

Subsequently four databases are designed. The first one needed is of temporary nature; it is the relevant table subset extracted from the original backup. It includes various changes due to anonymization. The second database maps IDs to new randomized ones. The mapping must be saved to maintain correlations through all backups. Its access permissions are restricted to the importing process only. The third stage is the destination database for all anonymized data. While the previous databases were needed only for the import, this one is the single database used by researchers. It also contains the aforementioned *views*. The last database contains user and group information including permission levels for the web client.

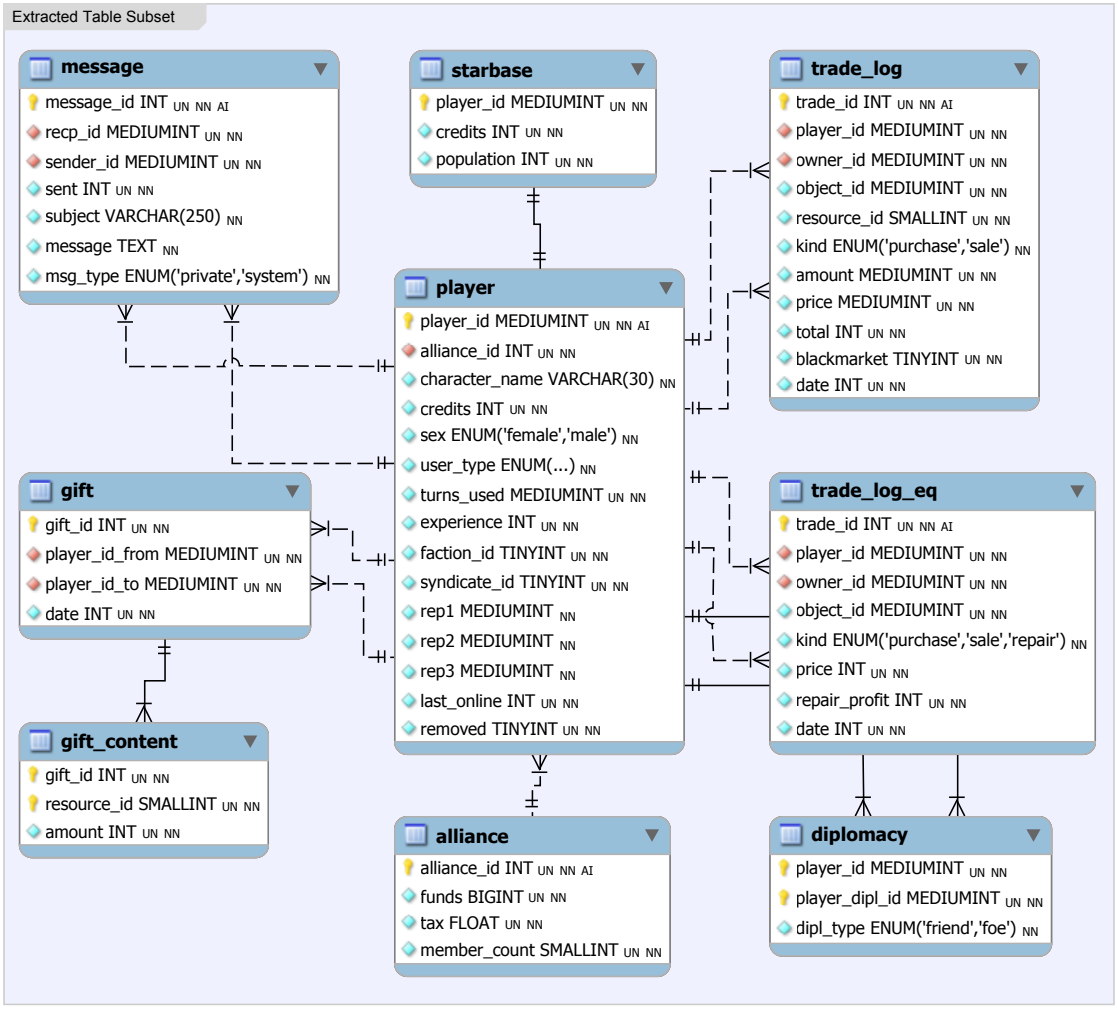
A high level of normalization is maintained for all non–temporary databases. To guarantee a logical, redundancy–free and anomaly–free layout they are in at least "projection–join normal form" (PJ/NF), also referred to as the 5th normal form [Fag79].

The databases are visualized with *MySQL Workbench* as ERDs using Crow's Foot notation. Abbrevations:

- *UN* — unsigned
- *NN* — not null
- *AI* — auto increment

The key icon next to fields indicates a *primary key*. Several key icons in the same table stand for a *primary key* made up of those fields. The red diamonds signify a foreign key and an index while the blue diamonds are non-prime values.

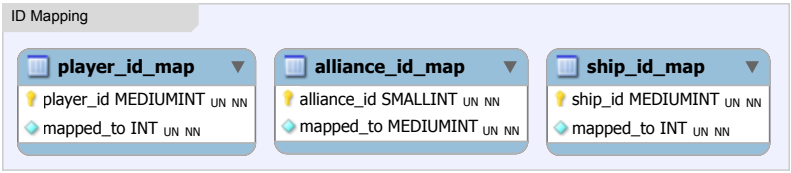
Figure 4.1.: Database Layout of the Extracted Table Subset.



Extracted Table Subset Temporary Database

Besides extraction of the relevant fields several more changes are done in Figure 4.1. All *datetime* or *timestamp* types are changed to simple *int* only counting the elapsed seconds since January 1st 1970. It is done for both smaller size and safer handling reasons – *timestamps* are usually updated with any modification of any field of the same row. Also noticeable is the existence of a new *gift* table and a *gift_content* table. These do not exist in the master data but are implicitly stored as system messages. When all data is extracted, messages must be parsed for this information. Once the gift tables are populated the *character_name* field in the *player* table and the *subject* and *message* fields

Figure 4.2.: Database Layout of the ID Mapping.



in the *message* table have to be dropped as anonymization measure. This leaves the externally visible *player_ids* and *alliance_ids* as identifying components. With help of the tables described in the next section their values can be randomized while keeping relations intact.

Tables in this database use the *MyISAM* engine to support faster and lock–free creation and population of all tables without any foreign key constraints. It is programmatically ensured that several extractions can be processed simultaneously.

ID Mapping Database

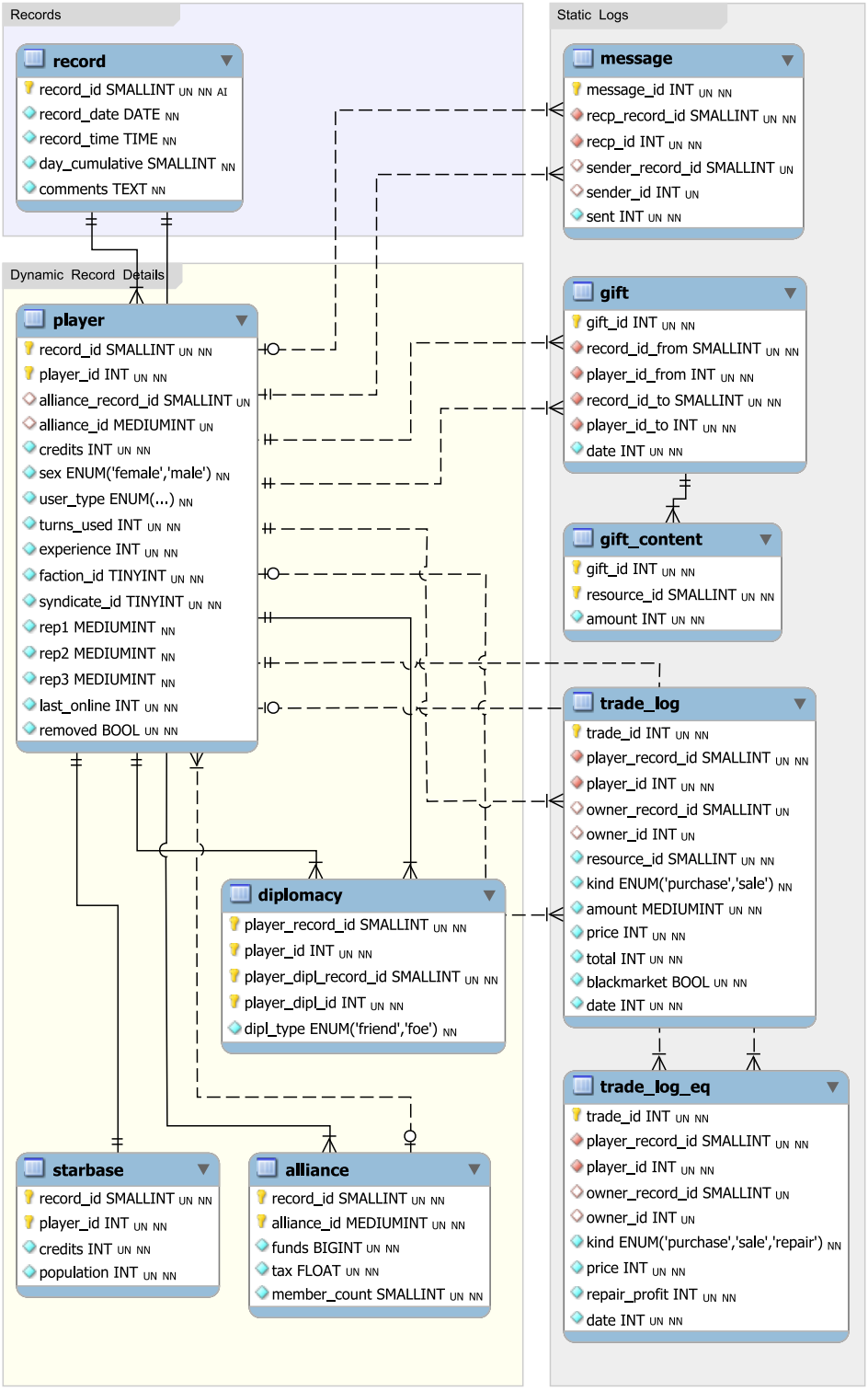
The tables in Figure 4.2 save a random unique value an ID is mapped to. The anonymization process goes through all IDs in the original table, retrieves the new mapping and then changes the ID in all relevant tables accordingly.

The tables are in *InnoDB* format to support row–level transactions as many processes may query this database simultaneously. Due to the small size of the tables they remain uncompressed in favor of speed.

Main Database

Once the extracted data set has been anonymized it is prepared for integration with the final data structure as shown in Figure 4.3. First, an entry in the *record* table is created. The date and the time of the backup as well as the amount of days since the first backup are stored along with the primary key *record_id*. A unique index is built on the columns *record_date* and *record_time*. This index ensures the same backup is not added multiple times and also allows faster queries for certain records. The rest of the tables can be divided into two categories: *Dynamic Record Details* and *Static Logs*.

Figure 4.3.: Main Data Mining Layout.



The tables *player*, *diplomacy*, *alliance* and *starbase* form the *Dynamic Record Details*. Entries in these tables change with each backup. It is important that these changes are stored so they can be visualized. Thus they include the *record_id* of the entry in the *record* table that was created for that backup. Together with the original primary key it forms the new primary key. All foreign keys referencing the *player* or the *record* tables feature the ''ON UPDATE CASCADE'' and ''ON DELETE CASCADE'' triggers. That way whole records or all information about a specific player can be removed with a single *DELETE* query.

Static Logs consist of the tables *message*, *gift*, *gift_content*, *trade_log* and *trade_log_eq*. An entry with a certain ID that is stored in one of these tables remains the same in each backup. Accordingly, the *record_id* does not need to be included in the primary key. If an entry is trying to be imported and its ID already exists in the target table, it can be safely ignored because the information does not change. This eliminates a considerable amount of storage space required by the original backups as logs are only deleted after a number of days. During that time they are redundantly saved in all backups.

Since most of the tables are referencing the *player* table through its primary key consisting of *record_id* and *player_id*, a surrogate key of *int* would remove the multi–valued foreign key indices. This way the referencing indices would become smaller and a faster in *JOIN* operations. On the other hand it would be impossible to deduce either the *record_id* or the *player_id*. Thus queries that work with these values would require an additional *JOIN* operation with the *player* table, making queries more expensive and complex again. Due to research concentrating on such queries a surrogate key for the *player* table is not used.

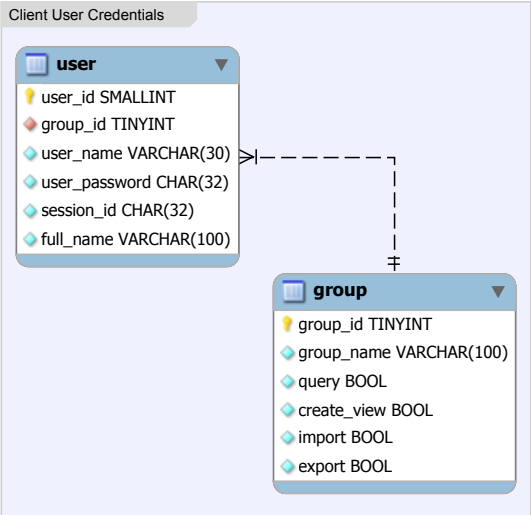
All tables use the *InnoDB* engine to guarantee full ACID compliance. Additionally they use the transparent compression capability of both user data and indices provided by the *InnoDB* plugin.

Client User Database

The database visualized in Figure 4.4 handles access to the web client. A user is identified by the primary key *user_id* and the unique index *user_name*. The password is saved as Message-Digest algorithm 5 (MD5) hash, as is the dynamically created session ID. A user is part of exactly one group that defines various permissions.

The tables are stored in uncompressed *InnoDB* format.

Figure 4.4.: Database Layout of the User Credentials.



4.3. Technical Implementation

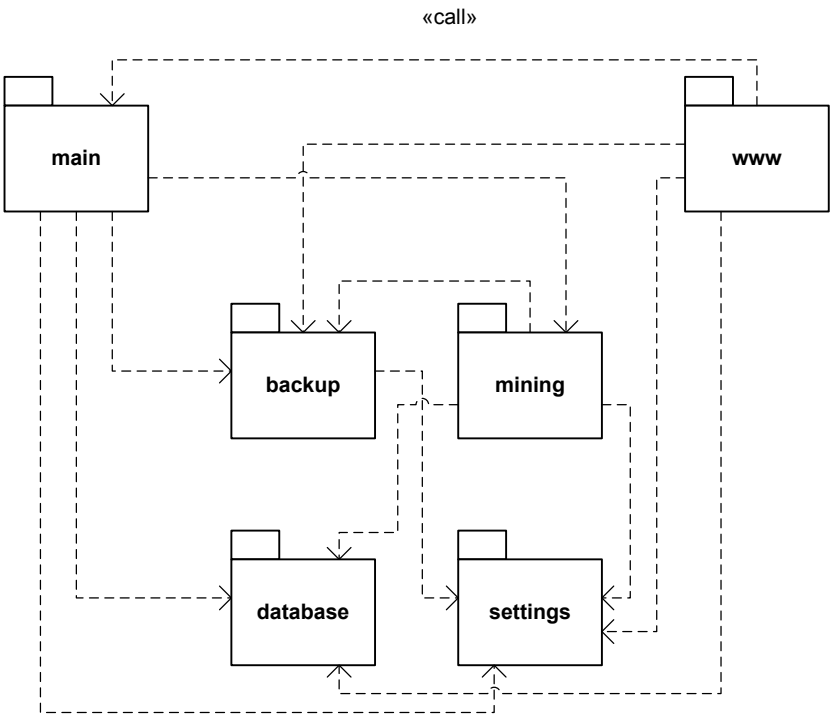
The program to integrate the backups into the newly created database is a Command–Line Interface (CLI) tool. A web interface serves as Graphical User Interface (GUI) front–end for authorized remote users. It handles integrating *Pardus* backup files (updates), exporting the database (backups), querying and creating views.

Both programs are written in PHP. PHP is an object–oriented programming language with support for namespaces, anonymous functions and closures in its *version 5.3*. A multitude of libraries exist, among others a *MySQL* connector supplying functions for the access of *MySQL* databases.

Source code is interpreted by the PHP binary. By default PHP comes with both a CLI and a Fast Common Gateway Interface (FCGI) interpreter. The FCGI binary allows easy integration with a web server. The web server forwards requests of certain files to the FCGI binary which interprets the requested file and sends the output back to the web server.

The entry point for the CLI program is class `AbsMain` in package `main`. The root for requests from the web server is package `www`. All requests are handled through the file `index.php`. The rest of the packages are shared by both the CLI and the GUI client. A high–level overview of the architecture is shown in the package diagram in Figure 4.5.

Figure 4.5.: Package Diagram of the Data Mining Suite.



4.3.1. Shared Mechanisms

Configuration

The application can be configured through command–line parameters as well as through modification of the `Settings` object or `Constants` file. The `Settings` class implements the *Singleton* programming pattern, only allowing one object of the class to be instantiated. The object can only be created through the static `getInstance` function as shown in the code snippet in Listing 4.1. If it already exists, a reference to the instance is returned. The standard constructor of the class is made private to avoid accidental invocations.

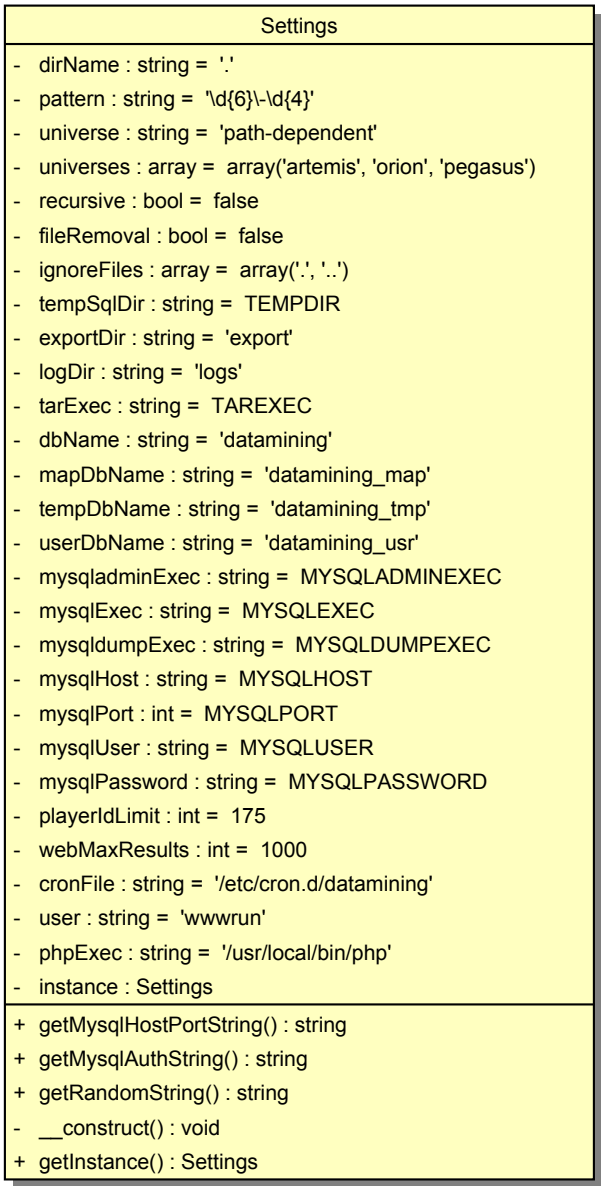
Listing 4.1: Settings::getInstance

```
1  /**
2   * Gets single instance
3   * @return Settings
4   */
5  public static function getInstance()
6  {
7      if (!isset(self::$instance)) {
8          $c = __CLASS__;
9          self::$instance = new $c;
10     }
11     return self::$instance;
12 }
```

Attributes of the `Settings` object – the configuration – can be retrieved and set using the appropriate get/set operations, see Figure 4.6. The default configuration is changed when command–line parameters are present. The default values of some attributes are constants defined in the `Constants` file. Constants are replaced with their actual values by the preprocessor. The purpose of the constants is to define system–wide default values once, saving the need to add them as command–line parameters with each invocation. For example, constants are used for the location of logging facilities, database authorization, the path to commands and the maximum *int* value.

Command–line parameters are parsed in the `Logic->process() : void` method using the operating–system dependent `getopt` function. Parameters are checked for validity as in Listing 4.2 line 7 and eventually saved in the `Settings` object.

Figure 4.6.: Settings Class Diagram (get/set operations hidden).



Listing 4.2: Logic->process

```
$params = getopt($options);
$settings = Settings::getInstance();
$settings->setDirName($params['d']);
$settings->setPattern('/(?Ui)' . $params['p'] . '/');
if (array_key_exists('u', $params)) {
    $params['u'] = strtolower($params['u']);
    if (!in_array($params['u'], $settings->getUniverses()) &&
        $params['u'] != 'path-dependent') {
        throw new Exception('Invalid universe parameter');
    }
    $settings->setUniverse($params['u']);
} else {
    $settings->setUniverse('path-dependent');
}
$settings->setRecursive(array_key_exists('r', $params));
$settings->setFileRemoval(array_key_exists('R', $params));
$settings->setTempDbName($settings->getTempDbName() .
                         $settings->getRandomString());
```

Database Abstraction

The connection to the database is maintained through the `Database` class. The constructor of the class takes the database server's information as parameters and connects to it. The connection ID is saved in a private attribute. All subsequent method calls use this ID to direct queries to the connection associated with the object. This enables connecting to several databases in a parallel fashion by instantiating a `Database` object for each.

The class includes functions for escaping strings to prevent SQL injection attacks, starting and ending transactions, getting the last inserted ID and the number of rows affected by a query (Figure 4.7). There are two separate functions for writing and reading queries. `Database->query` returns a boolean value indicating the success of a writing query unless an exception is thrown. The function for reading queries, `Database->execute`, returns an instance of the `Record` class containing the result set. The returned object allows iterating over all rows, selecting specific fields and generating associative arrays (Figure 4.8).

Figure 4.7.: Database Class Diagram.

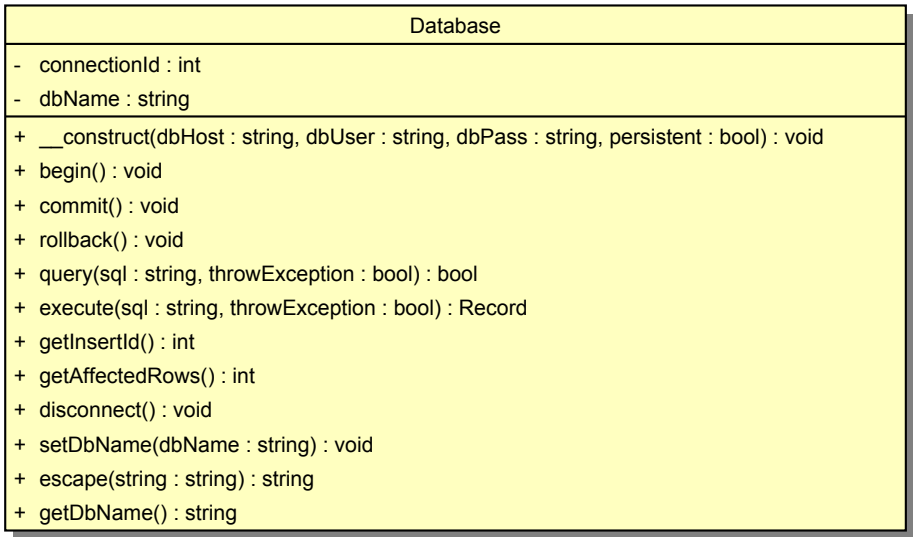
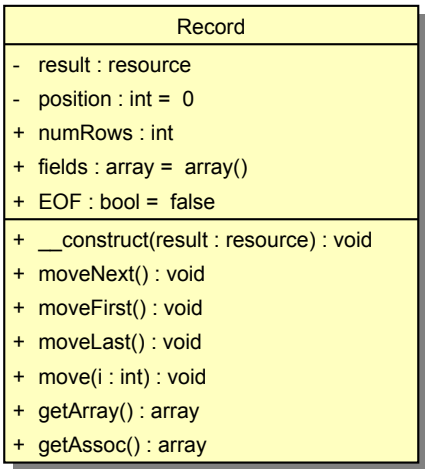


Figure 4.8.: Record Class Diagram.



Data Aggregation

Data is mined from backup files that are defined in the settings *directory*, *file pattern* and *recursive*. A directory can be given in absolute form or relative to the application's root directory. The file pattern is a perl–compatible regular expression. File names matching the pattern are included. Recursive is a boolean value which decides if sub–directories should be parsed as well.

The `backup` package contains the `File` class as well as several subclasses, see Figure 4.9. The `Directory` class is a special type of `File` inheriting the `name`, `path` and `fullPath` attributes but also storing a list of files and directories. These lists are populated in `Dir->scanDir` as shown in Listing 4.3. The list of files is filled with instances of the `File` class or any of its subclasses. Line 10 fills the array of directories with new `Dir` objects if recursive parsing is configured.

Listing 4.3: Snippet of Dir->scanDir

```
1  $dirContent = scandir($this->fullPath);
2  foreach ($dirContent as $key => $content) {
3      $path = $this->fullPath . DIRECTORY_SEPARATOR . $content;
4      if (is_file($path)) {
5          if (preg_match($this->pattern, $content)) {
6              ...
7              $this->files[] = $file;
8          }
9      } elseif ($this->recursive && is_dir($path)) {
10         $this->directories[] = new Dir($path, $this->recursive,
11                                        $this->pattern);
12     }
13 }
```

The `Logic` class coordinates the whole data mining operation (Figure 4.10). After processing any command–line parameters a `Dir` object is instantiated. Its constructor then calls the `scanDir` method to populate its attributes. This object is subsequently passed on to the `Logic->mineDirectory` function. This function starts the mining process on each of the directory's files and calls itself recursively on any directories it finds — see Listing 4.4 line 16.

Listing 4.4: Logic->mineDirectory

Figure 4.9.: Class Diagram of the backup package.

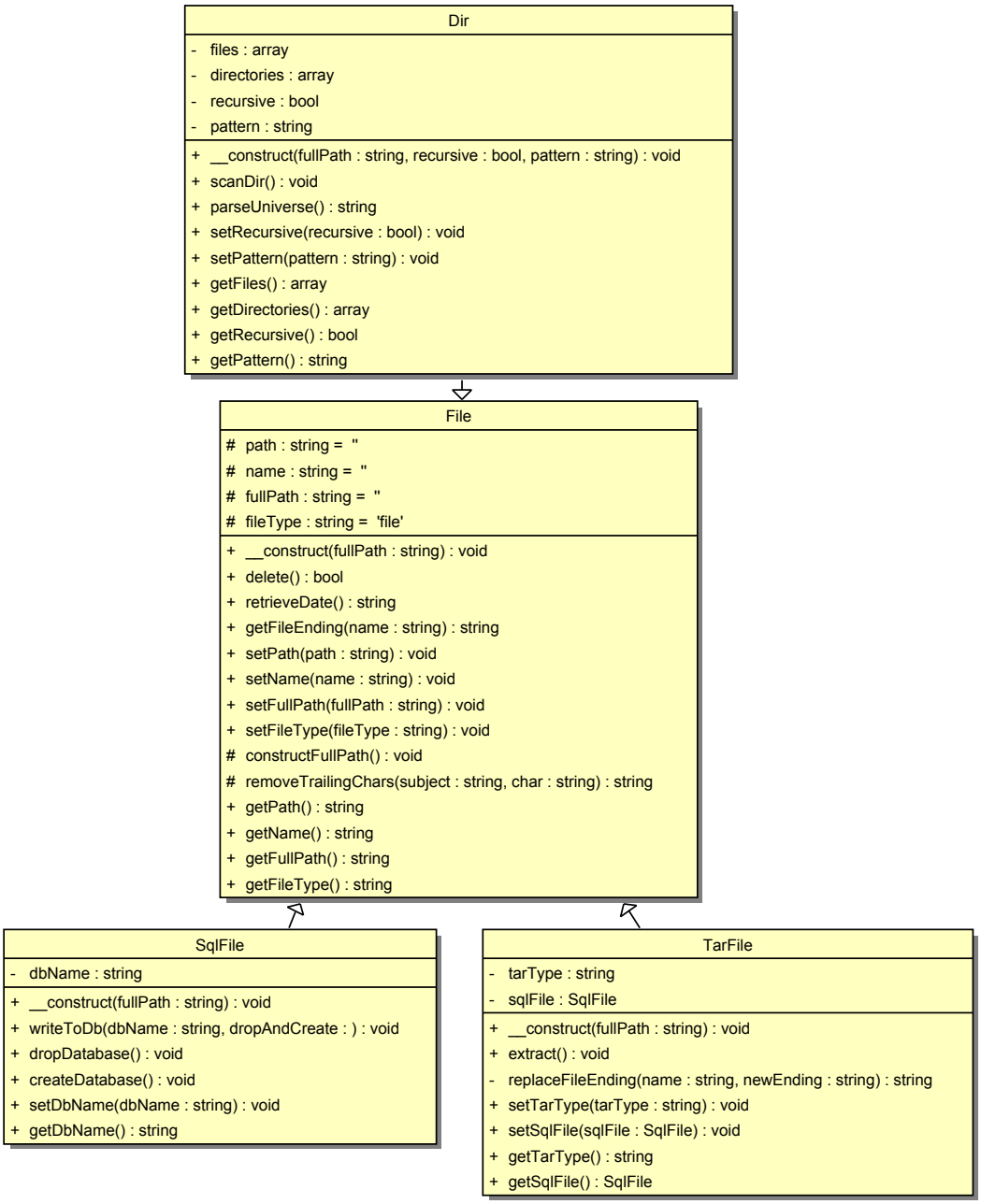
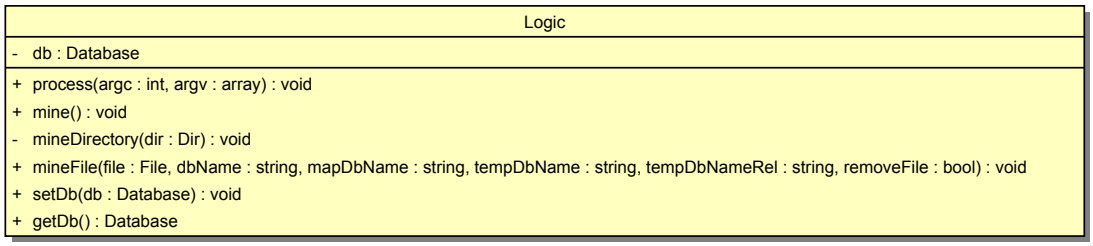


Figure 4.10.: Logic Class Diagram.



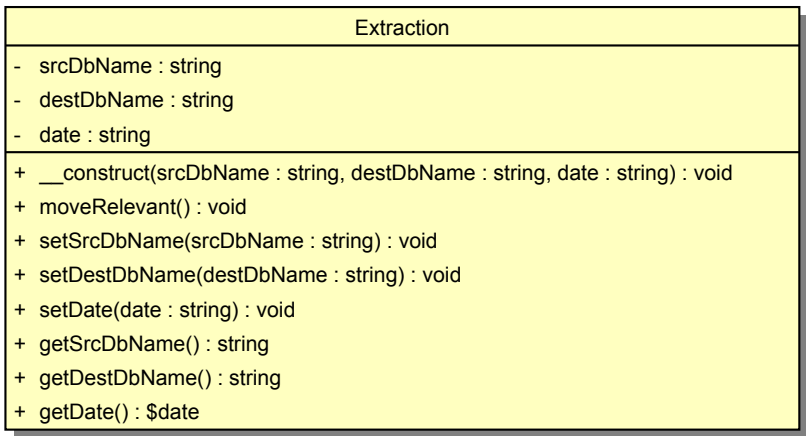

```
1  /**
2   *  Processes files for data mining recursively
3   *  @param Dir $dir
4   */
5  private function mineDirectory($dir)
6  {
7      ...
8      foreach ($files as $file) {
9          $this->mineFile($file, ...);
10     }
11     // recursively do the same for subdirectories
12     $directories = $dir->getDirectories();
13     foreach ($directories as $directory) {
14         $this->mineDirectory($directory);
15     }
16 }
```

4.3.2. Data Extraction

The backup files are daily database dumps from *Pardus* in SQL format. The data mining application supports both raw SQL files as well as *gzip*, *bzip2* and *lzma* compressed *tar* archives. In case of an archive the SQL file is temporarily extracted. The original or extracted SQL file is then written to a temporary database. The name of the database is randomly generated. Care is taken that it constitutes a unique identifier to support several mining operations simultaneously. The data extraction from that database is handled by the `Extraction` class displayed in Figure 4.11.

To remain flexible and adapt to database changes easily SQL code is put in external files wherever possible. This way database changes only require simple SQL file adjust-

Figure 4.11.: Extraction Class Diagram.



ments instead of a manipulation of the PHP code. Such an SQL file is used to transfer the relevant data set to a new temporary database. Listing 4.5 illustrates the transfer of the *trade_log_eq* table. Upper–case strings preceded and followed by __ are replaced by the application. The modified SQL code is saved in a new file and executed by *MySQL* to fill the new database. Listing 4.6 shows the PHP code responsible for the process. Replacing the __ placeholders sets the correct source and destination database and also allows to be considerate of intermittent database changes as mentioned in section 2.4.

Listing 4.5: Extraction SQL Code: trade_log_eq

```
1  __TRADE_LOG_EQ__ INSERT INTO `__DEST_SCHEMA__`.`trade_log_eq`
2      SELECT `trade_id`, `player_id`, `owner_id`, `kind`, `price`,
3      __REPAIR_PROFIT__, UNIX_TIMESTAMP(`date`) AS `date`
4      FROM `__SRC_SCHEMA__`.`trade_log_eq`
5      WHERE `player_id` > __PLAYERID_LIMIT__
6      AND (`owner_id` > __PLAYERID_LIMIT__
7      OR `owner_id` IS NULL);
8  -- The index is created later so it does not need to be updated
9  -- with each new row inserted above.
10 ALTER TABLE `__DEST_SCHEMA__`.`trade_log_eq`
11     ADD PRIMARY KEY (`trade_id`);
```

Listing 4.6: Extraction->moveRelevant

```
1  /**
2   * Moves relevant data to new schema
```

```
3   */
4  public function moveRelevant()
5  {
6      $recordTimestamp = strtotime($this->date . ' ' . '00:00:00');
7      $settings = Settings::getInstance();
8      // prepare the SQL statements
9      $sql = file_get_contents('sql/extract.sql');
10     $sql = str_replace('__DEST_SCHEMA__', $this->destDbName,
11                        $sql);
12     ...
13     // trade_log_eq is empty until November 13th 2005
14     $tradelogeqImpl = strtotime('2005-11-13 05:30:00');
15     if ($recordTimestamp <= $tradelogeqImpl) {
16         $sql = str_replace('__TRADE_LOG_EQ__', '--', $sql);
17     } else {
18         $sql = str_replace('__TRADE_LOG_EQ__', '', $sql);
19     }
20     ...
21     // save the modified SQL file
22     $tempSqlFile = $settings->getTempSqlDir()
23                        . '/' . $this->destDbName . '.sql';
24     file_put_contents($tempSqlFile, $sql);
25     // execute SQL
26     $sqlFile = new SqlFile($tempSqlFile);
27     $sqlFile->writeToDb($this->destDbName, true);
28     $sqlFile->delete();
29 }
```

Execution of the SQL code is done by an external call to the *mysql* binary, redirecting the SQL file's content to *mysql* as input. The data mining application blocks until an exit status code is returned. If the exit code is anything other than zero an exception is thrown and the application aborts. If the call was successful the database now containing the relevant data set is passed on to the anonymizing facility.

4.3.3. Anonymization

The extracted data set presented in Figure 4.1 includes identifying elements listed in Table 4.1.

Table	Field
message	message
message	subject
player	character_name
player	player_id
alliance	alliance_id

Table 4.1.: Table columns to anonymize

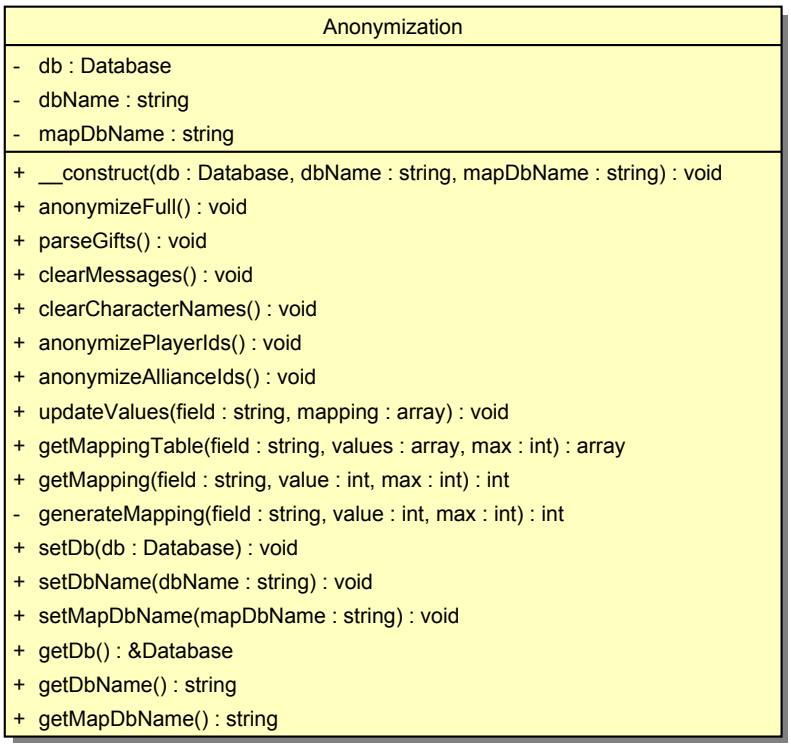
There are two types of messages, *private* messages and automatically generated *system* messages. A *private* message is sent from one person to another, while a *system* message has a sender value of `NULL`. The latter is important because gifts are logged only by the receipt of *system* messages. Such a message includes the the donor's character name and the contents of the gift. To anonymize that data the *gift* table shown in Figure 4.1 is created and populated by parsing the *message* table. This first step of the anonymization process is followed by dropping all *system* messages and clearing the messages' contents in the *message* and *subject* fields. The relevant information for network research remains, namely who sent a message to who and at what time. The *character_name* field of the *player* table was used to retrieve a donor's *player_id* and can simply be dropped as well.

The *integer* fields *player_id* and *alliance_id* must be anonymized while keeping both relations between the tables and between different backups intact.

One way would be using a cryptographic hash function on an ID. However, this could be easily traced back when the function is found out, simply by using the function on all *integer* numbers. This problem would be eliminated by adding salt to the cryptographic hash function. To make it more secure a different salt would have to be used for each ID. Two problems arise from that method. First, the salt would have to be safely stored for each value. Second, the value returned from the hash function would have to be sufficiently big to contain only unique values. In case of MD5, this would result in a 128–bit value used as primary key and foreign key in many tables. Besides the additional storage space required, *JOIN* operations on character fields are far more expensive.

A better way is to use a completely random mapping between *integer* values. The tables storing the original IDs and their mapped values are kept in a private database illustrated in Figure 4.2. Only the anonymization process has access to this database. There is a primary key on the original value to prohibit multiple mappings of the same ID. Another unique index is set for the mapped value to prevent more than one ID being mapped to

Figure 4.12.: Anonymization Class Diagram.



the same random value.

All of that is implemented in the `Anonymization` class (Figure 4.12). An `Anonymization` object is instantiated with parameters containing the name of the table holding the extracted data and the name of the private mapping database. The SQL commands responsible for changing all primary keys according to the mapping lie in external files again, one for the *player_id* and one for the *alliance_id*. The content of the file for anonymizing the *player_id* is shown in Listing 4.7.

Listing 4.7: Anonymization SQL Code: player_id

```
1 UPDATE `player` SET `player_id` = __MAPPED_TO__
2     WHERE `player_id` = __ORIGINAL__;
3 UPDATE `diplomacy` SET `player_id` = __MAPPED_TO__
4     WHERE `player_id` = __ORIGINAL__;
5 UPDATE `diplomacy` SET `player_dipl_id` = __MAPPED_TO__
6     WHERE `player_dipl_id` = __ORIGINAL__;
7 UPDATE `message` SET `recp_id` = __MAPPED_TO__
8     WHERE `recp_id` = __ORIGINAL__;
```

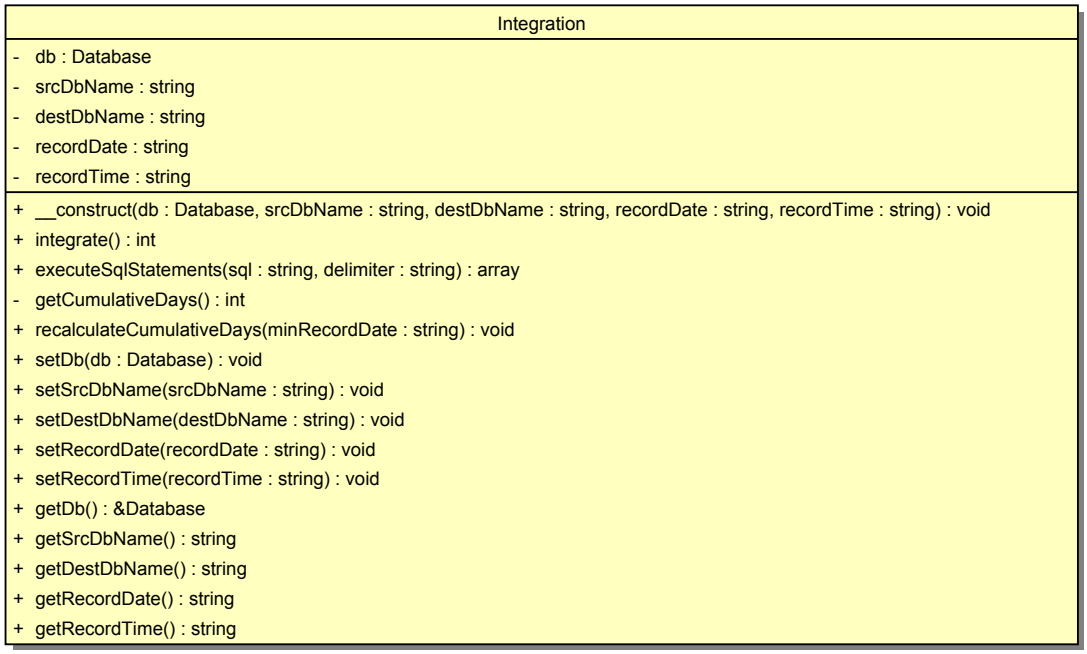
```
UPDATE `message` SET `sender_id` = __MAPPED_TO__
    WHERE `sender_id` = __ORIGINAL__;
...
UPDATE `trade_log_eq` SET `player_id` = __MAPPED_TO__
    WHERE `player_id` = __ORIGINAL__;
UPDATE `trade_log_eq` SET `owner_id` = __MAPPED_TO__
    WHERE `owner_id` = __ORIGINAL__;
UPDATE `gift` SET `player_id_from` = __MAPPED_TO__
    WHERE `player_id_from` = __ORIGINAL__;
UPDATE `gift` SET `player_id_to` = __MAPPED_TO__
    WHERE `player_id_to` = __ORIGINAL__;
```

As explained in section 4.2.1 foreign keys do not formally exist in the database that is operated on. This is why the single query on the *player* table does not suffice and all tables referring to the ID have to be modified accordingly. Also, in contrast to the previous SQL files, this one's content has to be executed for each ID that is anonymized. The implementation is shown in the code snippet in Listing 4.8.

Listing 4.8: Anonymization->updateValues

```
/**
 * Updates database fields according to a given mapping
 * @param string $field
 * @param array  $mapping
 */
public function updateValues($field, $mapping)
{
    // use SQL file as template for each update
    $pattern = file_get_contents('sql/anonymize_' . $field
                                 . '.sql');
    $sql = '';
    foreach ($mapping as $original => $mapped_to) {
        $updateString = $pattern;
        $updateString = str_replace('__MAPPED_TO__', $mapped_to,
                                    $updateString);
        $updateString = str_replace('__ORIGINAL__', $original,
                                    $updateString);
        $sql .= $updateString;
    }
    // save the modified SQL file
    $settings = Settings::getInstance();
```

Figure 4.13.: Integration Class Diagram.




```
22      $tempSqlFile = $settings->getTempSqlDir() . '/' .
23                     $this->dbName . '_anonymize_' . $field .
24                     '.sql';
25      file_put_contents($tempSqlFile, $sql);
26      // execute SQL
27      $sqlFile = new SqlFile($tempSqlFile);
28      $sqlFile->writeToDb($this->dbName, false);
29      $sqlFile->delete();
30  }
```

Anonymization of the *alliance_id* is handled using the same function, only substituting the included SQL file and using its own mapping table.

4.3.4. Data Integration

The final step consists of creating a new record in the researcher's database and transferring the data. This is done in the `Integration` class (Figure 4.13). A new `Integration` object is instantiated with the database connection object, the names of the source and des-

tination databases as well as the backup's date and time as parameters. Its `integrate() : int` method starts the integration. Again an external SQL file is used to allow easy adaption to any database changes, see Listing 4.9.

Listing 4.9: Integration SQL Code Snippet

```
1  INSERT INTO `__DEST_SCHEMA__`.`player`
2      SELECT __RECORD_ID__ AS `record_id`, `player_id`,
3      IF(`alliance_id` IS NULL, NULL, __RECORD_ID__)
4      AS `alliance_record_id`, `alliance_id`,
5      `credits`, `sex`, `user_type`, `turns_used`,
6      `experience`, `faction_id`, `syndicate_id`,
7      `repl`, `rep2`, `rep3`, `last_online`, `removed`
8      FROM `__SRC_SCHEMA__`.`player`;
9  INSERT IGNORE INTO `__DEST_SCHEMA__`.`trade_log`
10     SELECT `trade_id`, __RECORD_ID__ AS `player_record_id`,
11     `player_id`, IF(`owner_id` IS NULL, NULL, __RECORD_ID__)
12     AS `owner_record_id`, `owner_id`, `resource_id`,
13     `kind`, `amount`, `price`, `total`, `blackmarket`, `date`
14     FROM `__SRC_SCHEMA__`.`trade_log`;
15 ...
```

The `__RECORD_ID__` placeholder is substituted with the record ID of the newly created record. Static logs like the *trade_log* are only inserted if the corresponding ID is not yet in the database, as seen in line 9 of Listing 4.9. To ensure all operations including the creation of the record entry as well as the commands in the SQL file are done fully isolated in one atomic action, this file is not just forwarded to the *mysql* binary. A transaction is started with the creation of the record entry. Each query is then extracted from the file and executed directly through the `Database` object. If no errors occur the transaction is committed. During the integration process — which can take several minutes — the new record and all of its data is invisible to any other transactions. This level of consistency is realized through a serializable mode of isolation.

4.3.5. Web Front-end

The web application can be used in combination with any FCGI enabled web server. Its code follows the Model View Controller (MVC) software pattern to isolate the presentation layer from business logic. The database itself and the `User` class serve as model, the `Template` class exclusively returns output while the `Controller` class handles input

(Figure 4.14). All incoming requests go through *index.php* which uses the *view GET* parameter to decide which data to forward to the `Template` and `Controller` objects.

As can be seen in the ERD in Figure 4.4 access is permission–based. A user has to authenticate by providing a correct user name / password combination. The password is saved as 32 bytes MD5 hexadecimal character string. Upon logging in the MD5 function is used on the entered password and compared with the value stored in the database. Once logged in the user is authorized to use certain sections. The level of authorization depends on the user's group. Access permissions can be configured for the querying, views, import and export sections.

For users to stay logged in a random 32–bytes session ID is created and stored both in the database and as a session cookie. The session cookie expires as soon as the browser is closed. By reading the value of the session cookie the user can be identified transparently with each request.

The web server is run under the same user as the owner of the source files. Directories for log files and exported files are created. Their permissions are `0700` equaling read, write and execution rights for the owner. Any critical operations like failed login attempts are logged. Database access of the web application is limited to the main and user databases. The latter connection is dropped after logging in.

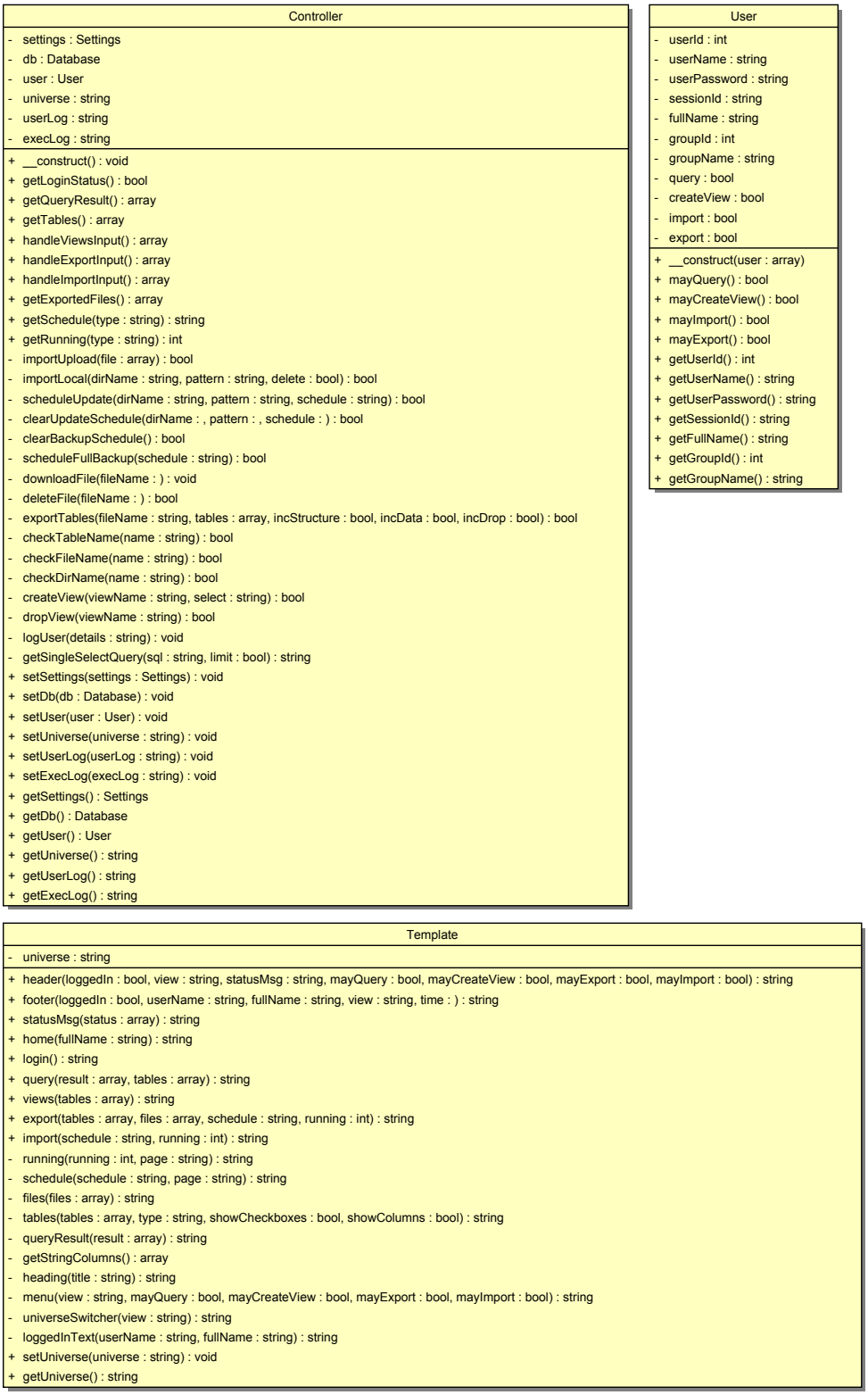
Routines

A *Query* section exists, supporting all kinds of *SELECT* SQL queries on the tables and any existing views. Submitted input is handled by the `Controller` object which discards anything but a single *SELECT* query. Even though the database user's permissions are only valid for the main database, it is checked again if only this database is queried. If any checks fail the request is ignored. In case the SQL syntax is incorrect an exception including the *MySQL* error message is thrown in the `Database` object. If the query is successful the result set is printed in a sortable table. The controller function handling input in the *Query* tab is shown in Listing 4.10.

Listing 4.10: Controller->getQueryResult

```
1 /**
2  * Executes SELECT query and returns results if any
3  * @return array
4  */
```

Figure 4.14.: Class Diagram of the www package.



```
5  public function getQueryResult()
6  {
7      if (!isset($_POST['query'])) {
8          return NULL;
9      }
10     $sql = $this->getSingleSelectQuery($_POST['query']);
11     if (!$sql) {
12         return NULL;
13     }
14     $this->logUser('query: ' . $sql);
15     $this->db->setDbName($this->settings->getDbName() .
16                          '_' . $this->universe);
17     $records = $this->db->execute($sql);
18     $result = $records->getArray();
19     return $result;
20 }
```

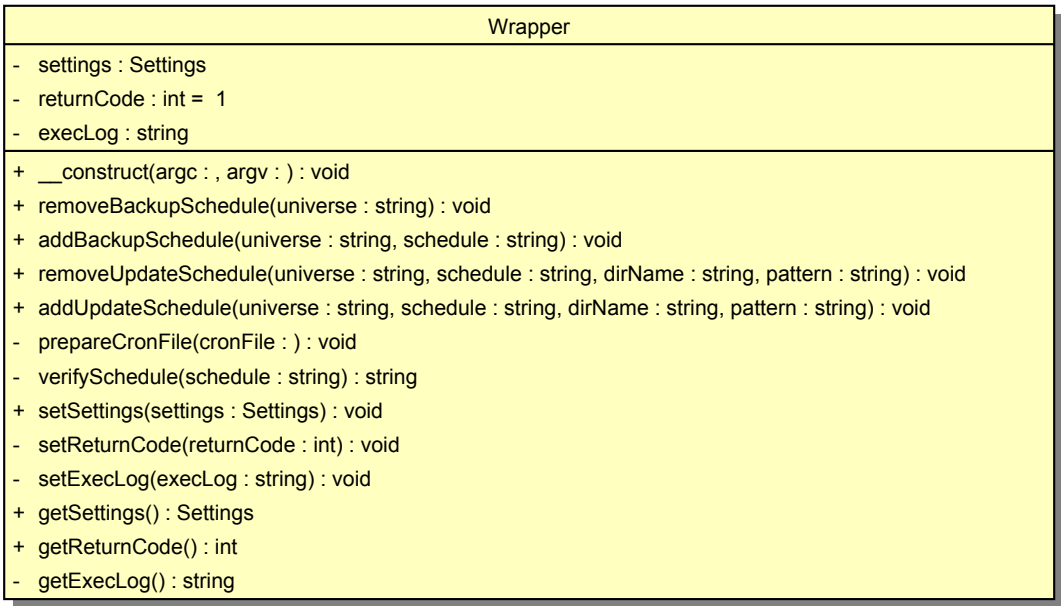
In the *Views* section views can be created and dropped, provided the user is permitted to. Creating a view works almost the same as before — a *SELECT* SQL statement and a name for the view are specified. The statement is again checked by the `getSingleSelectQuery(sql : string, limit : bool) : string` method. The name of the view must be clean of "`;`" characters since they can provide a way for an SQL injection attack by interrupting a *valid* statement and starting a new malicious one. If all checks are successful the view is created and can immediately be used for querying.

Export

The *Export* section offers functions to export the whole database or subsections of it. Exports, or "backups", can be made in SQL, CSV and XML format. To create the backup the web front–end calls the *mysqldump* executable in a non–blocking fashion. Its parameters make sure the exported data set is consistently done within a single transaction. The resulting backup file is created and written to in a temporary location. Once finished it is moved to the *export* directory and listed on the *Export* page. It can then be either downloaded or deleted again.

The user is also able to define a schedule for automatic exports. These are scheduled by the application by using a *crontab*. *Crontabs* are files that are parsed by *cron*, a *Unix* job scheduler. Commands specified in the *crontabs* are executed by *cron* according to

Figure 4.15.: Wrapper Class Diagram.



the parsed schedules. The web application uses */etc/cron.d/datamining* to specify any scheduled exports (or imports). However, this file has to be owned by *root*. To provide a secure solution to writing this file as *root* a command wrapper with *sudo* permission is used. The *sudo* command lets users execute certain permitted commands as *root*. The wrapper is implemented as standalone program in the class `Wrapper` (Figure 4.15) in the package `main`.

When a schedule is entered, the wrapper command along with the schedule is used as parameter for the *sudo* binary. The *sudo* program verifies that the web server's user is allowed to run the wrapper as *root*. The wrapper then runs under *root* permissions. It verifies the given schedule again, builds the export command and eventually modifies the *crontab* accordingly. If the verification fails an exit code not equaling zero is returned. The exit code is waited for and used by the web application to inform the user of success or failure. Clearing the schedule is done the same way, only supplying the wrapper with different parameters — the corresponding code segment is shown in Listing 4.11.

Listing 4.11: Controller->clearBackupSchedule

```
1 /**
2  * Calls external wrapper to clear backup Cron jobs
3  * @return bool
```

```
4   */
5   private function clearBackupSchedule()
6   {
7       $srcDir = realpath('./main/');
8       $cmd = 'cd ' . $srcDir . ' && ' .
9              'sudo ' . $this->settings->getPhpExec() .
10             ' -f Wrapper.php -H -- backup remove ' .
11             $this->universe;
12      system($cmd, $retval);
13      if ($retval == 0) {
14          $this->logUser('clear export schedule');
15          return true;
16      }
17      return false;
18  }
```

Import

The *Import* section serves as graphical front–end for the CLI data mining tool. Files can be imported in two ways. First, *Pardus* backups residing on the web host can be directly imported by calling the CLI binary. Second, a *Pardus* backup file can be uploaded from the remote computer. The directory and escaped name of the uploaded file is used as *directory* and *pattern* parameters for the CLI tool to match exactly that single file. After the file is imported it is removed again.

The typical code structure used to generate the output for web pages in the `Template` class is shown for the *Import* section in Listing 4.12.

Listing 4.12: Template->import

```
1   /**
2    *  Displays the import page
3    *  @param string $schedule
4    *  @param int    $running
5    *  @return string
6    */
7   public function import($schedule, $running)
8   {
9       $scheduleStr = $this->schedule($schedule, 'import');
10      $runningStr = $this->running($running, 'import');
```

```
11      $dirName = isset($_POST['dirname']) ? $_POST['dirname'] :
12                 '/backups/pardus/' . $this->universe;
13      $pattern = isset($_POST['pattern']) ? $_POST['pattern'] :
14                 '\d{6}\-\d{4}';
15      $when = isset($_POST['when']) ? $_POST['when'] : '55 7 * * *';
16      $out = $this->heading('IMPORT');
17      $out .= <<<TEMPLATE
18  ...
19  TEMPLATE;
20      return $out;
21  }
```

Automatic imports at scheduled times are possible through this interface as well. A path to import backups from, a pattern and a schedule have to be provided. All parameters are again verified and set with the use of the external wrapper. Listing 4.13 shows the wrapper function adding an import entry to the schedule.

Listing 4.13: Wrapper->addUpdateSchedule

```
1  /**
2   * Adds cron schedule entry for updates
3   * @param string $universe
4   * @param string $schedule
5   * @param string $dirName
6   * @param string $pattern
7   */
8  public function addUpdateSchedule($universe, $schedule,
9                                     $dirName, $pattern)
10 {
11      $schedule = $this->verifySchedule($schedule);
12      if ($schedule == NULL) {
13          $this->setReturnCode(1);
14          return;
15      }
16      $dirName = escapeshellarg($dirName);
17      $pattern = escapeshellarg($pattern);
18      $importCmd = './run.sh -d ' . $dirName .
19                   ' -p ' . $pattern .
20                   ' -u ' . $universe . ' -R';
21      $cmd = 'bash -c "cd ' . realpath('.') . '; ' .
22             $importCmd . '" >> ' . $this->execLog .
```

```
23              '␣2>&1␣&';
24      $user = $this->settings->getUser();
25      $cronFile = $this->settings->getCronFile();
26      $this->prepareCronFile($cronFile);
27      // add entry
28      $cronEntry = "\n" . $schedule . '␣' . $user .
29                   '␣' . $cmd . "\n";
30      file_put_contents($cronFile, $cronEntry, FILE_APPEND);
31      $this->setReturnCode(0);
32  }
```

Chapter 5.

Application

5.1. General Usage

5.1.1. Research

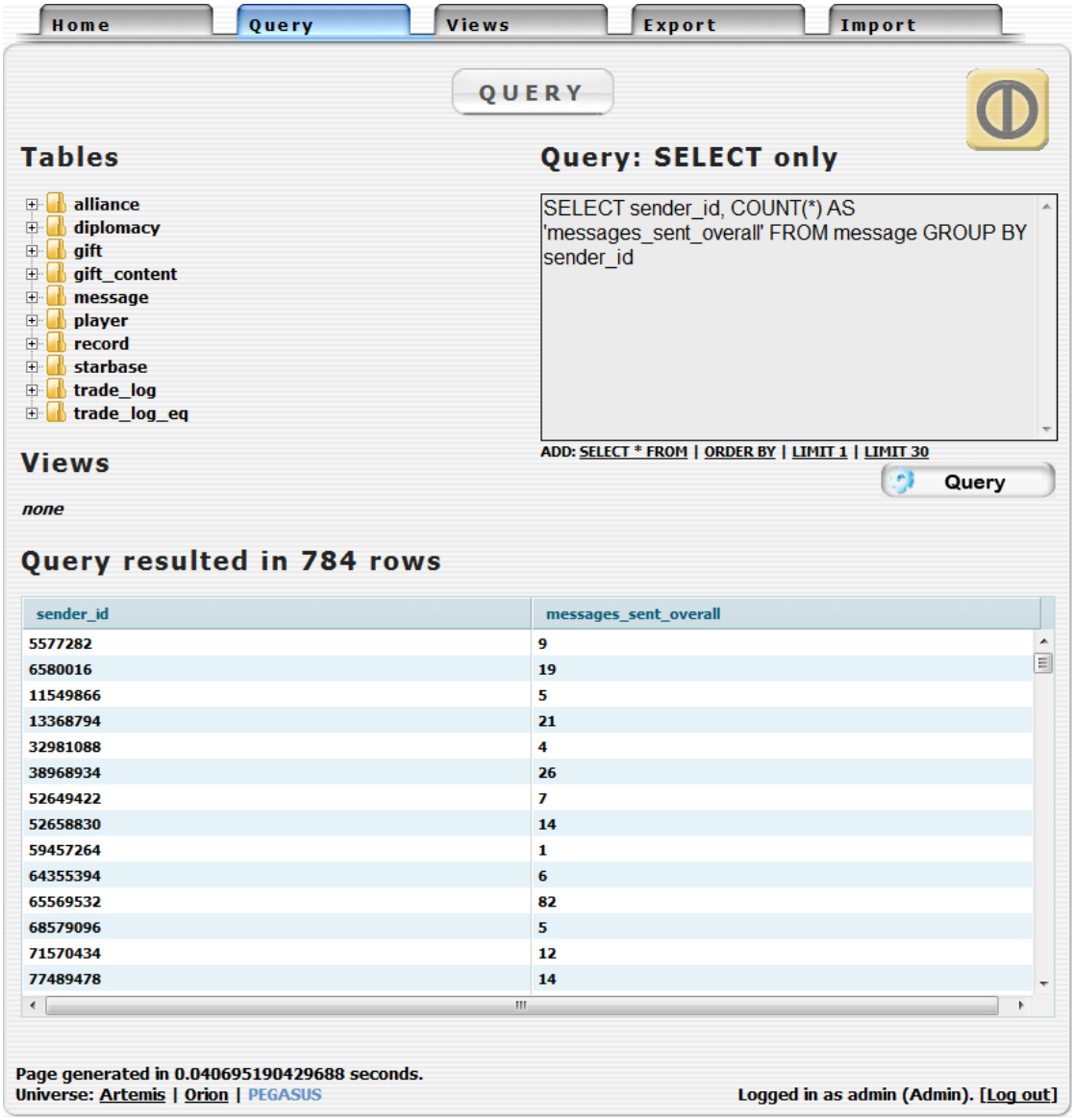
The *Query* tab of the web application forms the basis of access to the research data. It is illustrated in Figure 5.1. The query box supports any kind of *SELECT* statement that is supported by *MySQL*[1], inluding nested queries, joins and unions. Below the query box extensible shortcuts for the most common queries can be found.

On the left side available tables, views and their fields are listed in a dynamic tree structure. The tree can be browsed and any value can be copied into the query box by a double–click. Results of the query are shown in a table below. The resulting rows can be sorted without the need for a page reload or modifying the SQL code, simply by clicking on a column heading. Failed queries result in an exception containing a detailed SQL error message.

The *View* tab offers advanced functionality for senior researchers. Again, available tables, views and their fields are displayed on the left side. In addition views can be removed. On the right side, text fields for the creation of a new view exist. A view is defined by its name and an SQL query that determines its content. Newly created views are ready for use in the *Query* tab immediately. Views help breaking down complexity and having certain forms of the result set available in an up to date fashion at all times. Views can also work as pre–created queries for junior researchers with little or no SQL experience.

[1] Version 5.4 is used in this implementation.

Figure 5.1.: Screenshot of the web application: Query view.



Minute	Hour	Day of Month	Month	Day of Week
0–59	0–23	1–31	1–12	0–6

Values are separated by one or more white–spaces.
''*'' fits any value.
Day of Week starts with Monday (0).
Values are connected by logical AND operators to determine execution.

Table 5.1.: Cron syntax

In the top right corner of all pages the selected universe is represented by its symbol. The bottom of a page offers any debugging information as well as links to switch the selected universe or to log out.

5.1.2. Administration

The data set can be administered through the web front–end by users with export / import permissions, or directly through the CLI tool if shell access is present.

Like in the previous sections, the *Export* tab shows a list of the available tables and views on the left side. Below exported files can be downloaded and deleted. The tables and views do not include fields as in the previous sections but feature checkboxes to select them for exporting. They can be exported into SQL, CSV and XML format. Only data, only structure or both may be included. Optionally, *DROP TABLE* SQL statements can be added. Once started the exporting process runs in the background. Several export processes can run simultaneously. The user is notified of how many are running at a time.

Figure 5.2 shows an exported file and one process still running. It also displays the schedule of automatic database backups. The schedule is defined using *Cron* syntax as explained in Table 5.1. A full database backup will be initiated at each time matching the schedule.

The *Import* tab allows the integration of new *Pardus* backups into the research database. The first item on the left side makes it possible to upload and integrate single *Pardus* backup files stored on the client machine. The second option is specifying a path on the web server's host and a pattern of files to match. The default pattern is *\d{6}\-\d{4}*. It would include files of any format as long as a 6–digits long date and a 4–digits long time

Figure 5.2.: Screenshot of the web application: Export view.

Switch	Value	Description	Example	Required
-d	directory path	Directory to parse	/datamining	X
-p	pattern	File pattern to search for	\.sql	X
-u	universe	Backup's universe	orion	X
-r		Search directories recursively		
-R		Remove parsed files		

Table 5.2.: Command–line parameters

is part of the file name. The specified directory is searched for matching files in a non–recursive way. An option exists to delete any successfully integrated backup files after the operation.

The right side of the page displays the currently configured schedule as well as an interface to set a new schedule for automated import processes. Like previously, besides the schedule in *Cron* syntax a path and a pattern have to be specified. This function especially makes sense with an automatic retrieval of new backups. Successfully imported files are automatically deleted to avoid attempts of importing the same backups multiple times.

For further control the CLI tool can be called directly, assuming shell access to the data mining server is present. The available command–line parameters are listed in Table 5.2. If the recursive option is specified and the univere's value is set to ''path–dependent'', backups of all universes that are sorted in accordingly named directories can be integrated at once.

Aside from the command–line parameters system–wide configuration constants can be edited in *settings/Constants.php*. Default values as well as more advanced settings can be adjusted in the `Settings` class. HTML documentation of all classes and their members is found in the *doc* directory. The documentation is generated directly from the source code and can be refreshed or exported into a different format by the use of *phpdoc*.

5.2. Scientific Routines: An example

First research using the implemented data mining application has been done in [ST09]. An excerpt of some of the findings along with the process of getting these results is presented next.

5.2.1. Preferential Attachment

One hypothesis of social network dynamics is that the higher the degree of a node, the higher the probability of a ''newcomer'' to attach to the network through it than through lower–degree nodes. The degree is the amount of links connecting a node. A ''newcomer'' is considered a node attaching to the network for the first time. This model is called preferential attachment (PA) [BA99].

A node can be represented by a player. Links can either be private messages sent between two players, or the marking of a friend or foe. The steps to retrieve the required information using the data mining front–end are shown in Listing 5.1. The returned result–set can either be directly utilized through a *MySQL* library, for example in *Matlab*, or exported to a universally usable format. The networks in Figure 5.3 have been created using *Pajek*.

Listing 5.1: SQL queries to retrieve preferential attachment data

```
1  -- All of the following can be done in the web client's query tab.
2  -- Random players existing at all times are chosen.
3  -- For repetitive use of random IDs a view can be created.
4  SELECT pl.player_id FROM player pl JOIN player p2 ON
5          (pl.player_id = p2.player_id) WHERE pl.record_id =
6          (SELECT record_id FROM record WHERE day_cumulative = 1)
7          AND p2.record_id = (SELECT record_id FROM record
8          WHERE day_cumulative = 445) AND p2.removed = 0
9          ORDER BY RAND() LIMIT 78;
10 -- (a) Weighed PM communication between them is calculated.
11 SELECT DATE_FORMAT(FROM_UNIXTIME(m.sent), '%Y-%m-%d') AS
12         'day', m.sender_id, m.recp_id, COUNT(m.sender_id) AS
13         'messages_sent' FROM message m WHERE m.sent >=
14         (SELECT UNIX_TIMESTAMP(record_date) FROM record
15         WHERE day_cumulative = 1) AND m.sent <=
16         (SELECT UNIX_TIMESTAMP(record_date) FROM record
17         WHERE day_cumulative = 445) AND m.sender_id IN
18         (result from previous query delimited by commas) AND
19         m.recp_id IN (...)
20         GROUP BY day, m.sender_id ORDER BY day ASC
21 -- (b) All friendship(/foe) relations between them are selected.
22 SELECT r.day_cumulative, d.player_id, d.player_dipl_id
23         FROM diplomacy d JOIN record r ON (d.record_id =
24         r.record_id) WHERE d.player_id IN (result from previous
```

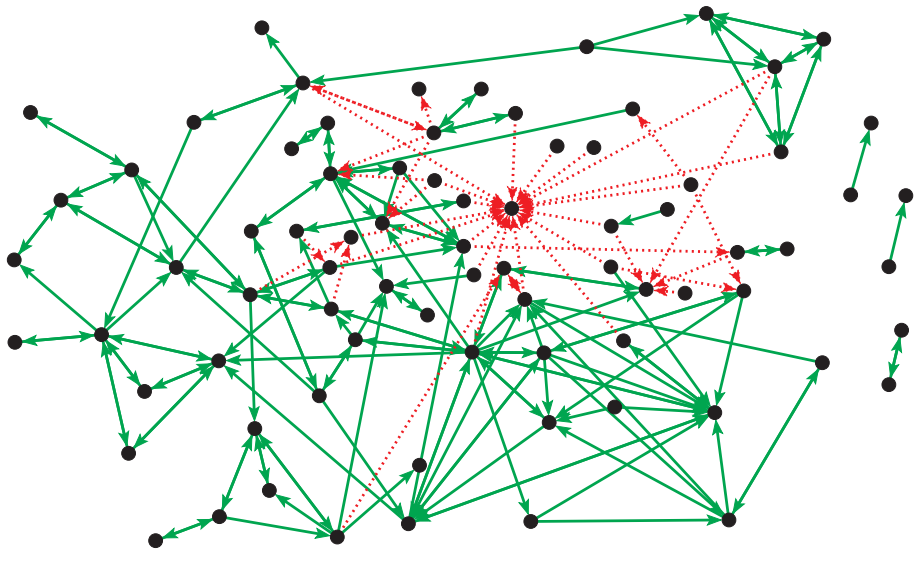
```
25          query delimited by commas) AND d.player_dipl_id IN (...)
26          AND r.day_cumulative >= 1 AND r.day_cumulative <= 445
27          AND d.dipl_type = 'friend'('foe')
28          ORDER BY r.day_cumulative ASC, d.player_id ASC;
```

[ST09] concludes with the following findings:

> ''In the classic model of PA it is assumed that the probability P of a newcomer connecting to an existing node n_i with in-degree k_i^{in} is $P(k^{\text{in}}) \propto \left(k^{\text{in}}\right)^{\alpha}$ with $\alpha = 1$. Figure 5.4 shows $P(k^{\text{in}})$ versus k^{in} for friend and enemy networks; all link events between newcomers and their destinations have been used from day 200 to 400. Least squares fits in double-logarithmic scale yield an exponent of $\alpha = 0.62$ for friend markings with $k^{\text{in}} < 30$, and $\alpha = 0.90$ for all enemy markings. We observe an increased upward bending for players having in-degrees larger than about 100, i.e. for very popular players. These findings are fully consistent with other game universes and other time ranges (not shown).''

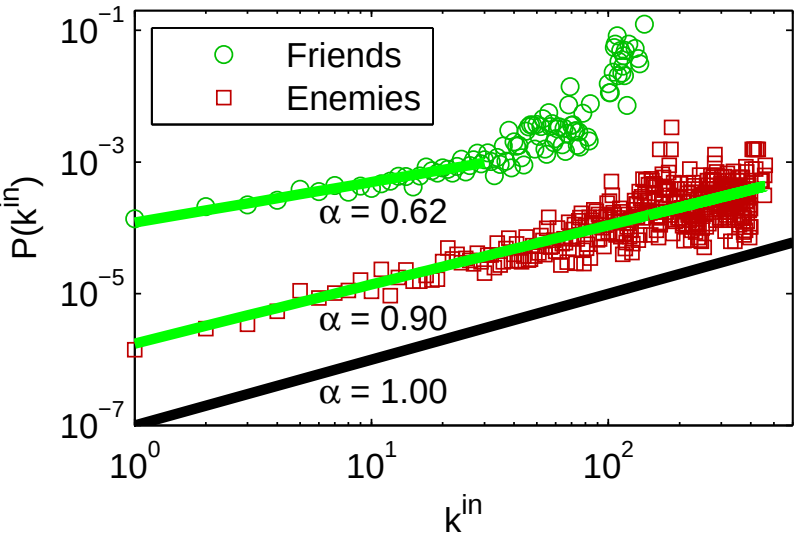
Figure 5.3.: Figures and description retrieved from [ST09]:
"(a) Accumulated PM communications over all 445 days between 78 randomly selected individuals who existed on the first and last day. Link colors of light gray, gray, and black correspond to 1–10, 11–100 and 101–1000 PMs sent, respectively.
(b) Friend (green, solid) and enemy (red, dashed) relations on day 445 between the same individuals. See our Youtube channel `http://www.youtube.com/user/complexsystemsvienna` for animated time evolutions of these networks."

(a)



(b)

Figure 5.4.: Figure and description retrieved from [ST09]:
"Empirical probability $P(k^{\text{in}})$ for newcomers connecting to nodes with in-degree k^{in}. Data is used between days 200 and 400. The black line depicts slope $\alpha = 1$ and indicates the linear dependence assumption needed in the PA model. Green lines denote least squares fits. Values for enemies are vertically displaced by a factor 0.1 for better visibility."



Chapter 6.

Concluding Remarks

6.1. Conclusion

Complications of gathering, storing and querying large volumes of data have been analyzed and addressed using data gathered over several years from the massive multiplayer online game *Pardus*.

Due to its high level of development and thus superiority in all evaluated areas, an RDBMS was chosen to serve as data back–end. Its ability to compress data and indices on–the–fly additionally reduced the hardware storage requirements. Outages and regular changes to the game and its data structure were addressed by outsourcing all direct database manipulations into external files. Backups were merged into a single database, allowing efficient querying as well as easy removal of duplicate information. To keep the possibility of researching unforeseen areas all relevant impersonal data was extracted from the backups practically unchanged. *Views* can be dynamically created, changed and removed at any time to provide special data compilations on an upper abstraction level — without changing the master data and without any waste of storage space.

In addition to the command–line extraction and integration tool a web front–end was developed. It offers researchers permission–based access to the database and administrative functions including manual as well as scheduled imports/updates and exports/backups of all or specified data.

Successful usage of the application is presented in [ST09]. This demonstrates large–scale and simultaneous usage of the high volume and variety of data output from a massive multiplayer online game is both possible and feasible.

6.2. Scientific Research

Use of this or similar data is valuable in nearly every field of study concerned with behavioral, social or group dynamics. Though the focus of this thesis has been slanted toward financial networking, even a first look at the existing Pardus data shows intriguing elements of group formation and evolution, social, financial and behavioral tendencies and patterns, friend and enemy networks, and much more — in both the game as a whole and in numerous micro–groups within.

In recent years there have been many efforts to gather varieties of this type of data using modern technology; including various social networking websites such as *Facebook* ([GWH07]) and cell phone networks ([OSH+07]). The *Pardus* data contains these same networks as they have evolved over years, and this data can continue to be gathered indefinitely with relatively little effort or expense; presenting a number of clear advantages over more traditional methods of data gathering. Data from an environment such as *Pardus* also provides researchers with a complete view of every aspect influencing players' virtual lives on a scale rarely seen in real–world studies.

6.3. Outlook

As technology, access to internet and popularity of multiplayer games grows steadily around the world, so does the opportunity to gather valuable data. A unique and important aspect of using a game environment, often lacking in the real world, is that the game environment can be manipulated to trigger specific events or phenomena, such as inflation, and the resulting behavior can easily be observed.

Along these same lines, the virtual environment as well as the overall design of the game can be specifically engineered to produce and record behavior, networks and patterns. Though *Pardus* provides a large volume of interesting data relevant to many fields of study, Pardus was not designed a priori with data mining in mind as the game's ultimate purpose.

6.3.1. New Virtual Environment

A massive multiplayer online game designed to replicate real–world systems could open up research in these areas to a degree never before imagined, with only a fraction of

the time and expense involved in real–world studies on a similar scale. Game mechanics could allow players to open banks, provide goods and services in dynamic, player–driven markets, trade stocks and other such real–world activities. Additionally anonymized data gathered about players when they sign up for the game could be expanded to include details such as level of education, average income, marital status and so forth.

A large virtual environment, such as exists now in Pardus, encourages natural grouping of players into micro communities and economies, allowing for study on both a macro and micro scale. This tendency could be further encouraged by providing players with similar ties to their community as can be found in the real world; such as dwellings and a localized trade which brings in income.

With data mining in mind as the game is developed, recording of all actions and interactions, both social and economic, in the game could be streamlined to be even more concise and efficient than the current *Pardus* system, along with ease of access to and protection of the data.

In closing, while existing methods such as agent–based models and massive multiplayer online games such as *Pardus* provide a wealth of valuable scientific data, a game designed from conception upward with data mining and subsequent research activities in mind could offer possibilities too great to ignore.

Appendix A.

Acknowledgements

I would like to thank Prof. Thurner for his guidance throughout the creation of this thesis.

I would also like to express my gratitude to Michael Szell for introducing me into the field of social network dynamics and providing me with data from his research.

Appendix B.

Bibliography

[AAS] Robert L. Axtell, Clinton J. Andrews, and Mitchell J. Small. Agent-based models of industrial ecosystems, `http://policy.rutgers.edu/andrews/projects/abm/abmarticle.htm`, retrieved in September 2009.

[ABMP07] Andrei Arion, Angela Bonifati, Ioana Manolescu, and Andrea Pugliese. Xquec: A query-conscious compressed xml database. *ACM Trans. Internet Technol.*, 7(2):10, 2007.

[Art05] W. Brian Arthur. Out-of-equilibrium economics and agent-based modeling. *Handbook of Computational Economics*, 2:1551–1564, 2005.

[BA99] A.L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509, 1999.

[Ber94] Elisa Bertino. Index configuration in object-oriented databases. *The VLDB Journal*, 3(3):355–399, 1994.

[Bon02] Eric Bonabeau. Agent-based modeling: Methods and techniques for simulating human systems. *PNAS*, 99:7280–7287, May 2002.

[Buc09] Mark Buchanan. Meltdown modeling. *Nature*, 460:680–682, 2009.

[Cas05] Edward Castronova. *Synthetic Worlds : The Business and Culture of Online Games*. University Of Chicago Press, 2005.

[CC04] Yangjun Chen and Yibin Chen. Signature file hierarchies and signature graphs: a new index method for object-oriented databases. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 724–728, 2004.

[DR01] Kurt W. Deschler and Elke A. Rundensteiner. B+ retake: sustaining high volume inserts into large data pages. In *DOLAP '01: Proceedings of the 4th ACM international workshop on Data warehousing and OLAP*, pages 56–63, 2001.

[dSGL09] M. de Smith, M. Goodchild, and P. Lenglay. *Geospatial Analysis*. Matador, 2nd edition, 2009.

[Ehr02] Norman Ehrentreich. The santa fe artificial stock market re-examined – suggested corrections, September 2002.

[Fag79] Ronald Fagin. Normal forms and relational database operators. In *SIGMOD '79: Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 153–160, 1979.

[FF09] J. Doyne Farmer and Duncan Foley. The economy needs agent-based modelling. *Nature*, 460:685–686, 2009.

[Fow94] Glenn Fowler. cql - a flat file database query language. In *USENIX Winter 1994 Conference*, 1994.

[GF99] S. Gächter and E. Fehr. Collective action as a social exchange. *Journal of Economic Behavior and Organization*, 39:341–369, 1999.

[GWH07] S. Golder, D. Wilkinson, and B. Huberman. Rhythms of social interaction: Messaging within a massive online network. In *Communities and Technologies 2007: Proceedings of the Third Communities and Technologies Conference*, 2007.

[HBB+05] J. Henrich, R. Boyd, S. Bowles, C. Camerer, E. Fehr, H. Gintis, R. McElreath, M. Alvard, A. Barr, J. Ensminger, et al. ''economic man'' in cross-cultural perspective: Behavioral experiments in 15 small-scale societies. *Behavioral and Brain Sciences*, 28:795–815, 2005.

[HH04] M. P. Haustein and T. Härder. Adjustable transaction isolation in xml database management systems. In *Database and XML technologies: Second International XML Database Symposium, XSym 2004*, pages 176–187, 2004.

[HLW94] Uwe Hohenstein, Regina Lauffer, and Petra Weikert. Object-oriented database systems: How much sql do they understand? In *DEXA '94:*

Proceedings of the 5th International Conference on Database and Expert Systems Applications, pages 15–26, 1994.

[Hol97] I. Holloway. *Basic Concepts for Qualitative Research*. Wiley-Blackwell, 1st edition, January 1997.

[HP03] Richard A. Hankins and Jignesh M. Patel. Effect of node size on the performance of cache-conscious b+-trees. In *SIGMETRICS '03: Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 283–294, 2003.

[IBM07] IBM. Innovation outlook 2007: Virtual worlds, real leaders, `http://www.ibm.com/ibm/gio/media/pdf/ibm_gio_gaming_report.pdf`, 2007.

[IEE02] The Open Group / IEEE. The single unix specification, `http://www.unix.org/single_unix_specification/`, January 2002.

[Kim02] Sang-Wook Kim. On batch-constructing b+-trees: algorithm and its performance evaluation. *Information Sciences*, 144:151–167, July 2002.

[KN01] Tomas B. Klos and Bart Nooteboom. Agent-based computational transaction cost economics. *Journal of Economic Dynamics & Control*, 25:503–526, 2001.

[Kul94] Krishna G. Kulkarni. Object-oriented extensions in sql3: a status report. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, page 478, 1994.

[Lea00] Neal Leavitt. Whatever happened to object-oriented databases? *Computer*, 33(8):16–19, 2000.

[NJ03] Matthias Nicola and Jasmi John. Xml parsing: a threat to database performance. In *CIKM '03: Proceedings of the twelfth international conference on Information and knowledge management*, pages 175–178, 2003.

[OEG04] Bayer & Szell OEG. Pardus, `http://www.pardus.at`, 2004.

[OSH+07] J.P. Onnela, J. Saramäki, J. Hyvönen, G. Szabó, M.A. de Menezes, K. Kaski, A.L. Barabási, and J. Kertész. Analysis of a largescale weighted network of one-to-one human communication. *New Journal of Physics*, 9:179, 2007.

[Ric05] Kurt Richardson. *Managing organizational complexity: philosophy, theory and application*. Information Age Publishing, 2005.

[Sch06] Robin Schumacher. A look at the mysql csv storage engine, `http://dev.mysql.com/tech-resources/articles/csv-storage-engine.html`, May 2006.

[SFGK03] E. Smith, J. Farmer, L. Gillemot, and S. Krishnamurthy. Statistical theory of the continuous double auction. *Quantitative Finance*, 3:481–514, 2003.

[SHS04] Gargi M. Sur, Joachim Hammer, and Jerome Simeon. An xquery-based language for processing updates in xml. In *PLAN-X 2004 - Programming Languages Technologies for XML*, 2004.

[ST09] Michael Szell and Stefan Thurner. Measuring social dynamics in a massive multiplayer online game. *Arxiv preprint 0911.1084v1*, November 2009.

[TH02] Pankaj M. Tolani and Jayant R. Haritsa. Xgrind: A query-friendly xml compressor. In *ICDE '02: Proceedings of the 18th International Conference on Data Engineering*, page 225, 2002.

[TLGXYJX06] Hu Tian-Lei, Chen Gang, Li Xiao-Yan, and Dong Jin-Xiang. Automatic relational database compression scheme design based on swarm evolution. *Journal of Zhejiang University - Science A*, 7(10):1642–1651, October 2006.