

POLITECHNIKA WROCŁAWSKA

WYDZIAŁ ELEKTRONIKI

KIERUNEK: Informatyka (INF)

SPECJALNOŚĆ: Inżynieria systemów informatycznych (INS)

PROJEKT INŻYNIERSKI

Indeksowanie temporalne w bazach
danych

Indexing temporal data in databases

AUTOR:

Anna Bomersbach

PROWADZĄCY PRAC:

dr inż. Tomasz Kubik

OCENA PRACY:

WROCŁAW, 2013

Spis treści

1. Wstęp	4
1.1. Cel pracy	4
1.2. Układ pracy	5
2. Projekt systemu	6
2.1. Analiza wymagań	6
2.1.1. Wymagania funkcjonalne	6
2.1.2. Wymagania нефункционалне	7
2.1.3. Wymagania dziedzinowe	8
2.2. Zarys architektury	13
2.2.1. Schemat systemu	13
2.2.2. Makiety interfejsu użytkownika	13
3. Opis rozwiązania	17
3.1. Zastosowanie modeli reprezentacji czasu do rozwiązania problemu	17
3.2. Narzędzia	17
3.3. Struktura logiczna	19
3.4. Schemat bazy danych	21
4. Przykład działania	22
4.1. Logowanie i profil użytkownika	22
4.2. System wypożyczeń	24
4.3. Zarządzanie zatrudnieniami	26
5. Implementacja	29
5.1. Modele	29
5.2. Widoki - logika aplikacji	30
5.3. Znane strategie indeksowania	34
5.4. Indeksowanie zastosowane w projekcie	35
6. Testy i wdrożenie	36
6.1. Index GIN na relacji wypożyczeń	36
6.2. Index GiST na relacji zatrudnień	37
6.3. Wdrożenie	39
6.4. Instalacja	40
7. Podsumowanie	41
7.1. Omówienie zastosowanych rozwiązań indeksowych	41
7.2. Podsumowanie wyników	42

7.3. Kierunki rozwoju	42
Literatura	43

Rozdział 1

Wstęp

Czas jest bardzo ważnym aspektem wielu dziedzin życia, dotyczącym m.in. zagadnień zarządzania zasobami, planowania rozkładów jazdy czy zagadnień naukowych (np. w fizyce, astronomii). Zarówno w ekonomii przy analizie trendów, jak i meteorologii przy prognozowaniu pogody, oprócz aktualnych wartości notowań czy temperatur, brana jest pod uwagę historia ich zmian. Stąd we współczesnych systemach coraz częściej istnieje konieczność zapisywania w bazach danych historii zmian wartości atrybutów oraz efektywnego wykonywania zapytań na przedziałach czasu.

Główną przyczyną sukcesu relacyjnego modelu jest jego prostota. Konwencjonalne (nie-temporalne) bazy danych reprezentują „urywek” modelowanej rzeczywistości, tzn. wspierają zapisywanie stanu modelu aktualnego jedynie w chwili zapisu. Relacje temporalne są bardziej złożone od konwencjonalnych relacji, ponieważ, oprócz powiązania wartości z faktami, określają, kiedy dany fakt jest prawdziwy w rzeczywistości oraz kiedy fakt jest obecny w bazie danych.

Zarządzanie temporalnym aspektem jest zwykle implementowane po stronie aplikacji korzystającej z bazy danych. W języku zapytań SQL dostępne są pewne typy danych dotyczące czasu, takie jak timestamp, date, time, jednak wsparcie dla temporalnych zapytań po stronie bazy danych jest bardzo ograniczone, a wykonywanie nawet prostych zapytań z uwzględnieniem aspektu czasu w języku SQL wymaga pisania długiego i skomplikowanego kodu.

Modelowanie czasu w bazach od dawna stanowi ważny problem w dziedzinie rozwoju baz danych. Na przykład, już w 1982 roku [1] po raz pierwszy zaproponowano uwzględnienie aspektu czasu na poziomie atrybutu. Od tego czasu powstało wiele modeli temporalnych relacyjnych baz danych, kilka spośród których zostało uznane za przystępne i jest obecnie stosowanych. Jednak kwestia modelowania baz temporalnych nie jest rozwiązana, a sposób reprezentacji czasu pozostaje tematem prac badawczych.

Otwartą kwestią pozostaje również problem efektywnego przechowywania i dostępu do danych temporalnych. Temat został poruszony po raz pierwszy w 1990 roku [2] w pracy naukowców z University of Houston, która stanowi w niniejszym projekcie ważny element wykorzystanej literatury.

1.1. Cel pracy

Praca ma na celu realizację systemu monitorującego wypożyczenia i zwroty sprzętu, np. urządzeń elektronicznych, aut, narzędzi w firmie. Dodatkowo, aplikacja powinna umożliwiać rejestrację zmian osób na stanowiskach w firmie, np. by sprawdzić w jakiej konfiguracji pracowników osiągnąć najlepsze rezultaty. Choć oba przykłady wymagają użycia temporalnej bazy

danych, zapytania wykonywane najczęściej dla każdego z nich są różne. Ponieważ tabele przechowujące dane temporalne znacznie przekraczają rozmiarami tradycyjne relacje, ważnym zagadnieniem jest indeksowanie takich relacji. Dla wydajnego funkcjonowania, system powinien korzystać ze stosownie dobranych struktur indeksowych. Omawiana praca stanowi zatem próbę znalezienia najodpowiedniejszego sposobu przechowywania i zarządzania danymi temporalnymi.

1.2. Układ pracy

W pracy przedstawiono projekt systemu, wykorzystującego temporalny aspekt bazy danych oraz zagadnienia z nim związane.

Pracę rozpoczyna Wstęp, po którym następuje rozdział z opisem wymagań stawianych przed tym systemem. Uwzględniono w nim wymagania funkcjonalne, нефункционалне oraz dziedzinowe. Przy okazji omawiania wymagań dziedzinowych przedstawiono informacje dotyczące sposobów modelowania czasu w relacyjnych bazach danych (w tym osiągnięcia w dziedzinie konstruowania temporalnych baz danych oraz ich indeksowania, niezbędne do implementacji systemu). Następnie, w rozdziale 3, opisano szczegóły opracowanego rozwiązania. Przedstawiono strukturę logiczną systemu, schemat bazy danych oraz wybrane narzędzia. Zaprezentowano też szczegóły implementacji. W kolejnym rozdziale opisano testowanie systemu oraz jego wyniki. W ostatnim rozdziale dokonano podsumowania projektu.

Rozdział 2

Projekt systemu

Efektom pracy nad projektem ma być system rejestrujący stan danych dotyczących firmy, zmiennych w czasie. Powinien on spełniać przede wszystkim dwie funkcje: pozwalać na efektywne wprowadzanie oraz odczytywanie informacji dotyczących pracowników na stanowiskach i wypożyczeń sprzętu. System ma wspierać wykonywanie operacji na danych wprowadzonych w niewielkim odstępie czasu od dnia obecnego, jak i umożliwiać prześledzenie kompletnej historii obiektów w bazie danych. Celem etapu projektowania działania systemu jest ustalenie potrzeb użytkowników oraz ograniczeń, dotyczących wdrażania i działania systemu. W dalszej części rozdziału, omówione zostaną wymagania oraz zarys architektury systemu, który miałby je spełniać.

2.1. Analiza wymagań

Celem tego etapu projektowania systemu jest zmiana potrzeb klienta na konkretne wymagania, zapewniające ich osiągnięcie. Poprawne zdefiniowanie wymagań w procesie budowania systemu informatycznego jest jednym z ważniejszych elementów projektowania oprogramowania. Decyzje podjęte na tym etapie są podstawą późniejszych działań, stąd błędy popełnione podczas definiowania wymagań są bardzo kosztowne i stanowią główną przyczynę niepowodzeń projektów informatycznych. Właśnie dlatego przed przystąpieniem do implementacji sprecyzowano listę wymagań z podziałem na:

- wymagania dotyczące funkcji, jakie system ma udostępniać użytkownikom aby sprostał ich oczekiwaniom;
- ograniczenia, przy zachowaniu których system powinien wykonywać wyznaczone zadania;
- prawa dotyczące zapisu czasu i manipulacji danymi temporalnymi oraz ich indeksowania.

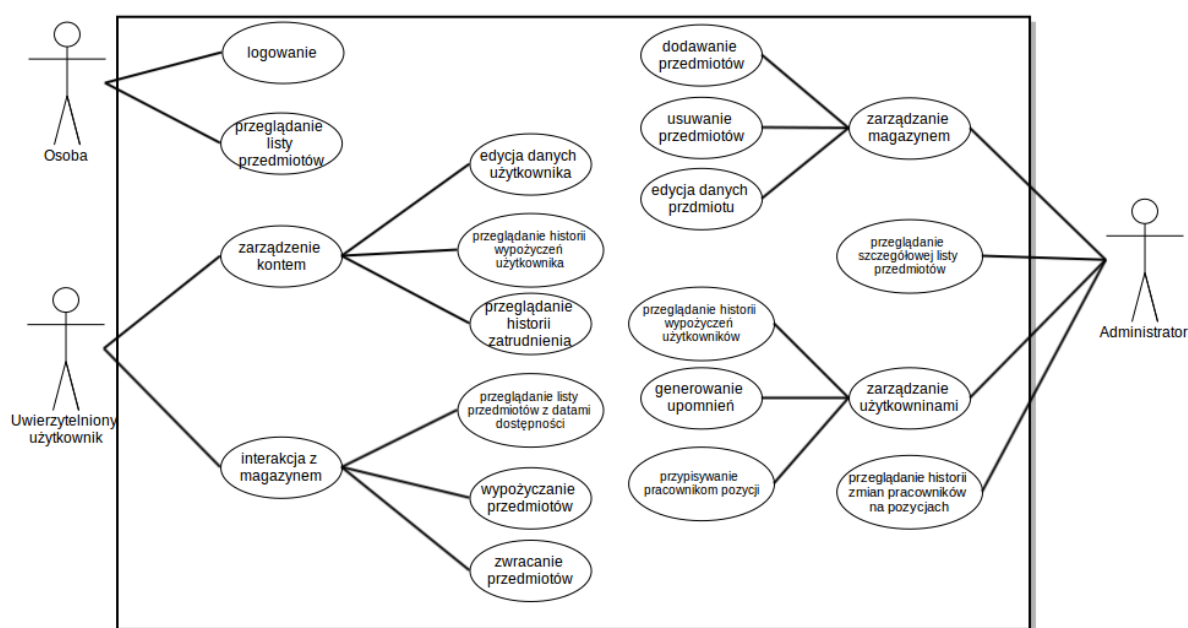
Poniżej zostaną one przedstawione zgodnie z tą kolejnością.

2.1.1. Wymagania funkcjonalne

Głównym celem systemu jest kontrola terminowości zwrotów sprzętu i prognoza terminu dostępności przedmiotów do następnych wypożyczeń, oraz śledzenie zmian pracowników na stanowiskach. Z systemu mają korzystać zarówno pracownicy, mający możliwość wypożyczania sprzętów, jak i administrator, zarządzający systemem.

Osoba niezarejestrowana powinna mieć opcję rejestracji, a osoba posiadająca konto, możliwość logowania do serwisu. Uwierzytelnieni użytkownicy powinni również mieć możliwość wglądu w spis dostępnych przedmiotów oraz dokonywania rezerwacji, przy czym każdy przedmiot może być w danym momencie wypożyczony tylko jednej osobie. W projekcie systemu wyróżniona została rola osoby odpowiedzialnej za zarządzanie systemem. Administrator jest odpowiedzialny za zarządzanie magazynem, tj. dodawanie i usuwanie sprzętów, oraz kontrolowanie wypożyczeń wszystkich użytkowników. System powinien umożliwiać administratorowi wprowadzanie informacji o zmianach na stanowiskach, jednocześnie dbając o to, by w danym momencie maksymalnie jedna osoba była przypisana do danego stanowiska.

Na rysunku 2.1 przedstawiono diagram przypadków wymienionych wyżej aktorów: osoby niezalogowanej, która być może spotkała się z systemem po raz pierwszy, osoby mającej konto i korzystającej z systemu oraz administratora, odpowiedzialnego za zarządzanie i monitorowanie zmian.



Rys. 2.1: Schemat funkcji dostępnych dla korzystających z systemu: zwykłego użytkownika (osoby niezalogowanej i osoby posiadającej już konto) i administratora.

2.1.2. Wymagania niefunkcjonalne

Aby z aplikacji korzystać mogło jednocześnie wielu użytkowników, musi być ona łatwo dostępna (jak aplikacja webowa lub mobilna). Ponadto powinna ona umożliwiać obsługę małego przedsiębiorstwa, gdzie przedmioty i stanowiska mogłyby być rozdzielane pomiędzy 300 pracowników. Czas odpowiedzi nie powinien przekraczać 1 sekundy, tak by użytkownicy nie doświadczyli uciążliwego czekania przy każdym zapytaniu. Interfejs użytkownika powinien być przejrzysty, a strona główna zawierać instrukcje, tak by zapoznanie użytkowników z obsługą systemu nie trwało więcej niż 15 minut.

Dla zachowania bezpieczeństwa, hasła użytkowników przechowywane w bazie powinny być szyfrowane. Każdy formularz ma posiadać zabezpieczenie przed atakiem CSRF.

2.1.3. Wymagania dziedzinowe

Przy projektowaniu systemu z historią zmian jego elementów najtrudniejszym zadaniem jest modelowanie aspektu czasu. Powinno być to wykonane tak, aby ważne informacje nie zostały utracone podczas zapisu, a jednocześnie wykonywanie najpopularniejszych zapytań było możliwie szybkie. Choć temporalne bazy danych są dziedziną stale rozwijającą się, opracowano pewne metody i standardy [6], których należy przestrzegać. Zostaną one opisane w tej części.

Aspekt czasu w bazach danych

Modelowanie czasu w bazach danych jest tematem wielu publikacji [1, 6, 5]. Niektóre z zaproponowanych modeli dostarczają bogatej struktury i są przydatne przy wyświetlaniu danych, inne natomiast dostarczają regularnej struktury przydatnej przy przechowywaniu danych temporalnych.

Modelowanie czasu

Reprezentacja czasu powinna dać się łatwo zintegrować z modelem danych. Wyróżniono dwie zasadnicze wielkości, które najczęściej stosuje się do opisywania czasu [5]:

- Transaction time
- Valid Time

Pierwszą z nich jest interwał czasu, w którym fakt był obecny w bazie danych. Druga natomiast, określa kiedy dany zapisany w bazie danych fakt był, jest lub będzie prawdziwy w modelowanej rzeczywistości. Przyjmuje się, że system bazy danych ma ograniczoną dokładność. Najmniejsze jednostki czasu nazywane są chrononami. Dziedzina czasu wartości *valid time* może być określona jako zbiór $D_{VT} = \{t_1, t_2, \dots, t_k\}$, a wartości *transaction time* jako $D_{TT} = \{t_1, t_2, \dots, t_j\} \cup \{UC\}$, gdzie *UC* (ang. *Until Changed*) jest specjalną wartością, używaną podczas aktualizowania bazy danych (podczas wykonywania poleceń *UPDATE* w języku *SQL*). Chronon *valid time* jest więc elementem D_{VT} , chronon *transaction time* elementem $D_{TT} \setminus \{UC\}$, a chronon bitemporalny - uporządkowaną parą dwóch chrononów: *transaction time* i *valid time*. W miarę upływu czasu, bitemporalne elementy powiązane z faktami są uaktualniane, jeśli fakt obecny w bazie danych pozostaje obowiązujący. Podczas wprowadzania faktu do relacji, związany z nim chronon *transaction time* ma wartość „UC”. Do momentu usunięcia faktu z bazy, wartość ta będzie uaktualniana, tzn. w pewnych odstępach czasu (równych ustalonej dokładności zapisu czasu, czy też gęstości czasu), w miejsce pól wpisywany jest aktualny czas systemowy, natomiast pola UC dopisywane są do wartości atrybutu. W ten sposób tworzona jest historia zapisów w bazie, a jednocześnie można rozróżnić fakty aktualnie obowiązujące dzięki wartości UC. Omówiony w następnej części model nazywany jest bitemporalnym ze względu na wsparcie dla obu typów czasu.

Koncepcyjny bitemporalny model danych

Bitemporal Conceptual Database Model (BCDM) prezentuje podstawową ideę dodawania temporalnego aspektu do tradycyjnych relacji. Miał on posłużyć jako punkt wyjściowy dla budowania struktur pozwalających na prezentację i przechowywanie danych oraz ocenę zapytań, stąd został nazwany koncepcyjnym. Według tego modelu w pojedynczej tabeli przechowywane są atrybuty opisujące fakt, obok atrybutów dotyczących wartości *transaction time* i *valid time* tego faktu.

W omawianym systemie faktem jest opis zdarzenia wypożyczenia przez użytkownika pewnego przedmiotu, w bazie danych reprezentowane jako para (id użytkownika, id przedmiotu). Dla każdej takiej pary przechowywana jest historia zmian, zapisywana za pomocą kolejnych wartości par (*transaction time*, *valid time*), jak opisano w [5].

W przykładzie dotyczącym wypożyczeń, wartość *transaction time* opisuje chwile, w których pewien użytkownik posiadał dany przedmiot. Każdy taki zapis ma odpowiadający mu zbiór wartości *valid time* - opis chwil, w których użytkownik miał prawo do wypożyczenia. Stąd zbiór par (1,1), (1,2), (1,3), (1,4) oznacza, że wypożyczenie dotyczyło dnia „1” oraz użytkownik miał prawo przechowywać przedmiot przez 4 kolejne dni. Natomiast zbiór (1,1), (1,2), (2,1), (2,2), (3,1), (3,2) przedstawia sytuację, gdy osoba korzystała z przedmiotu dłużej (3 dni), niż było wskazane (2 dni). Tak w uproszczeniu mogą wyglądać rekordy dotyczące zdarzeń przeszłych. Do opisywania *transaction time* definiuje się dodatkową wartość Until Changed (UC), oznaczającą fakt dotyczący chwili obecnej, np. zbiór (4,4), (4,5), (4,6), (UC,4), (UC,5), (UC,6) opisuje wypożyczenie trwające od dnia czwartego do chwili obecnej, w którym użytkownik ma prawo korzystać z przedmiotu przez 3 kolejne dni.

Każda z podstawowych operacji na bazie danych, przy manipulowaniu danymi temporalnymi staje się bardziej skomplikowane. Poniżej opisano podstawowe operacje: wprowadzania danych (INSERT), ich edycji (UPDATE) i usuwania (DELETE).

INSERT – podczas wprowadzania rekordów do bazy o takim modelu, należy rozpatrzyć 3 przypadki, gdy opisany fakt:

1. nie był obecny w bazie danych i zostaje on wprowadzony z *transaction time* = UC;
2. był wcześniej obecny w bazie i zostaje zaktualizowany - dopisana jest wartość *transaction time* = UC oraz odpowiednia wartość *valid time*;
3. znajduje się aktualnie w bazie danych, wprowadzenie jest odrzucone, a zamiast niego następuje edycja.

UPDATE – tutaj modyfikacja krotki polega na usunięciu, po którym następuje ponowne wprowadzenie danych. Taki zabieg sprawia, że dane dotyczące faktu przechowywane są raz, z całą historią zmian, ponieważ usunięcie (logiczne) nie powoduje utraty informacji o chrononach czasu, a ponowne wprowadzenie dodaje nowe informacje.

DELETE – dotyczy bazy temporalnej, dla której nie są wykonywane właściwe operacje usuwania, ponieważ taka baza historyczna jest typu „append-only”. Do takiej bazy można jedynie dopisywać rekordy oraz edytować obecne. Aby zaznaczyć, że dany fakt jest usunięty z bazy, należy znaleźć wszystkie chronony dotyczące tego faktu, które są postaci (UC, C_v), gdzie C_v jest pewnym chrononem *valid time* i je usunąć. W ten sposób fakt przestaje należeć do zbioru obecnie prawdziwych faktów i nie jest dłużej brany pod uwagę przy okresowym uaktualnianiu bazy.

Reprezentacja modelu koncepcyjnego w bazie danych

Choć przedstawiony koncepcyjny model jest prosty i intuicyjny, jego zastosowanie wymaga reprezentacji kolumny opisującej historię środkami dostępnymi w relacyjnej bazie danych. Wobec powyższego, prowadzono badania w tej dziedzinie i powstało kilka reprezentacji modelu koncepcyjnego, między innymi opisane w [6]:

- Snodgrass’ Tuple Timestamped Data Model,
- Jensen’s Backlog-based Data Model,

- Ben-Zvi's Tuple Timestamped Data Model,
- Gadia's Attribute Value Timestamped Data Model,
- McKenzie's Attribute Value Timestamped Data Model.

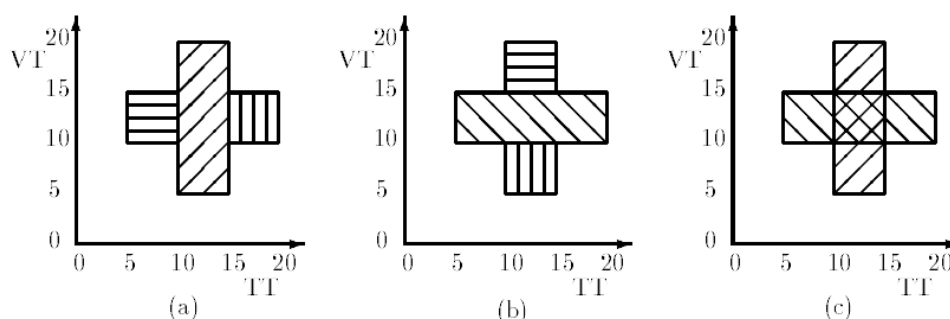
Są to jedynie propozycje reprezentacji, nie zawierające szczegółów dotyczących implementacji. W niektórych aspekt czasu wprowadzony jest na poziomie krotki, w innych - na poziomie atrybutu. Pierwsze mogą powodować redundancję, ponieważ wartości atrybutów, które zmieniają się w czasie, mogą powtórzyć się w kilku krotkach. W drugim przypadku, gdzie atrybutowi można przypisać zbiór interwałów, stosowanie prostych typów danych nie jest możliwe.

Snodgrass' Tuple Timestamped Data Model

Pierwszy z modeli opisuje każdy interwał dwoma wartościami dyskretnymi: chrononem jego początku i końca. Schemat relacji, w której atrybuty A_1, A_2, \dots, A_n definiują pewne zdarzenie, wygląda następująco:

$$R = (A_1, A_2, \dots, A_n, T_s, T_e, V_s, V_e) \quad (2.1)$$

Na wykresie o osi poziomej *transaction time* oraz pionowej *valid time* rekord T_s, T_e, V_s, V_e ma formę prostokątnego regionu. Aby record opisywany przez BCDM przedstawić za pomocą modelu Snodgrassa, należy region opisywany przed BCDM podzielić na prostokąty. Krotka modelu koncepcyjnego reprezentowana jest przez zbiór krotek, z których każda opisuje jeden prostokąt. Podziału można dokonać za względu na *transaction time* lub *valid time*. Prostokąty mogą być rozłączne lub posiadać części wspólne. Model nie ma na celu precyzować zasad implementacji, jednak przedstawić ogólną zasadę mapowania BCDM na postać możliwą do zaimplementowania.



Rys. 2.2: Możliwe sposoby podziału dwuwymiarowej przestrzeni *valid time* x *valid time* w celu przedstawienia danych w modelu Snodgrassa [6]

Jensen's Backlog-based Data Model

Schemat relacji zaproponowany przez Christiana Jensena z uniwersytetu w Aalborg w Danii przypomina omówiony poprzednio:

$$R = (A_1, A_2, \dots, A_n, V_s, V_e, T, O_p) \quad (2.2)$$

Atrybuty V_s, V_e również w tym przypadku opisują początkowy i końcowy czas, opisując interwał *valid time*. Wartość T określa *transaction time* jako moment wprowadzenia pewnych zmian. Na bazie wykonywane są operacje jedynie dwóch typów: INSERT (dodawanie krotki)

lub DELETE (usuwanie), ponieważ modyfikacje realizowane są przez usunięcie i ponowne wprowadzenie danych. Ostatni atrybut modelu determinuje, której z dwóch możliwych operacji dotyczy zapis.

Ben-Zvi's Tuple Timestamped Data Model

Model, w którym wśród przechowywanych atrybutów uwzględnia się pięć temporalnych wartości:

$$R = (A_1, A_2, \dots, A_n, T_{es}, T_{rs}, T_{ee}, T_{re}, T_d) \quad (2.3)$$

Każda określa moment pewnego zdarzenia: T_{es} – *effective start*, T_{rs} – *registration start*, T_{ee} – *effective end*, T_{re} – *registration end* oraz T_d – *deletion*. Pierwsza z nich określa, kiedy fakt zaczął być obowiązujący, druga kiedy zanotowano to w bazie, podobnie trzecia dotyczy końca obowiązywania faktu, natomiast T_{re} to moment, w którym zanotowano tę zmianę. Ostatnia wskazuje moment gdy fakt został logicznie usunięty z bazy i podobnie jak T_{ee} i T_{ee} nie musi przedstawiać żadnej wartości (co zapisywane jest jako „-”).

Gadia's Attribute Value Timestamped Data Model

Zaproponowane modele, nie będące w pierwszej normalnej formie, zawierają całą informację na temat obiektu w jednej krotce. Każda krotka zawiera n zbiorów, gdzie n jest liczbą atrybutów relacji. Każdy zbiór zawiera trójkę: interwał $[T_s, T_e]$ – *transaction time*, interwał $[V_s, V_e]$ – *valid time* oraz wartość atrybutu. Oto schemat takiej relacji:

$$R = (\{([T_s, T_e] \times [V_s, V_e], A_1)\}, \dots, \{([T_s, T_e] \times [V_s, V_e], A_n)\}) \quad (2.4)$$

McKenzie's Attribute Value Timestamped Data Model

Podobnie do omawianego wcześniej, model ten nie jest w pierwszej normalnej formie. Relacja zawiera stany wartości VR – *valid time*, indeksowane wartościami T – *transaction time*. Innymi słowy, dla każdej wartości czasu transakcji, przechowywana jest dla każdego atrybutu wartość interwału opisującego, kiedy atrybut stanowi o fakcie prawdziwym. Schemat to:

$$R = (T, VR), \text{ gdzie } VR = (A_1, V_1, \dots, A_n, V_n) \quad (2.5)$$

a wartość V_i odnosi się do czasu obowiązywania faktu zawartego w atrybucie A_i .

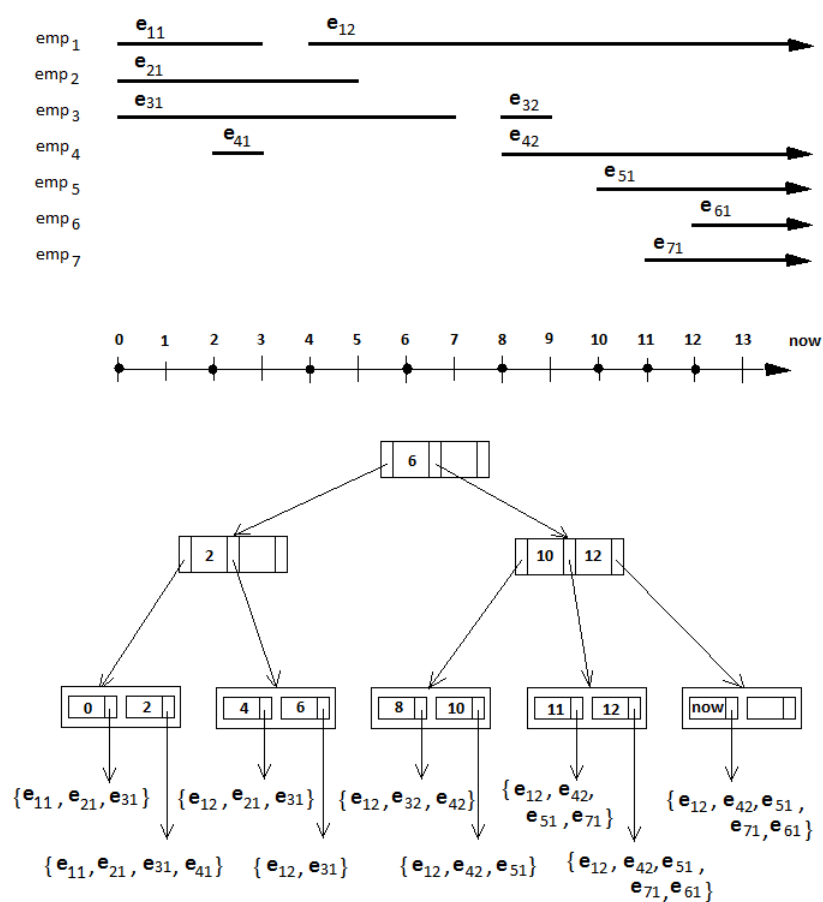
Indeksowanie danych temporalnych

Badania na temat temporalnych baz danych dotyczyły głównie definiowania modeli i operacji obsługujących wymiar czasu. Znacznie mniej uwagi poświęcono problemowi przechowywania danych temporalnych w sposób pozwalający na efektywną manipulację, podczas gdy właściwości wymiaru czasowego utrudniają wykorzystanie tradycyjnych indeksów. Po pierwsze, w bazie temporalnej uwzględnia się interwały czasowe zamiast pojedynczych chrononów. Ponadto, ponieważ w bazach temporalnych większość uaktualnień skutkuje dopisaniem wartości do bazy, nowe wersje obiektu pojawiają się ze wzrastającą wartością czasu. Przechowywanie danych historycznych powinno pozwolić na selekcję danych aktualnych w pewnym okresie czasu. Sekwencyjne porównywanie wartości zadanego interwału ze wszystkimi przedziałami opisanymi w bazie wymaga $O(M \cdot N)$ dostępów do systemu przechowywania, gdzie N jest liczbą obiektów, a M największą dopuszczalną liczbą wersji każdego z nich. Jest to wysoce nieefektywne, a właśnie takich zapytań dotyczących przedziałów należy się spodziewać.

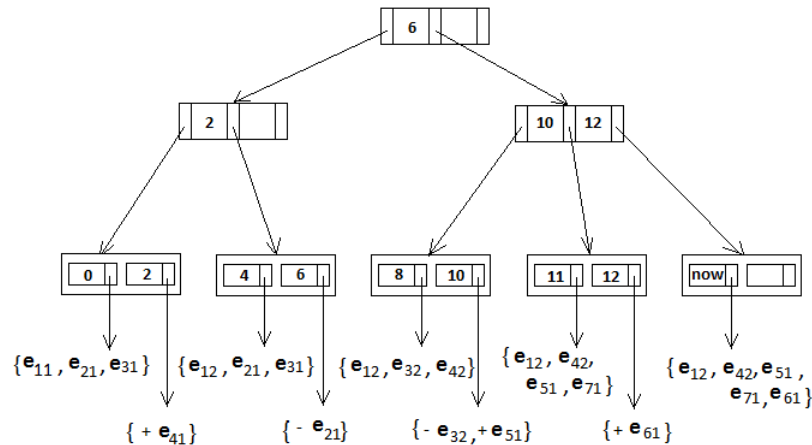
Pierwszą propozycję indeksu czasowego przedstawiono w [2]. Do zdefiniowania indeksu wprowadzono pomocniczą wielkość - system przechowywania wersji rekordów $TDB = \{e_1, e_2, \dots, e_n\}$, gdzie każda z wersji oznaczana jest indeksowaną literą e . Idea opiera się na zarządzaniu zbiorem uporządkowanych w wymiarze czasu *punktów indeksujących*. Takie punkty mają być tworzone w chwili początkowej interwału, lub w chwili następującej po zakończeniu interwału czasowego, opisującego wersję obiektu, jak pokazano na rysunku 2.3. Dla wartości *valid time*, dla której opisano indeks, można formalnie zdefiniować taki zbiór następująco:

$$BP = \{t_i | \exists e_j \in TDB((t_i = e_j.valid_time.t_s) \vee (t_i = e_j.valid_time.t_e + 1))\} \cup \{now\} \quad (2.6)$$

Każdy liść drzewa wskazuje wersje obiektów obowiązujące w pewnym momencie. Takich może być wiele i część z nich prawdopodobnie będzie się powtarzać dla sąsiednich liści. Zaproponowano, aby pełną informację o aktualnych wersjach przechowywać jedynie dla pierwszego elementu węzła, natomiast liście na prawo od niego wskazywać mają na zbiór przyrostowych zmian w stosunku do niego, co przedstawiono na rysunku 2.4.



Rys. 2.3: Sposób prezentacji wersji obiektów w strukturze B+-drzewa (na podstawie [2])



Rys. 2.4: Przechowywanie przyrostowych zmian w wartościach wskazywanych przez indeks czasowy (na podstawie [2])

2.2. Zarys architektury

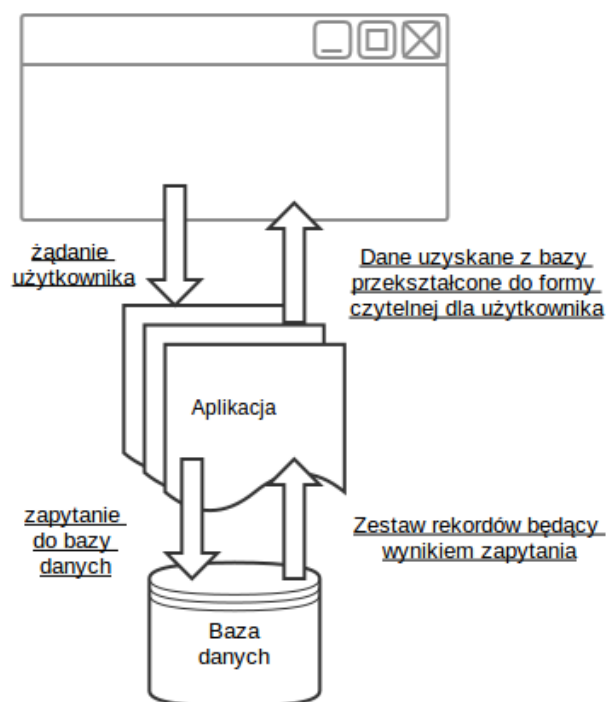
2.2.1. Schemat systemu

Przy projektowaniu systemu wykorzystano popularny wzorzec architektury trójwarstwowej. System zbudowany jest z kilku części, co pozwala na łatwiejsze testowanie i rozszerzanie aplikacji. Rolę interfejsu użytkownika pełni strona internetowa, wyświetlana przez przeglądarkę. Jej zadaniem jest prezentacja danych w sposób zrozumiały dla człowieka oraz przesyłanie żądań do warstwy niższej. Ta przyjmuje komunikaty użytkownika, przetwarza je oraz wykonuje odpowiednie operacje i obliczenia. Oprócz tego, warstwa logiki biznesowej jest odpowiedzialna za przekazywanie informacji pomiędzy sąsiednimi warstwami. Ma m.in. wysyłać zapytania do warstwy danych, której zadaniem jest pobranie żądanych danych z fizycznej bazy i przesłanie ich z powrotem do warstwy środkowej. Komunikację między elementami systemu przedstawiono na rysunku 2.5.

Aspekt temporalny implementowany jest zarówno na poziomie bazy danych, przez zastosowanie typów danych do zapisu czasu oraz odpowiednich indeksów, jak i aplikacji, której zadaniem jest interpretowanie danych temporalnych zapisanych w bazie.

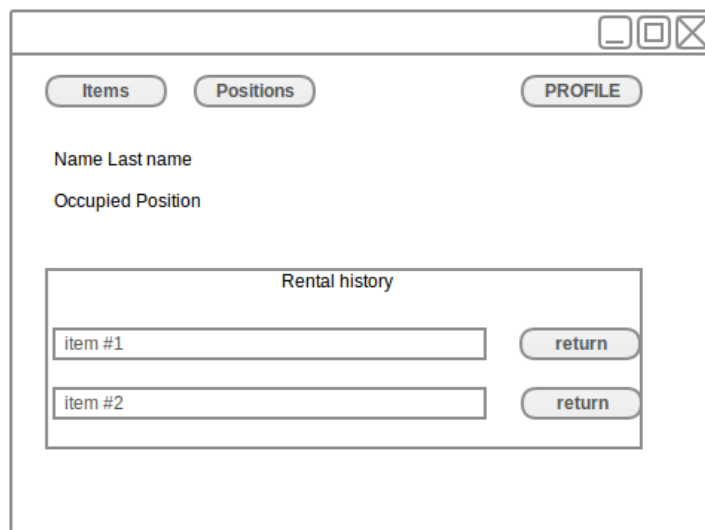
2.2.2. Makiety interfejsu użytkownika

Interfejs powinien w prosty sposób nawigować użytkownika do elementów pełniących funkcje zdefiniowane w wymaganiach funkcjonalnych. Osoba po zalogowaniu ma mieć możliwość wglądu w stan swojego konta, tzn. danych osobowych, pozycji, na której jest zatrudniona, oraz historii wypożyczeń, co przedstawiono na rysunku 2.6. Przedmioty zwrócone powinny zawierać informację o dacie zwrócenia, natomiast niezwrócone - przycisk pozwalający zgłosić oddanie przedmiotu. Użytkownik powinien mieć dostęp do listy dostępnych sprzętów, które mógłby wypożyczyć (rysunek 2.7). Lista przedmiotów wygląda inaczej w interfejsie dostępnym dla administratora. Zgodnie z wymaganiami funkcjonalnymi, administrator ma mieć dostęp do danych wypożyczeń wszystkich użytkowników, co przedstawiono na rysunku 2.8. Na rysunku 2.9 przedstawiono projekt interfejsu administratora, umożliwiający zarządzanie pracownikami na stanowiskach - ich zatrudnianie i zwalnianie. Dla zaoszczędzenia czasu, operacje te powinny

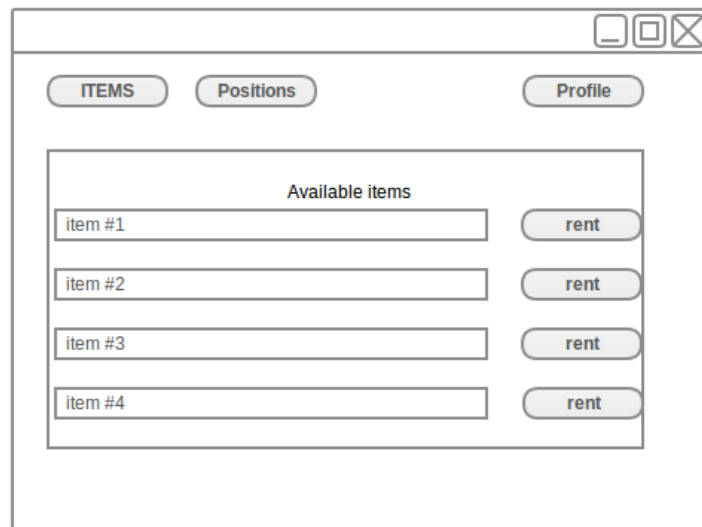


Rys. 2.5: Komunikacja między elementami systemu

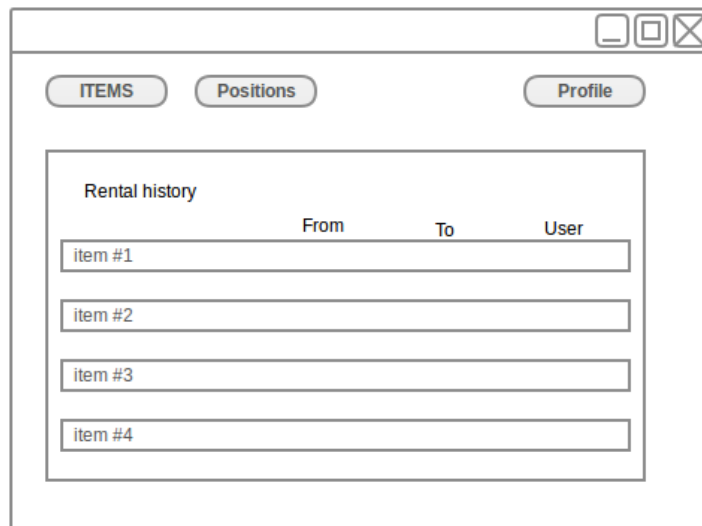
być wykonywane bez przeładowania strony, np. przy użyciu okna dialogowego, jak na rysunku 2.10.



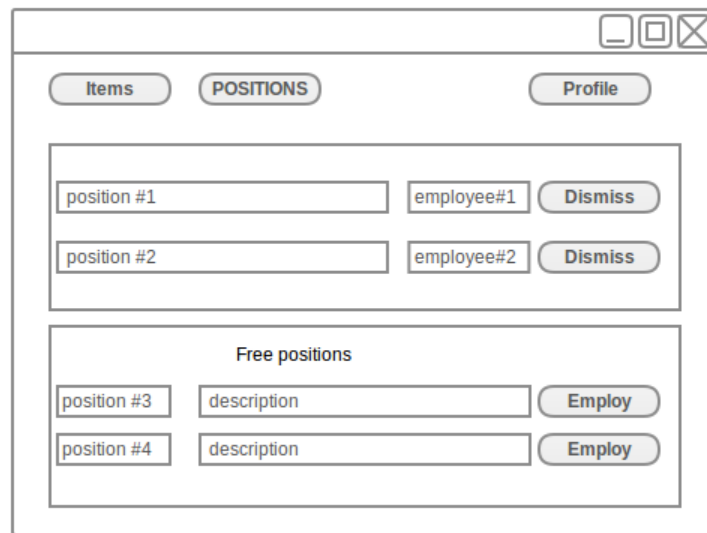
Rys. 2.6: Makieta interfejsu - strona profilowa użytkownika



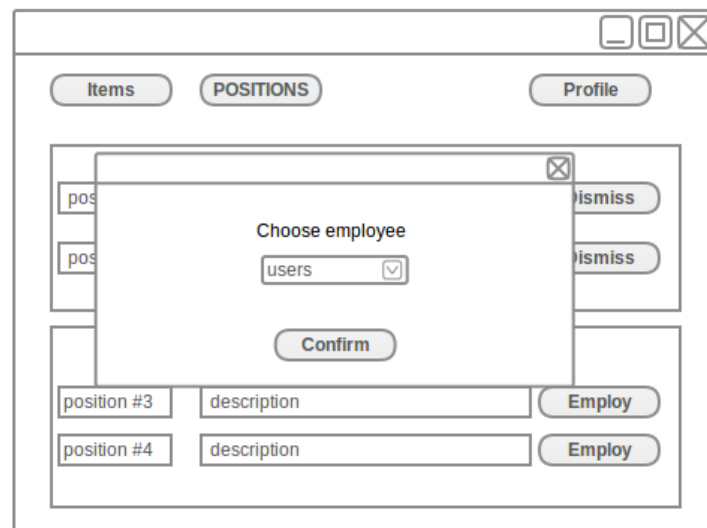
Rys. 2.7: Makieta interfejsu - strona z listą dostępnych przedmiotów



Rys. 2.8: Makieta interfejsu administratora - strona przedstawiająca historię wypożyczeń sprzętów z danymi dotyczącymi czasów wypożyczeń



Rys. 2.9: Makieta interfejsu administratora - strona do zarządzania zatrudnieniami



Rys. 2.10: Makieta interfejsu administratora - okno dialogowe do wyboru użytkownika do zatrudnieniu po wybraniu przycisku w liście wolnych stanowisk

Rozdział 3

Opis rozwiązania

3.1. Zastosowanie modeli reprezentacji czasu do rozwiązania problemu

W omawianej aplikacji obsłużony ma zostać każdy ze stanów, w którym może znaleźć się przedmiot należący do wypożyczalni: „wypożyczony”, „dostępny”, „przetrzymany”. W związku z tym należy użyć modelu zapisu danych pozwalającego na przechowywanie informacji na temat czasu zmiany wartości rekordów (*transaction time*) oraz tego, czy w danym momencie fakt obecny w bazie jest prawdziwy czy nie (*valid time*). Ponieważ wybrane do implementacji narzędzia są aktualnie na etapie rozwoju, przyjęto założenie, że informacja o czasie powinna być wprowadzana z dokładnością do dnia. W przyszłości łatwo można będzie przekształcić aplikację do pracy z danymi o innej gęstości, więc nie ma to wpływu na przydatność projektu.

Do zapisu informacji dotyczących wypożyczeń wykorzystane są dwie wspomniane wartości, w sposób zaproponowany przez Christiana Jensena [5]. *Transaction time* jest użyty do określenia, kiedy przedmiot był wypożyczony. Z kolei *valid time* informuje, do kiedy użytkownik miał prawo posiadać wypożyczony przedmiot. Dzięki takiemu ujęciu, za pomocą tych dwóch wartości można śledzić zarówno historię wypożyczeń danego użytkownika, jak i spisać listę wszystkich użytkowników, którzy kiedykolwiek posiadali dany przedmiot. Pierwsze pozwoli np. naliczyć karę użytkownikowi za wszystkie przedmioty przetrzymane od momentu działania systemu, drugie natomiast, w przypadku uszkodzenia sprzętu, umożliwi znalezienie potencjalnych winowajców.

Rejestrowanie okupowania stanowisk przez pracowników wymaga jedynie użycia wielkości *valid time*. Każdej parze (użytkownik, pozycja) przyporządkowane są interwały czasu, w których takie przyporządkowanie było aktualne. Ponieważ takich interwałów może być wiele, w odróżnieniu od systemu wypożyczeń z relacją opartą na BCDM, w tym przypadku w relacji zatrudnień pary (użytkownik, pozycja) mogą się powtarzać. Głównym celem tej części systemu jest umożliwienie przeglądu zmian na stanowiskach w pewnych przedziałach czasowych, dlatego takie zapytania powinny być wspierane przez strukturę indeksową.

3.2. Narzędzia

PostgreSQL

Wybór systemu zarządzania bazą danych jest kluczowy dla omawianego projektu. PostgreSQL jest dynamicznie rozwijającą się platformą, umożliwiającą użycie specyficznych typów

i modyfikowalnych struktur przechowywania danych. Typy danych dotyczących czasu, dostępne w momencie powstawania pracy, to `date`, `time`, `timestamp` i `interval`. Dodatkowe struktury, które okazały się pomocne w czasie realizowania projektu to typ tablicowy `array` (z dostępną od wersji 9.3 funkcją `array_remove(anyarray, anyelement)`) oraz typ `range` (dostępny od wersji 9.2).

PostgreSQL dostarcza również kilku typów indeksów: B-tree, Hash, GiST oraz GIN. Pierwszy z nich wspomaga zapytania, w których warunek filtrujący zawiera równość lub nierówność (zapytania o przedziały) i jest rozpatrywany przez plan systemu zarządzania gdy takie porównania mają miejsce. Jest również użyteczny, gdy zapytania zawierają którekolwiek ze słów kluczowych `BETWEEN`, `IN`, `IS NULL` lub `NOT NULL`. Ograniczeniem tego indeksu jest fakt, że można stosować go tylko dla podstawowych wbudowanych typów danych. Z kolei `hash index`, przydaje się jedynie przy równościach. Dwa pozostałe natomiast mogą być dostosowane przez użytkownika. General Inverted Index (GIN) przechowuje zestawy (*klucz*, *lista*), gdzie *lista* jest zbiorem identyfikatorów krotek zawierających dany *klucz*, przy czym każdy z tych identyfikatorów może pojawić się w wielu takich zestawach (*klucz*, *lista*). W przypadku gdy klucze pojawiają się w krotkach wiele razy, indeks zajmuje stosunkowo mało miejsca. GIN znajduje zastosowanie, gdy wartość zawiera więcej niż jeden klucz, tak jak w przypadku typu `array` lub `text`. Dodatkowo GIN wspiera zapytania na tablicach dotyczące przedziałów: „zawiera”, „jest zawarta w” oraz „posiada część wspólną z”. Przyspiesza on znacznie wykonywanie zapytań, natomiast spowalnia aktualizowanie tabel. Generalized Search Tree (GiST) pozwala na implementację różnych strategii, poprzez nadpisanie kilku metod dostępowych. J. M. Hellerstein, J. F. Naughton i A. Pfeffer. [4] pokazali, że można w ten sposób zaimplementować indeks o strukturze B+-drzewa, czyli taki, jaki zaproponowano [2] do implementacji indeksu czasowego. W projekcie wykorzystano indeksy GIN i GiST. Ich stosowność zostanie również uargumentowana w części dotyczącej reprezentacji aspektu czasu w bazach danych.

Python

Projekt został zaimplementowany w języku Python w wersji 2.7. Przyczyną takiego wyboru była chęć poznania nowego języka programowania oraz popularnego frameworka Django. Wykorzystano narzędzie do tworzenia niezależnych środowisk tego języka - `virtualenv`. Dzięki niemu, dla każdego realizowanego projektu można ustawić osobne środowisko. W łatwy sposób można również zarządzać pakietami, dzięki narzędziu o nazwie „`pip`”, które wspomaga `m.in.` instalację i wyświetlanie informacji na temat pakietów.

Django

Do implementacji systemu wykorzystano platformę programistyczną Django. Framework ten umożliwia tworzenie aplikacji o architekturze zbliżonej do znanego wzorca MVC (ang. *Model-View-Controller*). Django bazuje na wzorcu projektowym Model-View-Template, gdzie rolę kontrolera z MVC przejmują widoki (ang. *views*), a widoków - szablony HTML (ang. *templates*).

Platforma programistyczna Django została wybrana ze względu na przejrzystą strukturę aplikacji, przystępną i kompletną dokumentację oraz możliwość rozbudowania jej funkcjonalności dzięki zaimportowaniu gotowych pakietów. Jednym z takich dodatków, jest pakiet `schedule`, umożliwiający okresowe wykonywanie instrukcji, np. uaktualniających bazę danych. Innym jest `dbarray`, pozwalający na rozszerzenie listy dostępnych pól modelu o typ odpowiadający `array` w PostgreSQL. Aktualnie wspierane są tylko tablice jednowymiarowe, a jedynym obsługiwanym typem czasowym w tablicach jest `Date`.

Git

System kontroli wersji Git oprócz opcji wspólnych dla innych tego typu narzędzi, umożliwia binarne przeszukiwanie zbioru wersji w celu znalezienia tej, która powoduje błąd, oraz dodawanie plików do punktu pośredniego (tzw. *staging area*) przed ich zatwierdzeniem.

HTML5

HTML5 jest językiem do tworzenia i prezentowania stron internetowych. W projekcie jest on używany w plikach w folderze `templates` każdej z aplikacji Django.

jQuery

Biblioteka ułatwiająca korzystanie z języka JavaScript, w tym manipulację drzewem DOM, pozwoliła na wprowadzenie dynamiki do interfejsu użytkownika.

CSS/LESS

Do projektowania interfejsu użytkownika wykorzystano pliki LESS kompilowane do plików CSS. Język LESS umożliwia tworzenie zmiennych, funkcji i zagnieżdżeń, czym przewyższa wspomniane arkusze stylów. Dodatkowo użyto biblioteki graficznych interfejsów Bootstrap.

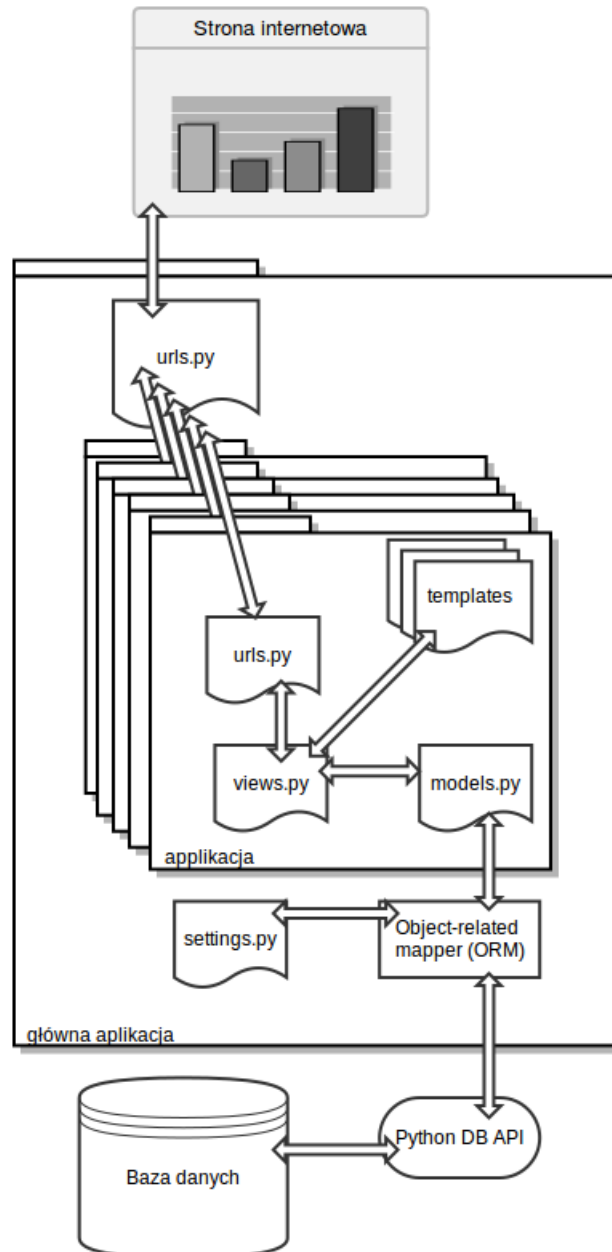
3.3. Struktura logiczna

Framework Django narzuca wykorzystanie schematu Model-View-Template, dostarcza przy tym metod ułatwiających implementację zgodną z tym schematem. Komunikacja z bazą danych odbywa się przy użyciu Django Object-relational mapper (ORM). Komponent ten umożliwia przekształcanie modelu aplikacji napisanego w całości w języku Python na model bazy danych oraz obsługuje zapytania i wprowadzanie zmian do bazy, korzystając z interfejsu języka Python. Jest to bardzo wygodne rozwiązanie, stanowiące jednak pewne ograniczenie. Dlatego obok korzystania z gotowych metod komunikacji z bazą danych, Django umożliwia również wywoływanie komend w języku SQL.

System składa się z aplikacji, z których każda powinna odpowiadać pewnemu zbiorowi funkcjonalności. Aplikacja posiada pliki wymagane przez strukturę projektu Django - MVT - i są to przedstawione na rysunku 3.1:

- plik `models.py`, zawierające klasy, będące dla ORM materiałem do tworzenia relacji w bazie danych;
- plik `views.py`, zawierający definicje klas i funkcji, stanowiących logiczną część systemu;
- folder `templates`, przechowujący w podfolderze z nazwą aplikacji pliki szablonów HTML;
- plik `urls.py`, składający się z par: adres url, klasa/funkcja z pliku `views.py`.

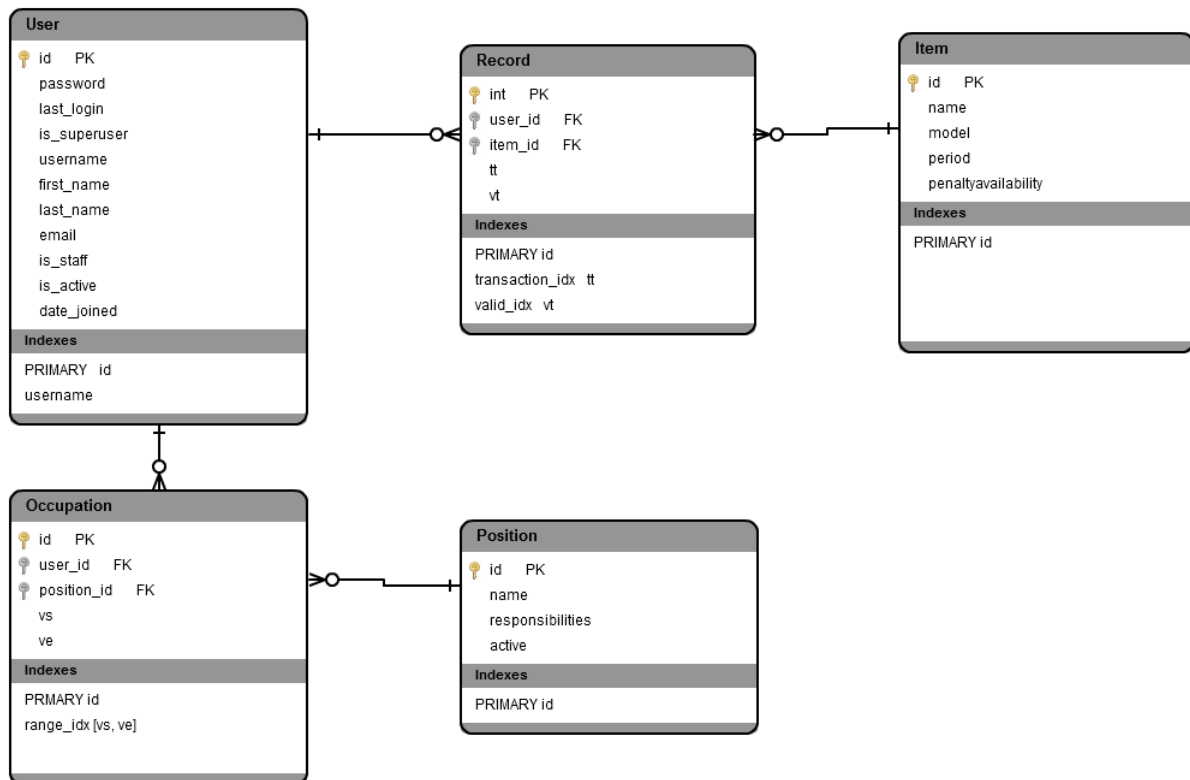
Logika aplikacji zawarta jest w widokach, które poprzez funkcje i klasy, przekazują do szablonów (`templates`) odpowiednie dane modeli, w odpowiedzi na żądania - adresy url. Główna aplikacja może zawierać plik `urls.py`, będący wyjściowym dla podobnych plików aplikacji potomnych. Plik `settings.py` specyfikuje, które aplikacje w folderze projektu mają zostać włączone do systemu, w tym, modele których z nich ma wziąć pod uwagę ORM w czasie budowania bazy. Plik ten dodatkowo definiuje parametry niezbędne do połączenia z bazą.



Rys. 3.1: Schemat systemu przedstawiający komunikację między użytkownikiem a bazą danych za pomocą Frameworku Django (na podstawie <http://www.oracle.com/ocom/groups/public/@otn/documents/digitalasset/113150.gif>)

3.4. Schemat bazy danych

Baza danych zawiera 5 tabel, utworzonych przez ORM z klas zdefiniowanych w plikach `models.py` każdej z aplikacji (przedstawiono je na rysunku 3.2) oraz dodatkowych tabel obsługujących działanie samego frameworka. Zastosowany typ `array` jest zbiorem elementów pewnego typu prostego. Jego zalety to m.in. możliwość dopisywania na początku i na końcu tablicy, a od wersji 9.3 także usuwanie elementów, lub zamiana wartości. Dodatkowo schemat bazy danych jest dzięki niemu bardziej czytelny, a w omawianych projekcie wiernie oddaje ideę koncepcyjnego modelu bitemporalnego.



Rys. 3.2: Schemat bazy danych

Każda z tabel tworzonych z klas zdefiniowanych w aplikacjach modeli, zawiera domyślnie automatycznie inkrementowany atrybut `id`, będący kluczem głównym relacji. Każdorazowo przy zapisywaniu nowo utworzonego obiektu klasy modelu, przyporządkowana zostaje mu taka unikalna, niezerowa wartość. W ten sposób framework Django kontroluje integralność encji.

Zgodnie z zasadami integralności krotki, każda powinna opisywać jeden obiekt świata rzeczywistego; Krotki relacji `User`, `Item`, `Positions` opisują kolejno instancje: użytkownika systemu, przedmiotu i pozycji w firmie. Natomiast `Records` - instancję wypożyczenia przedmiotu przez użytkownika, które mogło powtarzać się w czasie, a `Occupation` - jednorazowego zatrudnienia użytkownika na stanowisku. Dziedziny atrybutów wszystkich relacji definiowane są przy użyciu pól klasy `django.db.models.Model`. Dodatkowo, relacja `Item` posiada ograniczenie, by wartości atrybutów „name” oraz „model” krotek tworzyły pary unikalne w relacji.

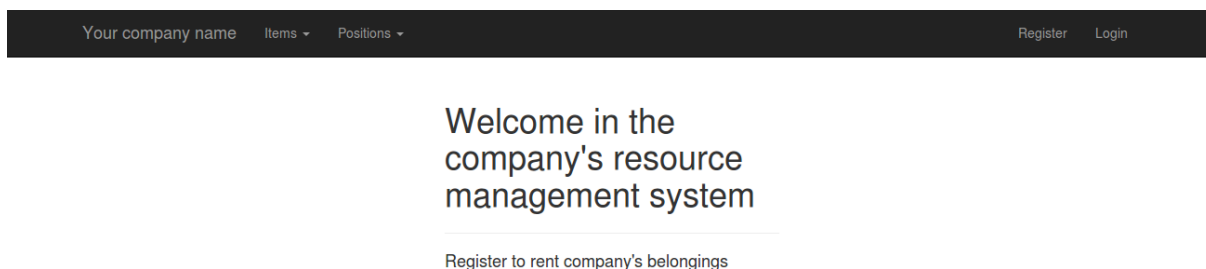
Więzy zbioru krotek określają zależności pomiędzy relacjami. W omawianym projekcie tabelami zależnymi od innych, są relacje temporalne: `Record` i `Occupation`, posiadające w zbiorze atrybutów klucze prywatne relacji, przedstawionych jako sąsiednie na rysunku 3.2.

Rozdział 4

Przykład działania

4.1. Logowanie i profil użytkownika

Zgodnie z postawionymi wymaganiami aplikacja powinna udostępniać przyjazny interfejs dla użytkowników, umożliwiając im poruszanie się po niej bez wcześniejszego doświadczenia. Ekran powitalny, przedstawiony na rysunku 4.1, zawiera informację o funkcji systemu oraz sugeruje założenie konta. Jest to możliwe poprzez formularz dostępny pod przyciskiem *Register*.



Rys. 4.1: Ekran powitalny aplikacji

Formularz, pokazany na rysunku 4.2, składa się z pól modelu użytkownika (dostępnego w pakiecie `django.contrib.auth`). Przy przetwarzaniu formularza, wykonywane jest sprawdzenie, czy:

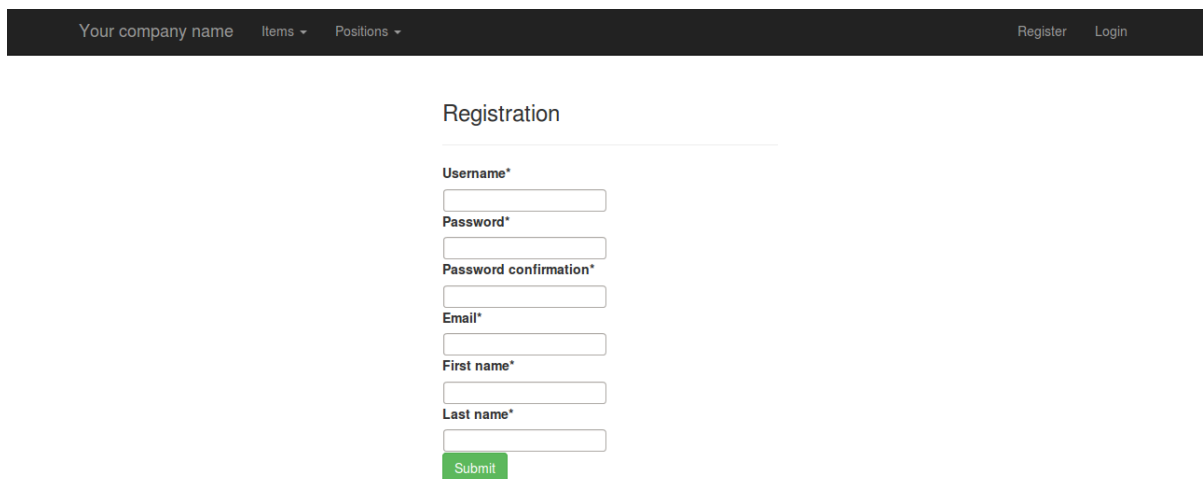
- identyfikator „username” jest unikalny,
- wprowadzony adres e-mail jest poprawnym adresem,
- wprowadzone hasła są zgodne.

Po walidacji, dane są zapisywane - tworzony zostaje nowy rekord tablicy „auth_user” w bazie danych - odpowiadającej klasie modelu „User”. Poza polami dostępnymi w formularzu, zapisywane są dane takie jak:

- `user_permissions`,
- `is_staff`,
- `is_active`,
- `is_superuser`,

- last_login,
- date_joined.

W projekcie wykorzystano pole „is_staff” do oznaczenia użytkowników, mających dostęp do funkcji zarządzania wypożyczeniami i zatrudnieniami.

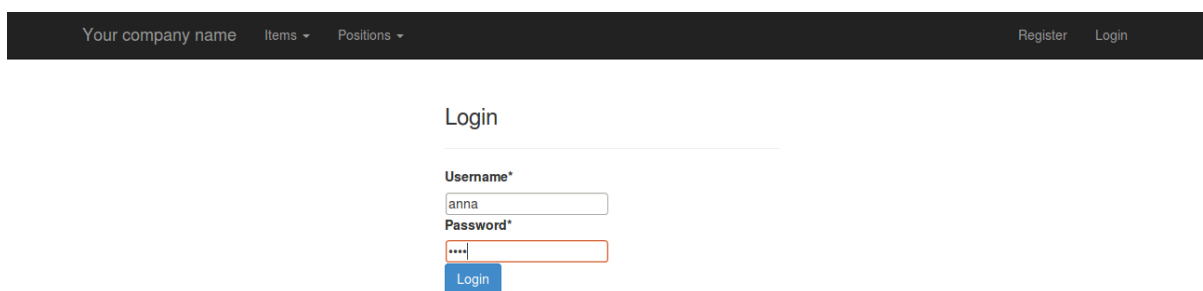


The screenshot shows a web application header with 'Your company name', 'Items', and 'Positions' dropdown menus, and 'Register' and 'Login' links. Below the header is a 'Registration' form with the following fields: Username*, Password*, Password confirmation*, Email*, First name*, and Last name*. Each field has a corresponding input box. At the bottom of the form is a green 'Submit' button.

Rys. 4.2: Formularz rejestracji

Podobne formularze skonstruowano dla pozostałych relacji: dla dodawania przedmiotów, stanowisk w firmie, rekordów wypożyczeń i zatrudnień. Jednak te strony dostępne są tylko dla użytkowników, których pole „is_staff” ma wartość True.

Uwierzytelnienie odbywa się po wprowadzeniu danych do formularza logowania, przedstawionego na rysunku 4.3.



The screenshot shows the same web application header as in Figure 4.2. Below the header is a 'Login' form with two fields: Username* and Password*. The Username field contains the text 'anna'. The Password field contains four asterisks '****'. Below the fields is a blue 'Login' button.

Rys. 4.3: Formularz logowania do systemu

Strona profilowa, przedstawiona na rysunku 4.4, dostępna jest po zalogowaniu i wyświetla dane użytkownika, wprowadzone przy rejestracji. Dodatkowo, pokazana jest informacja na temat aktualnej pozycji zajmowanej w firmie, lub w przypadku jej braku, odnośnik do strony listy dostępnych stanowisk. Poniżej pokazana jest lista wypożyczanych przedmiotów, wraz z datami dotyczącymi ostatniego wypożyczenia. Jeśli przedmiot nie został zwrócony, miejsce wartości „Return date” zajmuje odnośnik umożliwiający zwrot. Po jego naciśnięciu, pojawia się okno dialogowe z prośbą o potwierdzenie akcji, podobne do przedstawionych poniżej.

4.2. System wypożyczeń

Odnośnik pod przyciskiem „Items” przekierowuje do strony dostępnych przedmiotów, pokazanej na rysunku 4.5. Zalogowany użytkownik może użyć przycisku w kolumnie „Order” do zasygnalizowania chęci wypożyczenia sprzętu. Zobaczy wtedy okno dialogowe, przedstawione na rysunku 4.6, z prośbą o potwierdzenie.

CompanyName	Items ▾	Positions ▾	kasawi	Logout
-------------	---------	-------------	--------	--------

Your profile

[kasawi](#)

kasawi@example.com

Kasia Sawicka

Your current position: Senior JAVA programmer

Name	Model	Rent date	Return date	Due date
MacBook Air	0.1	02/02/2013	02/02/2013	02/02/2013
Skoda Octavia	Kombi	02/02/2013	02/02/2013	02/02/2013
Asus	X24235SF	02/01/2013	02/02/2013	02/02/2013
Samsung	X300	02/01/2013	02/02/2013	02/02/2013
Samsung	X300i	02/01/2013	02/02/2013	02/02/2013
Samsung	ghx	02/01/2013	02/02/2013	02/02/2013

Rys. 4.4: Strona profilowa użytkownika

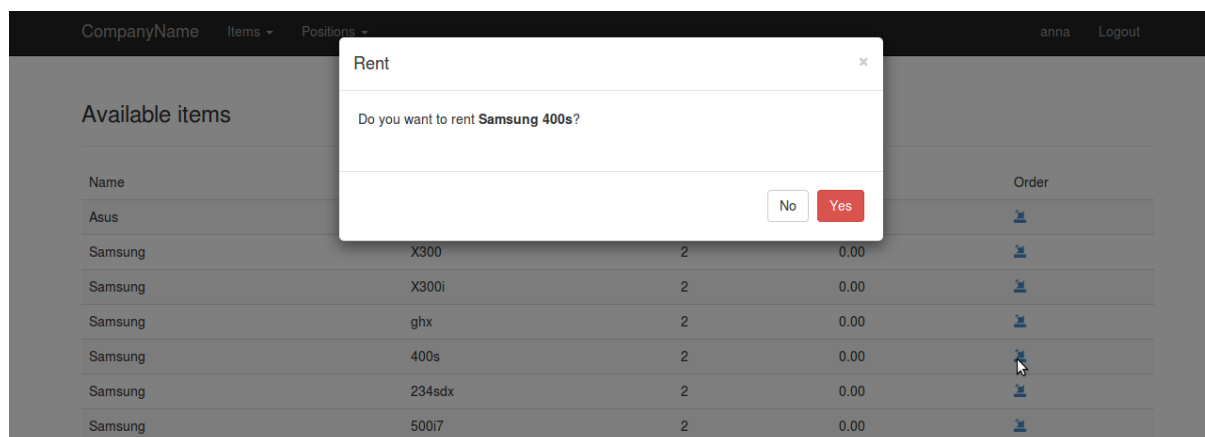
CompanyName	Items ▾	Positions ▾	kasawi	Logout
-------------	---------	-------------	--------	--------

Available items

Name	Model	Period	Penalty	Order
MacBook Air	0.1	1	10.00	📄
Skoda Octavia	Kombi	3	20.00	📄
Skoda Fabia	Sedan	6	1.00	📄
MacBook Pro	0.1	3	7.00	📄
Asus	X24235SF	2	0.00	📄
Samsung	X300	2	0.00	📄

Rys. 4.5: Lista obiektów dostępnych do wypożyczenia

Administrator systemu ma w stosunku do innych użytkowników dodatkowe prawa. Może dodawać informacje o nowych przedmiotach, poprzez formularz pokazany na rysunku 4.7. Podział identyfikatora przedmiotu na pola nazwy oraz modelu ma na celu ułatwienie wprowadzania danych o przedmiotach tych samych kategorii, tzn nazwa może oznaczać markę przedmiotu, a model - konkretny egzemplarz. Chociaż pole modelu nie jest wymagane, nie ma możliwości wprowadzenia dwóch przedmiotów o tej samej nazwie z pustym polem modelu, ponieważ dodano wymaganie, aby para (nazwa, model) była unikalna. Pole „Period” oznacza liczbę dni, na którą domyślnie ma zostać wypożyczony sprzęt, natomiast pole „Penalty” - kwotę kary naliczaną za każdy dzień opóźnienia zwrotu.



Rys. 4.6: Akcja wypożyczenia przedmiotu

The screenshot shows a web application interface with a dark header bar containing 'CompanyName', 'Items', 'Positions', 'anna', and 'Logout'. The main content area is titled 'Add new item' and contains a form with the following fields: Name* (text input), Model (text input), Period* (text input), Penalty* (text input with value 0.00), and an 'Add' button.

Add new item

Name*

Model

Period*

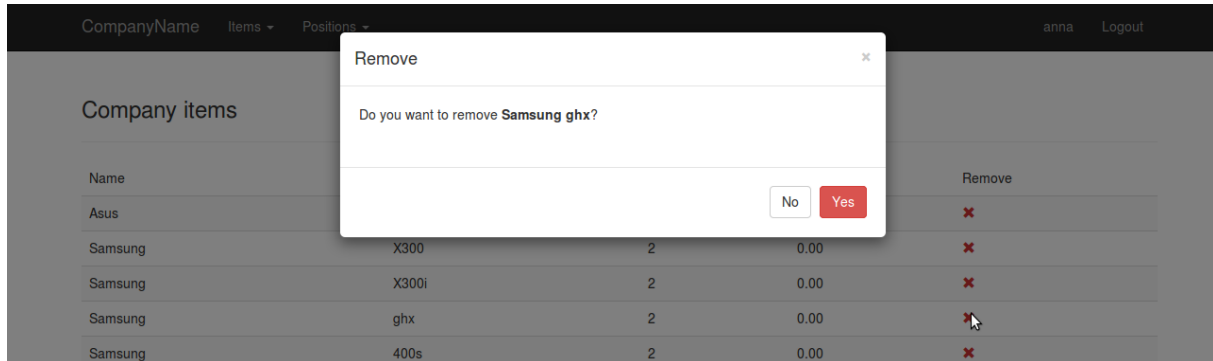
Penalty*

0.00

Add

Rys. 4.7: Formularz dodawania przedmiotu do bazy

Użytkownik z prawami administratora może dodatkowo usuwać przedmioty z listy dostępnych (rys. 4.8). Wycofanie produktu ma charakter logiczny, tzn. obiekt w dalszym ciągu jest obecny w bazie, tak aby dostępna była historia jego wypożyczeń. Taki produkt nie będzie jednak widoczny dla użytkowników.



Rys. 4.8: Akcja usuwania przedmiotu z zasobów firmy

Zastosowanie obu wartości: *transaction time* oraz *valid time*, do opisu czasu w systemie, pozwala na dokładne śledzenie wypożyczeń: momentów uzyskania i oddania przedmiotu, terminowości zwrotu, przewidywanego terminu dostępności przedmiotu. Szczegółowa historia wypożyczeń wszystkich przedmiotów dostępna jest jedynie dla administratora dzięki stronie przedstawionej na rysunku 4.9.

CompanyName

Items ▾

Positions ▾

anna

Logout

Items rental hisotry

Name	Model	Period	Penalty	Available	Rented	Returned	Due
Skoda Fabia	Sedan	2	1.00	True	01/01/2013	01/02/2013	01/02/2013
Skoda Fabia	Sedan	2	1.00	True	01/11/2013	01/12/2013	01/12/2013
Skoda Fabia	Sedan	2	1.00	True	01/21/2013	01/22/2013	01/22/2013
Skoda Fabia	Sedan	2	1.00	True	02/01/2013	02/02/2013	02/02/2013
Skoda Fabia	Sedan	2	1.00	True	02/11/2013	02/12/2013	02/12/2013
Skoda Fabia	Sedan	2	1.00	True	02/21/2013	02/22/2013	02/22/2013
Skoda Fabia	Sedan	2	1.00	True	03/01/2013	03/02/2013	03/02/2013
Skoda Fabia	Sedan	2	1.00	True	03/11/2013	03/12/2013	03/12/2013
Skoda Fabia	Sedan	2	1.00	True	03/21/2013	03/22/2013	03/22/2013
Skoda Fabia	Sedan	2	1.00	True	04/01/2013	04/02/2013	04/02/2013
Skoda Fabia	Sedan	2	1.00	True	04/11/2013	04/12/2013	04/12/2013

Rys. 4.9: Spis wszystkich zarejestrowanych wypożyczeń

4.3. Zarządzanie zatrudnieniami

W rozwijanej liście pod przyciskiem „Positions” w pasku menu, oprócz opcji dostępnych dla użytkowników:

- „Free” - kierującej do strony dostępnych przedmiotów, z możliwością wypożyczenia,

- „Taken” - prowadzącej do listy przedmiotów wypożyczonych,
- „List” - kierującej do listy wszystkich sprzętów, aktualnie dostępnych w firmie,

posiada dwie dodatkowe. Jedna prowadzi do formularza do dodawania pozycji, podobnego do omawianego przy dodawaniu przedmiotów. Ostatnia natomiast, umożliwia przeglądanie historii zmian na stanowiskach, jak na rysunku 4.10. Na rysunku 4.11 przedstawiono wspomniany

Company Name Items ▾ Positions ▾ anna Logout				
Employment history				
Employee	Position	Since	Until	Duration
ansob	Senior JAVA programmer	Jan. 1, 2013	✖ now.	345 days
pajas	Junior JAVA programmer	Jan. 1, 2013	🕒 Dec. 12, 2013	345 days
olpaw	Security Analyst	Jan. 1, 2013	✖ now.	345 days
pajas	HR manager	Jan. 1, 2013	✖ now.	345 days
juwis	CEO	Jan. 1, 2013	✖ now.	345 days
wadud	Senior JAVA programmer	Jan. 1, 2011	🕒 Dec. 31, 2012	730 days
bechmie	CEO	Jan. 1, 2011	🕒 Dec. 31, 2012	730 days
juwie	Junior JAVA programmer	Jan. 1, 2011	🕒 Dec. 31, 2012	730 days
ansob	Security Analyst	Jan. 1, 2011	🕒 Dec. 31, 2012	730 days
juwie	HR manager	Jan. 1, 2011	🕒 Dec. 31, 2012	730 days
kakow	HR manager	Jan. 1, 2010	🕒 Dec. 31, 2010	364 days
kakow	Junior JAVA programmer	Jan. 1, 2010	🕒 Dec. 31, 2010	364 days

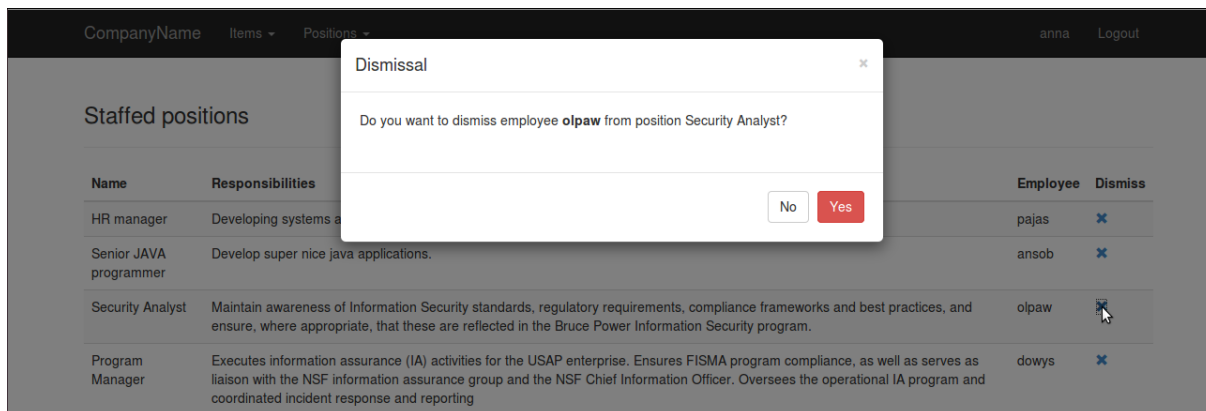
Rys. 4.10: Historia zatrudnienia pracowników

Company Name Items ▾ Positions ▾ anna Logout				
Staffed positions				
Name	Responsibilities	Employee	Dismiss	
HR manager	Developing systems and processes within the organization that address the strategic needs of the business	pajas	✖	
Senior JAVA programmer	Develop super nice java applications.	ansob	✖	
Security Analyst	Maintain awareness of Information Security standards, regulatory requirements, compliance frameworks and best practices, and ensure, where appropriate, that these are reflected in the Bruce Power Information Security program.	olpaw	✖	
Program Manager	Executes information assurance (IA) activities for the USAP enterprise. Ensures FISMA program compliance, as well as serves as liaison with the NSF information assurance group and the NSF Chief Information Officer. Oversees the operational IA program and coordinated incident response and reporting	dowys	✖	

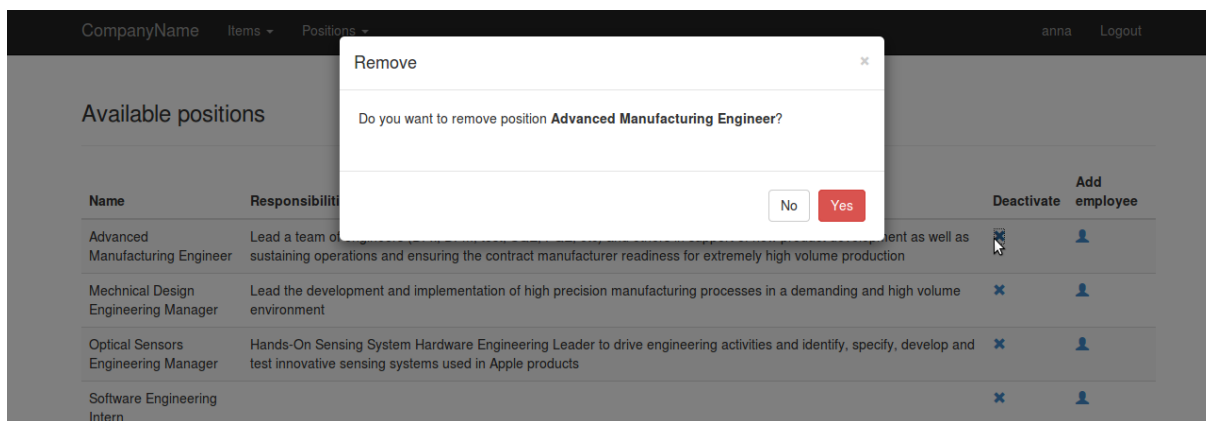
Rys. 4.11: Lista zatrudnionych użytkowników z możliwością ich zwolnienia

ekran z listą obsadzonych pozycji. Jest to widok dostępny dla administratora - zawiera przycisk umożliwiający usunięcie pracownika z pozycji (rys. 4.12). Pozycje zajęte to te, których rekordy zatrudnienia są obowiązujące w chwili obecnej, co w bazie oznaczone jest wartością „now” w miejscu końcowego chrononu wartości *valid time*. Zwolnienie użytkownika skutkuje zmianą wartości „now” na daty wykonania akcji dla tego rekordu.

Podobnie jak przy usuwaniu przedmiotów, rekordy dotyczące deaktywowanej pozycji nie są usuwane. Zamiast tego, zmieniane jest pole dostępności pozycji, tak, aby nie była ona wyświetlana w liście wolnych stanowisk. Deaktywację pozycji z listy aktualnie nieokupowanych przedstawiono na rysunku 4.13.



Rys. 4.12: Usunięcie użytkownika z listy zatrudnionych



Rys. 4.13: Usunięcie pozycji z dostępnych stanowisk w firmie

Rozdział 5

Implementacja

5.1. Modele

Framework Django posiada *Object-relational mapper*, który na podstawie klas modeli tworzy relacje w bazie danych. Aby możliwe było wykorzystanie typu `array` w bazie, należało zainstalować dodatkowy pakiet *django-dbarrray*.

Model użytkownika kreowany jest automatycznie przy tworzeniu aplikacji, co umożliwia korzystanie z wbudowanych funkcji, np. dekoratora `@staff_member_required` pakietu *django.contrib.admin.views.decorators*, gdy do pewnych stron mogą mieć dostęp jedynie osoby z odpowiednimi uprawnieniami. Model `Record` (definiowany kodem przedstawionym na listingu 5.1), stanowiący połączenie między użytkownikiem a wypożyczonymi przedmiotami, oraz model `Occupation` (listing 5.2), łączący osoby z pozycjami w firmie, musiały być szczególnie starannie zaprojektowane.

Listing 5.1: Model `Record`, zawierający pola typu `DateArrayField` klasy *dbarrray*

```
class Record(models.Model):
    user = models.ForeignKey(User)
    item = models.ForeignKey(Item)
    tt = dbarrray.DateArrayField(blank=True, null=True)
    vt = dbarrray.DateArrayField(blank=True, null=True)

    def __unicode__(self):
        return self.user + "_" + self.item.name
```

Dodawanie i edytowanie rekordów tej tabeli wykonywane jest zgodnie z regułami BCDM i jest opisane w następnej części.

Listing 5.2: Model `Occupation` z właściwościami, ułatwiającymi filtrowanie wyników

```
class Occupation(models.Model):
    user = models.ForeignKey(User)
    position = models.ForeignKey(Position)
    vs = models.DateField(auto_now_add=True)
    ve = models.DateField(default=common.future_now)

    def __unicode__(self):
        return self.user + "_as_" + self.position.name

    @property
    def is_current(self):
        if self.ve > date.today():
            return True
        return False

    @property
    def duration(self):
```

```

if self.is_current:
    return (date.today()-self.vs).days
return (self.ve-self.vs).days

```

Zgodnie z modelami danych z implementacją aspektu czasu na poziomie krotki, wartość *valid time* przedstawiona jest za pomocą wielkości *vs* oraz *ve*, gdzie pierwsza z nich to chronon oznaczający początek, a druga - chronon oznaczający koniec interwału. Dodatkowe właściwości pozwalają na manipulację danymi (filtrację wyników, obliczenia) na poziomie aplikacji.

5.2. Widoki - logika aplikacji

Widoki w Django stanowią połączenie między adresami URL a odpowiednimi szablonami stron oraz odpowiadają za pobieranie danych o polach modeli z bazy danych.

Dodawanie przedmiotu

Niektóre z widoków korzystają z formularzy, będącymi klasami dziedziczącymi po klasie *forms.ModelForm*. Formularz w najprostszej formie zawiera pola tekstowe odpowiadające polom modelu. Wybrane z nich można ukryć, by zdefiniować ich wartość w funkcji widoku, co zastosowano w kodzie z listingów 5.3 oraz 5.4. Poniżej przedstawiono przykład implementacji takiego rozwiązania. Na podobnej zasadzie odbywa się rejestracja użytkownika oraz dodawanie stanowisk pracy.

Listing 5.3: Przykład formularza: formularz dodawania przedmiotu

```

class AddItemForm(forms.ModelForm):
    class Meta:
        model = Item
        exclude = ('availability',)

```

Listing 5.4: Przykład obsługi formularza: pobranie danych wprowadzonych przez użytkownika, dodanie wartości pola wykluczonego z formularza oraz zapisanie powstałego rekordu

```

@staff_member_required
@render_to('items/add_new_item.html')
def add_item(request):
    item_form = AddItemForm(request.POST or None)

    if item_form.is_valid():
        obj = item_form.save(commit=False)
        obj.availability = True
        obj.save()

        return {'form': AddItemForm()}
    return {'form': item_form}

```

Dokonywanie wypożyczenia

Zgodnie z BCDM, polega na utworzeniu nowej krotki, bądź znalezieniu istniejącej i jej uaktualnieniu. Zarejestrowana osoba może wypożyczyć przedmiot z listy dostępnych sprzętów. Wtedy wykonywana jest funkcja z listingu 5.5, gdzie *id* jest identyfikatorem przedmiotu. Funkcja *get_or_create* wywołana na domyślnym menadżerze modelu *Record* zwraca istniejącą krotkę lub tworzy nową w razie jej braku. Programista musi zadbać o to, aby nie istniało więcej niż jedna krotka odpowiadająca zapytaniu. Wynik zapisany jest w bazie, następnie wywoływana jest funkcja z pakietu *common* (listing 5.6), kreująca odpowiedni ciąg znaków, do wywołania jako uaktualnienie w języku SQL. Funkcja ta jest również wykorzystywana przy okresowym uaktualnianiu bazy.

Listing 5.5: Funkcja `request_item(request, id)` wykonywana przy dokonywaniu wypożyczenia

```
@require_POST
def request_item(request, id):
    obj, created = Record.objects.get_or_create(user=request.user,
                                                item=Item.objects.get(id=id))

    obj.save()
    period = Item.objects.get(id=id).period
    common.update_record(request.user.id, id, period)
    return HttpResponse('OK')
```

Listing 5.6: Funkcja `update_record(user, item, period)` do aktualizacji rekordu wypożyczenia

```
def update_record(user, item, period):
    now = datetime.now().date()
    cursor = connection.cursor()
    update_tt = "UPDATE_records_record_SET_tt=_tt_||_array[_"
    update_vt = "UPDATE_records_record_SET_vt=_vt_||_array[_"
    #add tt - today + vt for each valid day
    for i in range(period):
        due = (datetime.now()+timedelta(days=i)).date()
        update_tt += "to_date(\'" + str(now) + "\',_\'YYYY-MM-DD\'),_"
        update_vt += "to_date(\'" + str(due) + "\',_\'YYYY-MM-DD\'),_"
    #add logical now at the end
    for i in range(period-1):
        due = (datetime.now()+timedelta(days=i)).date()
        update_tt += "to_date(\'" + logical_now + "\',_\'YYYY-MM-DD\'),_"
        update_vt += "to_date(\'" + str(due) + "\',_\'YYYY-MM-DD\'),_"
    due = (datetime.now()+timedelta(days=period-1)).date()
    selector = "_WHERE_user_id=_" + str(user) + "_AND_item_id=_" + str(item)
    update_tt += "to_date(\'" + logical_now + "\',_\'YYYY-MM-DD\'])" + selector
    update_vt += "to_date(\'" + str(due) + "\',_\'YYYY-MM-DD\'])" + selector

    cursor.execute(update_tt)
    cursor.execute(update_vt)
```

Oddanie przedmiotu

Jest równoważne z usunięciem elementów tablicy *transaction time*, których wartość jest równa „now()” (dla wygody oznaczona w projekcie jako najmniejsza możliwa data - '0001-01-01'), oraz odpadającym im zapisom w tablicy *valid time*. Odbywa się to przez wywołanie, w sposób pokazany na listingu 5.7, funkcji zdefiniowanej w języku SQL (przedstawionej na 5.8).

Listing 5.7: Funkcja `return_item(request, id)` wywołująca funkcję SQL o tej samej nazwie, gdzie `id` to identyfikator przedmiotu do zwrócenia

```
@require_POST
def return_item(request, id):
    cursor = connection.cursor()
    cursor.execute(common.return_item(request.user.id, id))
    return HttpResponse('OK')
```

Listing 5.8: Funkcja SQL `update_record(int, int)` do aktualizacji rekordu wypożyczenia przy zwracaniu przedmiotu; Jako argumentu przyjmuje `id` użytkownika i `id` zwracanego przedmiotu.

```
CREATE OR REPLACE FUNCTION return_item(int, int)
RETURNS text AS
$$
DECLARE
    the_user ALIAS FOR $1;
    the_item ALIAS FOR $2;
```

```

        myoutput text := 'OK';
BEGIN

UPDATE records_record
    SET tt = array_remove(
        (SELECT tt FROM records_record
         WHERE user_id = the_user
         AND item_id = the_item), to_date('0001-01-01','YYYY-MM-DD'))
    WHERE user_id = the_user AND item_id = the_item;

UPDATE records_record
    SET vt = vt[1:array_length(tt,1)]
    WHERE user_id = the_user AND item_id = the_item;

RETURN myoutput;
END;
$$
LANGUAGE plpgsql;

```

Wyświetlanie szczegółowej listy przedmiotów

Jest funkcją dostępną jedynie dla członków zespołu zarządzającego bazą. Administratorzy powinni w szczegółów wypożyczeń, m.in. daty wypożyczenia, daty zwrotu, jeśli sprzęt jest aktualnie w magazynie, oraz terminu zwrotu. Funkcja `item_details()`, przedstawiona na listingu 5.9, zwraca listę przedmiotów, z odpowiadającym im zbiorem tych trzech wielkości:

- datą wypożyczenia - *rented* - uzyskana z kolumny *valid time*, ponieważ dzień wypożyczenia jest pierwszym dniem, kiedy było ono prawomocne;
- datą zwrotu - *returned* - ostatnia wartość z kolumny *transaction time* lub wartość pusta, jeśli zwrot nie nastąpił;
- przewidywaną data oddania - *due* - ostatnia wartość z kolumny *valid time*.

Listing 5.9: Funkcja SQL `item_details()` zwracająca elementy tabeli `Item` ze szczegółami dotyczącymi ostatniego wypożyczenia

```

CREATE OR REPLACE FUNCTION item_details()
    RETURNS TABLE (
        id          int
        ,name       varchar(128)
        ,model      varchar(128)
        ,period     int
        ,penalty    numeric(8,2)
        ,available  boolean
        ,rented     varchar(10)
        ,returned   varchar(10)
        ,due        varchar(10)
    ) AS
$func$
BEGIN

    RETURN QUERY
SELECT i.id AS id,
       i.name AS name,
       i.model AS model,
       i.period AS period,
       i.penalty AS penalty,
       i.availability AS available,
       CASE WHEN array_length(r.tt, 1) = 1 THEN to_char(CURRENT_DATE, 'MM/DD/YYYY')
       ELSE rent_day(r.vt, i.period) END AS rented,
       CASE WHEN r.tt[array_length(r.tt, 1):array_length(r.tt, 1)] =
           ARRAY['0001-01-01']::date[]
       THEN '' ELSE get_day_or_null(r.tt) END AS returned,

```



```

        get_day_or_null(r.vt) AS due
FROM items_item i LEFT OUTER JOIN records_record r ON i.id = r.item_id;

END
$func$ LANGUAGE plpgsql;

```

Wyświetlanie danych wypożyczeń użytkownika

Ma miejsce przy przeglądaniu profilu użytkownika. Wszyscy członkowie powinni mieć dostęp do dat dotyczących ostatnich wypożyczeń sprzętów, stąd funkcja użyta w tym przypadku jest zbliżona do opisanej powyżej.

Lista dostępnych przedmiotów

Jest zbiorem tych rekordów tabeli Item, których kolumna *transaction time* nie zawiera wartości „now()” (tzn przedmiot nie jest aktualnie nikomu wypożyczony). W języku SQL użyto operatora @>, oznaczającego zawieranie przy operacjach na przedziałach, co pokazano na 5.10. Do szablonu przekazany jest zbiór takich przedmiotów, oraz jego liczebność, ponieważ w razie braku takich przedmiotów należy poinformować o tym użytkowników.

Listing 5.10: Zapytanie SQL wywoływane w funkcji `free_items()`

```

SELECT * FROM items_item i WHERE availability = TRUE
AND i.id NOT IN
  (SELECT item_id FROM records_record
   WHERE tt @> ARRAY[to_date('0001-01-01', 'YYYY-MM-DD')]);

```

Usunięcie przedmiotu odbywa się na poziomie logicznym. Ponieważ w razie usunięcia pewnego sprzętu z zasobów firmy wciąż istnieje potrzeba przechowywania historii jego wypożyczeń, przedmiot jest jedynie deaktywowany i nie będzie wyświetlany w liście dostępnych do wypożyczenia, co pokazano na listingu 5.11.

Listing 5.11: Funkcja zmieniająca pole dostępności przedmiotu

```

@staff_member_required
@require_POST
def remove_item(request, id):
    obj = get_object_or_404(Item, id=id)
    obj.availability = False
    obj.save()
    return HttpResponse('OK')

```

Zatrudnienie użytkownika, czyli przypisanie osobie aktualnie niezatrudnionej dostępnego stanowiska, odbywa się przez dodanie do tabeli Occupation rekordu z id użytkownika, id pozycji, daty dzisiejszej jako początkowy czas zatrudnienia oraz wartości „now()”, jako czas końcowy. Ta ostatnia, w celu skonstruowania poprawnego indeksu, musi mieć wartość większą niż data zapisu. W odróżnieniu od omawianego systemu wypożyczeń, w tym przypadku wartość reprezentowana jest przez datę '3001-01-01'.

Zwolnienie stanowiska jest równoważne z wprowadzeniem daty końcowej zatrudnienia w miejsce domyślnie wpisanej wartości „now()”.

Ponowne zatrudnienie pracownika na stanowisku, które piastował już wcześniej, powoduje dodanie nowego rekordu do tabeli, jak pokazano na listingu 5.12.

Listing 5.12: Funkcja przywracająca pracownika na stanowisko

```

@staff_member_required
@require_POST
def renew_occupation(request, id):
    o_record = get_object_or_404(Occupation, id=id)
    employee = o_record.user
    job = o_record.position
    obj = Occupation.objects.create(user=employee, position=job)
    obj.vs = date.today()
    obj.ve = common.future_now
    obj.save()
    return HttpResponse('OK')

```

5.3. Znane strategie indeksowania

Indeks czasowy powinien przede wszystkim wspierać wyszukiwanie wersji obiektów aktualnych w pewnym okresie czasu. PostgreSQL oferuje dwie struktury indeksowe wspierające operacje na przedziałach: GiST oraz GIN. Obie przyspieszają zapytania z filtrami, używającymi operatorów:

- $< @$ „zawiera”,
- $@ >$ „zawiera się w”,
- $\&\&$ „posiada część wspólną z”.

Twórcy idei indeksu temporalnego [2] zaproponowali użycie struktury B+-drzewa do indeksowania danych zmiennych w czasie. Choć indeks GiST ma domyślnie strukturę B-drzewa, można go dostosować do zachowania B+-drzewa przez nadpisanie kilku kluczowych metod [4]. Do ich implementacji wykorzystano funkcje pomocnicze [4]:

$Contains([x, y], v)$ zwraca *True* jeśli przedział $[x, y]$ zawiera wartość v .

$Equal(x, y)$ zwraca wartość *True*, jeśli $x = y$.

Metody opisujące zachowanie indeksu GiST to według [4]:

$Consistent(E, q)$

Jest funkcją używaną przy wyszukiwaniu. Wielkość E oznacza wystąpienie $E = (p, ptr)$, gdzie p jest twierdzeniem takim, że $p = Contains([x_p, y_p], v)$, a ptr wskaźnikiem to węzła wskazującego do wartości zgodnych z tym twierdzeniem. Drugi argument funkcji, q , jest zależny od zapytania i ma postać $q = Contains([x_q, y_q], v)$ lub $q = Equal(x_q, v)$. W pierwszym przypadku, zwracana jest wartość *True*, wtedy i tylko wtedy, gdy $(x_p < y_q) \wedge (y_p > x_q)$. W drugim, jeśli $x_p \leq x_q < y_p$.

$Union\{E_1, E_2, \dots, E_n\}$

Dla wystąpień $E_1 = ([x_1, y_1], prt_1), \dots, E_n = ([x_n, y_n], prt_n)$ zwraca przedział od najmniejszego do największego krańca przedziałów wystąpień, tzn $[MIN(x_1, x_2, \dots, x_n), MAX(y_1, y_2, \dots, y_n)]$.

Compress($E = ([x, y], ptr)$)

Dokonyje kompresji wystąpienia, zawierającego przedział jako jeden z elementów. Jeśli węzeł jest najbardziej wysuniętym na lewo, zwracany jest pusty obiekt; W przeciwnym wypadku, zwracana jest wartość x .

Decompress($E = (\pi, ptr)$)

Dekompresja polega na rekonstrukcji pierwotnego przedziału w wystąpieniu E . Jeśli E jest kluczem na najbardziej wysuniętym na lewo węźle, nie będącym liściem, $x = -\infty$, w przeciwnym wypadku $x = \pi$. Jeśli E jest kluczem na najbardziej wysuniętym na prawo węźle, nie będącym liściem, $y = \infty$. Jeśli E jest kluczem na jakimkolwiek innym węźle, nie będącym liściem, y przypisana jest wartość sąsiedniego klucza. Jeśli natomiast E jest kluczem liścia, $y = x + 1$. Zwracana jest wartość $([x, y], ptr)$

Penalty($E = ([x_1, y_1], prt_1), F = ([x_2, y_2], prt_2)$)

Wykorzystywana jest przy wstawianiu nowych rekordów. Rekord zostanie wprowadzony do tej gałęzi, dla której wartość funkcji *Penalty* będzie najmniejsza. Metoda wykonywana na coraz niższych poziomach, aż przy porównywaniu wyników wywołań na liściach, wskaże ten, który będzie wskazywał na rekord. Jeśli E jest najbardziej wysuniętym na lewo wskaźnikiem węzła, funkcja zwraca $MAX(y_2 - y_1, 0)$. Jeśli E jest najbardziej wysuniętym na prawo wskaźnikiem węzła, funkcja zwraca $MAX(x_1 - x_2, 0)$. W innym wypadku, $MAX(y_2 - y_1, 0) + MAX(x_1 - x_2, 0)$.

Picksplit(P)

Opisuje, jak powinien zostać podzielony węzeł wierzchołkowy, gdy nie będzie w nim miejsca na dodanie nowych rekordów. Pierwsze $\lfloor \frac{|P|}{2} \rfloor$ wystąpień w węźle przeniesione zostaje do lewej grupy, a ostatnie $\lceil \frac{|P|}{2} \rceil$ - do prawej.

Funkcje rozszerzenia powinny być zdefiniowane w języku C i skompilowane do biblioteki dzielonej. Każdej funkcji musi towarzyszyć jej deklaracja w języku SQL.

5.4. Indeksowanie zastosowane w projekcie

Obecnie administratorzy baz danych stosują dostępny typ *range* i obsługujący go indeks GiST. Takie rozwiązanie zastosowano w niniejszym projekcie budując indeks na przedziale czasowym (v_s, v_e) .

Indeks GIN ma pewne zalety nad indeksem GiST. Oba są powszechnie stosowane przy przeszukiwaniu tekstów dla słów kluczowych, jednak pierwszy jest znany z szybszego wykonywania zapytań `SELECT`. Samo tworzenie indeksu trwa dłużej, podobnie jak wprowadzanie nowych rekordów, jednak przy znacznej przewadze zapytań wyszukiwujących, nie na to wielkiego znaczenia. GIN może być zastosowany dla typu tablicowego, dlatego został użyty w projekcie na relacji `Record`.

Listing 5.13: Indeksy użyte na relacjach temporalnych

```
CREATE INDEX duration_idx ON occupations_occupation USING GIST(daterange(vs, ve));
CREATE INDEX transaction_idx ON records_record USING GIN(tt);
CREATE INDEX valid_idx ON records_record USING GIN(vt);
```

Rozdział 6

Testy i wdrożenie

W omawianym projekcie najważniejsze znaczenie ma niezawodność systemu (zdolność rekonstrukcji wszystkich danych zapisywanych do bazy) oraz jego wydajność.

Pierwszą właściwość gwarantuje użycie bitemporalnego koncepcyjnego modelu danych do zapisu wypożyczeń, oraz modelu z aspektem czasu wprowadzonym na poziomie krotki w relacji opisującej zatrudnienia.

Wydajność systemu przetestowano dzięki funkcji `EXPLAIN` systemu PostgreSQL. Pozwala ona przeanalizować czas wykonania zapytania, oraz plan, jaki do jego realizacji wybrał system zarządzania bazą danych. Według informacji zamieszczonej na stronie PostgreSQL <http://www.postgresql.org/docs/9.3/static/indexes-examine.html>, indeksy będą używane jedynie przy relacjach dużych rozmiarów, np przy wybieraniu 1000 spośród 100000 wierszy, ponieważ relacja o małej liczbie krotek może być pamięta na obszarze jednej strony pamięci fizycznej. W takim przypadku sekwencyjne skanowanie relacji jest najszybsze i indeks nie jest używany. Przy przeprowadzaniu testów sprawdzono, jaka liczba wierszy jest wystarczająca, aby indeks przyspieszył wyszukiwanie.

6.1. Index GIN na relacji wypożyczeń

W relacji `records_record` w kolumnach dotyczących czasu wykorzystano typ `array`. General Inverted Index jest dostępny dla tego typu. GIN łączy wartości pól w tablicach `array` z listą tablic, w której one wystąpiły. Idealnie nadaje się do zapytań o listę tablic spełniających pewien warunek, np zawierających zadane wartości. W projekcie przewidziano m.in. użycie indeksu przy wyszukiwaniu aktualnych rekordów wypożyczeń - zawierających wartość „now”. Pierwsze testy przeprowadzono dla 300 wierszy. To jednak nie wystarczyło do wybrania planu z indeksem. Natomiast już przy ponad 800 wierszach, zastosowany został indeks GIN. Łatwo sobie wyobrazić sytuację, w której każdy z 300 pracowników firmy chociaż raz wypożyczy 3 przedmioty, co w bazie danych byłoby zapisane w 900 wierszach. Poniżej przedstawiono wyniki funkcji `EXPLAIN` dla omawianej relacji przy założonym indeksie oraz bez niego.

Listing 6.1: Wyniki wywołania funkcji `EXPLAIN` na zapytaniu `SELECT` bez założonego indeksu

```
EXPLAIN ANALYZE SELECT * FROM records_record
WHERE tt @> ARRAY[to_date('0001-01-01'::text, 'YYYY-MM-DD'::text)];

QUERY PLAN

-----
Seq Scan on records_record  (cost=0.00..26.05 rows=20 width=86)
    (actual time=0.148..2.193 rows=20 loops=1)
    Filter: (tt @> ARRAY[to_date('2013-10-01'::text, 'YYYY-MM-DD'::text)])
```

Rows Removed by Filter: 850
Total runtime: 2.236 ms

```
EXPLAIN ANALYZE SELECT * FROM records_record
WHERE tt @> ARRAY[to_date('2013-10-01'::text, 'YYYY-MM-DD'::text)];
```

QUERY PLAN

```
Seq Scan on records_record (cost=0.00..26.05 rows=20 width=86)
    (actual time=0.174..2.173 rows=20 loops=1)
    Filter: (tt @> ARRAY[to_date('2013-10-01'::text, 'YYYY-MM-DD'::text)])
    Rows Removed by Filter: 850
Total runtime: 2.215 ms
```

Listing 6.2: Wyniki wywołania funkcji EXPLAIN na zapytaniu SELECT dla relacji z indeksem GIN

```
EXPLAIN ANALYZE SELECT * FROM records_record
WHERE tt @> ARRAY[to_date('0001-01-01'::text, 'YYYY-MM-DD'::text)];
```

QUERY PLAN

```
Bitmap Heap Scan on records_record (cost=8.04..17.44 rows=4 width=86)
    (actual time=0.045..0.045 rows=0 loops=1)
    Recheck Cond: (tt @> ARRAY[to_date('0001-01-01'::text, 'YYYY-MM-DD'::text)])
    -> Bitmap Index Scan on transaction_idx (cost=0.00..8.04 rows=4 width=0)
        (actual time=0.042..0.042 rows=0 loops=1)
    Index Cond: (tt @> ARRAY[to_date('0001-01-01'::text, 'YYYY-MM-DD'::text)])
Total runtime: 0.122 ms
```

```
EXPLAIN ANALYZE SELECT * FROM records_record
WHERE tt @> ARRAY[to_date('2013-10-01'::text, 'YYYY-MM-DD'::text)];
```

QUERY PLAN

```
Bitmap Heap Scan on records_record (cost=8.16..21.87 rows=20 width=86)
    (actual time=0.068..0.093 rows=20 loops=1)
    Recheck Cond: (tt @> ARRAY[to_date('2013-10-01'::text, 'YYYY-MM-DD'::text)])
    -> Bitmap Index Scan on transaction_idx (cost=0.00..8.15 rows=20 width=0)
        (actual time=0.052..0.052 rows=20 loops=1)
    Index Cond: (tt @> ARRAY[to_date('2013-10-01'::text, 'YYYY-MM-DD'::text)])
Total runtime: 0.153 ms
```

Porównując wyniki wywołania funkcji SQL na listingach 6.1 oraz 6.2 można zauważyć, że wykorzystanie indeksu na relacji wypożyczeń przyspieszyło filtrowanie wyników ponad dwudziestokrotnie.

6.2. Index GiST na relacji zatrudnień

Indeks GiST zastosowano na przedziale wartości $[v_s, v_e)$, oznaczających początek i koniec interwału *valid time*. Sądzone, że przyspieszy to zapytania o obsadę stanowisk w firmie na przedziale czasowym. W testach sprawdzono wyniki dla zapytania o pracowników aktualnie zatrudnionych (wartość v_e) równa wartości oznaczającej umownie „now” - '3001-01-01') oraz dla przypadkowo wybranej innej daty. Listingi 6.3 i 6.4 zawierają rezultaty wywołań: pierwszy bez indeksów na kolumnach dotyczących danych temporalnych, drugi - z indeksem GiST na wartościach $range(v_s, v_e)$.

Listing 6.3: Wyniki wywołania funkcji EXPLAIN na zapytaniu SELECT dla relacji bez indeksu temporalnego

```
EXPLAIN ANALYZE SELECT * FROM occupations_occupation
WHERE daterange(vs, ve) @> to_date('0001-01-01'::text, 'YYYY-MM-DD'::text);
```

```

QUERY PLAN
-----
Seq Scan on occupations_occupation (cost=0.00..9.09 rows=2 width=20)
  (actual time=0.834..0.834 rows=0 loops=1)
  Filter: (daterange(vs, ve) @> to_date('0001-01-01'::text, 'YYYY-MM-DD'::text))
  Rows Removed by Filter: 348
Total runtime: 0.873 ms

EXPLAIN ANALYZE SELECT * FROM occupations_occupation
  WHERE daterange(vs, ve) @> to_date('0001-01-01'::text, 'YYYY-MM-DD'::text);

QUERY PLAN
-----
Seq Scan on occupations_occupation (cost=0.00..9.09 rows=2 width=20)
  (actual time=1.068..1.068 rows=0 loops=1)
  Filter: (daterange(vs, ve) @> to_date('0001-01-01'::text, 'YYYY-MM-DD'::text))
  Rows Removed by Filter: 435
Total runtime: 1.106 ms

```

Listing 6.4: Wyniki wywołania funkcji EXPLAIN na zapytaniu SELECT dla relacji z indeksem GiST

```

EXPLAIN ANALYZE SELECT * FROM occupations_occupation
  WHERE daterange(vs, ve) @> to_date('0001-01-01'::text, 'YYYY-MM-DD'::text);

QUERY PLAN
-----
Index Scan using duration_idx
  on occupations_occupation (cost=0.15..8.17 rows=1 width=20)
  (actual time=0.027..0.027 rows=0 loops=1)
  Index Cond: (daterange(vs, ve) @> to_date('0001-01-01'::text, 'YYYY-MM-DD'::text))
Total runtime: 0.081 ms

EXPLAIN ANALYZE SELECT * FROM occupations_occupation
  WHERE daterange(vs, ve) @> to_date('0001-01-01'::text, 'YYYY-MM-DD'::text);

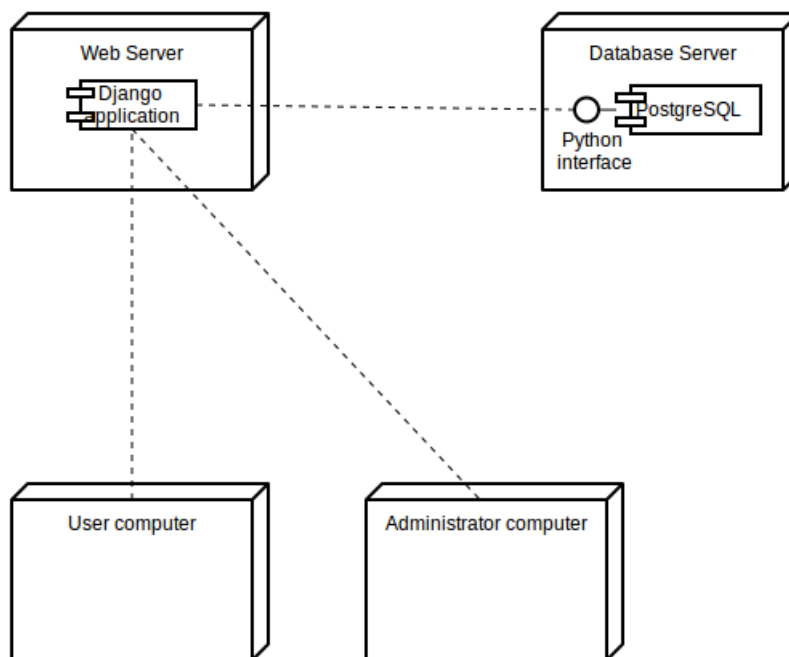
QUERY PLAN
-----
Bitmap Heap Scan
  on occupations_occupation (cost=4.16..7.30 rows=2 width=20)
  (actual time=0.028..0.028 rows=0 loops=1)
  Recheck Cond: (daterange(vs, ve) @> to_date('0001-01-01'::text, 'YYYY-MM-DD'::text))
  -> Bitmap Index Scan on duration_idx (cost=0.00..4.16 rows=2 width=0)
    (actual time=0.024..0.024 rows=0 loops=1)
    Index Cond: (daterange(vs, ve) @> to_date('0001-01-01'::text, 'YYYY-MM-DD'::text))
Total runtime: 0.088 ms

```

Dla wybranych do testów przypadków, założenie indeksu na relację zatrudnień przyspieszyło otrzymanie wyników ponad dziesięciokrotnie.

6.3. Wdrożenie

Zgodnie ze wzorcem architektury trójwarstwowej system tworzą 3 główne komponenty (pokazane na rysunku 6.1). Zadania warstwy prezentacji realizowane są przez interfejs użytkownika, dostępny przez przeglądarkę internetową. Użytkownicy zatem, muszą posiadać urządzenie do wyświetlania zawartości stron www. Aplikacja zarządzająca systemem powinna być ogólnie dostępna, należy ją zatem wyeksportować na serwer. Z kolei baza danych, z którą aplikacja ma się łączyć może być składowana na tym samym serwerze, co jest zalecane, lub może znajdować się na innym. Aplikacja łączy komunikuje się z bazą danych za pomocą interfejsu w języku Python.



Rys. 6.1: Diagram wdrożenia dla systemu

6.4. Instalacja

Projekt zbudowano przy pomocy języka Python w wersji 2.7. Należało zainstalować zależności, potrzebne do obsługi środowiska i pracy z językiem Python. Na systemie typu Unix wystarczy w tym celu wpisać podane komendy: `sudo apt-get install python-pip python-dev build-essential`. Podczas pracy nad projektem użyto narzędzia do tworzenia niezależnych środowisk tego języka - `virtualenv` - i jest to zalecane przy instalacji aplikacji. Jrgo instalację umożliwia program „pip”: `pip install virtualenv`. Tworzenie nowego środowiska odbywa się przez wykonanie komendy `virtualenv nazwa_środowiska`, a jego aktywacja - `source nazwa_środowiska/bin/activate`. Dodatkowo można użyć programu `virtualenvwrapper`, ułatwiającego m.in. aktywację środowiska. Lista potrzebnych pakietów mieści się w folderze projektu, w pliku `requirements.txt`. Po aktywacji, można je zainstalować poleceniem `pip install -r requirements.txt`.

Wykorzystano system zarządzania bazą Postgres 9.3. System w tej posiada wsparcie jedynie dla systemów operacyjnych typu LTS (ang. **Long-Term Support**). Wykorzystano system Ubuntu 12.04, gdzie instalacja polega na dodaniu pozycji Postgres 9.3 do repozytorium „Apt” i uaktualnieniu stanu systemu. Dokładniejsze informacje na ten temat dostępne są na stronie <http://www.postgresql.org/download/linux/ubuntu/>.

Następnie można już utworzyć bazę przeznaczoną do przechowywania danych systemu i skonfigurować połączenie z nią w pliku `rental_service/rental_service/settings.py`, gdzie należy wpisać nazwę bazy, nazwę użytkownika, mającego do niej prawo oraz hasło i port.

Po zainstalowaniu aplikacji na serwerze należy dodatkowo zadbać o odświeżanie bazy danych. Można to zrobić dzięki dodatkowi „schedule” dostępnej dla platformy Django i cyklicznemu wywoływaniu funkcji, która będzie dokonywać zwrotu wszystkich aktualnie wypożyczonych przedmiotów oraz ich ponownego wypożyczenia. Przykład takiej funkcji pokazano na listingu 6.5.

Listing 6.5: Funkcja aktualizująca dane w relacji wypożyczeń

```
from occupations.models import Occupation

def daily_update():
    record_list = Occupation.objects.filter(ve=future_now)
    for i in record_list:
        return_item(i.user, i.item)
        update_record(i.user, i.item, i.item.period)
```


Rozdział 7

Podsumowanie

Wprowadzenie aspektu temporalnego do bazy danych z jednej strony pozwala na obsługę przypadków użycia często spotykanych w praktyce, z drugiej jednak strony komplikuje wykonywanie operacji. W projekcie podjęto próbę implementacji modelu koncepcyjnego w PostgreSQL poprzez zastosowanie typu `array` dostępnego. W porównaniu do relacji, korzystającej z modelu reprezentacyjnego z aspektem czasu wprowadzonym na poziomie krotki, model ten miał prostszą postać, jednak wykonywanie na nim operacji było znacznie bardziej skomplikowane. Przedstawiono wyniki badań, dotyczące reprezentacji danych temporalnych oraz ich indeksowania. Pokazano, że możliwe jest odtworzenie zaproponowanej struktury Time Index dzięki implementacji metod struktury indeksowej GiST, używanej obecnie m.in. do implementacji struktury R-drzewa, wykorzystywanej w przestrzennych bazach danych.

Wykonano przegląd dostępnych struktur indeksowych i typów danych, udostępnionych w najnowszych wersjach systemu zarządzania bazą. Dzisiejsze implementacje aspektu czasowego odbiegają od zaproponowanych wcześniej rozwiązań, dzięki dostępności typów danych, takich jak `range` czy `array`.

7.1. Omówienie zastosowanych rozwiązań indeksowych

Autorzy koncepcji indeksu czasowego [2] sugerowali użycie B+-drzewa do indeksowania danych temporalnych. Przypuszczano więc, że zastosowanie indeksu o struktury indeksowej w formie drzewa binarnego, jaką jest GiST, będzie najlepszym rozwiązaniem problemu indeksowania. Współczynnik przyspieszenia filtracji wyników przy jego użyciu jest jednak gorszy, niż przy podobnym zapytaniu na typie `array`, przy użyciu indeksu GIN.

Typ `array` ma dodatkową zaletę: cała historia zmiany obiektu - w tym przypadku pary (użytkownik, przedmiot) - przechowywana jest w jednym wierszu. Natomiast, aby odczytać najbardziej aktualne dane dotyczące zdarzenia wypożyczenia, wystarczy odczytać kilka ostatnich pól tablicy `array`. Istnieją jednak pewne ograniczenia. Do niedawna do manipulowania danymi w tablicach dostępnych było tylko kilka podstawowych funkcji. Od wersji 9.3. systemu zarządzania bazą PostgreSQL możliwe jest dodatkowo usuwanie lub zamienianie wartości w tabeli. Wciąż jednak autorzy systemu zastrzegają, że typ ten powinien być używany dla niewielkiej ilości danych. Nie jest to więc najlepsze rozwiązanie, jeśli system ma służyć do przechowywania historii z wielu lat. Odtworzenie bitemporalnego koncepcyjnego modelu danych wymaga również okresowego aktualizowania danych. Jeśli więc dokonywane są zapisy z dokładnością do dnia, jak w omawianym projekcie, codziennie musi być wywoływana funkcja aktualizująca.

Zastosowanie indeksu GiST wydaje się być opcją optymalną. Przyspiesza on przeszukiwanie rekordów dziesięciokrotnie oraz jest bardziej wydajny przy wprowadzaniu nowych rekordów niż indeks GIN. Implementacja obsługi aspektu temporalnego na poziomie krotki i przy użyciu wartości $[v_s, v_e)$ była prostsza niż implementacja obsługi relacji wypożyczeń. Jednak w przypadku tabeli zatrudnień, wykorzystany był tylko jeden z wymiarów czasowych - *valid time*.

7.2. Podsumowanie wyników

Oba rozwiązania zaproponowane w projekcie funkcjonują poprawnie. Czasy pozyskiwania danych przy użyciu indeksów są na tyle niewielkie, że można przypuszczać iż system spełniałby swoją rolę w firmie liczącej 300 pracowników, co założono w wymaganiach.

Podsumowując - podczas realizacji projektu udało się zrealizować wszystkie postawione cele.

7.3. Kierunki rozwoju

Powstała w rezultacie pracy nad projektem aplikacja stanowi doskonały punkt wyjściowy dla systemu zarządzania firmą. Do pełnego wykorzystania możliwości, jakie udostępnia wykorzystanie baz temporalnych, należy jednak rozwinąć pewne elementy systemu. Można m.in. dodać stronę wyświetlającą historie wypożyczeń lub zatrudnień na zadanych przedziałach czasowych. Ułatwiłoby to tworzenie statystyk, których dodanie umożliwiłoby rozszerzenie projektu o nowe funkcjonalności.

Modele tabel zostały skonstruowane z myślą o dodatkowych, nie omawianych wcześniej funkcjach. Pola relacji `Item:Period` oraz `Penalty` pozwalają na wprowadzenie elastyczności do systemu - różne przedmioty mogą mieć inne okresy wypożyczeń, a za spóźnione zwroty - może być naliczana kara. Baza temporalna umożliwia - dzięki zachowywaniu historii - naliczanie pracownikom kar za spóźnienia.

Literatura

- [1] J. Clifford. A model for historical databases. *Information Systems Working Papers Series*, November 1982.
- [2] R. Elmasri, G. T. J. Wu, Y.-J. Kim. The time index: An access structure for temporal data. D. McLeod, R. Sacks-Davis, H.-J. Schek, redaktorzy, *VLDB*, strony 1–12. Morgan Kaufmann, 1990. <http://dblp.uni-trier.de/db/conf/vldb/vldb90.html#ElmasriWK90>.
- [3] C. H. Goh, H. Lu, B.-C. Ooi, K.-L. Tan. Indexing temporal data using existing b+-trees. *Data Knowl. Eng.*, 18(2):147–165, Mar. 1996.
- [4] J. M. Hellerstein, J. F. Naughton, A. Pfeffer. Generalized search trees for database systems. *IN PROC. 21 ST INTERNATIONAL CONFERENCE ON VLDB*, strony 562–573, 1995.
- [5] C. S. Jensen. Temporal database management; chapter 1: Introduction to temporal database research. Raport instytutowy, April 2000. <http://people.cs.aau.dk/~csj/Thesis/>.
- [6] C. S. Jensen, M. D. Soo, R. T. Snodgrass. Unifying temporal data models via a conceptual model. *Information Systems*, 19:513–547, 1993.
- [7] K. Turczyn, D. Urban. Komputerowe przetwarzanie wiedzy - Kolekcja prac 2009/2010 pod redakcją... Tomasza Kubika; rozdział, 10: Reprezentacja wiedzy zmieniającej się w czasie. 2011.