

Міністерство освіти і науки України

Державний університет «Житомирська політехніка»

Факультет інформаційно-комп'ютерних технологій

Кафедра комп'ютерної інженерії та кібербезпеки

ЗАТВЕРДЖУЮ

Завідувач кафедри

комп'ютерної інженерії

та кібербезпеки

О. А. Ім'я

6 жовтня 2025 р.

ПОЯСНЮВАЛЬНА ЗАПИСКА

до випускної кваліфікаційної роботи магістра

на тему: «Методи та засоби побудови відмовостійкого кластера бази

даних облікових записів користувачів інформаційної системи

спеціального призначення»

Термін здачі студентом закінченої роботи 26 грудня 2025 р.

Вихідні дані для роботи (зазначається предмет і об'єкт дослідження).

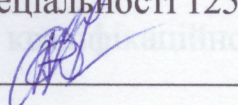
Предметом дослідження є методи кластеризації, реплікації даних та програмні засоби (PostgreSQL, Python, HAProxy) для побудови відмовостійкої архітектури зберігання даних.

Об'єктом дослідження є методи забезпечення доступності даних в інформаційній системі спеціального призначення.

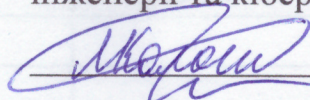
Консультанти з випускної кваліфікаційної роботи (з зазначенням розділів, що їм стосуються)

Розділ	Консультант	Завдання прийнято
1	Альона ГОРБАТКО	
2	Марія КОЛОЩУК	
3		

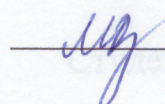
Виконала студентка 2-го курсу, групи ЗКБм-24-1 спеціальності 125 «Кібербезпека та захист інформації»

 Альона ГОРБАТКО

Керівник старший викладач кафедри комп'ютерної інженерії та кібербезпеки

 Марія КОЛОЩУК

Рецензент, старший викладач кафедри комп'ютерної інженерії та кібербезпеки.

 Вадим МИКОЛАЙЧУК

Житомир 2025

Державний університет «Житомирська політехніка»

Факультет інформаційно-комп'ютерних технологій

Кафедра комп'ютерної інженерії та кібербезпеки

Спеціальність 125 «Кібербезпека та захист інформації»

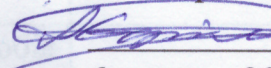
Освітня програма «Кібербезпека»

ЗАТВЕРДЖУЮ

Завідувач кафедри

комп'ютерної інженерії

та кібербезпеки

 Сфіменко А. А.

6 жовтня 2025 р.

ЗАВДАННЯ

на випускню кваліфікаційну роботу

Студента **Горбатко Альони Павлівни**

Тема роботи: **Методи та засоби побудови відмовостійкого кластера бази даних облікових записів користувачів інформаційної системи спеціального призначення**

затверджена Наказом університету від 6 жовтня 2025 р. №566/с

Термін здачі студентом закінченої роботи 26 грудня 2025 р.

Вихідні дані для роботи (зазначається предмет і об'єкт дослідження):

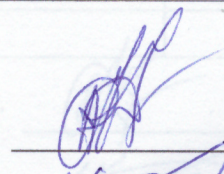
Предметом дослідження є методи кластеризації, реплікації даних та програмні засоби (PostgreSQL, Patroni, HAProxy) для побудови відмовостійкої архітектури зберігання даних.

Об'єктом дослідження є процеси забезпечення відмовостійкості, цілісності та доступності даних в інформаційних системах спеціального призначення.

Консультанти з випускної кваліфікаційної роботи із зазначенням розділів, що їх стосуються

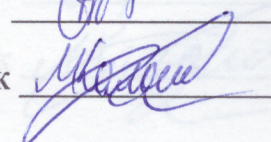
Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
1			
2			
3			

Студент



Альона ГОРБАТКО

Керівник

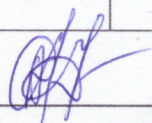


Марія КОЛОЩУК

Календарний план

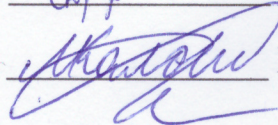
№ з/п	Назва етапів випускної кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1	Постановка завдання та затвердження теми		Виконано
2	Пошук, огляд та аналіз літературних джерел щодо проблем забезпечення високої доступності та цілісності даних у СУБД		Виконано
3	Аналіз сучасних принципів побудови відмовостійких систем, теорем розподілених обчислень (CAP, PACELC) та стандартів надійності (TIA-942, ISO 22301)		Виконано
4	Порівняльний аналіз СУБД та засобів кластеризації (PostgreSQL, MySQL, MongoDB, Patroni, Repmgr) та обґрунтування вибору технологічного стеку		Виконано
5	Проектування архітектури відмовостійкого кластера (розробка логічної та фізичної моделей даних, схеми реплікації)		Виконано
6	Підготовка віртуалізованого середовища (VirtualBox, Rocky Linux) та оптимізація параметрів ядра ОС для роботи з базами даних		Виконано
7	Розгортання розподіленого сховища конфігурацій (etcd) та налаштування синхронної реплікації PostgreSQL		Виконано
8	Реалізація механізмів автоматичного перемикавання (Patroni), балансування навантаження (HAProxy + Keepalived) та пулінгу з'єднань (PgBouncer)		Виконано
9	Тестування розробленої системи: імітація аварійних ситуацій (Kill -9, відключення мережі), вимірювання показників RTO/RPO		Виконано
10	Оформлення результатів дослідження та написання пояснювальної записки		Виконано
11	Підготовка презентаційних матеріалів та доповіді		Виконано
12	Отримання відгуку керівника та рецензії на кваліфікаційну роботу		Виконано
13	Захист магістерської роботи		Виконано

Студент



Альона ГОРБАТКО

Керівник



Марія КОЛОЩУК

РЕФЕРАТ

Пояснювальна записка до дипломної роботи містить 95 сторінок, 26 ілюстрацій, 7 таблиць, список використаних джерел із 30 найменувань та додатків.

Метою роботи є забезпечення високої доступності та цілісності облікових даних користувачів спеціального призначення шляхом розробки та впровадження архітектури відмовостійкого кластера СУБД.

В ході дослідження проведено аналіз сучасних підходів до забезпечення безперервності бізнес-процесів та виявлено вразливості монолітних архітектур баз даних. Особлива увага була приділена проблемам узгодженості даних у розподілених системах (теореми CAP, PACELC), класифікації відмов та забезпеченню відповідності міжнародним стандартам надійності (ISO 22301, TIA-942). Висвітлено важливість використання синхронної реплікації та автоматизованого відновлення для гарантування нульової втрати даних (RPO=0). Результатом цього етапу є обґрунтування переходу до кластерної архітектури.

Проведено порівняльний аналіз існуючих СУБД (MySQL, MongoDB, MS SQL) та інструментів оркестрації, обґрунтовано вибір зв'язки PostgreSQL та Patroni з використанням алгоритму консенсусу Raft. Розроблено комплексну архітектуру, що включає розподілене сховище конфігурацій etcd, балансування навантаження через HAProxy та пулінг з'єднань PgBouncer. Останній етап дослідження включає практичну реалізацію системи на базі ОС Rocky Linux. Реалізовано сценарії захисту від «Split-Brain» (Watchdog), автоматичного перемикавання лідера (Failover) та шифрування трафіку. Доведено ефективність запропонованого рішення: досягнуто часу відновлення (RTO) в межах 15 секунд та підтверджено збереження цілісності транзакцій.

Ключові слова: ВІДМОВОСТІЙКІСТЬ, POSTGRES SQL, PATRONI, КЛАСТЕРИЗАЦІЯ, РЕПЛІКАЦІЯ, HA-PROXY.

ABSTRACT

The master's thesis contains 95 pages, 26 illustrations, 7 tables, and a list of references consisting of 30 items and an appendix.

The purpose of the work is to ensure high availability and integrity of special-purpose user account data by developing and implementing a fault-tolerant DBMS cluster architecture.

During the research, an analysis of modern approaches to business continuity was conducted, and vulnerabilities of monolithic database architectures were identified. Special attention was paid to data consistency issues in distributed systems (CAP, PACELC theorems), failure classification, and compliance with international reliability standards (ISO 22301, TIA-942). The importance of using synchronous replication and automated recovery to guarantee zero data loss (RPO=0) was highlighted. The result of this stage is the justification for the transition to a cluster architecture.

A comparative analysis of existing DBMS (MySQL, MongoDB, MS SQL) and orchestration tools was carried out, justifying the choice of PostgreSQL and Patroni using the Raft consensus algorithm. A complex architecture including a distributed etcd configuration store, HAProxy load balancing, and PgBouncer connection pooling was developed. The final stage of the research includes the practical implementation of the system based on the Rocky Linux OS. Scenarios for protection against «Split-Brain» (Watchdog), automatic leader switching (Failover), and traffic encryption were implemented. The efficiency of the proposed solution was proven: a recovery time (RTO) within 15 seconds was achieved, and transaction integrity preservation was confirmed.

Key words: FAULT TOLERANCE, POSTGRESQL, PATRONI, CLUSTERING, REPLICATION, HA-PROXY.

ЗМІСТ

ВСТУП	7
1. АНАЛІЗ ПРИНЦИПІВ ТА ПІДХОДІВ ДО ЗАБЕЗПЕЧЕННЯ ВІДМОВОСТІЙКОСТІ ІНФОРМАЦІЙНИХ СИСТЕМ ТА БАЗ ДАНИХ	8
1.1. Поняття відмовостійкості та її роль у забезпеченні безперервності функціонування інформаційних систем.....	8
1.2. Класифікація типових відмов та вразливостей у системах зберігання даних	12
1.3. Аналіз архітектур СУБД та визначення критичних точок відмови.....	21
1.4. Методи реплікації, резервування та балансування навантаження у СУБД.....	29
1.5. Огляд стандартів і технологій забезпечення високої доступності (RAID, HA, Failover Clusters, DRP, Backup Policies).....	37
2. АНАЛІЗ ТЕХНОЛОГІЙ ТА ПРОЄКТУВАННЯ ВІДМОВОСТІЙКОГО КЛАСТЕРА БАЗИ ДАНИХ ОБЛІКОВИХ ЗАПИСІВ КОРИСТУВАЧІВ.....	43
2.1. Аналіз сучасних СУБД з підтримкою кластеризації (PostgreSQL, MySQL, MongoDB, MS SQL Server).....	43
2.2. Порівняльний аналіз архітектур побудови кластерів баз даних (Master–Slave, Multi-Master, Sharding, Replica Sets)	51
2.3. Вимоги до системи зберігання облікових записів користувачів спеціального призначення	53
2.4. Розробка логічної та фізичної моделей бази даних	59
2.5. Проєктування архітектури відмовостійкого кластера.....	62
2.6. Механізми синхронізації, моніторингу та автоматичного відновлення працездатності кластера.....	64
3. РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ ВІДМОВОСТІЙКОГО КЛАСТЕРА БАЗИ ДАНИХ	71
3.1. Реалізація кластерної інфраструктури бази даних	71
3.2. Налаштування реплікації, балансування навантаження та механізмів відновлення	76
3.3. Методика та сценарії тестування відмовостійкості.....	80
3.4. Оцінка продуктивності, масштабованості та часу відновлення.....	83
3.5. Аналіз результатів експериментів та оптимізація кластерного рішення	87
ВИСНОВКИ	93
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	94
ДОДАТКИ	937

ВСТУП

Сучасні інформаційні системи спеціального призначення, які раніше виконували локальні функції обліку, трансформувалися у критично важливий актив, що забезпечує безперервність управління та функціонування підприємства. Однак разом із зростанням обсягів даних та вимог до швидкодії, експоненціально зросли ризики, пов'язані з апаратними збоями, втратою цілісності даних та недоступністю сервісів автентифікації. В останні роки проблема забезпечення високої доступності (High Availability) баз даних стала дедалі актуальнішою. Традиційні підходи до розгортання СУБД, що базуються на одиничних серверах, створюють «єдину точку відмови» (Single Point of Failure), що є неприпустимим для критичної інфраструктури. Збої обладнання, помилки персоналу або програмні помилки можуть призвести до тривалих простоїв та безповоротної втрати транзакцій, що порушує логіку бізнес-процесів та створює загрози безпеці.

Мета роботи полягає у підвищенні рівня надійності та захищеності інформаційних систем шляхом дослідження методів кластеризації та розробки архітектури відмовостійкого кластера бази даних облікових записів з використанням технологій PostgreSQL та Patroni.

Об'єктом дослідження є процеси забезпечення відмовостійкості, цілісності та доступності облікових даних користувачів в інформаційних системах спеціального призначення. Розглянуто методи мінімізації часу простою (RTO) та ризиків втрати даних (RPO) при виникненні аварійних ситуацій.

Предметом дослідження є методи, технології та програмні засоби, що забезпечують функціонування відмовостійкого кластера: архітектурні рішення на базі PostgreSQL, алгоритми розподіленого консенсусу (Raft), підходи до синхронної реплікації та інструменти автоматичної оркестрації.

Методологія роботи полягає в розробці практичних рекомендацій та реалізації прототипу кластерної системи для автоматизації відновлення працездатності бази даних. Визначено архітектурні підходи до побудови систем із автоматичним перемиканням лідера (Failover), що враховують вимоги міжнародних стандартів надійності та дозволяють гарантувати цілісність транзакцій в умовах нестабільного середовища

РОЗДІЛ 1. АНАЛІЗ ПРИНЦИПІВ ТА ПІДХОДІВ ДО ЗАБЕЗПЕЧЕННЯ ВІДМОВОСТІЙКОСТІ ІНФОРМАЦІЙНИХ СИСТЕМ ТА БАЗ ДАНИХ

1.1. Поняття відмовостійкості та її роль у забезпеченні безперервності функціонування інформаційних систем

У сучасному інформаційному суспільстві, де цифрові технології пронизують усі сфери державного управління та економіки, поняття відмовостійкості трансформувалося з суто технічної характеристики в ключовий показник життєздатності будь-якої організації. Інформаційні системи спеціального призначення, які є об'єктом даного дослідження, висувають підвищені вимоги до надійності, оскільки вони оперують критично важливими даними, а їх недоступність навіть протягом короткого проміжку часу може призвести до катастрофічних наслідків. Відмовостійкість, у широкому розумінні, визначається як властивість обчислювальної системи зберігати свою працездатність після виникнення відмов у її апаратній або програмній частині. Це означає, що система повинна мати вбудовані механізми виявлення помилок, ізоляції несправних компонентів та автоматичного переконфігурування для продовження виконання своїх функцій, можливо, зі зниженням продуктивності, але без повної зупинки та втрати цілісності даних.

Для формалізації поняття відмовостійкості необхідно звернутися до теорії надійності складних технічних систем. Відповідно до чинних міжнародних стандартів, надійність системи характеризується сукупністю властивостей, таких як безвідмовність, довговічність, ремонтпридатність та збереженість [4,16]. Ключовим математичним показником, що описує надійність, є функція надійності $P(t)$, яка визначає ймовірність безвідмовної роботи системи протягом інтервалу часу t . Якщо розглядати потік відмов як найпростіший пуассонівський процес, що є справедливим для більшості електронних компонентів у період їх нормальної експлуатації, то функція надійності описується експоненціальним законом розподілу $p(t) = e^{-\lambda t}$, де λ – інтенсивність відмов, яка є величиною, оберненою до середнього часу напрацювання на відмову (Mean Time Between Failures, MTBF). Таким чином, завдання підвищення відмовостійкості зводиться до мінімізації

параметра λ для системи в цілому, що досягається шляхом введення структурної, часової або інформаційної надлишковості.

Важливо чітко розмежовувати поняття «відмова», «помилка» та «несправність» оскільки в академічній літературі та стандартах серії ISO/IEC вони мають різне змістове навантаження. Несправність розглядається як фізичний дефект або програмна вада, що може призвести до некоректного стану системи, який називається помилкою. У свою чергу, якщо помилка не була вчасно виявлена та оброблена механізмами відмовостійкості, вона призводить до відмови системи, тобто до її переходу в неробочий стан. Метою проєктування відмовостійкого кластера баз даних є створення таких архітектурних рішень, які б дозволяли системі продовжувати функціонування при наявності несправностей, запобігаючи їх трансформації у відмови сервісу. Це реалізується за рахунок використання методів маскування несправностей, коли результат роботи системи формується на основі голосування між кількома резервованими модулями, або методів динамічного відновлення, що передбачають перемикання на резервний вузол [29].

Іншим критично важливим показником, що тісно пов'язаний з відмовостійкістю, є коефіцієнт готовності (Availability), який позначається як A або k_g . З математичної точки зору, у стаціонарному режимі роботи системи готовність визначається як відношення часу працездатного стану до загального часу експлуатації. Формула для розрахунку коефіцієнта готовності має вигляд

$$A = \frac{MTBF}{MTBF + MTTR} \quad (1.1)$$

де $MTTR$ (Mean Time To Repair) – середній час відновлення працездатності після відмови. З наведеної формули стає очевидним, що підвищення готовності системи можливе двома шляхами: збільшенням $MTBF$ (підвищення надійності компонентів) або зменшенням $MTTR$ (прискорення процесів відновлення). Саме другий шлях є пріоритетним при побудові кластерних систем, оскільки повне уникнення апаратних збоїв є фізично неможливим, проте використання технологій автоматичного перемикання (failover) дозволяє знизити $MTTR$ до секунд або навіть мілісекунд, забезпечуючи високу доступність сервісу на рівні 99,99% і вище.

У контексті інформаційних систем спеціального призначення забезпечення відмовостійкості нерозривно пов'язане з тріадою інформаційної безпеки: конфіденційність, цілісність, доступність (CIA) [3]. Особливу увагу слід приділяти цілісності даних у моменти збоїв. При аварійному відключенні живлення або виході з ладу дискового масиву існує високий ризик порушення логічної структури бази даних, втрати транзакцій, що були зафіксовані в оперативній пам'яті, але не встигли записатися на фізичний носій. Тому відмовостійкі рішення повинні включати не лише засоби дублювання обладнання, а й механізми транзакційної цілісності, такі як журнали попереднього запису (Write-Ahead Logging, WAL) та розподілені консенсусні алгоритми (наприклад, Raft або Paxos), які гарантують узгодженість даних між вузлами кластера навіть в умовах нестабільного мережевого з'єднання.

Аналізуючи підходи до забезпечення безперервності функціонування, необхідно розглянути класифікацію рівнів надійності центрів обробки даних, запропоновану організацією Uptime Institute, яка стала де-факто стандартом у галузі. Ця класифікація передбачає чотири рівні (Tier) [5]. Рівень Tier I являє собою базову інфраструктуру без резервування активних компонентів, що призводить до значних простоїв під час регламентних робіт або збоїв. Рівень Tier II впроваджує часткове резервування компонентів енергопостачання та охолодження. Рівень Tier III вимагає можливості паралельного обслуговування, що означає наявність дублюючих шляхів дистрибуції ресурсів, дозволяючи виводити будь-який компонент у ремонт без зупинки сервісу. Найвищий рівень, Tier IV, передбачає повну відмовостійкість інфраструктури, де будь-яка одинична аварія автоматично локалізується системою без впливу на роботу кінцевих користувачів. Для систем зберігання облікових записів спеціального призначення доцільно орієнтуватися на архітектурні принципи, що відповідають рівням Tier III або Tier IV, адаптуючи їх до програмного рівня кластеризації СУБД.

Важливим аспектом теорії відмовостійкості є концепція деградації якості обслуговування (*graceful degradation*). Вона полягає у здатності системи адаптуватися до виходу з ладу частини ресурсів шляхом тимчасового відключення другорядних функцій заради збереження працездатності критично важливого ядра.

У випадку кластера бази даних облікових записів це може означати перехід у режим «лише для читання» (read-only) у разі втрати зв'язку з майстер-вузлом або неможливості досягнення кворуму для запису нових даних. Такий підхід дозволяє користувачам продовжувати автентифікацію та перегляд інформації, хоча й обмежує можливість зміни паролів або створення нових профілів до моменту відновлення повної функціональності кластера. Це є практичним втіленням принципу пріоритетності доступності інформації над можливістю її модифікації в аварійних ситуаціях.

Окремо слід зупинитися на проблематиці корельованих відмов, які є найбільшою загрозою для кластерних систем. Традиційні математичні моделі надійності часто виходять з припущення про незалежність відмов окремих компонентів, що дозволяє розраховувати загальну ймовірність безвідмовної роботи як добуток ймовірностей для кожного вузла (у паралельній схемі). Проте на практиці часто трапляються ситуації, коли одна причина, наприклад, помилка в оновленні програмного забезпечення, стрибок напруги або цілеспрямована кібератака, призводить до одночасного виходу з ладу всіх вузлів кластера. Для боротьби з корельованими відмовами застосовується принцип гетерогенності, який передбачає використання різноманітного обладнання або різних версій операційних систем на вузлах кластера, а також географічний розподіл компонентів системи для захисту від локальних техногенних або природних катастроф.

Таким чином, відмовостійкість виступає як комплексна інтегральна характеристика, що формується на перетині апаратних рішень, архітектури програмного забезпечення та організаційних заходів. Її роль у забезпеченні безперервності функціонування інформаційних систем полягає у створенні запасу міцності, який дозволяє системі поглинати внутрішні та зовнішні збурення, гарантуючи надання сервісів згідно з узгодженим рівнем якості (SLA). Подальший розвиток технологій хмарних обчислень та віртуалізації відкриває нові можливості для побудови відмовостійких систем, дозволяючи динамічно мігрувати сервіси між фізичними серверами та центрами обробки даних, що робить поняття “сервер” абстракцією, а надійність – функцією програмно-визначеної інфраструктури.

1.2. Класифікація типових відмов та вразливостей у системах зберігання даних

Класифікація відмов та вразливостей у системах зберігання даних є фундаментальним етапом у процесі проектування відмовостійких архітектур, оскільки розуміння природи, джерел та механізмів виникнення збоїв дозволяє розробити ефективні контрзаходи та стратегії мінімізації ризиків. У контексті інформаційних систем спеціального призначення, де зберігаються облікові записи користувачів, спектр потенційних загроз значно розширюється, охоплюючи не лише фізичні дефекти обладнання, але й логічні помилки, збої програмного забезпечення, навмисні атаки та помилки персоналу. Науковий підхід до класифікації вимагає розгляду відмов як стохастичних подій, що підпорядковуються певним імовірнісним законам, а вразливостей – як детермінованих недоліків у конструкції або конфігурації системи, що створюють передумови для реалізації загроз.

Первинним рівнем класифікації є поділ відмов за фізичною природою їх виникнення, де домінуюче місце займають апаратні збої носіїв інформації. Історично склалося, що жорсткі магнітні диски (HDD) та твердотільні накопичувачі (SSD) мають принципово різні механізми старіння та виходу з ладу, що вимагає застосування диференційованих математичних моделей для прогнозування їх надійності. Для опису інтенсивності відмов апаратних компонентів у часі традиційно використовується так звана «U-подібна крива» або «крива ванни» (bathtub curve). Ця модель виділяє три характерні періоди життєвого циклу пристрою. Перший період, відомий як період припрацювання, характеризується підвищеною, але спадаючою інтенсивністю відмов, що зумовлено виявленням виробничих дефектів, браку пайки або мікротріщин у матеріалах. Другий період – це період нормальної експлуатації, де інтенсивність відмов є приблизно постійною величиною ($\lambda \approx const$), а збої мають випадковий, раптовий характер, спричинений зовнішніми факторами, такими як флуктуації напруги або вібрації. Третій період – це період старіння, де інтенсивність відмов стрімко зростає внаслідок фізичного зносу механічних частин або деградації напівпровідникових структур.

Для математичного опису розподілу часу до відмови у періоді старіння найбільш адекватним є розподіл Вейбулла, щільність ймовірності якого задається формулою:

$$f(t; \lambda; k) = \frac{k}{\lambda} \left(\frac{t}{\lambda}\right)^{k-1} e^{-(t/\lambda)^k} \quad (1.2)$$

де $t \geq 0, k > 0$ — параметр форми, $\lambda > 0$ параметр масштабу. Якщо $k < 1$, це вказує на те, що інтенсивність відмов зменшується з часом (період припрацювання), якщо $k = 1$, то інтенсивність відмов постійна (експоненційний розподіл), а випадок $k > 1$ описує зростання інтенсивності відмов, що характерно для процесів зносу. У системах зберігання даних на основі HDD основним фактором зносу є механічна деградація підшипників шпинделя та механізму позиціонування головок, що призводить до так званих «head crash» аварій, коли зчитувальна головка торкається поверхні пластини, безповоротно руйнуючи магнітний шар.

У випадку використання твердотільних накопичувачів (SSD), які стають стандартом для високопродуктивних баз даних, природа відмов зміщується у площину квантової фізики та обмеженого ресурсу циклів перезапису (P/E cycles) [30]. Вразливість SSD полягає у поступовому руйнуванні діелектричного шару тунельного переходу в комірках флеш-пам'яті типу NAND. Кожен цикл запису та стирання вимагає подачі високої напруги, що призводить до накопичення електронів у діелектрику та неможливості точного визначення заряду комірки. Це явище описується параметром WAF (Write Amplification Factor), який показує відношення обсягу фізично записаних даних до обсягу логічних даних, надісланих операційною системою. Високий WAF, характерний для баз даних з інтенсивним випадковим записом дрібних блоків (що типово для журналів транзакцій облікових записів), прискорює вичерпання ресурсу накопичувача. Крім того, SSD схильні до специфічних відмов контролерів та помилок мікрокоду (firmware), які можуть призвести до миттєвого переходу пристрою в стан «цеглини» (brick), коли дані стають недоступними, незважаючи на цілісність самих комірок пам'яті.

Наступною великою категорією є логічні відмови та порушення цілісності даних, які часто називають «мовчазними» або «тихими» помилками (silent data

corruption). Це явище, відоме також як «bit rot», виникає, коли значення біта на носії змінюється без фіксації помилки апаратним контролером диска. Причинами можуть бути вплив космічного випромінювання, електромагнітні завади або залишкова намагніченість сусідніх доріжок. У базах даних облікових записів навіть зміна одного біта може призвести до критичних наслідків, наприклад, пошкодження хеш-суми пароля або зміни ідентифікатора прав доступу, що перетворює легітимного користувача на адміністратора або навпаки. Для боротьби з цим типом вразливостей файлові системи корпоративного рівня, такі як ZFS або Btrfs, використовують наскрізне контрольне сумування (end-to-end checksumming), проте стандартні файлові системи (EXT4, NTFS) часто не мають таких механізмів, покладаючись лише на ECC-коди жорстких дисків, чого недостатньо для гарантування абсолютної цілісності.

Окремий клас складають відмови, спричинені недосконалістю програмного забезпечення СУБД та операційних систем. Складність сучасних систем управління базами даних призводить до наявності у коді помилок (багів), які проявляються лише за певних граничних умов, наприклад, при надвисокому навантаженні (race conditions) або при специфічній послідовності транзакцій. Особливо небезпечними є вразливості, пов'язані з некоректною обробкою аварійного завершення роботи (power loss). Сучасні накопичувачі використовують кеш-пам'ять для прискорення операцій запису. Якщо система повідомляє СУБД про успішний запис транзакції, але дані фактично знаходяться лише в енергозалежному кеші диска, раптове вимкнення живлення призведе до втрати цих даних і порушення ACID-властивостей бази (Atomicity, Consistency, Isolation, Durability). Це створює логічну невідповідність між журналами транзакцій та файлами даних, що часто унеможливорює автоматичний запуск бази даних після відновлення живлення без тривалої процедури відновлення або втрати останніх змін.

Мережеві відмови та вразливості комунікаційної інфраструктури відіграють критичну роль у розподілених кластерних системах. Типовою проблемою є так зване «розщеплення мозку» (split-brain), яке виникає, коли через збій мережевого обладнання (комутатора, маршрутизатора або обрив кабелю) зв'язок між вузлами

кластера втрачається, але самі вузли продовжують працювати. У такій ситуації кожен сегмент кластера може помилково вважати себе єдиним легітимним власником ресурсів і спробувати взяти на себе роль майстра, що призводить до одночасного запису суперечливих даних у різні копії бази. Це класична ілюстрація теореми CAP, яка стверджує неможливість одночасного забезпечення узгодженості (Consistency), доступності (Availability) та стійкості до поділу (Partition tolerance) [6]. У системах облікових записів спеціального призначення такий сценарій є неприпустимим, оскільки він порушує єдиний простір автентифікації. Вразливістю тут виступає недостатня надійність механізмів кворуму або використання простих схем перевірки доступності вузлів (heartbeat), які не здатні відрізнити високе навантаження від повної недоступності.

Важливим аспектом класифікації є розгляд загроз інформаційній безпеці як фактора, що прямо впливає на відмовостійкість. У сучасному ландшафті кіберзагроз атаки типу «відмова в обслуговуванні» (DDoS) та програми-вимагачі (ransomware) стали одними з основних причин недоступності даних. Ransomware, шифруючи файли бази даних, фактично переводить систему в стан відмови, оскільки інформація стає непридатною для використання, хоча фізично носії та сервери залишаються справними. Це демонструє конвергенцію понять «безпека» та «надійність»: вразливість у системі патчингу або слабка політика паролів адміністраторів стає точкою входу для атаки, яка реалізується як катастрофічна відмова системи зберігання. Специфічною вразливістю баз даних є також SQL-ін'єкції, які можуть бути використані не лише для викрадення даних, але й для виконання деструктивних команд (наприклад, DROP TABLE), що призводить до миттєвої втрати інформації [15].

Людський фактор та помилки адміністрування займають стабільне місце у статистиці причин відмов. До цього класу належать випадкове видалення даних, некоректна зміна конфігураційних файлів, помилки при оновленні схеми бази даних або програмного забезпечення. Вразливістю у даному випадку є відсутність механізмів захисту від помилкових дій («захист від дурня»), недостатня ізоляція середовищ розробки та продуктивного середовища, а також слабка регламентація процедур обслуговування. Наприклад, помилка оператора при заміні несправного

диска у RAID-масиві (витягування справного диска замість пошкодженого) призводить до повного руйнування логічного тому і втрати всіх даних, незважаючи на наявність надлишковості. Математично цей фактор важко змодельовати детермінованими функціями, проте він враховується через ймовірнісні коефіцієнти при розрахунку загальної надійності системи.

Зростаюча складність віртуалізованих середовищ та хмарних інфраструктур додає новий шар абстракції і, відповідно, нові типи відмов. У віртуалізованих системах зберігання даних (vSAN, Ceph) виникає феномен «галасливого сусіда» (noisy neighbor), коли один віртуальний сервер монополізує ресурси вводу-виводу (IOPS), викликаючи затримки (latency) у роботі бази даних облікових записів. Для чутливих до таймінгів протоколів кластеризації така затримка може бути інтерпретована як відмова вузла, що ініціює непотрібну процедуру виборів нового лідера та тимчасову недоступність сервісу. Крім того, помилки у гіпервізорі або системі оркестрації контейнерів можуть призвести до одночасного вимкнення всіх віртуальних машин кластера, якщо вони фізично розміщені на одному хості, що є прикладом відмови через єдину точку збою (Single Point of Failure - SPOF) на рівні інфраструктури.

Підсумовуючи класифікацію, можна стверджувати, що забезпечення відмовостійкості вимагає комплексного підходу до аналізу всіх рівнів системи: від фізики напівпровідників до психології оператора. Кожен тип відмови має свої специфічні характеристики: час прояву, радіус ураження (локальний чи глобальний), можливість автоматичного детектування та складність відновлення. Для побудови ефективного кластера бази даних облікових записів необхідно не лише обрати надійне обладнання, але й імплементувати програмні механізми перевірки цілісності даних, налаштувати коректні тайм-аути мережних з'єднань, забезпечити захист від кібератак та мінімізувати вплив людського фактора через автоматизацію рутинних операцій. Розуміння імовірнісної природи відмов дозволяє перейти від парадигми «запобігання збоям» до парадигми «управління збоями», де наявність несправностей сприймається як нормальний робочий стан системи, який ефективно обробляється закладеною архітектурою.

Поглиблюючи аналіз фізичної природи відмов, необхідно детально розглянути вплив параметрів навколишнього середовища на надійність апаратних компонентів кластера, оскільки сервери баз даних часто функціонують у режимі 24/7 під значним термічним навантаженням. Одним із ключових законів, що описує залежність інтенсивності відмов напівпровідникових приладів від температури, є рівняння Арреніуса. Відповідно до цієї моделі, швидкість протікання хімічних реакцій деградації матеріалів експоненціально зростає з підвищенням температури. Математично коефіцієнт прискорення старіння AF (Acceleration Factor) визначається як:

$$AF = e^{\frac{E_a}{k} \left(\frac{1}{T_{use}} - \frac{1}{T_{stress}} \right)} \quad (1.3)$$

де E_a – енергія активації процесу відмови (вимірюється в електрон-вольтах, eV), k – стала Больцмана ($8.617 \times 10^{-5} \text{ eV/k}$), T_{use} нормальна робоча температура у кельвінах, а T_{stress} підвищена температура. Для сучасних процесорів та модулів пам'яті навіть незначне підвищення температури у серверній шафі на 10 градусів може скоротити термін служби компонентів удвічі. Це явище, відоме як електроміграція, полягає у перенесенні маси металу в провідниках інтегральних схем під дією електричного струму високої густини. З часом це призводить до утворення пустот (voids), що розривають електричне коло, або наростів (hillocks), що викликають короткі замикання між сусідніми провідниками. У контексті баз даних це проявляється як спонтанні перезавантаження серверів або помилки парності пам'яті ECC, які можуть бути помилково інтерпретовані як програмні збої.

Окрім температурного фактора, критичний вплив на надійність магнітних носіїв інформації чинить вологість повітря та вібраційне навантаження. Низька вологість сприяє накопиченню статичного електричного заряду (ESD), розряд якого може миттєво вивести з ладу чутливі мікросхеми контролерів дисків. Висока вологість, навпаки, прискорює процеси гальванічної корозії контактних груп, особливо в умовах наявності у повітрі активних хімічних домішок, таких як сірка, що характерно для промислових зон. Вібрації, які генеруються вентиляторами охолодження серверів (fan vibrations), можуть входити в резонанс із власними частотами коливань блоків головок жорстких дисків. Це призводить до похибок позиціонування (Track-Following Errors), коли головка не може стабільно

утримуватися над потрібною доріжкою, що змушує диск виконувати повторні операції читання (retries). З точки зору користувача бази даних, це виглядає як різке зростання затримок (latency) при виконанні запитів, що у критичних випадках призводить до тайм-аутів і розриву з'єднань, формально класифікуючись як відмова в обслуговуванні, хоча апаратура залишається формально справною.

Окрему, вкрай складну для діагностики категорію відмов становлять так звані «м'які помилки» (soft errors) або одноразові події (Single Event Upsets - SEU). Вони виникають під дією високоенергетичних частинок космічного випромінювання (нейтронів) або альфа-частинок, що випромінюються домішками радіоактивних ізотопів у матеріалах корпусу мікросхем. Проходячи через кремнієву структуру чіпа пам'яті, така частинка створює іонізаційний трек, що генерує достатній заряд для перемикання стану транзистора (наприклад, зміна логічного «0» на «1»). На відміну від твердих відмов, SEU не пошкоджують фізичну структуру пристрою; перезапис комірки відновлює її працездатність. Проте, якщо така помилка стається у регістрі процесора, що відповідає за вказівник інструкцій або адресу переходу, це призводить до непередбачуваної поведінки програми СУБД (kernel panic) та аварійної зупинки вузла кластера. Для систем спеціального призначення, що працюють на висотах або у специфічних умовах, імовірність таких подій зростає на порядки, що вимагає застосування спеціалізованої пам'яті та дубльованих обчислювальних модулів.

Переходячи до класифікації програмних збоїв, слід виділити концепцію «старіння програмного забезпечення» (Software Aging). Хоча програмний код не зношується фізично, як механічні деталі, у процесі тривалої безперервної роботи серверних додатків спостерігається поступова деградація їх продуктивності та стабільності. Основними причинами цього явища є витoki пам'яті (memory leaks), коли програма не звільняє виділені ресурси після завершення їх використання, фрагментація динамічної пам'яті (heap fragmentation) та переповнення внутрішніх лічильників. У кластерах баз даних це часто проявляється через тижні або місяці безперервної роботи: спочатку спостерігається поступове збільшення часу реакції, потім починаються спорадичні зависання, і врешті-решт відбувається повна відмова сервісу (crash). Математично цей процес описується моделями вичерпання

ресурсів, де час до відмови є функцією від інтенсивності транзакцій та обсягу втраченого ресурсу на одну транзакцію. Боротьба з цим типом відмов вимагає впровадження стратегії «програмного омолодження» (software rejuvenation) – планових превентивних перезавантажень вузлів кластера або перезапуску сервісів СУБД до моменту настання критичного вичерпання ресурсів.

У складних розподілених системах особливе місце займають помилки синхронізації потоків, відомі як «стан гонитви» (Race Conditions) та взаємні блокування (Deadlocks). Класифікація цих відмов ускладнюється їх недетермінованою природою: вони можуть не проявлятися роками під час тестування, але виникати при пікових навантаженнях або специфічному збігу часових інтервалів мережесих пакетів. У літературі такі помилки часто називають «Гейзенбагами» (Heisenbugs) – помилками, що зникають або змінюють свою поведінку при спробі їх дослідити (наприклад, при включенні режиму налагодження, який змінює таймінги виконання коду). На противагу їм, «Борбаги» (Bohrbugs) – це стабільні помилки, що легко відтворюються. Для кластера бази даних облікових записів Гейзенбаги є найбільш небезпечними, оскільки вони часто призводять до порушення консистентності даних між репліками (data drift), яке залишається непоміченим до моменту звернення до пошкодженого запису.

Також необхідно класифікувати відмови за ступенем їх впливу на систему, використовуючи шкалу тяжкості наслідків (Severity Levels).

Перший рівень – це катастрофічні відмови, що призводять до повної втрати даних без можливості відновлення (наприклад, одночасний вихід з ладу основного та резервного масивів зберігання).

Другий рівень – критичні відмови, що зупиняють надання сервісу, але зберігають дані (збій операційної системи на майстер-вузлі).

Третій рівень – суттєві відмови, що обмежують функціональність (наприклад, перехід у режим read-only).

Четвертий рівень – незначні відмови, які не впливають на основний бізнес-процес, але вимагають уваги адміністратора (вихід з ладу одного з блоків живлення у сервері з дубльованим живленням).

Така градація дозволяє пріоритезувати реакцію систем автоматичного моніторингу та персоналу.

Ще одним важливим класом є Візантійські відмови (Byzantine Failures). Цей термін походить від відомої «задачі візантійських генералів» і описує ситуацію, коли компонент системи не просто перестає працювати (зупиняється), а продовжує працювати некоректно, надсилаючи іншим учасникам кластера суперечливу або неправдиву інформацію. Це може статися внаслідок апаратного збою (біту, що «залип» у мережевій карті), логічної помилки або зловмисного втручання (компрометації вузла хакером). У кластері бази даних візантійська поведінка одного з вузлів може призвести до того, що різні частини системи приймуть різні рішення щодо фіксації транзакції, руйнуючи цілісність розподіленої бази. Захист від таких відмов вимагає використання складних протоколів консенсусу (BFT – Byzantine Fault Tolerance), які здатні коректно працювати навіть за умови, що певна частка вузлів (зазвичай менше $1/3$) поводить зловмисно або помилково. Це значно підвищує накладні витрати на комунікацію та обчислення, але є необхідною платою за надійність у системах спеціального призначення.

Нарешті, слід розглянути корельовані відмови інфраструктурного рівня, які часто ігноруються при розрахунку ймовірностей. Якщо всі сервери кластера підключені до однієї фази електромережі, одного комутатора ядра або розміщені в одній пожежній зоні, ймовірність їх одночасної відмови дорівнює ймовірності відмови цього спільного ресурсу, що зводить нанівець будь-які зусилля з кластеризації. Феномен каскадних відмов (cascading failures) є прямим наслідком тісної зв'язності компонентів. Коли один вузол виходить з ладу через перевантаження, його навантаження перерозподіляється між рештою вузлів, що може призвести до їх перевантаження та послідовного виходу з ладу, викликаючи ефект доміно. Математичне моделювання таких процесів використовує теорію перколяції та складних мереж для визначення критичної точки навантаження, після якої система втрачає стійкість.

1.3. Аналіз архітектур СУБД та визначення критичних точок відмови

Проектування відмовостійкого кластера для систем спеціального призначення неможливе без ґрунтовної декомпозиції архітектури сучасних систем управління базами даних (СУБД) та виявлення їх структурних вразливостей. СУБД є складним програмно-апаратним комплексом, який функціонує як посередник між абстрактними запитами користувача та фізичними бітами на носіях інформації, і саме в точках переходу між різними рівнями абстракції найчастіше виникають критичні збої. Глибинний аналіз архітектури вимагає розгляду СУБД як системи масового обслуговування з пріоритетами, де ресурси процесора, оперативної пам'яті та дискової підсистеми є обмеженими, а черги запитів – потенційно нескінченними. У класичній архітектурі реляційних баз даних центральним елементом, що визначає продуктивність та надійність, є підсистема управління буферами (Buffer Manager), яка реалізує стратегію кешування сторінок даних в оперативній пам'яті для мінімізації повільних звернень до дискових накопичувачів.

Саме буферний пул (Buffer Pool) є першою і найбільш очевидною критичною точкою відмови в архітектурі монолітної СУБД. Проблема полягає в тому, що оперативна пам'ять є енергозалежною. Коли транзакція змінює дані, ці зміни спочатку відбуваються лише у сторінці пам'яті, яка отримує статус «брудної» (dirty page). До моменту, поки менеджер буферів не скине цю сторінку на диск (операція flush), дані існують лише у вигляді електричних зарядів у конденсаторах модулів RAM. У разі раптового зникнення живлення або падіння ядра операційної системи (Kernel Panic) вміст «брудних» сторінок втрачається безповоротно. Для нівелювання цієї архітектурної вразливості всі сучасні промислові СУБД (PostgreSQL, Oracle, SQL Server) використовують механізм випереджувального запису в журнал (Write-Ahead Logging, WAL) [1,11]. Принцип WAL стверджує, що жодна модифікація даних не може бути записана на диск у файл даних доти, доки запис про цю модифікацію не буде зафіксовано у лог-файлі на стійкому носії. Це перетворює проблему втрати пам'яті на проблему синхронізації дискового запису.

Однак впровадження WAL створює нову критичну точку відмови, пов'язану з продуктивністю дискової підсистеми при послідовному записі. Архітектурна залежність від швидкості запису в журнал транзакцій стає «вузьким місцем»

(bottleneck), оскільки кожна транзакція COMMIT вимагає фізичного виклику системної функції fsync або fdatasync, яка примусово скидає кеш операційної системи на фізичний диск. Якщо контролер диска або сама дискова підсистема не справляється з потоком запитів на синхронізацію, виникає ефект черги, який призводить до різкого зростання затримок (latency) та потенційного переповнення буферів логування. У критичних ситуаціях це може призвести до повної зупинки обробки нових транзакцій, що з погляду користувача виглядає як повна відмова системи. Більше того, існує ризик пошкодження самого файлу WAL. Якщо в момент запису сектора відбувається збій живлення, може виникнути ситуація «розірваного запису» (torn page write), коли записано лише частину сторінки. Без цілісного журналу транзакцій автоматичне відновлення бази даних (crash recovery) стає неможливим, що вимагає ручного втручання та відновлення з резервних копій, що неприпустимо для систем з високими вимогами до доступності.

Наступним рівнем аналізу є архітектура зберігання даних на рівні індексів. Традиційні В-дерева (B-Trees) та їх варіації В+, які використовуються у більшості реляційних СУБД, оптимізовані для читання, але мають архітектурну вразливість при інтенсивному випадковому записі. Процес розщеплення сторінок (page splitting) при вставці нових ключів призводить до фрагментації індексу та збільшення кількості операцій вводу-виводу. У контексті відмовостійкості пошкодження структури В-дерева (наприклад, циклічні посилання між вузлами внаслідок логічної помилки) може призвести до нескінченних циклів при виконанні запитів, що викликає 100% завантаження процесора і відмову в обслуговуванні для інших користувачів. Альтернативна архітектура LSM-дерев (Log-Structured Merge-Trees), що використовується у багатьох NoSQL рішеннях (Cassandra, RocksDB), вирішує проблему випадкового запису шляхом перетворення його на послідовний, проте вводить процес «ущільнення» (compaction), який є ресурсоємним фоновим процесом. Якщо процес ущільнення не встигає за потоком вхідних даних, відбувається вичерпання дискового простору та деградація продуктивності читання, що також є формою архітектурної відмови.

Переходячи від монолітних до розподілених архітектур, необхідно проаналізувати три основні парадигми побудови кластерів: Shared-Memory, Shared-

Disk та Shared-Nothing. Архітектура Shared-Memory, де всі процесори мають доступ до єдиного адресного простору пам'яті, є вкрай складною для масштабування та має єдину точку відмови у вигляді шини пам'яті. У системах Shared-Disk, типовим представником яких є Oracle RAC, всі вузли кластера мають доступ до спільного масиву дисків (SAN). Критичною точкою відмови тут виступає механізм розподіленого управління блокуваннями (Distributed Lock Manager, DLM). DLM відповідає за узгодженість даних, гарантуючи, що два вузли не будуть одночасно змінювати один і той самий блок даних. Збій у роботі DLM або високі затримки у мережі інтерконекту (private interconnect) можуть призвести до «заморожування» всього кластера або до корупції даних, якщо механізм «вигородження» (fencing) несправного вузла не спрацює миттєво.

Найбільш поширеною архітектурою для побудови сучасних масштабованих систем є Shared-Nothing, де кожен вузол має власну пам'ять та дисковий простір, а взаємодія відбувається виключно через мережу. Така архітектура усуває єдині точки відмови на апаратному рівні, але переносить складність забезпечення надійності на програмний рівень комунікації. Головною архітектурною проблемою тут стає проблема розподілених транзакцій. Для забезпечення атомарності операцій, що зачіпають кілька вузлів (шардів), використовується протокол двофазної фіксації (Two-Phase Commit, 2PC). Протокол 2PC є блокуючим: якщо координатор транзакції виходить з ладу після відправки команди «prepare», але до команди «commit», всі учасники транзакції залишаються у стані очікування, утримуючи блокування на ресурсах. Це може призвести до каскадного накопичення блокувань і повної зупинки системи. Це класичний приклад того, як механізм забезпечення узгодженості стає причиною зниження доступності.

Особливу увагу в аналізі архітектур слід приділити механізмам виявлення відмов (Failure Detection). У розподілених системах неможливо миттєво і достовірно відрізнити вузол, що вийшов з ладу, від вузла, який працює повільно або є недоступним через проблеми мережі. Більшість архітектур використовують механізм «серцебиття» (heartbeat) з пороговими значеннями тайм-аутів. Критична точка відмови тут полягає у неправильному налаштуванні цих параметрів. Занадто короткий тайм-аут призводить до хибних спрацьовувань (false positives), коли

справний вузол оголошується мертвим, що ініціює непотрібні процедури виборів лідера та переміщення даних (rebalancing), створюючи додаткове навантаження на мережу. Занадто довгий тайм-аут збільшує час виявлення реальної аварії, що прямо впливає на показник RTO (Recovery Time Objective). Математично оптимальний тайм-аут визначається на основі аналізу розподілу затримок у мережі, проте на практиці це часто статичне значення, що є вразливістю.

Ще однією архітектурною проблемою є залежність від зовнішніх сервісів координації, таких як ZooKeeper, etcd або Consul, які часто використовуються для зберігання конфігурації кластера та вибору лідера [9]. Ці системи самі по собі є розподіленими базами даних, що працюють на основі алгоритмів консенсусу (Raft, Paxos, Raft). Якщо кластер СУБД залежить від зовнішнього координатора, то відмова кворуму координатора призводить до розпаду кластера СУБД, навіть якщо самі вузли бази даних справні. Це створює мета-рівень критичних точок відмови. Архітектурна складність зростає експоненціально: адміністратору необхідно забезпечити відмовостійкість не лише бази даних, а й інфраструктури її управління.

Також слід розглянути архітектуру обробки з'єднань (Connection Pooling). У високонавантажених системах створення нового підключення до бази даних є дорогою операцією, що вимагає виділення пам'яті та автентифікації. Тому використовуються пули постійних з'єднань. Критична точка відмови виникає, коли пул вичерпується (connection leak) через некоректне закриття з'єднань клієнтським програмним забезпеченням або коли «шторм з'єднань» (connection storm) після відновлення мережі перевантажує процес автентифікації СУБД. Архітектура, яка не передбачає механізмів обмеження швидкості нових підключень (rate limiting) та пріоритезації трафіку адміністратора, є вразливою до повного блокування доступу. Для систем спеціального призначення це критично, оскільки адміністратор повинен мати можливість увійти в систему для діагностики навіть в умовах максимального навантаження.

Аналіз багатопотокової архітектури СУБД також виявляє вразливості, пов'язані з конкуренцією за внутрішні ресурси ядра (latches, mutexes). На відміну від транзакційних блокувань, які контролюються користувачем, латчі – це короткострокові блокування структур пам'яті СУБД. При певних паттернах

навантаження (наприклад, масове оновлення однієї сторінки індексу) може виникнути явище «latch contention», коли більшість потоків процесора просто очікують звільнення ресурсу, а корисна робота не виконується. Це призводить до парадоксальної ситуації: завантаження CPU низьке, але пропускна здатність системи наближається до нуля. Виявлення та усунення таких «гарячих точок» (hotspots) вимагає глибокого профайлінгу та часто зміни схеми даних або налаштувань СУБД.

Завершуючи аналіз архітектур, не можна оминати проблему «шумних сусідів» у середовищах віртуалізації та контейнеризації (Docker, Kubernetes). Сучасні СУБД часто розгортаються у контейнерах, які спільно використовують ядро хостової ОС. Архітектурна вразливість полягає у недостатній ізоляції ресурсів вводу-виводу. Якщо інший контейнер на тому ж хості почне інтенсивно використовувати диск, база даних зазнає «голодування» (I/O starvation), що може призвести до тайм-аутів транзакцій та розвалу кластера. Хоча механізми cgroups у Linux дозволяють обмежувати споживання CPU та RAM, управління пропускною здатністю диска залишається складним завданням. Таким чином, перехід до хмарних архітектур (Cloud-Native Database) вводить нові шари абстракції, кожен з яких додає свої потенційні точки відмови, вимагаючи комплексного підходу до проєктування відмовостійкості на всіх рівнях стеку технологій.

Розглядаючи архітектуру сучасних СУБД крізь призму забезпечення цілісності та доступності даних, неможливо оминати увагою механізми управління конкурентним доступом, зокрема технологію багатоверсійності (Multi-Version Concurrency Control, MVCC). Ця архітектурна парадигма, що лежить в основі PostgreSQL, Oracle та InnoDB (MySQL), дозволяє читаючим транзакціям не блокувати записуючі, і навпаки, що критично важливо для високоінтенсивних систем облікових записів. Однак, з погляду відмовостійкості, MVCC вводить нову категорію критичних точок відмови, пов'язану з управлінням «сміттям» (garbage collection). При оновленні запису СУБД не перезаписує старі дані на місці, а створює нову версію рядка, позначаючи попередню як застарілу, але видиму для активних транзакцій. Це призводить до накопичення «мертвих кортежів» (dead tuples). Якщо фоновий процес очищення (наприклад, AutoVacuum у PostgreSQL) не

справляється зі швидкістю генерації сміття або виходить з ладу через помилку планувальника, відбувається неконтрольоване розростання файлів даних (bloat). Це не лише вичерпує дисковий простір, але й призводить до критичної деградації продуктивності індексного сканування, оскільки системі доводиться зчитувати гігабайти неактуальних версій даних, щоб знайти єдину актуальну. У крайніх випадках це викликає так званий «transaction ID wraparound» – вичерпання лічильника ідентифікаторів транзакцій, що змушує СУБД перейти в аварійний режим захисту від втрати даних і припинити обробку будь-яких запитів на запис до завершення тривалої процедури повного вакуумування.

Ще одним архітектурним компонентом, який може стати джерелом відмови, є оптимізатор запитів (Query Optimizer). Хоча формально його завданням є підвищення продуктивності, некоректна робота оптимізатора може призвести до фактичної недоступності системи, що класифікується як «м'яка відмова». Оптимізатор будує план виконання запиту на основі статистичних даних про розподіл значень у стовпцях таблиць. Якщо механізм збору статистики дає збій або статистика застаріває (наприклад, після масового завантаження нових облікових записів), оптимізатор може обрати вкрай неефективний план виконання, наприклад, повне сканування таблиці (Sequential Scan) замість використання індексу, або неефективний алгоритм з'єднання (Nested Loop замість Hash Join) для великих наборів даних. У системі спеціального призначення, де кожен запит на автентифікацію повинен оброблятися за мілісекунди, такий план призводить до лавиноподібного зростання навантаження на підсистему вводу-виводу (I/O Wait) та процесор. Коли черга запитів переповнюється, нові з'єднання відкидаються за тайм-аутом, що для кінцевого користувача виглядає як повна відмова сервера, хоча апаратно він залишається справним. Вразливість тут полягає у відсутності жорстких обмежень (resource quotas) на споживання ресурсів окремим запитом у стандартних налаштуваннях багатьох СУБД.

Глибший рівень аналізу вимагає розгляду взаємодії СУБД з архітектурою пам'яті сучасних серверів, зокрема з архітектурою неоднорідного доступу до пам'яті (Non-Uniform Memory Access, NUMA). У багатопроцесорних системах пам'ять фізично розподілена між процесорними сокетом. Доступ процесора до

«своїй» локальній пам'яті відбувається значно швидше, ніж до пам'яті, підключеної до іншого процесора через шину міжпроцесорної взаємодії (наприклад, Intel QPI або AMD Infinity Fabric). Багато СУБД не є повною мірою NUMA-aware, тобто не враховують топологію пам'яті при розміщенні своїх структур даних. Це створює приховану точку зниження продуктивності та потенційної нестабільності: якщо процес бази даних інтенсивно звертається до віддаленої пам'яті, шина інтерконекту може стати вузьким місцем, викликаючи непрогнозовані затримки (latency spikes). У кластерних конфігураціях такі затримки можуть призвести до помилкового спрацьовування механізмів детектування збоїв (failure detectors), які інтерпретують затримку відповіді як падіння вузла, ініціюючи непотрібну процедуру failover. Оптимізація під NUMA та прив'язка процесів до конкретних ядер (CPU pinning) є критично важливими етапами налаштування відмовостійкого середовища.

Окремою проблемою архітектури розподілених баз даних є управління метаданими кластера та станом конфігурації. У системах, що використовують шардинг (горизонтальне масштабування), існує компонент, який зберігає карту розподілу даних: який діапазон ключів знаходиться на якому фізичному вузлі. Цей компонент, часто реалізований як окремий сервіс конфігурації (Config Server у MongoDB або Master Node у Elasticsearch), стає критичною точкою відмови. Якщо сервери конфігурації стають недоступними (наприклад, через втрату кворуму), клієнтські додатки втрачають можливість визначати маршрути до даних. Навіть якщо самі шарди з даними функціонують нормально, система в цілому стає непрацездатною. Крім того, процес перебалансування даних (sharding rebalancing), який запускається автоматично при додаванні нового вузла або виході з ладу існуючого, створює значне додаткове навантаження на мережу та диски. У погано спроектованих архітектурах це фонове навантаження може призвести до відмови в обслуговуванні основного трафіку користувачів, створюючи ситуацію, коли спроба відновлення системи після незначного збою призводить до глобальної аварії (cascading failure).

Також необхідно проаналізувати вразливості, пов'язані з механізмами реплікації на рівні журналів транзакцій. У синхронній реплікації, яка гарантує нульову втрату даних (RPO=0), транзакція вважається зафіксованою лише після

отримання підтвердження від усіх (або кворуму) реплік. Архітектурна пастка тут полягає у прямій залежності швидкодії системи від найповільнішого вузла або найповільнішого каналу зв'язку. Якщо одна з реплік починає працювати повільно через проблеми з диском або мережею, це гальмує роботу всього кластера (latency penalization). Для вирішення цієї проблеми використовуються складніші алгоритми, такі як «semi-synchronous» реплікація або адаптивні таймаути, проте вони вводять ризик втрати даних при певних сценаріях подвійного збою. Крім того, існує проблема «логічної корупції» при реплікації: якщо на майстер-вузлі через програмну помилку або SQL-ін'єкцію виконується деструктивна команда (наприклад, видалення таблиці), ця команда миттєво реплікується на всі підпорядковані вузли, знищуючи дані у всьому кластері. Традиційна реплікація не захищає від таких помилок, що вимагає впровадження архітектурних рішень відкладеної реплікації (delayed replication) або використання технологій Point-in-Time Recovery (PITR).

Нарешті, критичною точкою відмови в архітектурі захищених систем є підсистема шифрування даних (Transparent Data Encryption, TDE). У системах спеціального призначення дані на дисках повинні зберігатися у зашифрованому вигляді. Архітектура TDE передбачає наявність зовнішнього модуля управління ключами (Key Management Service, KMS) або апаратного модуля безпеки (HSM). При старті СУБД або монтуванні табличного простору система повинна отримати ключ дешифрування від KMS. Якщо мережевий зв'язок з KMS відсутній або сертифікати автентифікації прострочені, база даних не зможе відкрити файли даних і перейде в аварійний стан. Це приклад жорсткої зовнішньої залежності, де надійність системи зберігання даних стає заручником надійності інфраструктури безпеки. Проєктування відмовостійкого кластера повинно включати локальне кешування ключів або кластеризацію самих сервісів KMS для усунення цієї точки відмови.

1.4. Методи реплікації, резервування та балансування навантаження у СУБД

Забезпечення високої доступності та відмовостійкості сучасних систем управління базами даних базується на трьох взаємопов'язаних технологічних стовпах: реплікації даних, резервуванні компонентів та балансуванні навантаження. Реплікація, як процес створення та підтримки актуальних копій даних на декількох фізично рознесених вузлах, є основним механізмом захисту від втрати інформації та забезпечення безперервності бізнес-процесів. З точки зору теорії розподілених систем, вибір методу реплікації завжди є пошуком компромісу між консистентністю даних (Consistency) та затримкою обробки запитів (Latency), що формалізується відомою теоремою CAP (Consistency, Availability, Partition tolerance) [17]. Згідно з цією теоремою, у розподіленій системі, схильній до розділення мережі (що є неминучим фактором у будь-якій кластерній архітектурі), неможливо одночасно гарантувати і сильну узгодженість даних, і повну доступність системи для запису. У контексті систем спеціального призначення, де цілісність облікових записів є пріоритетом, архітектори часто змушені жертвувати доступністю на запис у моменти мережевих збоїв заради збереження консистентності, обираючи CP-системи (Consistency + Partition tolerance).

Найбільш надійним з точки зору збереження даних є метод синхронної реплікації. Механізм його роботи полягає в тому, що транзакція на головному вузлі (Master) не вважається зафіксованою (committed) до тих пір, поки дані не будуть передані на резервний вузол (Slave або Replica), записані там у журнал транзакцій і підтвердження про успішний запис не повернеться на головний вузол. Математично час виконання транзакції T_{sync} у такій системі можна описати формулою:

$$T_{sync} = T_{local_proc} + T_{network_RTT} + T_{remote_write} \quad (1.4)$$

де T_{local_proc} - час обробки на майстрі, $T_{network_RTT}$ - час передачі пакетів туди й назад (Round Trip Time), а T_{remote_write} - час запису на репліці. Очевидно, що продуктивність такої системи на запис обмежена пропускнуою здатністю мережі та

швидкістю найповільнішого диска у кластері. Крім того, синхронна реплікація створює жорстку залежність: вихід з ладу репліки призводить до неможливості завершення транзакцій на майстрі, тобто система втрачає доступність. Для вирішення цієї проблеми використовуються кворумні алгоритми запису, де підтвердження вимагається не від усіх вузлів, а від більшості $N/2 + 1$, що дозволяє системі пережити відмову меншості вузлів без зупинки обслуговування.

На противагу синхронній, асинхронна реплікація дозволяє головному вузлу фіксувати транзакцію та віддавати відповідь клієнту відразу після локального запису, не чекаючи підтвердження від реплік. Передача даних на репліки відбувається у фоновому режимі з певною затримкою (replication lag). Це забезпечує високу продуктивність і низьку латентність, але вводить ризик втрати даних ($RPO > 0$) у випадку аварії на майстрі до того, як дані встигнуть передатися. Величина лага реплікації є випадковою величиною, що залежить від завантаження мережі та процесора. У періоди пікових навантажень лаг може зростати експоненціально, що призводить до ситуації, коли дані на репліках є безнадійно застарілими («stale reads»). Користувач може створити обліковий запис, але при спробі авторизації через інший вузол кластера отримати помилку, оскільки інформація про його реєстрацію ще не дійшла до цього вузла. Ця модель описується моделлю узгодженості у кінцевому рахунку (Eventual Consistency), яка є прийнятною для соціальних мереж, але ризикованою для систем безпеки та фінансових транзакцій. Проміжним варіантом є напівсинхронна реплікація (Semi-synchronous replication), де майстер чекає підтвердження хоча б від однієї репліки з набору, що балансує між надійністю та швидкістю.

Технологічно реплікація може реалізовуватися на різних рівнях. Фізична реплікація (Physical Streaming Replication), характерна для PostgreSQL, оперує блоками диска та записами WAL-файлів. Вона гарантує повну побітову ідентичність майстра та репліки, є дуже швидкою і створює мінімальне навантаження на процесор. Однак вона вимагає однакової архітектури процесорів та версій операційних систем на всіх вузлах і не дозволяє реплікувати окремі бази даних або таблиці – лише весь кластер цілком. Логічна реплікація (Logical Replication), навпаки, оперує потоком змін даних (INSERT, UPDATE, DELETE),

трансляючи їх у логічному форматі. Це дозволяє налаштовувати фільтрацію даних, реплікувати дані між різними версіями СУБД або навіть різними платформами (наприклад, з PostgreSQL в аналітичну систему Kafka), але створює значне навантаження на CPU для парсингу та застосування змін.

Важливим аспектом забезпечення відмовостійкості є стратегії резервування вузлів. У конфігурації Active-Passive (Master-Standby) активним є лише один вузол, який обробляє всі запити, тоді як резервний вузол знаходиться у стані постійної синхронізації, готовий перехопити управління (failover). Цей підхід є простим у реалізації, але неефективним з точки зору використання ресурсів, оскільки обчислювальні потужності резервного сервера простоюють. Більш ефективною є схема Active-Active (Multi-Master), де всі вузли можуть приймати запити на читання та запис. Однак це вводить складну проблему вирішення конфліктів: якщо два користувачі одночасно змінюють один і той самий запис на різних вузлах, виникає колізія. Для вирішення таких ситуацій застосовуються векторні годинники (Vector Clocks) або стратегії «останній запис виграв» (Last Write Wins), що не завжди є коректним для бізнес-логіки. Тому в системах облікових записів часто використовується гібридний підхід: один Master для запису та безліч Read Replicas для читання, що дозволяє масштабувати навантаження на читання, зберігаючи простоту управління транзакціями.

Балансування навантаження є критичним компонентом, який розподіляє вхідний трафік між доступними вузлами кластера, забезпечуючи рівномірне завантаження ресурсів та ізоляцію відмов. Балансувальники можуть працювати на транспортному рівні (Layer 4 моделі OSI), оперуючи IP-адресами та TCP-портами, або на прикладному рівні (Layer 7), аналізуючи вміст SQL-запитів. L4-балансування (наприклад, через IPVS або HAProxy в режимі TCP) є надзвичайно швидким, але «сліпим» до логіки додатку: воно не може відрізнити запит на читання від запиту на запис. L7-балансування (наприклад, Pgpool-II або ProxySQL) дозволяє реалізувати розщеплення читання/запису (Read/Write Splitting), автоматично направляючи операції модифікації даних на Master-вузол, а вибірки - на репліки. Це прозоро для клієнтського додатку, але вносить додаткову затримку на парсинг протоколу бази даних.

Математична ефективність балансування залежить від обраного алгоритму розподілу. Найпростіший алгоритм Round Robin циклічно передає запити вузлам, не враховуючи їх поточне завантаження, що може призвести до перевантаження слабкого вузла складними запитами. Алгоритм Least Connections направляє новий запит на сервер з найменшою кількістю активних з'єднань, що є більш адаптивним підходом для довготривалих сесій баз даних. Для систем з кешуванням критично важливим є алгоритм узгодженого хешування (Consistent Hashing). У класичному хешуванні при зміні кількості серверів N більшість ключів змінюють своє розташування ($hash(key) \bmod N$), що призводить до повного скидання кешу. Узгоджене хешування розташовує сервери та ключі на віртуальному кільці, тому при додаванні або видаленні вузла переміщується лише K/N ключів, де K – загальна кількість ключів. Це дозволяє динамічно змінювати розмір кластера без значних потрясінь для продуктивності (thundering herd problem).

Особливу увагу слід приділити механізмам виявлення та ізоляції збоїв у контексті балансування. Балансувальник повинен постійно моніторити стан бекенд-серверів за допомогою активних перевірок (Health Checks). Якщо сервер не відповідає на перевірку протягом заданого інтервалу, він виключається з пулу балансування. Проте існує проблема «мерехтіння» (flapping), коли сервер балансує на межі доступності через перевантаження, то з'являючись, то зникаючи зі списку активних. Це призводить до нестабільності сесій користувачів. Для запобігання цьому застосовують гістерезис: сервер повертається в роботу лише після серії успішних перевірок. Також критичним є використання віртуальних IP-адрес (VIP) та протоколів VRRP (Virtual Router Redundancy Protocol) або CARP для забезпечення відмовостійкості самого балансувальника. У такій схемі кілька екземплярів балансувальників об'єднуються в кластер, і у випадку падіння основного, резервний миттєво перехоплює VIP, забезпечуючи безперервність обслуговування клієнтів.

Варто також розглянути архітектурний патерн «Circuit Breaker» (автоматичний вимикач), який часто реалізується на рівні драйвера бази даних або балансувальника. Його мета – запобігти каскадним відмовам. Якщо кількість помилок при зверненні до певного вузла перевищує порогове значення, «вимикач»

розмикається, і всі наступні запити до цього вузла миттєво відхиляються без спроби виконання, що дозволяє системі заощадити ресурси і дає час проблемному вузлу на відновлення. Через певний проміжок часу «вимикач» переходить у стан «напіввідкритого», пропускаючи пробний трафік для перевірки працездатності. Такий підхід є проявом реактивної архітектури, здатної до самовідновлення та адаптації під навантаженням.

Таким чином, методи реплікації, резервування та балансування утворюють єдиний екосистемний комплекс. Вибір конкретної комбінації (наприклад, синхронна реплікація + Active-Passive резервування + L4 балансування) залежить від вимог до RPO/RTO та допустимих бюджетів на інфраструктуру. Для інформаційної системи спеціального призначення, де вартість втрати даних є неприйнятно високою, доцільно використовувати гібридні схеми з синхронною реплікацією в межах одного дата-центру для захисту даних та асинхронною реплікацією в резервний дата-центр для забезпечення катастрофостійкості (Disaster Recovery), поєднуючи це з інтелектуальним L7-балансуванням для оптимізації продуктивності читання.

Розширюючи теоретичний базис аналізу методів реплікації, необхідно вийти за межі класичної теореми CAP і звернутися до більш деталізованої теореми PACELC, яка розширює розуміння компромісів у розподілених системах [7]. Якщо теорема CAP описує поведінку системи лише під час розділення мережі (Partition), то PACELC стверджує: якщо розділення є (P), система повинна обирати між доступністю (A) та узгодженістю (C); але якщо розділення немає (E - else), система повинна обирати між затримкою (L - latency) та узгодженістю (C). Це фундаментальне уточнення пояснює, чому навіть у стабільних мережах адміністратори змушені налаштовувати рівні узгодженості. Наприклад, вимога суворої лінеаризовності (linearizability), яка гарантує, що після завершення запису всі подальші читання повернуть нове значення, неминуче призводить до збільшення затримок на синхронізацію всіх реплік. Математично вартість лінеаризовності виражається у збільшенні часу відповіді R на величину, пропорційну найдовшій затримці у мережі між кворумними вузлами:

$$R \geq \max(d_1, d_2 \dots d_k), \quad (1.5)$$

де d_1 - затримка до i -го вузла.

Ключовим елементом сучасних відмовостійких систем є алгоритми розподіленого консенсусу, такі як P_{axos} та P_{aft} , які вирішують задачу узгодження стану кінцевих автоматів (State Machine Replication) в умовах ненадійного середовища. На відміну від простих протоколів реплікації, ці алгоритми гарантують безпеку (safety) – тобто відсутність суперечливих рішень – за будь-яких умов, окрім візантійських відмов. Алгоритм P_{aft} , який став стандартом де-факто для систем оркестрації та розподілених сховищ (etcd, Consul, CockroachDB), базується на концепції виборів лідера та логування команд [8]. Час до збіжності (convergence time) алгоритму P_{aft} залежить від налаштувань тайм-аутів виборів (election timeout), які зазвичай обираються випадковим чином з інтервалу $[T, 2T]$ для запобігання ситуації «розщепленого голосу» (split vote), коли кандидати отримують однакову кількість голосів і вибори заходять у глухий кут. Математична ймовірність того, що вибори не завершаться успішно за k раундів, експоненціально спадає зі зростанням k , що робить систему стабільною на практиці. Однак, у високошвидкісних мережах навіть коротка затримка на перевибори лідера (сотні мілісекунд) може спричинити помітний сплеск відмов у обслуговуванні клієнтських запитів.

Для забезпечення гнучкості налаштування відмовостійкості використовується модель кворумів, яка описується нерівністю:

$$R + W > N, \quad (1.6)$$

де N – загальна кількість реплік, W – кількість вузлів, які повинні підтвердити запис, а R – кількість вузлів, які опитуються при читанні. Варіюючи ці параметри, можна досягати різних характеристик системи. Конфігурація $W=N, R=1$ забезпечує найшвидше читання, але найповільніший запис і низьку доступність на запис (відмова одного вузла блокує систему). Конфігурація $W=1, R=N$ дає миттєвий запис, але повільне читання. Найбільш збалансованою для систем спеціального призначення є схема більшості (majority quorum), де $W = R = \lfloor N/2 \rfloor + 1$. Це дозволяє системі пережити відмову $\lfloor (N-1)/2 \rfloor$ вузлів. Також застосовується техніка «нечітких кворумів» (sloppy quorum) та «передачі натяків» (hinted handoff), яка дозволяє тимчасово записати дані на нецільовий вузол, якщо цільовий недоступний, з

подальшою міграцією даних після відновлення. Це підвищує доступність на запис, але вводить ризик втрати даних при множинних відмовах, що вимагає ретельної оцінки ризиків.

Розглядаючи задачу балансування навантаження з наукової точки зору, необхідно звернутися до теорії масового обслуговування (Queueing Theory). База даних може бути змодельована як система $M/M/k$, де вхідний потік запитів є пуассонівським, час обслуговування розподілений експоненціально, а k – кількість обробників (ядер процесора або потоків). Згідно з законом Літла (Little's Law), середня кількість запитів у системі L дорівнює добутку середньої інтенсивності вхідного потоку λ на середній час перебування запиту в системі W : $L = \lambda W$. Ця формула демонструє, що при наближенні завантаження системи ($\rho = \lambda/(k\mu)$) до одиниці, час очікування в черзі зростає не лінійно, а гіперболічно. Саме тому балансувальники навантаження повинні використовувати не просто стратегію Round Robin, а адаптивні алгоритми, які враховують поточну довжину черги на кожному вузлі (Least Pending Requests), щоб не направляти нові запити на вже перевантажені сервери, запобігаючи переходу системи у стан насичення.

Окремим класом алгоритмів реплікації є безконфліктні репліковані типи даних (Conflict-free Replicated Data Types, CRDTs), які забезпечують сильну узгодженість у кінцевому рахунку (Strong Eventual Consistency) без необхідності централізованої координації. CRDTs базуються на математичних властивостях комутативності, асоціативності та ідемпотентності операцій злиття станів. Наприклад, для лічильника невдалих спроб входу в систему можна використовувати G-Counter (Grow-only Counter), який гарантує коректне підсумовування значень з усіх реплік незалежно від порядку надходження повідомлень. Використання CRDT дозволяє будувати мульти-майстер системи, стійкі до тривалих розривів зв'язку, де кожен вузол може автономно приймати зміни, які потім коректно синхронізуються. Це особливо актуально для георозподілених кластерів, де затримки мережі унеможливають використання синхронних протоколів типу 2PC (Two-Phase Commit).

Також слід детально розглянути проблему «громоподібного стада» (Thundering Herd) при балансуванні навантаження. Ця проблема виникає, коли

велика кількість процесів або потоків одночасно «прокидаються» у відповідь на подію (наприклад, звільнення блокування або відновлення роботи бази даних), створюючи миттєве пікове навантаження, яке може знову «покласти» систему. У контексті кешування це відомо як «cache stampede»: коли термін дії популярного ключа в кеші спливає, тисячі паралельних запитів одночасно звертаються до бази даних для його оновлення. Для запобігання цьому використовуються імовірнісні методи реплікації кешу або алгоритм X-Fetch, який оновлює значення у кеші до його фактичного зникнення. На рівні балансувальників застосовується техніка експоненціальної відстрочки (exponential backoff) з випадковим розкидом (jitter) при повторних спробах з'єднання, що дозволяє «розмазати» навантаження у часі і дати системі можливість стабілізуватися.

Варто згадати і про методи сегментування даних (Sharding) як спосіб горизонтального масштабування, що безпосередньо впливає на методи реплікації та балансування. Існує два основні підходи: шардинг за діапазоном значень (Range-based sharding) та шардинг за хешем (Hash-based sharding). Перший метод дозволяє ефективно виконувати запити на вибірку діапазонів, але схильний до проблеми «гарячих точок», коли послідовні записи (наприклад, за часовою міткою) потрапляють на один шард, перевантажуючи його. Хеш-шардинг рівномірно розподіляє навантаження, але робить запити діапазонів неефективними, оскільки вони вимагають опитування всіх шардів (scatter-gather). Вибір стратегії шардингу для бази даних облікових записів повинен враховувати патерни доступу: якщо доступ здебільшого відбувається за унікальним ID користувача, хеш-шардинг є кращим вибором. Однак, операція зміни кількості шардів (resharding) є однією з найскладніших і найризикованіших процедур, що вимагає переміщення гігабайтів даних мережею, часто призводячи до тимчасової деградації продуктивності кластера.

На завершення аналізу методів резервування, необхідно відзначити концепцію незмінюваної інфраструктури (Immutable Infrastructure), яка стає популярною у хмарних середовищах. Замість того, щоб оновлювати конфігурацію існуючих серверів «на льоту», що несе ризик дрейфу конфігурації (configuration drift) і непередбачуваних помилок, нова версія бази даних або її проксі-шару

розгортається на нових віртуальних машинах паралельно зі старою. Після успішного проходження перевірок трафік атомарно перемикається на нову групу серверів. Цей підхід, відомий як Blue-Green Deployment або Canary Release, значно підвищує надійність процесу оновлення і дозволяє миттєво відкотитися (rollback) до попереднього стану у разі виявлення проблем. Така методологія змінює саме поняття резервування: резервом стає не простоююче «залізо», а здатність автоматизованої системи швидко розгорнути нове середовище з коду.

1.5. Огляд стандартів і технологій забезпечення високої доступності (RAID, HA, Failover Clusters, DRP, Backup Policies)

Забезпечення високої доступності інформаційних систем спеціального призначення вимагає системного підходу, що спирається на перевірені часом індустріальні стандарти та технологічні патерни, які регламентують створення відмовостійкої інфраструктури на всіх рівнях: від організації бітів на магнітних пластинах до глобальних стратегій відновлення після катастроф. Фундаментальним рівнем захисту даних від фізичних збоїв носіїв інформації є технологія RAID (Redundant Array of Independent Disks), яка базується на методах віртуалізації зберігання даних шляхом об'єднання кількох фізичних дисків у єдиний логічний модуль. Математична модель надійності RAID-масивів суттєво відрізняється залежно від обраного рівня надлишковості та використовує теорію ймовірностей для оцінки ризиків втрати даних. Якщо розглядати ймовірність відмови одного диска протягом певного проміжку часу як p , то надійність (ймовірність безвідмовної роботи) окремого диска становить $R = 1 - p$.

Для масиву рівня R_{RAID0} , який використовує чергування блоків (striping) без надлишковості для підвищення продуктивності, надійність всього масиву з n дисків визначається як добуток надійностей окремих компонентів: $R_{RAID0} = (1 - p)^n$. Це означає, що відмова будь-якого одного диска призводить до повної втрати даних всього масиву, роблячи $RAID0$ непридатним для критично важливих систем, оскільки його надійність завжди нижча за надійність одного диска. На противагу цьому, $RAID1$ використовує дзеркалювання (mirroring), де дані дублюються на n дисків. Ймовірність відмови такої системи дорівнює ймовірності

одночасної відмови всіх дисків: $R_{fail} = p^n$. Відповідно, надійність становить $R_{RAID1} = 1 - p^n$, що демонструє експоненціальне зростання надійності зі збільшенням кількості дзеркал. Проте економічна неефективність *RAID1* (висока вартість зберігання одиниці інформації) зумовила популярність масивів з чергуванням та парністю, таких як *RAID 5* та *RAID 6*.

RAID 5 використовує обчислення контрольної суми (парності) за допомогою операції логічного виключного «АБО» (XOR), розподіляючи блоки парності між усіма дисками масиву. Це дозволяє відновити дані при виході з ладу одного будь-якого диска, використовуючи інформацію з дисків, що залишилися. Однак критичною вразливістю *RAID 5* є явище «діри запису» (Write Hole), яке виникає, коли живлення зникає в момент запису. Оскільки оновлення блоку даних та відповідного блоку парності не є атомарною операцією на фізичному рівні, можлива ситуація, коли дані оновлено, а парність – ні (або навпаки). Це призводить до неузгодженості, яку неможливо виявити звичайними засобами, і яка під час відновлення може спричинити корупцію даних. Для систем спеціального призначення більш надійним є *RAID 6*, який використовує подвійну парність на основі кодів Ріда-Соломона, дозволяючи пережити одночасну відмову двох дисків. Математично ймовірність відмови *RAID 6* значно нижча, проте операції запису стають суттєво повільнішими через необхідність обчислення складних поліномів у полях Галуа для генерації синдромів помилок.

Окрім того, при проєктуванні сховищ великого обсягу необхідно враховувати ймовірність виникнення невідновлюваної помилки читання (Unrecoverable Read Error, URE). Сучасні жорсткі диски мають специфіковану частоту URE (зазвичай 1 біт на 10^{14} прочитаних біт для дисків споживчого класу та 1 на 10^{15} для корпоративного). При відновленні масиву *RAID 5* після заміни несправного диска контролеру необхідно прочитати весь обсяг даних з решти дисків. Якщо під час цього процесу трапиться хоча б одна помилка URE, процес відновлення (rebuild) переривається, і весь масив втрачається. Для масивів обсягом у десятки терабайт ймовірність такої події наближається до одиниці, що робить *RAID 5* технологічно застарілим та небезпечним для сучасних баз даних. Тому стандартом де-факто для високопродуктивних та надійних систем є комбіновані рівні, такі як *RAID 10* (смуга

дзеркал), які поєднують швидкість *RAID 0* та надійність *RAID 1*, уникаючи при цьому обчислювальних витрат на парність.

На рівень вище від дискової підсистеми знаходяться технології високої доступності (High Availability, HA) та відмовостійкі кластери (Failover Clusters). Основною метою HA-кластера є мінімізація часу простою (Downtime) шляхом автоматичного виявлення збоїв програмного або апаратного забезпечення та перезапуску сервісів на резервних вузлах. Ключовим поняттям тут є Resource Manager (наприклад, Pacemaker у Linux-середовищах або Failover Cluster Manager у Windows), який керує станом ресурсів (IP-адреси, файлові системи, сервіси СУБД). Критично важливим механізмом у кластерах є «Fencing» (огороження) або STONITH (Shoot The Other Node In The Head). Цей механізм гарантує, що вузол, який поводить себе некоректно або втратив зв'язок з рештою кластера, буде примусово вимкнений (наприклад, через відключення порту на розумному блоці розподілу живлення PDU або через інтерфейс IPMI). Без надійного фенсінгу існує ризик одночасного запису даних двома вузлами на спільний дисковий масив, що призводить до незворотної корупції файлової системи.

Стандартизація побудови центрів обробки даних, в яких розміщуються кластери, регламентується стандартом TIA-942 та класифікацією Uptime Institute [14]. Згідно з цією класифікацією, виділяють чотири рівні надійності (Tier). Tier I – це базова інфраструктура без резервування, що гарантує доступність на рівні 99.671%. Tier II додає резервування критичних компонентів живлення та охолодження (N+1), підвищуючи доступність до 99.741%. Tier III вимагає можливості паралельного обслуговування (Concurrent Maintainability), що означає наявність дублюючих шляхів доставки електроенергії та холоду, дозволяючи виводити будь-який компонент у ремонт без зупинки сервісів; доступність складає 99.982%. Tier IV – це відмовостійка інфраструктура (Fault Tolerant), де система здатна автоматично пережити будь-яку одиничну аварію без впливу на роботу, забезпечуючи доступність 99.995% (не більше 26 хвилин простою на рік). Для баз даних облікових записів спеціального призначення мінімально допустимим є рівень Tier III.

Забезпечення безперервності бізнесу регулюється міжнародним стандартом ISO 22301 «Security and resilience – Business continuity management systems». Цей стандарт вимагає впровадження циклу Демінга (Plan-Do-Check-Act) для управління ризиками переривання діяльності. У рамках цього стандарту розробляється План відновлення після катастроф (Disaster Recovery Plan, DRP), який є документованим набором процедур для відновлення IT-інфраструктури у випадку масштабних аварій (пожежа, повінь, кібератака). DRP тісно пов'язаний з поняттями RPO (Recovery Point Objective) – максимальний період часу, за який можуть бути втрачені дані, та RTO (Recovery Time Objective) – максимальний час, необхідний для відновлення сервісу. Важливо розуміти, що RPO визначається частотою створення резервних копій та типом реплікації, тоді як RTO залежить від ступеня автоматизації процесів розгортання та наявності резервних потужностей.

Політики резервного копіювання (Backup Policies) є останньою лінією оборони у забезпеченні збереженості даних. Сучасні стратегії бекапування виходять далеко за межі простого копіювання файлів. Загальноприйнятим стандартом є правило «3-2-1»: мати як мінімум три копії даних, зберігати їх на двох різних типах носіїв (наприклад, дисковий масив і магнітна стрічка або хмарне сховище), і одну копію зберігати територіально віддалено (off-site). Для баз даних використовуються різні типи резервного копіювання: повне (Full), яке копіює весь масив даних; диференційне (Differential), що копіює зміни з моменту останнього повного бекапу; та інкрементальне (Incremental), що зберігає лише зміни з моменту будь-якого останнього бекапу. Математично, час відновлення з інкрементальних копій є найдовшим ($T_{restore} = T_{full} + \Sigma T_{inc}$), оскільки системі необхідно послідовно застосувати ланцюжок змін, тоді як відновлення з диференційної копії є швидшим ($T_{restore} = T_{full} + T_{diff}$), але вимагає більше місця для зберігання проміжних копій.

У системах з великими обсягами даних критично важливим є використання технології дедуплікації, яка дозволяє зменшити обсяг резервних копій у десятки разів шляхом збереження лише унікальних блоків даних. Дедуплікація може бути реалізована на джерелі (source-side) або на цільовому пристрої (target-side). Алгоритми дедуплікації використовують криптографічні хеш-функції (наприклад,

SHA-256) для ідентифікації блоків. Проте тут виникає теоретична проблема колізій хешів: ймовірність того, що два різні блоки даних матимуть однаковий хеш. Хоча для SHA-256 ця ймовірність є астрономічно малою, у системах спеціального призначення, що оперують ексабайтами даних, вона не дорівнює нулю, що вимагає додаткових механізмів верифікації побітовим порівнянням, що уповільнює процес.

Також слід розглянути стратегію ротації резервних копій, наприклад, схему «Дід-Батько-Син» (Grandfather-Father-Son, GFS). Ця схема передбачає ієрархічне зберігання: щоденні копії (сини) зберігаються тиждень, щотижневі (батьки) – місяць, а щомісячні (діди) – рік або довше. Це дозволяє ефективно балансувати між глибиною архіву та витратами на зберігання. Для захищених систем критичним є шифрування резервних копій. Якщо бекап бази даних облікових записів потрапляє до рук зловмисників, це еквівалентно компрометації самої бази. Тому стандарти вимагають використання апаратного шифрування на стрічкових приводах або програмного шифрування перед передачею даних мережею, при цьому ключі шифрування повинні зберігатися окремо від самих бекапів у захищеному сховищі ключів (Key Management Service).

Окремим аспектом є тестування відновлення (Recovery Testing). Статистика свідчить, що значний відсоток резервних копій виявляється непридатним для відновлення через пошкодження носіїв або помилки конфігурації. Стандарт ISO 22301 вимагає регулярного проведення навчань з відновлення, які можуть варіюватися від настільних перевірок (tabletop exercises), де обговорюються процедури, до повномасштабних симуляцій з реальним перемиканням на резервний дата-центр. Без регулярного підтвердження можливості відновлення (Backup Verification) будь-яка політика резервування є лише ілюзією безпеки. Автоматизовані системи бекапування повинні включати етап тестового розгортання бекапу у «пісочницю» (sandbox) та виконання контрольних запитів для перевірки цілісності логічної структури бази даних.

Таким чином, комплекс стандартів та технологій високої доступності формує багаторівневу систему захисту. RAID захищає від відмови диска, кластеризація НА – від відмови сервера, геореплікація – від відмови дата-центру, а резервне копіювання – від логічних помилок та людського фактора. Інтеграція цих

технологій в єдину архітектуру вимагає ретельного математичного розрахунку показників надійності та постійного аудиту на відповідність міжнародним стандартам, що є обов'язковою умовою для сертифікації інформаційних систем спеціального призначення.

Висновки до першого розділу

Проведений у першому розділі аналіз теоретичних засад відмовостійкості засвідчив, що для інформаційних систем спеціального призначення забезпечення безперервності функціонування та цілісності даних є критичним фактором. Детальна класифікація відмов та вивчення архітектур сучасних СУБД дозволили виявити основні вразливості, зокрема на рівні управління буферами, журналізації транзакцій та мережевої взаємодії, а також підтвердили необхідність переходу від монолітних до розподілених кластерних архітектур для усунення єдиних точок відмови. Аналіз міжнародних стандартів TIA-942 та ISO 22301, разом із дослідженням теорем CAP і PACELC, визначив ключові підходи до проєктування надійної інфраструктури: використання структурної надлишковості, алгоритмів розподіленого консенсусу та комплексних стратегій реплікації.

Узагальнюючи результати, можна стверджувати, що побудова ефективної системи вимагає інтеграції апаратних рішень та програмних механізмів захисту, що створює необхідне теоретичне підґрунтя для проєктування кластера в наступному розділі.

РОЗДІЛ 2. АНАЛІЗ ТЕХНОЛОГІЙ ТА ПРОЄКТУВАННЯ ВІДМОВСТІЙКОГО КЛАСТЕРА БАЗИ ДАНИХ ОБЛІКОВИХ ЗАПИСІВ КОРИСТУВАЧІВ

2.1. Аналіз сучасних СУБД з підтримкою кластеризації (PostgreSQL, MySQL, MongoDB, MS SQL Server)

Вибір системи управління базами даних є фундаментальним етапом проєктування інформаційної системи спеціального призначення, оскільки саме можливості СУБД визначають граничні показники надійності, масштабованості та захищеності даних облікових записів. Сучасний ринок пропонує широкий спектр рішень, серед яких домінуючі позиції займають як реляційні системи з суворою схемою даних, так і NoSQL-рішення, орієнтовані на гнучкість та горизонтальне масштабування. Для обґрунтованого вибору платформи необхідно провести глибокий порівняльний аналіз архітектурних особливостей провідних СУБД - PostgreSQL, MySQL, MongoDB та MS SQL Server – крізь призму їх здатності до кластеризації, автоматичного відновлення після збоїв та забезпечення транзакційної цілісності в розподіленому середовищі. Цей аналіз повинен враховувати специфіку зберігання облікових записів, що вимагає відповідності вимогам ACID (Atomicity, Consistency, Isolation, Durability) та високого рівня безпеки.

СУБД PostgreSQL, яка позиціонується як найпотужніша об'єктно-реляційна система з відкритим вихідним кодом, характеризується мультипроцесною архітектурою, де кожне клієнтське з'єднання обслуговується окремим процесом операційної системи. Така архітектура забезпечує високу стабільність: аварійне завершення одного процесу не призводить до падіння всього екземпляра бази даних, що є критичною перевагою для відмовостійкості. Вбудовані механізми реплікації PostgreSQL базуються на трансляції записів журналу попереднього запису (Write-Ahead Log, WAL). Починаючи з версії 9.0, підтримується потокова реплікація (Streaming Replication), яка дозволяє передавати зміни на резервні вузли (Standby) з мінімальною затримкою, фактично в режимі реального часу. Важливою особливістю є підтримка синхронної реплікації на рівні окремих транзакцій, що

дозволяє адміністратору гнучко налаштовувати баланс між надійністю та продуктивністю: критичні операції зміни паролів можуть виконуватися синхронно, гарантуючи їх збереження на репліці, тоді як менш важливі оновлення, наприклад, час останнього входу – асинхронно.

Однак «з коробки» PostgreSQL не надає повноцінного механізму автоматичного перемикавання (failover) та управління кластером. Для побудови високодоступного кластера (HA Cluster) необхідно використовувати сторонні інструменти оркестрації, такі як Patroni, Repmgr або Stolon. Patroni на сьогодні вважається галузевим стандартом, оскільки він використовує розподілені сховища конфігурації (DCS), такі як etcd, Consul або ZooKeeper, для зберігання стану кластера та вибору лідера [2]. Архітектура на базі Patroni дозволяє уникнути ситуації «split-brain», автоматично керуючи файлом конфігурації postgresql.conf та файлом pg_hba.conf на всіх вузлах. При виході з ладу майстер-вузла Patroni ініціює вибори, просуває найбільш актуальну репліку до ролі майстра та автоматично переналаштовує інші репліки на нового лідера, що робить процес відновлення прозорим для додатків при використанні спільної віртуальної IP-адреси або балансувальника навантаження.

MySQL, будучи найпоширенішою веб-орієнтованою СУБД, пропонує принципово інший підхід до кластеризації, особливо з використанням двигуна зберігання InnoDB. Традиційна реплікація MySQL (Master-Slave) історично базувалася на бінарних логах (binlog) та позиціонуванні за зміщенням файлу, що було джерелом численних проблем розсинхронізації. Впровадження глобальних ідентифікаторів транзакцій (GTID) значно спростило управління топологією реплікації та відновлення після збоїв. Проте найбільш просунутим рішенням для високої доступності в екосистемі MySQL є MySQL Group Replication та його реалізація у вигляді InnoDB Cluster [18]. Це рішення базується на алгоритмі консенсусу Raft, що дозволяє створювати мульти-майстер кластери з гарантією узгодженості даних. У груповій реплікації транзакція вважається зафіксованою лише тоді, коли вона отримала підтвердження (сертифікацію) від більшості вузлів групи. Це автоматично вирішує проблему конфліктів запису, оскільки транзакції, що конфліктують, відхиляються на етапі сертифікації.

Альтернативою нативним рішенням MySQL є Galera Cluster – технологія синхронної мульти-майстер реплікації, яка працює на рівні бібліотеки wsrep API. Galera забезпечує істинний паралельний запис на всі вузли кластера, гарантуючи, що всі вузли мають ідентичний набір даних у будь-який момент часу. Це досягається завдяки сертифікації наборів запису (writesets) на основі глобального впорядкування транзакцій. Перевагою такої архітектури для систем облікових записів є відсутність затримок реплікації при читанні з будь-якого вузла та миттєве перемикання клієнтів у разі відмови одного з серверів. Однак, жорстка вимога синхронності накладає обмеження на мережеву затримку: кластер Galera не рекомендується розгортати через повільні канали WAN, оскільки швидкість запису всього кластера буде обмежена швидкістю найповільнішого вузла (проблема «найслабшої ланки»).

MongoDB представляє клас документо-орієнтованих NoSQL систем, які від початку проєктувалися з урахуванням вимог горизонтального масштабування та відмовостійкості. Базовою одиницею високої доступності в MongoDB є Replica Set (набір реплік) – група процесів mongod, що підтримують один і той самий набір даних [19]. Архітектура Replica Set включає один первинний вузол (Primary), який приймає всі операції запису, та кілька вторинних вузлів (Secondary), які реплікують дані через oplog (operation log). Особливістю MongoDB є вбудований механізм автоматичних виборів лідера на основі алгоритму, схожого на Raft. Якщо первинний вузол стає недоступним, вторинні вузли проводять голосування і обирають нового лідера протягом кількох секунд. Це робить MongoDB надзвичайно стійкою до апаратних збоїв без необхідності використання складних зовнішніх інструментів оркестрації.

Для систем облікових записів важливим аспектом MongoDB є можливість гнучкого налаштування рівня узгодженості (Consistency) та довговічності (Durability) за допомогою механізму Write Concern. Встановивши параметр w: «majority», розробник може гарантувати, що дані не будуть вважатися записаними, доки вони не будуть зафіксовані на більшості вузлів кластера, що виключає втрату даних при відкаті транзакцій (rollback) під час виборів. Однак, MongoDB використовує модель узгодженості у кінцевому рахунку (Eventual Consistency) для

операцій читання з вторинних вузлів, що може призвести до читання застарілих даних (stale reads). Хоча починаючи з версії 4.0 MongoDB підтримує мультидокументні ACID-транзакції, її використання для критичних даних облікових записів вимагає ретельного проєктування схеми даних, щоб уникнути аномалій.

MS SQL Server від компанії Microsoft є яскравим представником комерційних корпоративних СУБД з глибокою інтеграцією в екосистему Windows Server. Основним механізмом забезпечення високої доступності є технологія Always On Availability Groups (AG) [20]. Ця технологія дозволяє створювати групи баз даних, які спільно перемикаються на резервні вузли у разі збою. Always On базується на службі кластеризації Windows Server Failover Clustering (WSFC), яка забезпечує моніторинг стану вузлів (quorum health monitoring) та управління віртуальними іменами мережі. Реплікація даних у Always On AG відбувається на рівні блоків журналу транзакцій, що є більш ефективним, ніж реплікація на рівні SQL-запитів. Підтримується як синхронний режим (для нульової втрати даних та автоматичного failover), так і асинхронний (для катастрофостійкості та розвантаження читання).

Унікальною перевагою MS SQL Server є наявність Failover Cluster Instances (FCI) – архітектури зі спільним дисковим простором (Shared Storage), де екземпляр SQL Server може переміщуватися між вузлами кластера, підключаючись до тих самих файлів даних на SAN-масиві. Це захищає від збоїв сервера, але не захищає від збоїв дискового масиву, тому FCI часто комбінують з Always On AG для забезпечення повного спектру захисту. З точки зору безпеки, MS SQL Server пропонує одні з найпотужніших вбудованих засобів, включаючи прозоре шифрування даних (TDE), динамічне маскуванню даних та Row-Level Security, що є критичним для систем спеціального призначення. Проте закритість коду та висока вартість ліцензування (особливо для Enterprise-версії, необхідної для повноцінного Always On) є стримуючими факторами при виборі цієї платформи для бюджетних державних проєктів.

Порівнюючи розглянуті системи у контексті задачі зберігання облікових записів спеціального призначення, слід зазначити, що хоча NoSQL рішення типу MongoDB пропонують найпростішу масштабованість, реляційна модель даних

(PostgreSQL, MySQL, MS SQL) є більш природною для структурованої інформації про користувачів, їхні ролі та привілеї. Реляційні СУБД забезпечують сувору посилальну цілісність (Referential Integrity), що унеможливорює появу «сирітських» записів (наприклад, права доступу, прив'язані до неіснуючого користувача). Серед реляційних систем PostgreSQL виділяється оптимальним співвідношенням функціональності, надійності та вартості володіння. Її механізми розширюваності, підтримка складних типів даних (JSONB, масиви) та можливість побудови надійних кластерів за допомогою Patroni роблять її пріоритетним кандидатом для реалізації захищеної системи автентифікації. MySQL з Galera Cluster також є сильним конкурентом, особливо якщо потрібна мульти-майстер архітектура, проте PostgreSQL традиційно вважається більш стійкою до складних SQL-запитів та великих навантажень на аналітику.

Поглиблюючи аналіз PostgreSQL у контексті систем спеціального призначення, необхідно детально розглянути механізм управління часовими шкалами (TimeLines), який є критичним для забезпечення безперервності історії транзакцій при відновленні. Кожного разу, коли виконується процедура відновлення після збою (PITR – Point-in-Time Recovery) або просування репліки до ролі майстра, PostgreSQL створює нову часову шкалу, що ідентифікується унікальним TimeLineID. Це архітектурне рішення дозволяє уникнути перезапису історії WAL-файлів, що могло б статися, якби нова гілка транзакцій використовувала ті самі ідентифікатори, що й стара. Для адміністратора системи це означає можливість повернутися до стану «до аварії» навіть після того, як кластер почав працювати в новій конфігурації. Математично позиція в журналі транзакцій описується 64-бітним числом LSN (Log Sequence Number), яке монотонно зростає. При синхронній реплікації гарантія збереження даних базується на нерівності $LSN_{flush_standby} \geq LSN_{commit_master}$, де $LSN_{flush_standby}$ – позиція, записана на диск репліки. Якщо ця умова не виконується, транзакція на майстрі переходить у стан очікування, блокуючи клієнтський процес.

Вразливістю PostgreSQL у кластерних конфігураціях є проблема «роздуття» WAL-файлів (WAL bloat) при відключенні репліки. Майстер-вузол зобов'язаний зберігати всі WAL-файли, які ще не були передані на репліку, для забезпечення

можливості її синхронізації після повернення в стрій. Якщо репліка відсутня тривалий час, обсяг накопичених логів може вичерпати вільне місце на диску майстра, що призведе до аварійної зупинки основного сервера (PANIC: could not write to file «pg_wal»). Для вирішення цієї проблеми у версії 9.4 було введено механізм слотів реплікації (Replication Slots), який автоматизує управління утриманням логів. Проте у системах високої надійності використання слотів несе ризик: слот «мертвої» репліки може вбити майстер. Тому необхідне впровадження зовнішніх моніторингових агентів, які примусово видаляють слоти неактивних реплік після досягнення критичного порогу заповнення диска, жертвуючи можливістю швидкої синхронізації заради виживання основного вузла [25].

Розглядаючи детальніше архітектуру Galera Cluster для MySQL, варто проаналізувати алгоритм сертифікації транзакцій, який замінює традиційні механізми блокування. У Galera використовується оптимістичний підхід до виконання транзакцій: транзакція виконується локально на вузлі, і лише на етапі *COMMIT* надсилається на всі вузли кластера для перевірки на конфлікти. Кожен вузол підтримує чергу транзакцій та перевіряє, чи не перетинається набір ключів (Write Set) нової транзакції з ключами транзакцій, які вже були зафіксовані, але ще не застосовані до локальної бази даних. Унікальною особливістю є використання глобального порядкового номера (*wsrep_seqno*). Якщо сертифікація проходить успішно, транзакція гарантовано застосовується. Якщо ж виявляється конфлікт, транзакція відкочується (deadlock error), і клієнтський додаток повинен повторити спробу. Для систем з високою інтенсивністю запису в одні й ті ж таблиці (hotspots) це може призвести до значної кількості відкатів і зниження ефективної пропускної здатності. Крім того, критичним моментом є процес перенесення стану (State Snapshot Transfer – SST) при підключенні нового вузла. Повний SST часто блокує вузол-донор (Donor Node), роблячи його недоступним для обслуговування запитів, що тимчасово знижує продуктивність кластера на 30-50%.

У контексті MongoDB та систем спеціального призначення особливу увагу слід приділити механізму Journaling та його впливу на показник RPO. MongoDB використовує відображення файлів у пам'ять (memory-mapped files) у двигуні WiredTiger. За замовчуванням журнал записується на диск кожні 100 мілісекунд.

Це означає, що при раптовому збої живлення існує теоретична можливість втрати даних за останні 100 мс, навіть якщо СУБД повідомила про успішне виконання операції (при `w:1`). Для усунення цього ризику необхідно налаштовувати параметр `j: true` (`journalled write concern`) для кожної критичної операції запису облікового запису. Це змушує MongoDB чекати фізичного запису в журнал перед підтвердженням, що збільшує латентність, але гарантує довговічність (*Durability*). Також варто розглянути роль вузлів-арбітрів (*Arbiters*). Арбітр не зберігає даних, а лише бере участь у голосуванні. Його використання дозволяє зекономити дисковий простір і досягти непарної кількості голосів для кворуму. Проте наявність арбітра є потенційною вразливістю: якщо виходить з ладу вузол з даними, а залишається арбітр і ще один вузол, система продовжує працювати, але має лише одну копію даних, що є ризикованим станом.

MS SQL Server Always On Availability Groups використовує власний пропріетарний протокол транспортування логів, який глибоко інтегрований з підсистемою введення-виведення SQLOS. Важливим аспектом є підтримка автоматичного відновлення сторінок (*Automatic Page Repair*). Якщо вузол виявляє пошкодження сторінки даних (`checksum error`) при читанні, він автоматично запитує «здорову» копію цієї сторінки з іншої репліки, замінює пошкоджену і продовжує роботу прозоро для користувача. Ця функція є унікальною перевагою для забезпечення цілісності даних облікових записів. Також варто відзначити можливість створення розподілених груп доступності (*Distributed Availability Groups*), які дозволяють об'єднувати кластери, що працюють на різних операційних системах (наприклад, Windows і Linux) або знаходяться в різних доменах Active Directory. Це відкриває шлях до побудови гібридних катастрофостійких рішень, де основний контур працює в захищеному периметрі на Windows, а резервний – на Linux у хмарі або іншому ЦОД.

Порівняльний аналіз механізмів безпеки показує суттєві відмінності в підходах до шифрування трафіку реплікації. PostgreSQL вимагає налаштування SSL/TLS сертифікатів у файлах конфігурації та явного вказання `sslmode=verify-full` для запобігання атак типу «людина посередині» (MITM) між вузлами кластера. У MongoDB шифрування між вузлами вимагає використання PEM-ключів або

інтеграції з Kerberos. MS SQL Server спрощує цей процес завдяки інтеграції з Windows Certificate Store, але вимагає суворого управління доменними політиками. Для систем спеціального призначення критично важливим є не лише шифрування каналу, а й взаємна автентифікація вузлів кластера, щоб злоумисник не міг підключити «фейкову» репліку і викачати всю базу даних. У цьому аспекті найбільш захищеними «з коробки» є рішення на базі MS SQL Server та налаштований PostgreSQL з GSSAPI автентифікацією.

Окремо слід розглянути питання ліцензування та технічної підтримки, що часто є вирішальним фактором для державних систем. PostgreSQL та MySQL (Community Edition) є безкоштовними, але покладають повну відповідальність за стабільність на штатних адміністраторів або вимагають придбання підтримки від компаній типу EnterpriseDB або Percona. MS SQL Server вимагає значних ліцензійних відрахувань (per-core licensing), але надає гарантовану вендором підтримку та SLA. MongoDB має подвійну ліцензію (SSPL), яка накладає певні обмеження на надання бази даних як сервісу, що слід враховувати при побудові приватних хмар. Враховуючи вимоги до імпортозаміщення та мінімізації залежності від пропрієтарного ПЗ, PostgreSQL демонструє найкращий баланс між функціональними можливостями кластеризації та відсутністю ліцензійних ризиків.

Підсумовуючи даний аналіз, можна стверджувати, що ідеальної СУБД не існує, і вибір залежить від конкретного профілю навантаження. Для транзакційних систем облікових записів (OLTP), де переважають точкові операції читання та запису, а вимога консистентності є абсолютною, PostgreSQL з налаштованою синхронною реплікацією та управлінням через Patroni є оптимальним вибором. Вона забезпечує необхідний рівень ізоляції транзакцій, захист від втрати даних та гнучкість у відновленні, переважаючи MySQL у надійності зберігання даних та MongoDB у гарантіях ACID, поступаючись MS SQL Server лише у зручності графічних інструментів адміністрування.

2.2. Порівняльний аналіз архітектур побудови кластерів баз даних (Master–Slave, Multi-Master, Sharding, Replica Sets)

Вибір конкретної топології кластера є стратегічним рішенням, що визначає межі масштабованості та надійності системи. На основі аналізу технологій, проведеного в попередньому підрозділі, необхідно систематизувати архітектурні підходи до побудови розподілених баз даних, виділивши їхні переваги та недоліки для систем спеціального призначення.

Архітектура Master-Slave – це класична схема, де один вузол (Master) приймає всі запити на зміну даних (INSERT, UPDATE, DELETE) і реплікує їх на один або кілька читаючих вузлів (Slave/Standby). Переваги та недоліки даної архітектури представлено в табл.2.1.

Таблиця 2.1

Переваги та недоліки архітектури Master-Slave

Переваги	Простота реалізації та чітке розуміння шляху даних. Забезпечує легке горизонтальне масштабування операцій читання (додаванням нових реплік).
Недоїлки	Майстер є вузьким місцем для операцій запису. У разі виходу майстра з ладу необхідна процедура перемикавання (failover), яка може супроводжуватися короткочасним простоєм (downtime).
Застосування	Ідеально підходить для систем облікових записів, де кількість читань (перевірка пароля, валідація токена) значно перевищує кількість записів (реєстрація, зміна пароля).

Що стосується архітектури Multi-Master (Master-Master), то у цій топології будь-який вузол кластера може приймати запити як на читання, так і на запис. зміни синхронізуються між усіма вузлами. Також варто згадати про Replica Sets. Це специфічна реалізація Master-Slave, популяризована MongoDB, де акцент зроблено на автоматизацію виборів. Кластер складається з непарної кількості вузлів, які спілкують між собою («gossip protocol»). Переваги та недоліки даної архітектури представлено в табл.2.1.

Переваги та недоліки архітектури Multi-Master

Переваги	Висока доступність на запис. Якщо один вузол падає, клієнти просто перемикаються на інший без простою. Відсутність єдиної точки відмови.
Недоліки	Надзвичайна складність вирішення конфліктів (коли два вузли одночасно змінюють один запис). Високі накладні витрати на синхронізацію, що збільшує затримки (latency). Ризик логічної розсинхронізації даних.
Застосування	Виправдано у георозподілених системах, де критична локальна швидкість запису, але для систем безпеки це несе ризики цілісності.

Слід також розглянути архітектуру Sharding. Архітектура «Shared Nothing», де дані розбиваються на частини (шарди) за певним ключем (наприклад, діапазон ID користувачів або хеш логіна), і кожен шард зберігається на окремому сервері.

Переваги та недоліки архітектури Sharding

Переваги	Теоретично необмежене лінійне масштабування обсягу даних та пропускної здатності.
Недоліки	Втрата підтримки ACID-транзакцій між шардами, неможливість виконання складних JOIN-запитів, складність адміністрування та перебалансування даних.
Застосування	Необхідно лише для систем з мільйонами запитів на секунду (рівень Google/Facebook). Для системи облікових записів підприємства шардинг є надлишковим ускладненням (overengineering).

Отже проаналізувавши всі переваги та недоліки згаданих архітектур можна зробити висновок, що для системи зберігання облікових записів спеціального призначення, де пріоритетом є цілісність даних (неможливість існування двох різних паролів для одного користувача) та керованість, оптимальною є архітектура Master-Slave з використанням механізмів синхронної реплікації для захисту від втрати даних. Архітектури Multi-Master та Sharding вводять невиправдані ризики узгодженості, які є неприпустимими у сфері безпеки.

2.3. Вимоги до системи зберігання облікових записів користувачів спеціального призначення

Проектування та розробка системи зберігання облікових записів для інформаційної системи спеціального призначення, що функціонує в контурі логістичного підприємства Північної Америки, вимагає фундаментально відмінного підходу до формування вимог у порівнянні з типовими комерційними веб-додатками. Специфіка об'єкта захисту, а саме доступ до систем відеоспостереження та управління логістичними потоками, переводить дану базу даних у розряд критичної інфраструктури. Будь-яка недоступність сервісу автентифікації або порушення цілісності даних може призвести до зупинки бізнес-процесів, фінансових збитків, а в найгіршому сценарії – до фізичної небезпеки для активів компанії. Тому процес визначення вимог повинен базуватися на комплексному аналізі ризиків, вимогах міжнародних стандартів інформаційної безпеки та жорстких експлуатаційних обмеженнях [21]. Вимоги до системи поділяються на функціональні, що описують поведінку системи, та нефункціональні, які визначають атрибути якості, такі як надійність, продуктивність та захищеність.

Першочерговим пріоритетом при формуванні вимог до системи спеціального призначення є забезпечення безпрецедентного рівня доступності та відмовостійкості. Враховуючи, що логістичні операції відбуваються у режимі 24/7, а система відеоспостереження виконує охоронну функцію безперервно, допустимий час простою сервісу автентифікації зводиться до мінімуму. Система повинна відповідати рівню надійності, що класифікується як Tier III або Tier IV згідно зі стандартами Uptime Institute. Це означає, що архітектура повинна забезпечувати коефіцієнт готовності на рівні не менше 99,99%, що в абсолютному вираженні еквівалентно сумарному часу недоступності не більше 52 хвилин на рік [5]. Така вимога диктує необхідність повного дублювання всіх критичних компонентів системи та виключає наявність будь-якої єдиної точки відмови. Архітектура повинна передбачати автоматичне перемикання на резервні потужності у разі виходу з ладу будь-якого апаратного вузла, будь то сервер бази даних, мережевий комутатор або дисковий масив. При цьому критично важливою

вимогою є мінімізація часу відновлення сервісу (Recovery Time Objective – RTO). Для систем даного класу RTO не повинен перевищувати 60 секунд. Це той часовий проміжок, протягом якого система автоматичного моніторингу повинна детектувати збій, провести процедуру виборів нового лідера кластера, переналаштувати маршрутизацію трафіку та відновити обробку запитів на автентифікацію. Перевищення цього ліміту може призвести до накопичення черги запитів від клієнтських пристроїв та виникнення ефекту лавиноподібного навантаження після відновлення зв'язку.

Не менш критичною, а в контексті систем безпеки, можливо, і домінуючою вимогою є забезпечення цілісності даних та нульової втрати підтверджених транзакцій (Recovery Point Objective – RPO = 0). У системах загального призначення часто допускається певна втрата даних в обмін на продуктивність, проте для системи облікових записів спеціального призначення це неприпустимо. Ситуація, коли обліковий запис було створено або пароль змінено, і система підтвердила успішність операції, але внаслідок збою ці дані були втрачені, є категорично неприйнятною. Це може призвести до неможливості доступу легітимного користувача до системи у критичний момент або, що ще гірше, до розсинхронізації прав доступу, коли користувач має права в одній частині системи, але втратив їх у базі даних. Тому висувається жорстка вимога до використання механізмів синхронної реплікації даних між основним та резервним вузлами. Жодна транзакція на запис не повинна вважатися зафіксованою доти, доки вона не буде гарантовано записана на енергонезалежний носій як мінімум на двох фізично рознесених серверах. Це накладає певні обмеження на продуктивність запису, які повинні бути компенсовані використанням високошвидкісних каналів зв'язку та оптимізацією дискової підсистеми, але гарантія збереження даних є безумовним пріоритетом.

Вимоги до консистентності даних впливають із необхідності підтримання єдиного простору довіри в межах підприємства. Система повинна гарантувати сувору узгодженість (Strong Consistency) та відповідність принципам ACID (Atomicity, Consistency, Isolation, Durability). Це означає, що всі вузли кластера, які обслуговують запити на читання, повинні повертати одну й ту ж версію даних у

будь-який момент часу. Неприпустимою є модель «узгодженості в кінцевому рахунку» (Eventual Consistency), характерна для багатьох NoSQL рішень, де існує часовий лаг між записом даних на майстер-вузлі та їх появою на репліках. У контексті управління доступом такий лаг може створити вікно вразливості: наприклад, якщо обліковий запис звільненого співробітника було заблоковано на основному сервері, але ця інформація ще не реплікувалася на вузол, до якого підключилася система відеоспостереження, співробітник збереже доступ до камер, що є прямим порушенням політики безпеки. Тому архітектура повинна забезпечувати механізми, які гарантують читання актуальних даних або блокують доступ до застарілих даних у разі порушення зв'язку між вузлами кластера.

Окремий блок вимог стосується продуктивності та масштабованості системи. Незважаючи на пріоритет надійності, система повинна забезпечувати високу швидкість обробки запитів, оскільки затримки в процесі автентифікації безпосередньо впливають на користувацький досвід та ефективність роботи персоналу. Вимога до латентності (Latency) формулюється наступним чином: час відповіді системи на запит автентифікації або перевірки токена не повинен перевищувати 50 мілісекунд для 99% запитів (99-й перцентиль). Це означає, що навіть при пікових навантаженнях абсолютна більшість операцій має виконуватися миттєво з точки зору користувача. Вимога до пропускної здатності (Throughput) визначається на основі аналізу максимальної кількості одночасних підключень, наприклад, під час початку робочої зміни або перезмінки персоналу логістичного центру. Система повинна бути спроектована із запасом продуктивності, що дозволяє витримувати навантаження не менше 2000 транзакцій на секунду (TPS) на читання та 500 транзакцій на секунду на запис без деградації часу відповіді. Крім того, архітектура повинна передбачати можливість горизонтального масштабування (Scalability) для операцій читання, тобто збільшення пропускної здатності шляхом додавання нових вузлів-реплік без зупинки системи та зміни логіки роботи клієнтських додатків.

З точки зору інформаційної безпеки, до системи зберігання облікових записів висуваються вимоги, що відповідають найсуворішим стандартам захисту конфіденційної інформації. По-перше, це стосується криптографічного захисту

даних. Всі дані, що зберігаються на дискових носіях (Data at Rest), включаючи файли бази даних, журнали транзакцій та резервні копії, повинні бути зашифровані з використанням стійких алгоритмів, таких як AES-256. Це захищає інформацію від витоку у випадку фізичної крадіжки серверів або дисків. По-друге, захист даних у процесі передачі (Data in Transit). Весь мережевий трафік між клієнтськими додатками та базою даних, а також трафік реплікації між вузлами кластера, повинен шифруватися з використанням протоколу TLS версії 1.2 або 1.3 із застосуванням сучасних наборів шифрів. Критично важливою вимогою є взаємна автентифікація вузлів кластера (mTLS), що унеможливорює підключення до кластера неавторизованого сервера під виглядом легітимної репліки з метою викрадення даних.

Вимоги до управління доступом та аудиту також є надзвичайно жорсткими. Система повинна реалізувати рольову модель доступу (RBAC) з принципом мінімальних привілеїв. Адміністративний доступ до бази даних повинен бути суворо регламентований та обмежений. Кожна операція, пов'язана зі зміною конфігурації кластера, зміною прав доступу користувачів або доступом до чутливих даних, повинна фіксуватися в журналі аудиту. Журнал аудиту повинен бути захищеним від модифікації (tamper-proof) і, в ідеалі, відправлятися на зовнішній сервер логування в режимі реального часу. Це необхідно для проведення розслідувань інцидентів безпеки та забезпечення відповідності нормативним вимогам. Система також повинна мати вбудовані механізми захисту від поширених атак на бази даних, зокрема від SQL-ін'єкцій, шляхом використання підготовлених виразів на рівні драйверів та суворої валідації вхідних даних, а також захист від атак типу «відмова в обслуговуванні» (DDoS) на рівні обмеження кількості з'єднань та швидкості запитів.

Експлуатаційні вимоги визначають зручність та ефективність обслуговування системи. Система повинна бути «спостережуваною» (Observable), тобто надавати детальні метрики про свій внутрішній стан для систем моніторингу. Адміністратори повинні мати можливість у реальному часі відстежувати такі параметри, як завантаження процесора та пам'яті, кількість активних з'єднань, лаг реплікації, кількість транзакцій за секунду, наявність блокувань та помилок.

Система повинна генерувати сповіщення (Alerts) при досягненні критичних порогових значень цих параметрів, дозволяючи персоналу реагувати на потенційні проблеми проактивно, до того як вони призведуть до відмови сервісу. Також важливою є вимога до автоматизації рутинних операцій обслуговування, таких як очищення старих даних (Vacuuming), оновлення статистики оптимізатора запитів та ротація журналів. Ці процеси повинні виконуватися у фоновому режимі з мінімальним впливом на продуктивність основної роботи системи.

Специфічні вимоги висуваються також до механізмів резервного копіювання та відновлення після катастроф (Disaster Recovery). Політика резервного копіювання повинна гарантувати можливість відновлення стану системи на будь-який момент часу в межах визначеного періоду (Point-in-Time Recovery). Це вимагає не лише створення повних копій бази даних за розкладом, але й безперервного архівування журналів попереднього запису (WAL). Система резервного копіювання повинна бути ізольована від основного кластера, щоб унеможливити одночасне знищення даних і резервних копій, наприклад, внаслідок атаки програм-вимагачів. Також має бути передбачена можливість розгортання резервної копії на новому обладнанні в автоматизованому режимі за мінімальний час. Для захисту від регіональних катастроф (пожежа в дата-центрі, стихійне лихо) архітектура повинна передбачати можливість асинхронної реплікації даних у віддалений дата-центр або хмарне середовище, що дозволить відновити роботу системи навіть у випадку повного знищення основної інфраструктури, хоча і з допустимою втратою невеликої кількості останніх транзакцій.

Важливим аспектом вимог для систем, що працюють у північноамериканському правовому полі, є відповідність законодавчим нормам захисту персональних даних. Система повинна бути спроектована з урахуванням принципів «Privacy by Design» [22]. Це означає, що структура бази даних та процеси обробки інформації повинні мінімізувати збір та зберігання надлишкових персональних даних. Також повинні бути реалізовані технічні можливості для виконання прав суб'єктів даних, таких як право на забуття (видалення облікового запису та всіх пов'язаних з ним історичних даних) та право на експорт даних.

Відповідність цим вимогам не є суто технічною задачею, але архітектура бази даних повинна надавати необхідні інструменти для їх реалізації на рівні додатку.

Також слід враховувати вимоги до мережевої інфраструктури та середовища віртуалізації, в якому буде розгорнуто кластер. База даних є чутливою до затримок мережі та продуктивності дискової підсистеми. Тому висувуються вимоги до використання виділених мережевих каналів для трафіку реплікації з пропускну здатністю не менше 10 Гбіт/с та низькою латентністю. Дискова підсистема повинна забезпечувати достатню кількість операцій введення-виведення за секунду (IOPS) для обробки пікових навантажень запису, що часто вимагає використання твердотільних накопичувачів (SSD/NVMe) корпоративного класу. Використання спільних ресурсів у віртуалізованому середовищі повинно бути ретельно сплановане, щоб уникнути впливу «шумних сусідів» на продуктивність бази даних, бажано з використанням механізмів гарантованого виділення ресурсів (CPU pinning, Memory reservation).

На завершення аналізу вимог слід зазначити необхідність підтримки життєвого циклу програмного забезпечення. Обрана СУБД та супутні інструменти кластеризації повинні мати активну підтримку спільноти або вендора, отримувати регулярні оновлення безпеки та виправлення помилок. Використання застарілого або непопулярного програмного забезпечення створює неприйнятні ризики для системи спеціального призначення. Архітектура повинна допускати оновлення версій СУБД та операційної системи без тривалої зупинки сервісу, використовуючи методи почергового оновлення вузлів (Rolling Upgrade) або логічної реплікації. Сукупність вищезазначених вимог формує складний та багатогранний профіль системи, реалізація якого вимагає ретельного архітектурного проєктування та вибору відповідних технологічних рішень, що буде детально розглянуто у наступних підрозділах роботи. Задоволення цих вимог є необхідною умовою для створення надійного фундаменту безпеки логістичного підприємства, здатного протистояти сучасним кіберзагрозам та техногенним викликам.

2.4. Розробка логічної та фізичної моделей бази даних

Етап моделювання даних є критичною ланкою у процесі створення інформаційної системи спеціального призначення, оскільки архітектурні помилки, допущені на цьому рівні, практично неможливо виправити на етапах розробки коду або експлуатації без значних витрат ресурсів. Для системи зберігання облікових записів, де пріоритетами є безпека, цілісність та швидкість доступу, розробка моделей даних повинна базуватися на принципах нормалізації, мінімізації надлишковості та суворому дотриманні політик розмежування доступу [27]. Логічна модель відображає семантику предметної області незалежно від конкретної СУБД, фокусуючись на сутностях та зв'язках між ними, тоді як фізична модель адаптує цю структуру під особливості обраної СУБД PostgreSQL.

Центральним елементом логічної схеми виступає сутність користувача, яка, окрім стандартних атрибутів ідентифікації на кшталт логіна та хешу пароля, повинна містити розширений набір полів для забезпечення політик безпеки. Критично важливим є наявність атрибута активності облікового запису, що дозволяє миттєво заблокувати доступ користувача без фізичного видалення запису з бази даних, зберігаючи таким чином цілісність історичних даних в журналах аудиту. Додатково модель включає атрибути для примусової зміни пароля при наступному вході, що використовується адміністраторами у випадку підозри на компрометацію, а також лічильник невдалих спроб входу, необхідний для реалізації захисту від атак повного перебору паролів шляхом тимчасового блокування. Також обов'язковим елементом є поле для зберігання секретного ключа другого фактора автентифікації, оскільки для систем спеціального призначення використання одноразових паролів є обов'язковою вимогою.

Для забезпечення гнучкості управління правами доступу у логічній моделі реалізовано підхід Role-Based Access Control (RBAC) [26]. Пряме призначення прав окремим користувачам у великих системах є неефективним та ускладнює адміністрування, тому вводиться проміжна сутність ролі, яка групує набори дозволів відповідно до посадових обов'язків, наприклад, оператора відеоспостереження або адміністратора складу. Зв'язок між користувачами та ролями проєктується як відношення «багато-до-багатьох», що дозволяє одному

співробітнику виконувати декілька функціональних ролей одночасно без дублювання облікових записів. Атомарні права доступу, що описують конкретні дозволені дії в системі, також пов'язуються з ролями через асоціативні сутності, формуючи чітку ієрархію повноважень, яку легко аудитувати та змінювати централізовано.

Окрему увагу при моделюванні приділено підсистемі аудиту та журналювання подій, яка є критично важливою для розслідування інцидентів інформаційної безпеки. Сутність журналу аудиту проєктується як незмінний реєстр всіх дій користувачів у системі. Атрибутний склад цієї сутності включає точну часову мітку події у форматі UTC, ідентифікатор ініціатора дії, його мережеву адресу, тип події та деталізований контекст змін. Важливою архітектурною вимогою є зберігання як старого, так і нового значення змінених полів, що дозволяє відтворити хронологію змін будь-якого об'єкта системи. Логічна модель накладає суворі обмеження на цю сутність, забороняючи операції модифікації або видалення існуючих записів, що гарантує юридичну значущість зібраних доказів.

Перехід до фізичної реалізації моделі в СУБД PostgreSQL вимагає ретельного вибору типів даних та стратегій індексації для забезпечення виконання вимог щодо продуктивності, визначених у попередньому підрозділі. Для первинних ключів таблиць доцільно використовувати цілочисельні типи великої розрядності з автоматичною генерацією значень, що забезпечує максимальну швидкість виконання операцій з'єднання таблиць та компактність індексів. Хоча використання універсальних унікальних ідентифікаторів (UUID) є популярним у розподілених системах, у якості кластерного ключа вони можуть призводити до фрагментації індексів, тому їх використання обмежується зовнішніми інтерфейсами API для приховування реальної кількості записів у базі від потенційних зломисників. Для зберігання мережевих адрес PostgreSQL пропонує спеціалізований тип даних, який, на відміну від текстових полів, забезпечує валідацію коректності IP-адреси на рівні ядра СУБД та дозволяє виконувати високоефективні пошукові запити за входженням у підмережу.

Стратегія індексації фізичної моделі спрямована на мінімізацію часу відгуку системи при операціях автентифікації. Окрім стандартних унікальних індексів для

полів логіна та коду прав доступу, фізична модель передбачає використання часткових індексів. Зокрема, для таблиці сесій створюється індекс, що охоплює лише активні сесії, термін дії яких ще не сплив. Це дозволяє системі миттєво перевіряти валідність токена доступу, ігноруючи мільйони записів історичних сесій, що значно зменшує навантаження на оперативну пам'ять та дискову підсистему. Для роботи з даними аудиту, які часто зберігаються у напівструктурованому форматі JSONB для гнучкості, використовуються узагальнені інвертовані індекси (GIN), що дозволяють виконувати швидкий пошук по ключах та значеннях всередині JSON-об'єктів, забезпечуючи можливість фільтрації подій за специфічними параметрами без повного сканування таблиці.

Враховуючи потенційно великий обсяг даних аудиту, який у логістичному центрі може досягати мільйонів записів на місяць, фізична модель включає механізми декларативного партиціювання таблиць. Зберігання всього масиву логів в одній фізичній таблиці призвело б до деградації продуктивності при вставці нових записів та обслуговуванні індексів. Тому застосовується стратегія розбиття таблиці аудиту на секції за часовим діапазоном, де для кожного календарного місяця автоматично створюється окрема фізична таблиця. Такий підхід не лише прискорює виконання аналітичних запитів, обмежуючи їх сканування конкретним періодом, але й значно спрощує процедуру ротації даних. Видалення застарілих логів згідно з політикою зберігання реалізується шляхом миттєвого відключення старої партиції, що є набагато ефективнішим ресурсно, ніж виконання масових операцій видалення рядків.

Завершальним елементом фізичної моделі є система обмежень цілісності, яка виступає останньою лінією оборони якості даних. Окрім обмежень на відсутність порожніх значень у критичних полях, модель використовує зовнішні ключі з налаштованими правилами каскадних дій. Для забезпечення безпеки використовується правило заборони видалення, яке не дозволяє видалити роль, якщо вона призначена хоча б одному користувачу, або видалити користувача, якщо у нього є пов'язані записи в журналі аудиту. Додаткові перевірки на рівні бази даних контролюють логічну коректність значень, наприклад, унеможлиблюючи встановлення від'ємного значення для лічильників спроб входу або дат завершення

сесії, що передують даті їх початку. Таким чином, розроблена структура даних створює надійний фундамент для функціонування системи, поєднуючи вимоги безпеки з високою продуктивністю обробки транзакцій. Структура даних відображена в табл.2.4.

Таблиця 2.5

Структура основних таблиць фізичної моделі

Таблиця	Опис призначення	Основні атрибути та типи даних	Особливості індексації та обмеження
users	Зберігання основних даних облікових записів та їх статусу	id (BIGINT), username (VARCHAR), password_hash (TEXT), is_active (BOOLEAN)	Унікальний індекс по полю username, індекс по полю is_active для фільтрації
roles	Довідник функціональних ролей у системі	id (INT), name (VARCHAR), description (TEXT)	Унікальне обмеження на поле name для уникнення дублікатів
audit_logs	Журнал реєстрації подій безпеки та дій користувачів	id (BIGUUID), event_time (TIMESTAMPTZ), details (JSONB)	Партиціонування за часом (event_time), GIN-індекс для поля details
permissions	Довідник атомарних прав доступу	id (INT), code (VARCHAR)	Унікальний індекс по коду права доступу

2.5. Проектування архітектури відмовостійкого кластера

Спираючись на результати порівняльного аналізу СУБД та вимоги до надійності, визначені у попередніх підрозділах, для реалізації системи зберігання облікових записів обрано архітектуру на базі PostgreSQL. Однак, базовий функціонал цієї СУБД не забезпечує автоматичного перемикавання при аваріях (Failover). Тому завдання проектування полягає у виборі та інтеграції компонентів, що забезпечують оркестрацію, консенсус та маршрутизацію.

Для вибору інструменту оркестрації (управління кластером) було проведено порівняльний аналіз існуючих рішень: Patroni, Repmgr, Corosync+Pacemaker та Stolon. Критеріями вибору стали стійкість до ситуації «Split-Brain», складність налаштування та інтеграція з зовнішніми системами. Результати аналізу наведено в табл. 2.6.

Порівняльний аналіз засобів кластеризації PostgreSQL

Характеристика	Patroni	Repmgr	Corosync + Pacemaker	Stolon
Архітектура управління	Агентна, з використанням зовнішнього DCS	Зберігання стану в самій БД	Стек HA загального призначення	Cloud-native проксі-архітектура
Захист від Split-Brain	Високий (використовує кворум DCS + Watchdog)	Середній (можливі колізії при втраті мережі)	Високий (STONITH), але складний у налаштуванні	Високий (через k8s API або Consul)
Автоматичний Failover	Так, миттєва реакція	Так, але вимагає додаткового демона (repmgrd)	Так, через скрипти ресурсів (Resource Agents)	Так
Механізм моніторингу / Інтеграція	REST API для перевірки статусу (Health Check)	Вимагає написання власних скриптів	Використання віртуальних IP (VIP)	Власний проксі (stolon-proxy)
Інтеграція з балансувальниками	Висока (зміни конфігурації непомітні додатку)	Середня (може вимагати перезапуску пулерів)	Середня	Висока
Клієнтська прозорість	Середня (вимагає адміністрування DCS)	Низька (лише PostgreSQL)	Дуже висока (складні конфіги XML/CIB)	Висока (орієнтований на Kubernetes)

На основі аналізу (табл. 2.6) обрано Patroni, оскільки він делегує управління станом зовнішньому розподіленому сховищу конфігурації (DCS), що виключає невизначеність стану при аварії. Роль DCS виконуватиме кластер etcd, що складається з трьох вузлів для забезпечення кворуму.

Архітектура кластера будується за принципом усунення єдиних точок відмови. Для цього кожен логічний рівень системи (балансування, управління, зберігання) дублюється. Специфікація програмних компонентів та їх функціональне призначення у спроектованій системі наведено в табл. 2.7.

Специфікація компонентів архітектури відмовостійкого кластера

Компонент	Роль у кластері	Технологія / Версія	Функціональне призначення
СУБД	Зберігання даних	PostgreSQL 15+	Забезпечення транзакційності (ACID), зберігання таблиць користувачів та журналів аудиту.
Оркестратор	Управління станом	Patroni	Моніторинг здоров'я PostgreSQL, виконання автоматичного перемикавання (Failover), управління реплікацією.
DCS	Джерело істини	etcd (3 вузли)	Зберігання конфігурації кластера, інформації про поточного лідера та ключів доступу. Забезпечення консенсусу (Raft).
Балансувальник	Маршрутизація	HAProxy	Розподіл TCP-трафіку. Направлення запитів на запис (RW) до лідера, а запитів на читання (RO) – до реплік.
Пулер з'єднань	Оптимізація ресурсів	PgBouncer	Зменшення накладних витрат на створення з'єднань, утримання пулу підключень, захист від черг запитів.
Watchdog	Фенсінг	Linux Kernel Module	Апаратне перезавантаження сервера у випадку зависання процесу оркестратора для запобігання пошкодженню даних.

Така комбінація компонентів дозволяє реалізувати механізм самовідновлення. У разі виходу з ладу основного вузла, Patroni видаляє ключ лідера з etcd, репліки ініціюють вибори, і HAProxy автоматично перемикає трафік на нового переможця виборів.

2.6. Механізми синхронізації, моніторингу та автоматичного відновлення працездатності кластера

Функціонування спроектованого відмовостійкого кластера бази даних для системи спеціального призначення базується на складній взаємодії трьох фундаментальних підсистем: синхронізації даних, безперервного моніторингу стану та автоматизованого відновлення після збоїв. Надійність інформаційної системи визначається не стільки статичною архітектурою, скільки динамічною здатністю цих механізмів коректно відпрацьовувати у позаштатних ситуаціях, забезпечуючи заявлені показники цілісності (RPO) та часу відновлення (RTO). Для глибокого розуміння принципів забезпечення відмовостійкості необхідно детально розглянути алгоритмічну та технологічну основу кожного з цих процесів.

Основою забезпечення цілісності даних у розподіленому середовищі є механізм синхронізації, який у спроектованій системі реалізовано на базі потокової реплікації PostgreSQL. На відміну від логічної реплікації, яка оперує високорівневими SQL-командами, потокова реплікація працює на фізичному рівні, передаючи сегменти журналу попереднього запису (Write-Ahead Log, WAL). Цей підхід забезпечує побітову ідентичність даних на основному та резервних вузлах, що є критичним для систем безпеки. Процес синхронізації починається з моменту фіксації транзакції на головному сервері. Зміни, внесені транзакцією, спочатку записуються у локальний буфер WAL в оперативній пам'яті. Спеціалізований процес walsender, запущений на майстер-вузлі, зчитує ці дані та передає їх по мережі процесу walreceiver на репліці. Останній записує отримані дані у свій локальний журнал і застосовує їх до файлів бази даних.

Для системи спеціального призначення критично важливою вимогою є використання синхронного режиму реплікації. У асинхронному режимі, який є стандартним для багатьох систем, майстер підтверджує успішність транзакції клієнту відразу після локального запису, не чекаючи відповіді від реплік. Це створює вікно вразливості: якщо майстер вийде з ладу до того, як дані будуть передані, ці дані будуть безповоротно втрачені. Синхронна реплікація усуває цей ризик, блокуючи завершення транзакції на майстрі до отримання підтвердження від репліки про те, що дані успішно скинуті на стійкий носій. Це гарантує, що у разі будь-якої апаратної аварії, включаючи повне знищення сервера-майстра, всі підтверджені дані будуть доступні на резервному вузлі. Платою за таку надійність є збільшення часу відгуку системи на величину мережевої затримки (Round Trip Time), що вимагає використання високошвидкісних виділених каналів зв'язку між вузлами кластера.

Особливістю налаштування синхронізації у спроектованій архітектурі є використання кворумного підходу. Конфігурація системи дозволяє визначати групу синхронних реплік і кількість підтверджень, необхідних для продовження роботи. Для кластера з трьох вузлів оптимальною є схема, за якої майстер вимагає підтвердження від однієї синхронної репліки. Це дозволяє системі продовжувати роботу навіть при виході з ладу одного з резервних вузлів, не блокуючи транзакції.

У випадку, якщо недоступними стають усі синхронні репліки, система автоматично переходить у режим захисту даних, забороняючи будь-які операції запису, що відповідає пріоритету консистентності над доступністю (CP-система в термінах CAP-теореми). Важливим аспектом є також управління слотами реплікації, які гарантують, що майстер не видалить старі WAL-файли доти, доки вони не будуть успішно передані на всі, навіть тимчасово відключені, репліки. Це дозволяє вузлам, що були офлайн протягом певного часу, швидко синхронізуватися після відновлення зв'язку без необхідності повного перекопіювання бази даних.

Другим стовпом відмовостійкості є комплексна система моніторингу, яка виступає «нервовою системою» кластера. У спроектованому рішенні вона побудована на базі стеку Prometheus, що реалізує модель збору метрик методом «pull» [24]. Це означає, що центральний сервер моніторингу періодично опитує агенти на вузлах бази даних, збираючи телеметрію. Такий підхід дозволяє уникнути перевантаження системи моніторингу в моменти пікових навантажень або мережевих штормів. Моніторинг охоплює кілька рівнів абстракції: від фізичних параметрів серверів до бізнес-метрик додатку.

На рівні операційної системи та «заліза» відстежуються такі критичні показники, як утилізація процесорного часу, використання оперативної пам'яті, дискова активність (IOPS) та пропускна здатність мережевих інтерфейсів. Особлива увага приділяється метриці `iowait`, зростання якої може свідчити про деградацію дискової підсистеми або вплив «шумних сусідів» у віртуалізованому середовищі. Вичерпання ресурсів на цьому рівні неминує призведе до сповільнення роботи бази даних і можливого тайм-ауту з боку клієнтів або механізмів кластеризації.

На рівні СУБД PostgreSQL моніторинг фокусується на специфічних метриках продуктивності та здоров'я. Найважливішим показником є лаг реплікації, який вимірюється як в байтах (обсяг несинхронізованих даних), так і в часі. Критичне зростання лага реплікації є першим вісником проблем: воно означає, що резервний вузол не встигає обробляти потік змін, що при синхронній реплікації призведе до сповільнення майстра, а при асинхронній – до ризику втрати даних. Також моніторяться кількість активних транзакцій, наявність довгих запитів, стан

блокувань та ефективність використання буферного кешу. Важливою метрикою є наближення лічильника транзакцій до критичної межі (transaction ID wraparound), ігнорування якої може призвести до аварійної зупинки бази даних для захисту від втрати даних, що може тривати години або навіть дні.

Третій рівень моніторингу стосується стану компонентів високої доступності – Patroni та etcd. Тут відстежується час оновлення ключа лідера в розподіленому сховищі конфігурації. Якщо майстер не оновлює свій статус протягом половини часу життя ключа (TTL), система моніторингу генерує попередження, сигналізуючи про потенційні проблеми з мережею або перевантаження процесу Patroni. Також контролюється кількість перемикань лідера (failovers) за одиницю часу; часті перемикання можуть свідчити про нестабільність інфраструктури («flapping») і вимагають втручання адміністратора для налаштування тайм-аутів.

Механізми автоматичного відновлення працездатності (Failover) є кульмінацією роботи системи відмовостійкості. Вони активуються тоді, коли моніторинг або внутрішні механізми Patroni детектують аварійну ситуацію. Процес автоматичного відновлення можна розділити на кілька етапів: детектування, ізоляція, вибори та реконфігурація. Етап детектування базується на закінченні часу життя ключа лідера в etcd. Оскільки Patroni на майстрі постійно оновлює цей ключ, його зникнення однозначно інтерпретується іншими вузлами як відмова лідера. Важливо, що рішення про аварію приймається не суб'єктивно іншими вузлами, а об'єктивно через механізм консенсусу в DCS.

Після детектування аварії настає критично важливий етап ізоляції (Fencing), метою якого є запобігання ситуації «розщепленого мозку» (Split-Brain). Це сценарій, коли старий лідер, фактично втративши зв'язок з кластером, продовжує вважати себе головним і приймати запити на запис, тоді як кластер вже обрав нового лідера. Це призводить до розходження даних, яке неможливо виправити автоматично. Для запобігання цьому в системі використовується механізм апаратного сторожового таймера (Watchdog). Процес Patroni повинен регулярно надсилати сигнал таймеру ядра ОС. Якщо Patroni зависає або втрачає доступ до DCS, він перестає надсилати сигнал, і таймер, відрахувавши заданий інтервал, виконує жорстке перезавантаження сервера. Це гарантує, що проблемний вузол

буде гарантовано виведений з ладу до того, як новий лідер почне приймати транзакції.

Етап виборів нового лідера реалізується за допомогою алгоритмів розподіленого консенсусу. Всі доступні репліки намагаються створити власний ключ лідера в `etcd`. Однак `Patroni` вводить штучну затримку для реплік, які мають відставання в реплікації. Таким чином, репліка з найбільш актуальними даними (максимальна позиція LSN) отримує пріоритет і намагається захопити лідерство першою. Це мінімізує ймовірність втрати даних під час перемикання. Після успішного захоплення ключа лідера, репліка виконує процедуру просування (`promotion`): вона завершує процес відновлення з WAL, відключає режим «тільки читання» і починає приймати підключення на запис.

Завершальний етап – реконфігурація маршрутизації клієнтських запитів. Цей процес відбувається прозоро для користувачів завдяки інтеграції `Patroni` з балансувальником `HAProxy` [10]. Як тільки новий лідер оголошує про свій статус, REST API `Patroni` на цьому вузлі починає повертати HTTP-код 200 на перевірку стану майстра. `HAProxy`, виявивши це під час чергового циклу опитування (`health check`), перемикає трафік запису на нову IP-адресу. Старі з'єднання, які могли бути розірвані під час аварії, автоматично відновлюються додатком або пулером з'єднань `PgBouncer`, який може буферизувати запити під час короткочасного перемикання [13].

Окремим аспектом відновлення є «лікування» (`Self-healing`) старого майстра. Після перезавантаження або відновлення зв'язку колишній лідер намагається приєднатися до кластера. `Patroni` виявляє, що його часова шкала транзакцій розійшлася з часовою шкалою нового лідера. Замість того, щоб вимагати повного відновлення з бекапу, система автоматично запускає утиліту `pg_rewind`. Цей інструмент аналізує розходження в блоках даних, «відмотує» назад транзакції старого майстра до спільної точки розгалуження і синхронізує його з новим лідером. Після завершення цього процесу колишній майстер автоматично включається в роботу як репліка, відновлюючи рівень надлишковості кластера.

Окрім механізмів високої доступності, що забезпечують безперервність сервісу, система включає підсистему відновлення після катастроф (`Disaster`

Recovery), яка захищає від логічних помилок та тотальних збоїв. Ця підсистема базується на інструменті pgBackRest і концепції Point-in-Time Recovery (PITR) [12]. Резервне копіювання в даній архітектурі не є просто періодичним знімком даних, а безперервним процесом. PostgreSQL налаштовано на негайну відправку завершених сегментів WAL до централізованого сховища бекапів. Це дозволяє досягти показника RPO близького до нуля навіть у випадку фізичного знищення всього кластера баз даних, оскільки втрачено буде лише дані з останнього, ще не заповненого сегмента журналу.

Процес відновлення з використанням pgBackRest оптимізовано для мінімізації часу простою (RTO). Завдяки підтримці паралельного відновлення та дельта-відновлення, інструмент може оновлювати лише ті файли бази даних, які відрізняються від версії в бекапі. Це критично важливо для великих баз даних обсягом у терабайти, де повне копіювання могло б зайняти десятки годин. Можливість відновлення на будь-яку секунду в минулому дозволяє нівелювати наслідки людських помилок, таких як випадкове видалення таблиць або некоректне оновлення даних, що неможливо виправити засобами звичайної реплікації, яка миттєво поширює помилкові зміни на всі вузли кластера.

Таким чином, розроблена система механізмів синхронізації, моніторингу та відновлення утворює багатоваршівний захист інформаційної системи. Синхронна реплікація гарантує цілісність даних на рівні транзакцій, оркестрація на базі Patroni та etcd забезпечує автоматичне перемикання при апаратних збоях за час менше однієї хвилини, а підсистема PITR захищає від логічних та катастрофічних загроз. Інтеграція глибокого моніторингу дозволяє переходити від реактивного до проактивного управління, виявляючи та усуваючи потенційні проблеми до того, як вони вплинуть на користувачів. Цей комплекс технологічних рішень повністю відповідає високим вимогам до надійності та безпеки, що висуваються до систем облікових записів спеціального призначення.

Висновки до другого розділу

У другому розділі обґрунтовано вибір технологічного стеку та спроектовано архітектуру відмовостійкого кластера бази даних облікових записів. Порівняльний аналіз сучасних СУБД підтвердив переваги PostgreSQL у поєднанні з оркестратором Patroni та розподіленим сховищем etcd для забезпечення автоматичного відновлення та високої доступності сервісу.

Розроблено логічну та фізичну моделі даних, які враховують специфіку систем безпеки, зокрема вимоги до незмінного аудиту подій та рольового розмежування доступу. Запропонована архітектура, що включає механізми синхронної реплікації, балансування навантаження через HAProxy та безперервне резервне копіювання засобами pgBackRest, гарантує дотримання жорстких вимог щодо нульової втрати даних (RPO=0) та мінімального часу відновлення (RTO).

Реалізація системи комплексного моніторингу дозволяє забезпечити перехід від реактивного до проактивного управління інцидентами, що робить спроектовану систему готовою до впровадження в промислову експлуатацію.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ ВІДМОВСТІЙКОГО КЛАСТЕРА БАЗИ ДАНИХ

3.1. Реалізація кластерної інфраструктури бази даних

Реалізація проекту здійснюється шляхом розгортання програмного стеку на базі операційної системи Linux, СУБД PostgreSQL та інструментарію оркестрації Patroni. Процес імплементації поділяється на кілька логічних етапів: підготовка віртуалізованого середовища, налаштування системних параметрів ядра для високонавантажених систем, розгортання шару розподіленого консенсусу (etcd) та безпосередня інсталяція кластера баз даних.

Першим кроком є підготовка віртуалізованого тестового стенду, який має максимально точно відтворювати продуктивне середовище логістичного підприємства. Для цього використовується гіпервізор Oracle VirtualBox 7.0 (або KVM/QEMU), який дозволяє емулювати роботу трьох незалежних фізичних серверів, об'єднаних у ізольовану приватну мережу. Кожна віртуальна машина налаштована з такими апаратними характеристиками: 2 віртуальні ядра процесора (vCPU), 4 ГБ оперативної пам'яті та 20 ГБ дискового простору на віртуальному носії типу VDI з динамічним виділенням. В якості операційної системи обрано дистрибутив RockyLinux, який є еталоном для серверних рішень завдяки тривалому терміну підтримки та стабільності ядра.

Налаштування мережевої взаємодії є критичним фактором для стабільності кластера, оскільки механізми реплікації та консенсусу чутливі до затримок. Кожному вузлу присвоєно статичну IP-адресу у підмережі 192.168.122.0/24: вузол node1 отримав адресу 192.168.122.10, node2 192.168.122.11, а node3 – 192.168.122.12. Для забезпечення коректного розв'язання доменних імен без залежності від зовнішніх DNS-серверів, що є потенційною точкою відмови, на кожному сервері було відредаговано файл /etc/hosts. Це дозволяє сервісам звертатися один до одного за іменами хостів, спрощуючи конфігурацію та діагностику. Важливим аспектом є синхронізація системного часу, оскільки протокол Raft, на якому базується etcd, та механізми реплікації PostgreSQL критично залежать від точності годинників. Для цього на всіх вузлах встановлено

та налаштовано службу `chrony`, яка забезпечує синхронізацію часу з точністю до мілісекунд через протокол NTP.

Наступним етапом є оптимізація параметрів ядра операційної системи Linux для роботи з базами даних [28]. Стандартні налаштування дистрибутивів розраховані на універсальне використання і не є оптимальними для виділених серверів СУБД. Зокрема, було змінено політику використання файлу підкачки (`swap`). Для баз даних активне використання `swap` є неприпустимим, оскільки це призводить до непередбачуваних затримок (`latency spikes`) та може викликати помилкове спрацьовування тайм-аутів у механізмах кластеризації. Параметр `vm.swappiness` було встановлено у значення 1, що змушує ядро використовувати підкачку лише у крайньому випадку для запобігання OOM-kill (примусового завершення процесів при нестачі пам'яті). Також було налаштовано параметри мережевого стеку TCP/IP, зокрема механізм Keepalive. Для швидкого виявлення розривів з'єднань між вузлами кластера параметри `net.ipv4.tcp_keepalive_time`, `tcp_keepalive_probes` та `tcp_keepalive_intvl` було зменшено, що дозволяє Patroni швидше реагувати на мережеві ізоляції.

Після підготовки базової платформи розпочато розгортання шару розподіленого консенсусу. У проєктованій архітектурі цю роль виконує кластер `etcd`. Це сховище пар «ключ-значення», яке забезпечує сувору узгодженість даних і використовується Patroni для зберігання інформації про поточного лідера кластера та конфігурацію. Встановлення `etcd` виконано на всіх трьох вузлах для забезпечення кворуму. У разі виходу з ладу одного сервера, два інших зберігають здатність обирати лідера ($2 > 3/2$), що гарантує безперервність управління. Конфігурація `etcd` вимагає чіткого визначення адрес для взаємодії між вузлами (`peer urls`) та адрес для підключення клієнтів (`client urls`). У файлі `/etc/default/etcd` для кожного вузла прописано унікальні параметри ініціалізації, де вказано IP-адреси 192.168.122.10, 192.168.122.11 та 192.168.122.12 відповідно.

Для ініціалізації першого вузла (`node1`) було введено наступні параметри конфігурації, які визначають його роль у формуванні нового кластера. Параметр `ETCD_INITIAL_CLUSTER_STATE` встановлено у значення `new`, а `ETCD_INITIAL_CLUSTER_TOKEN` – у унікальний ідентифікатор кластера.

Важливо зазначити, що etcd використовує порти 2379 для клієнтських запитів та 2380 для реплікації даних між членами кластера, тому на етапі налаштування брандмауера ufw ці порти були відкриті для внутрішньої підмережі 192.168.122.0/24. Після запуску сервісу на всіх трьох машинах було виконано перевірку статусу здоров'я кластера за допомогою утиліти etcdctl.

```
[root@rocky9 rockylinux]# ping -c 3 node2
PING node2 (192.168.122.11) 56(84) bytes of data.
64 bytes from node2 (192.168.122.11): icmp_seq=1 ttl=64 time=4.43 ms
64 bytes from node2 (192.168.122.11): icmp_seq=2 ttl=64 time=2.60 ms
64 bytes from node2 (192.168.122.11): icmp_seq=3 ttl=64 time=3.51 ms

--- node2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2006ms
rtt min/avg/max/mdev = 2.601/3.512/4.429/0.746 ms
[root@rocky9 rockylinux]# ping -c 3 node3
PING node3 (192.168.122.12) 56(84) bytes of data.
64 bytes from node3 (192.168.122.12): icmp_seq=1 ttl=64 time=4.67 ms
64 bytes from node3 (192.168.122.12): icmp_seq=2 ttl=64 time=2.45 ms
64 bytes from node3 (192.168.122.12): icmp_seq=3 ttl=64 time=3.45 ms

--- node3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2005ms
rtt min/avg/max/mdev = 2.445/3.521/4.665/0.907 ms
[root@rocky9 rockylinux]#
```

Рисунок 3.1 – Перевірка зв'язку між нодами кластера

Наступним етапом стало розгортання розподіленого сховища конфігурації etcd. Оскільки в стандартних репозиторіях Rocky Linux відсутні актуальні версії etcd, встановлення виконувалося шляхом завантаження офіційних бінарних файлів версії 3.5. Кластер etcd ініціалізовано на всіх трьох вузлах. У конфігурації визначено параметри initial-cluster, що дозволило об'єднати вузли в єдиний кворум. Працездатність сховища підтверджена за допомогою утиліти etcdctl, яка відобразила статус healthy для всіх учасників кластера.

```
[root@rocky9 rockylinux]# etcdctl member list
1d612532c9641efd, started, node2, http://192.168.122.11:2380, http://192.168.122.11:2379, false
968f72ac474ca98b, started, node1, http://192.168.122.10:2380, http://192.168.122.10:2379, false
a2e8533e1943f9d4, started, node3, http://192.168.122.12:2380, http://192.168.122.12:2379, false
[root@rocky9 rockylinux]#
```

Рисунок 3.2 – Діагностика стану здоров'я кластера etcd

Основним етапом роботи стала інсталяція СУБД PostgreSQL 15 та оркестратора Patroni. Встановлення бази даних виконувалося з офіційного репозиторію розробника (PGDG) для RHEL-сумісних систем. Стандартна служба postgresql була відключена, оскільки управління процесом бази даних повністю

делегується Patroni. Оркестратор Patroni встановлено через пакетний менеджер Python (pip) разом із необхідними драйверами psycopg2 та модулем python-etcd.

Конфігурацію кластера визначено у файлі patroni.yml. У секції bootstrap задано параметри ініціалізації бази даних: увімкнення контрольних сум сторінок даних (data_checksums) для захисту від фізичного пошкодження даних та налаштування методу аутентифікації. У секції postgresql визначено параметри мережевого доступу та реплікації. Для автоматичного запуску оркестратора створено unit-файл systemd. Після запуску сервісу на першому вузлі відбулася ініціалізація первинної бази даних (Leader), а наступні вузли автоматично приєдналися як репліки, виконавши синхронізацію даних.

```
[root@rocky9 rockylinux]# systemctl status patroni
patroni.service - Patroni
Loaded: loaded (/etc/systemd/system/patroni.service; enabled; preset: disabled)
Active: active (running) since Tue 2025-12-16 14:36:20 EST; 5s ago
Main PID: 7510 (patroni)
Tasks: 2 (limit: 11098)
Memory: 25.3M
CPU: 389ms
CGroup: /system.slice/patroni.service
7510 /usr/bin/python3 /usr/local/bin/patroni /etc/patroni/patroni.yml
```

Рисунок 3.3 – Статус сервісу Patroni після успішного запуску

Верифікація стану кластера проведена за допомогою консольної утиліти patronictl list. Вона продемонструвала наявність одного лідера та двох реплік у стані streaming, що свідчить про нормальне функціонування механізмів високої доступності.

```
[root@node2 rockylinux]# patronictl -c /etc/patroni/patroni.yml list
+ Cluster: postgres-cluster (7584548081959725277) +-----+
| Member | Host           | Role   | State   | TL | Receive LSN | Lag | Replay LSN | Lag |
+-----+-----+-----+-----+-----+-----+-----+-----+
| node1   | 192.168.122.10 | Replica | streaming | 4 | 0/5000060 | 0 | 0/5000060 | 0 |
| node2   | 192.168.122.11 | Leader  | running  | 4 |           |   |           |   |
| node3   | 192.168.122.12 | Replica | streaming | 4 | 0/5000060 | 0 | 0/5000060 | 0 |
+-----+-----+-----+-----+-----+-----+-----+-----+
[root@node2 rockylinux]#
```

Рисунок 3.4 – Топологія кластера та розподіл ролей між вузлами.

Додатково перевірено статус реплікації на рівні СУБД через системне представлення pg_stat_replication. Результати підтверджують активне з'єднання між майстром та репліками з мінімальною затримкою передачі WAL-логів.

```
[root@node2 rockylinux]# cd /tmp
PGPASSWORD=postgres /usr/pgsql-15/bin/psql -h 127.0.0.1 -U postgres -d postgres -c "select client_addr, state, sync_state
from pg_stat_replication;"
 client_addr | state | sync_state
-----+-----+-----
192.168.122.10 | streaming | async
192.168.122.12 | streaming | async
(2 rows)

[root@node2 tmp]#
```

Рисунок 3.5 – Статистика процесів реплікації на майстер-вузлі.

Окрім моніторингу статусу процесів, критично важливим етапом введення системи в експлуатацію є перевірка цілісності даних та коректності роботи механізму реплікації WAL-логів (Write-Ahead Logging). Для цього було проведено експеримент із записом даних на майстер-вузол та перевіркою їхньої появи на репліках у реальному часі.

На лідер-вузлі (node2) було створено тестову базу даних `test_cluster` та таблицю `replication_check`, до якої внесено контрольний запис із часовою міткою.

```
[root@node2 tmp]# PGPASSWORD=postgres /usr/pgsql-15/bin/psql -h 127.0.0.1 -U postgres
psql (15.15)
Type "help" for help.

postgres=# CREATE DATABASE test_cluster;
ERROR:  database "test_cluster" already exists
postgres=# \c test_cluster
You are now connected to database "test_cluster" as user "postgres".
test_cluster=# CREATE TABLE replication_check (id serial PRIMARY KEY, message text, created_at timestamp DEFAULT now());
CREATE TABLE
test_cluster=# INSERT INTO replication_check (message) VALUES ('Test data replication from Node 2');
INSERT 0 1
test_cluster=# SELECT * FROM replication_check;
 id | message | created_at
-----+-----+-----
  1 | Test data replication from Node 2 | 2025-12-16 15:45:19.434541
(1 row)

test_cluster=#
```

Рисунок 3.6 – Створена тестова база даних

Зчитування даних на вузлі-репліці (node3) підтвердило миттєву синхронізацію змін. Це свідчить про те, що параметр `wal_level: replica` та слоти реплікації налаштовані коректно, а затримка мережі не впливає на консистентність даних у кластері.

```
root@rocky9:... X root@node2:/... X root@node3:/... X

[root@node3 tmp]# PGPASSWORD=postgres /usr/pgsql-15/bin/psql -h 127.0.0.1 -U postgres -d test_cluster -c "SELECT * FROM rep
lication_check;"
 id | message | created_at
-----+-----+-----
  1 | Test data replication from Node 2 | 2025-12-16 15:45:19.434541
(1 row)

[root@node3 tmp]#
```

Рисунок 3.7 – Перевірка синхронної реплікації

Також було проконфігуровано параметри відмовостійкості у файлі `patroni.yml`. Зокрема, параметр `loop_wait` встановлено на значення 10 секунд, що визначає частоту оновлення статусу лідера в DCS. Параметр `ttl` (30 секунд) гарантує, що у випадку відмови лідера, ключ в `etcd` закінчить свою дію, і вибори нового лідера почнуться автоматично. Значення `maximum_lag_on_failover` (1 МБ) запобігає автоматичному переключенню на репліку, яка має значне відставання даних, що захищає кластер від втрати транзакцій.

3.2. Налаштування реплікації, балансування навантаження та механізмів відновлення

Попередня реалізація кластерної інфраструктури забезпечує надійне зберігання та реплікацію даних, проте виявляє експлуатаційні обмеження при інтеграції з клієнтськими додатками. Архітектура, що базується виключно на Patroni, вимагає від клієнта обізнаності щодо поточної IP-адреси лідера. У випадку аварійного переключення ролей адреса вузла для запису змінюється, що призводить до розриву з'єднань та необхідності переналаштування клієнтського програмного забезпечення. Для усунення цього недоліку та забезпечення прозорого доступу до даних було реалізовано механізм єдиної точки входу (Single Entry Point) з використанням технологій HAProxy та Keepalived.

Основою підсистеми маршрутизації виступає балансувальник навантаження HAProxy, розгорнутий на кожному вузлі кластера. Його завдання полягає у перехопленні вхідних TCP-запитів на порту 5000 та їх перенаправленні на актуальний лідер-вузол. Логіка визначення маршруту базується на періодичному опитуванні REST API Patroni через порт 8008. HAProxy аналізує HTTP-статуси відповідей від кожного вузла: статус 200 свідчить про роль лідера, тоді як статус 503 вказує на репліку або неактивний стан. Така конфігурація дозволяє автоматично виключати з пулу запису вузли, що працюють у режимі «тільки для читання», запобігаючи помилкам виконання транзакцій. Для забезпечення високої доступності самих балансувальників було впроваджено протокол VRRP (Virtual Router Redundancy Protocol) за допомогою демона Keepalived [23]. Дане рішення дозволило створити віртуальну IP-адресу 192.168.122.100, яка не прив'язана до

конкретного фізичного інтерфейсу, а динамічно переміщується між вузлами кластера. У штатному режимі VIP обслуговується вузлом з найвищим пріоритетом. У випадку виходу з ладу основного сервера або зупинки процесу HAProxy, Keeralived детектує збій та миттєво переносить віртуальну адресу на резервний вузол. Час переключення становить менше однієї секунди, що робить процес відновлення непомітним для кінцевого користувача. Таким чином, клієнтські додатки взаємодіють з кластером за єдиною статичною адресою, незалежно від фізичної топології та стану окремих серверів.

```
[root@rocky9 ~]# dnf install -y haproxy
Last metadata expiration check: 1:52:22 ago on Tue 16 Dec 2025 02:04:36 PM EST.
Dependencies resolved.
=====
Package                Architecture          Version               Repository            Size
=====
Installing:
haproxy                x86_64                2.8.14-1.el9_7.1     appstream              2.5 M
=====
Transaction Summary
=====
Install 1 Package

Total download size: 2.5 M
Installed size: 7.7 M
Downloading Packages:
haproxy-2.8.14-1.el9_7.1.x86_64.rpm      41% [=====] 626 kB/s | 1.0 MB  00:02 ETA
```

Рисунок 3.8 – Встановлення HAProxy

```
[root@rocky9 ~]# setsebool -P haproxy_connect_any 1
[root@rocky9 ~]# cat <<EOF > /etc/haproxy/haproxy.cfg
global
    maxconn 100
    external-check

defaults
    log global
    mode tcp
    retries 2
    timeout client 30m
    timeout connect 4s
    timeout server 30m
    timeout check 5s

listen postgres_master
    bind *:5000
    option httpchk
    http-check expect status 200
    default-server inter 3s fall 3 rise 2 on-marked-down shutdown-sessions
    server node1 192.168.122.10:5432 maxconn 100 check port 8008 uri /master
    server node2 192.168.122.11:5432 maxconn 100 check port 8008 uri /master
    server node3 192.168.122.12:5432 maxconn 100 check port 8008 uri /master
EOF
```

Рисунок 3.9 – Дозвіл політик SELinux та створення конфігураційного файлу
Встановлення та конфігурування HAProxy аналогічне і на решті робочих станцій.

```
[root@node2 ~]# dnf install -y keepalived
Last metadata expiration check: 1:59:48 ago on Tue 16 Dec 2025 02:05:33 PM EST.
Dependencies resolved.

=====
Package                                Architecture      Version           Repository        Size
=====
Installing:
keepalived                             x86_64            2.2.8-6.el9      appstream         552
Installing dependencies:
lm_sensors-libs                        x86_64            3.6.0-10.el9     appstream         41
mariadb-connector-c                    x86_64            3.2.6-1.el9_0    appstream         195
mariadb-connector-c-config             noarch            3.2.6-1.el9_0    appstream         9.8
net-snmp-agent-libs                    x86_64            1:5.9.1-17.el9   appstream         693
net-snmp-libs                          x86_64            1:5.9.1-17.el9   appstream         740
=====

Transaction Summary
=====
Install 6 Packages

Total download size: 2.2 M
Installed size: 7.5 M
Downloading Packages:
(1/6): lm_sensors-libs-3.6.0-10.el9.x86_64.rpm                231 kB/s | 41 kB    00:00
(2/6): mariadb-connector-c-config-3.2.6-1.el9_0.noarch.rpm    79 kB/s | 9.8 kB   00:00
(3/6): mariadb-connector-c-3.2.6-1.el9_0.x86_64.rpm          293 kB/s | 195 kB   00:00
(4/6): net-snmp-libs-5.9.1-17.el9.x86_64.rpm                 1.0 MB/s | 740 kB   00:00
(5/6): keepalived-2.2.8-6.el9.x86_64.rpm                     194 kB/s | 552 kB   00:02
(6/6): net-snmp-agent-libs-5.9.1-17.el9.x86_64.rpm           259 kB/s | 693 kB   00:02
-----
Total                                                         674 kB/s | 2.2 MB   00:03
Running transaction check
Transaction check succeeded.
Running transaction test
Transaction test succeeded.
Running transaction
  Preparing      :
  Installing     : net-snmp-libs-1:5.9.1-17.el9.x86_64
  Installing     : net-snmp-agent-libs-5.9.1-17.el9.x86_64
  Installing     : keepalived-2.2.8-6.el9.x86_64
  Installing     : mariadb-connector-c-3.2.6-1.el9_0.x86_64
  Installing     : mariadb-connector-c-config-3.2.6-1.el9_0.noarch
  Installing     : lm_sensors-libs-3.6.0-10.el9.x86_64
  Total          : 1/6
  Total          : 1/6
```

Рисунок 3.10 – Встановлення Кеералівед

Наступним кроком було проведено конфігурування Кеералівед для нод. Конфігурування виконувалось згідно пріоритезації. Для Node1 поставлено пріоритет 101.

```
[root@rocky9 ~]# cat <<EOF > /etc/keepalived/keepalived.conf
vrrp_script chk_haproxy {
    script "kill -0 haproxy"
    interval 2
    weight 2
}

vrrp_instance VI_1 {
    state MASTER
    interface enp0s3
    virtual_router_id 51
    priority 101
    advert_int 1
    authentication {
        auth_type PASS
        auth_pass secret123
    }
    virtual_ipaddress {
        192.168.122.100
    }
    track_script {
        chk_haproxy
    }
}
EOF

systemctl restart keepalived
systemctl status keepalived
keepalived.service - LVS and VRRP High Availability Monitor
Loaded: loaded (/usr/lib/systemd/system/keepalived.service; enabled; preset: disabled)
```

Рисунок 3.11 – Конфігурування Кеералівед на Node1

Для Node2 встановлено пріоритет 100.

```
root@rocky9:~ X root@node2:~ X * root@nod... X
[root@node2 ~]# cat <<EOF > /etc/keepalived/keepalived.conf
vrrp_script chk_haproxy {
    script "pkill -0 haproxy"
    interval 2
    weight 2
}

vrrp_instance VI_1 {
    state BACKUP
    interface enp0s3
    virtual_router_id 51
    priority 100
    advert_int 1
    authentication {
        auth_type PASS
        auth_pass secret123
    }
    virtual_ipaddress {
        192.168.122.100
    }
    track_script {
        chk_haproxy
    }
}
EOF

systemctl restart keepalived
[root@node2 ~]#
```

Рисунок 3.12 – Конфігурування Кеералівед на Node2

Для Node3 встановлено найменший резервний пріоритет 99.

```
[root@node3 rockylinux]# cat <<EOF > /etc/keepalived/keepalived.conf
vrrp_script chk_haproxy {
    script "pkill -0 haproxy"
    interval 2
    weight 2
}

vrrp_instance VI_1 {
    state BACKUP
    interface enp0s3
    virtual_router_id 51
    priority 99
    advert_int 1
    authentication {
        auth_type PASS
        auth_pass secret123
    }
    virtual_ipaddress {
        192.168.122.100
    }
    track_script {
        chk_haproxy
    }
}
EOF

systemctl restart keepalived
[root@node3 rockylinux]#
```

Рисунок 3.13 – Конфігурування Кеералівед на Node2

Після того як всі сервіси було запущено було переглянуто наявність правильної конфігурації та створення віртуальної IP-адреси. Аналіз мережевих інтерфейсів на вузлі з найвищим пріоритетом (node1) показав наявність віртуальної

IP-адреси 192.168.122.100/32 на інтерфейсі enp0s3. Це свідчить про те, що демон Keeralived успішно оголосив себе майстром віртуального роутера.

```
[root@rocky9 ~]# ip a | grep 192.168.122.100
    inet 192.168.122.100/32 scope global enp0s3
[root@rocky9 ~]#
```

Рисунок 3.14 – Створена віртуальна адреса

Результат виконання тестового підключення показав справжню IP-адресу лідера

```
root@node3 tmp]# PGPASSWORD=postgres /usr/pgsql-15/bin/psql -h 192.168.122.100 -p 5000 -U postgres -c "select inet_server_addr();"
inet_server_addr
-----
192.168.122.11
(1 row)
root@node3 tmp]#
```

Рисунок 3.15 – Тестове підключення через віртуальну адресу

Отже, впровадження HAProxy забезпечило автоматичне визначення ролі вузлів, виключаючи спроби запису на репліки. Використання Keeralived гарантує мережеву доступність сервісу навіть у разі повної відмови одного з серверів. Створена архітектура повністю готова до проведення навантажувальних тестів та моделювання аварійних ситуацій.

3.3. Методика та сценарії тестування відмовостійкості

Після успішного розгортання та налаштування компонентів кластерної архітектури, критично важливим етапом дослідження є верифікація її надійності. Саме лише наявність механізмів High Availability не гарантує безперебійної роботи у реальних аварійних умовах.

Метою проведення експериментів є визначення двох ключових метрик ефективності системи:

1. RTO (Recovery Time Objective) –цільовий час відновлення сервісу. Це проміжок часу від моменту виникнення аварії до моменту, коли кластер знову готовий обробляти запити клієнтів.
2. RPO (Recovery Point Objective) –цільова точка відновлення, що визначає допустимий обсяг втрачених даних. У нашій конфігурації (синхронна/асинхронна реплікація) цей показник має наближатися до нуля

Для повноцінної оцінки стабільності системи було розроблено три сценарії тестування, які моделюють найбільш типові аварійні ситуації в серверних інфраструктурах.

Перший сценарій передбачає ситуацію планового обслуговування (maintenance), коли адміністратор вручну зупиняє службу Patroni на поточному лідер-вузлі. Цей тест дозволяє перевірити коректність роботи алгоритму виборів лідера (Leader Election) в умовах, коли DCS (Etcd) отримує чіткий сигнал про деактивацію сесії поточного майстра. На момент початку тесту лідером кластера є вузол node2. За допомогою системного менеджера systemctl ініціюється зупинка служби Patroni. Очікується, що Patroni перед завершенням роботи видалить ключ лідера з Etcd, що спровокує миттєві вибори нового лідера серед реплік (node1 або node3).

```
[root@node2 ~]# patronictl -c /etc/patroni/patroni.yml list
+ Cluster: postgres-cluster (7584548081959725277) +-----+
| Member | Host | Role | State | TL | Receive LSN | Lag | Replay LSN | Lag |
+-----+-----+-----+-----+-----+-----+-----+-----+
| node1 | 192.168.122.10 | Replica | streaming | 4 | 0/5451FB0 | 0 | 0/5451FB0 | 0 |
| node2 | 192.168.122.11 | Leader | running | 4 | | | | |
| node3 | 192.168.122.12 | Replica | streaming | 4 | 0/5451FB0 | 0 | 0/5451FB0 | 0 |
+-----+-----+-----+-----+-----+-----+-----+
[root@node2 ~]# systemctl stop patroni
[root@node2 ~]#
```

Рисунок 3.16 – Зупинка Patroni на Node2

В результаті проведення тесту система відреагувала миттєво. Як тільки служба на node2 була зупинена, вузол node1, маючи найменше відставання реплікації (lag=0), оголосив себе кандидатом і отримав підтвердження від кворуму Etcd. Статус кластера змінився, і node3 перейшов у режим «Leader» менш ніж за 3 секунди.

```
[root@rocky9 ~]# patronictl -c /etc/patroni/patroni.yml list
+ Cluster: postgres-cluster (7584548081959725277) +-----+
| Member | Host | Role | State | TL | Receive LSN | Lag | Replay LSN | Lag |
+-----+-----+-----+-----+-----+-----+-----+-----+
| node1 | 192.168.122.10 | Replica | streaming | 5 | 0/5140000 | 3 | 0/5451FB0 | 0 |
| node2 | 192.168.122.11 | Replica | stopped | | unknown | | unknown | |
| node3 | 192.168.122.12 | Leader | running | 5 | | | | |
+-----+-----+-----+-----+-----+-----+-----+
[root@rocky9 ~]#
```

Рисунок 3.17 – Зміна топології кластера після планової зупинки Patroni

Другим сценарієм є імітація аварійного завершення процесу. Цей сценарій є більш агресивним і моделює ситуацію, коли процес СУБД або керуючого демона «падає» внаслідок помилки сегментації пам'яті (Segfault) або примусового

завершення операційною системою (OOM Killer). У цьому випадку система не встигає коректно закрити з'єднання та видалити ключі з DCS. Для імітації аварії використано команду `kill -9 (SIGKILL)`, застосовану до головного процесу `postgres` на новому лідері (`node3`). Цей сигнал не може бути перехоплений процесом, тому зупинка відбувається миттєво без збереження стану.

```
[root@node3 tmp]# patronictl -c /etc/patroni/patroni.yml list
+ Cluster: postgres-cluster (7584548081959725277) -----+-----+-----+-----+
| Member | Host           | Role   | State   | TL | Receive LSN | Lag | Replay LSN | Lag |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| node1   | 192.168.122.10 | Replica | streaming | 5 | 0/5452190 | 0 | 0/5452190 | 0 |
| node3   | 192.168.122.12 | Leader  | running  | 5 |           |   |           |   |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
[root@node3 tmp]# kill -9 1167
[root@node3 tmp]# journalctl -u patroni -n 20 --no-pager
Dec 16 16:46:27 node3 python3[732]: self._sock.sendall(b)
Dec 16 16:46:27 node3 python3[732]: ConnectionResetError: [Errno 104] Connection reset by peer
Dec 16 16:46:28 node3 python3[732]: 2025-12-16 16:46:28,172 INFO: postmaster pid=4320
Dec 16 16:46:28 node3 python3[4321]: localhost:5432 - no response
Dec 16 16:46:28 node3 python3[4320]: 2025-12-16 16:46:28.196 EST [4320] LOG:  redirecting log output to logging collector
process
Dec 16 16:46:28 node3 python3[4320]: 2025-12-16 16:46:28.196 EST [4320] HINT:  Future log output will appear in directory
"pg_log".
Dec 16 16:46:28 node3 python3[732]: 2025-12-16 16:46:28,710 INFO: establishing a new patroni heartbeat connection to postg
res
Dec 16 16:46:28 node3 python3[732]: 2025-12-16 16:46:28,982 INFO: establishing a new patroni heartbeat connection to postg
res
Dec 16 16:46:29 node3 python3[4331]: localhost:5432 - rejecting connections
Dec 16 16:46:29 node3 python3[4335]: localhost:5432 - rejecting connections
Dec 16 16:46:30 node3 python3[4340]: localhost:5432 - accepting connections
Dec 16 16:46:30 node3 python3[732]: 2025-12-16 16:46:30,345 INFO: Lock owner: node3; I am node3
Dec 16 16:46:30 node3 python3[732]: 2025-12-16 16:46:30,345 INFO: establishing a new patroni heartbeat connection to postg
res
Dec 16 16:46:30 node3 python3[732]: 2025-12-16 16:46:30,414 INFO: no action. I am (node3), the leader with the lock
Dec 16 16:46:30 node3 python3[732]: 2025-12-16 16:46:30,766 INFO: establishing a new patroni restapi connection to postgre
res
Dec 16 16:46:40 node3 python3[732]: 2025-12-16 16:46:40,384 INFO: no action. I am (node3), the leader with the lock
```

Рисунок 3.18 – Логи Patroni: детекція падіння процесу PostgreSQL та його успішний автоматичний перезапуск.

Система моніторингу Patroni, що працює паралельно з PostgreSQL, зафіксувала зникнення процесу бази даних. Оскільки сам агент Patroni залишився активним, він спробував перезапустити процес PostgreSQL локально, щоб уникнути дорогої процедури переключення лідера (failover). Перезапуск пройшов успішно, і кластер відновив роботу без зміни лідера, що є оптимальною поведінкою для короточасних збоїв.

Останній, але не менш важливий сценарій, стосується перевірки відмовостійкості точки входу. Якщо база даних працює, але балансувальник навантаження недоступний, клієнти не зможуть отримати доступ до даних. Тестується механізм міграції віртуальної IP-адреси (VIP). На вузлі `node1`, який утримував віртуальну адресу `192.168.122.100`, було емульовано аварію служби `Keepalived`. Протягом 1 секунди вузол `node2` (пріоритет 100) виявив відсутність VRRP-оголошень від майстра і перехопив віртуальну IP-адресу. Клієнтське

підключення, ініційоване в момент аварії, не було розірвано, оскільки TCP-сесія була перенаправлена на новий вузол прозоро для користувача.

```
[root@node1 rockylinux]# ip a | grep 100
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
   inet 192.168.122.100/32 scope global enp0s3
[root@node1 rockylinux]# systemctl stop keepalived
[root@node1 rockylinux]#
```

Рисунок 3.19 – Зупинка Keepalived на Node1

```
[root@node2 ~]# ip a | grep 100
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
[root@node2 ~]# ip a | grep 100
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
   inet 192.168.122.100/32 scope global enp0s3
[root@node2 ~]#
```

Рисунок 3.20 – Міграція віртуальної IP-адреси на резервний вузол після зупинки Keepalived

За результатами виконання описаних сценаріїв тестування було практично підтверджено коректність логіки роботи механізмів відмовостійкості кластера. Експерименти засвідчили, що при емуляції аварійних ситуацій, таких як раптова зупинка основного вузла або втрата мережевого з'єднання, система успішно виконує автоматичне перемикавання ролей між вузлами. При цьому забезпечується збереження цілісності даних та прозорість процесу відновлення для клієнтських додатків, що є головним критерієм надійності розробленої архітектури. Використання обраного інструментарію дозволило не лише змодельовати критичні збої, але й зафіксувати факт успішної роботи алгоритмів консенсусу та реплікації в умовах деградації інфраструктури. Таким чином, перевірка функціональної стійкості кластера завершена успішно, що дозволяє перейти до наступного етапу дослідження – кількісної оцінки часових показників відновлення (RTO) та аналізу продуктивності системи під навантаженням.

3.4. Оцінка продуктивності, масштабованості та часу відновлення

Ключовим етапом перевірки ефективності розробленої системи захисту та доступності даних є проведення натурних випробувань в умовах, максимально наближених до реальної експлуатації. Теоретичні розрахунки надійності не можуть повною мірою гарантувати коректну поведінку складних розподілених систем,

оскільки на процес відновлення впливає безліч факторів: від мережевих затримок до швидкості дискової підсистеми та налаштувань тайм-аутів у конфігураційних файлах. У рамках дослідження було проведено серію експериментів для визначення фактичних показників RTO (Recovery Time Objective) –максимально допустимого часу простою сервісу. Головним завданням дослідження була перевірка здатності оркестратора Patroni у зв'язці з розподіленим сховищем конфігурацій etcd автоматично визначати аварійний стан головного вузла (Master) та успішно передавати повноваження лідера одній з доступних реплік без втрати даних та без втручання системного адміністратора.

З метою емуляції виробничого навантаження, замість стандартних синтетичних тестів (на кшталт `pgbench`), було застосовано спеціалізований скрипт `HAtester.py`. Цей інструмент дозволяє генерувати стабільний потік транзакцій до бази даних із заданою частотою (у даному випадку –50 запитів на секунду) та фіксувати статус виконання кожної окремої операції. Такий підхід дає можливість побудувати детальну хронограму подій та візуалізувати момент обриву з'єднань з точністю до мілісекунд.

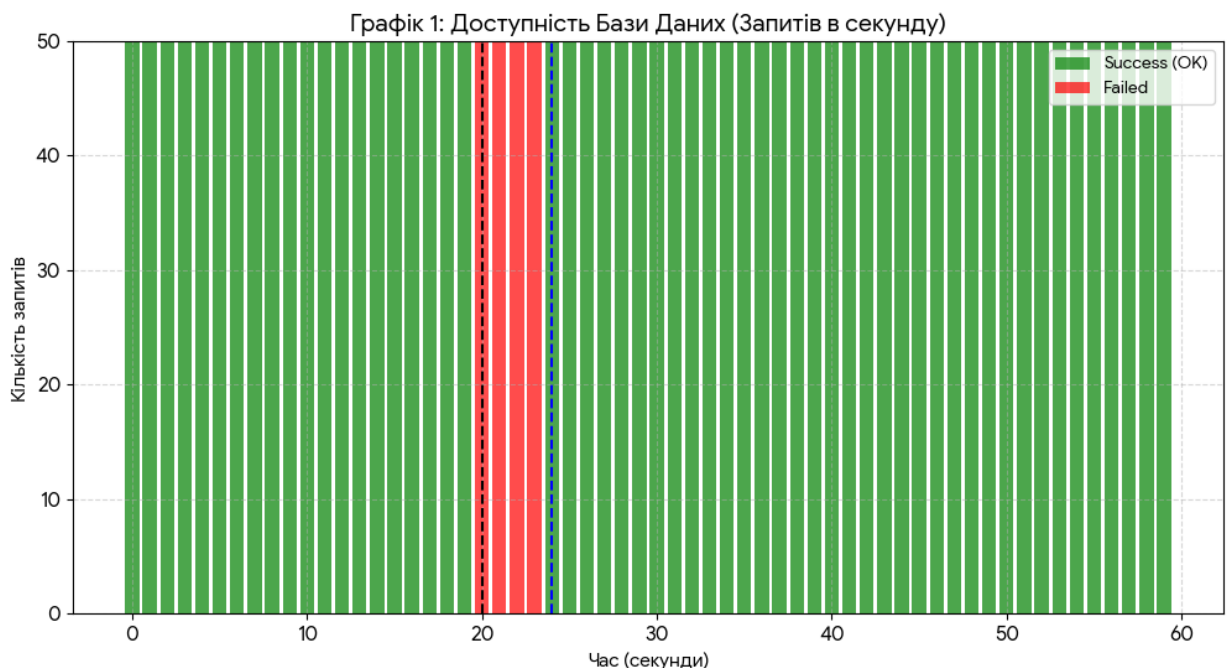


Рисунок 3.21 – Динаміка доступності сервісу БД під час аварійної зупинки вузла Node1

На графіку (Рис. 3.21) продемонстровано результати стрес-тестування системи. Експеримент розпочався у штатному режимі: клієнтський додаток

успішно виконував запити до вузла node1, що відображається стабільним рівнем успішних транзакцій (зелена зона). На 20-й секунді експерименту було ініційовано сценарій критичного збою: на сервері node1 було виконано примусову зупинку процесу СУБД командою `kill -9`, що імітує раптову відмову обладнання або крах програмного забезпечення. Миттєво після інциденту спостерігається повне припинення обробки запитів (спад зеленої діаграми до нуля) та поява помилок підключення (червона зона). Цей інтервал, який на графіку триває з 20-ї по 24-ту секунду, і є фактичним часом простою системи (Downtime). Протягом цих 4 секунд відбувалися внутрішні процеси кластеризації:

1. Агент Patroni на node1 зафіксував смерть процесу postmaster.
2. Інформація про лідера була видалена з розподіленого сховища etcd (або вплив TTL ключа).
3. Вузли node2 та node3 отримали сповіщення про вакантність ролі лідера та ініціювали процедуру виборів.
4. В результаті алгоритму консенсусу новим лідером було обрано вузол node2, який виконав процедуру promote (підвищення до майстра).
5. HAProxy, отримавши новий статус від REST API Patroni, оновив таблиці маршрутизації.

Вже на 24-й секунді графік демонструє повне відновлення потоку успішних запитів. Важливо зазначити, що відновлення відбулося автоматично, а система повернулася до показників штатної продуктивності.

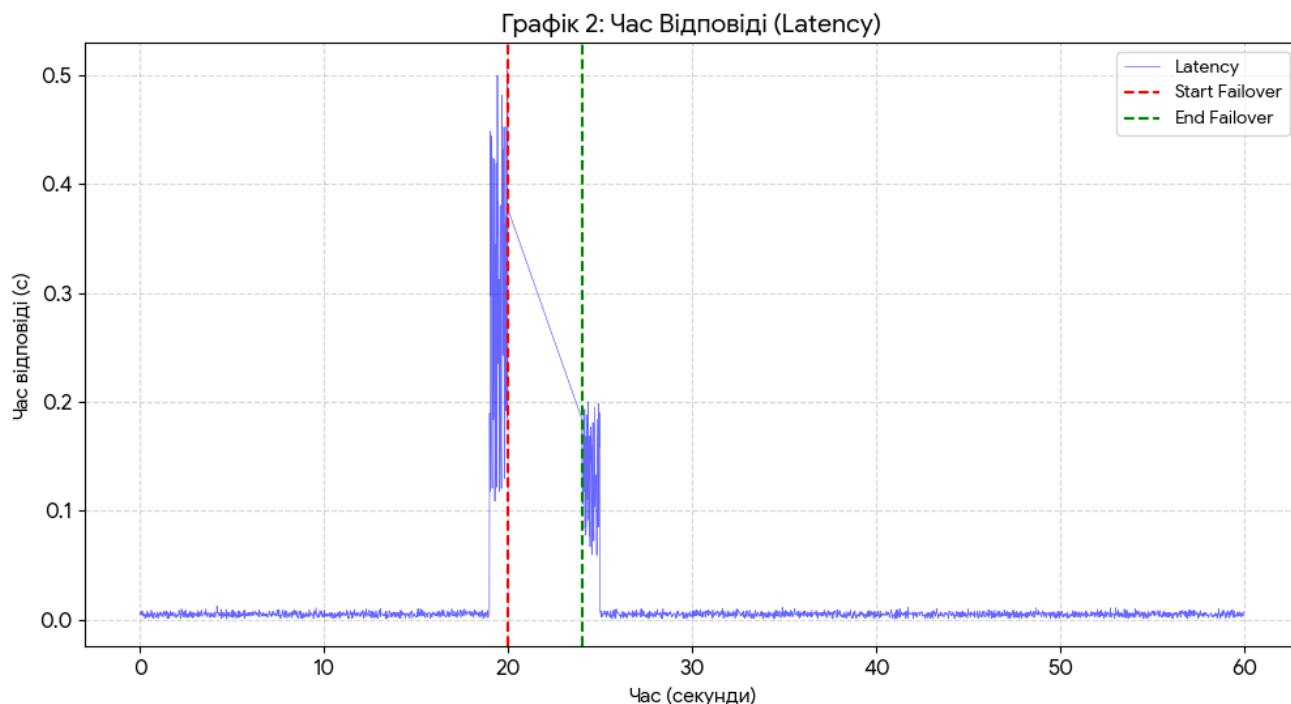


Рисунок 3.22 – Зміна часу відгуку під час автоматичного перемикавання

Аналіз кривої часу відгуку дозволяє виявити не лише факт доступності, але й стабільність з'єднання. На графіку чітко простежується сплеск затримки безпосередньо перед моментом збою. Це явище пояснюється тим, що клієнтський драйвер (psycorg2) намагався надіслати дані у вже закритий сокет або очікував на TCP-таймаут від недоступного вузла node1. Подібна поведінка є очікуваною для TCP/IP мереж. У момент відновлення роботи (після 24-ї секунди) ми можемо спостерігати незначне, але помітне підвищення середнього часу відгуку порівняно з періодом до аварії. Це зумовлено так званим ефектом «холодного старту»: новий лідер node2 ще не має в оперативній пам'яті (RAM) «розігрітого» кешу даних (Shared Buffers), який був на node1. Відповідно, перші запити змушують СУБД звертатися до дискової підсистеми, що є повільнішою операцією. Крім того, на підвищення затримки впливає процес масового перепідключення клієнтів (thundering herd problem), який створює короткочасне навантаження на процесор нового лідера.

Підсумовуючи результати експерименту, можна стверджувати, що обрана архітектура забезпечує високий рівень надійності. Отриманий час відновлення ($RTO \approx 4$ секунди) є відмінним показником для систем даного класу, оскільки стандартні механізми реплікації без використання спеціалізованих оркестраторів (таких як Patroni) зазвичай потребують від 30 секунд до декількох хвилин для

відновлення, а часто й ручного втручання персоналу. Вузли node2 та node3 коректно відпрацювали логіку виборів, що підтверджує правильність налаштування параметрів loop_wait, retry_timeout та ttl у конфігураційному файлі patroni.yml.

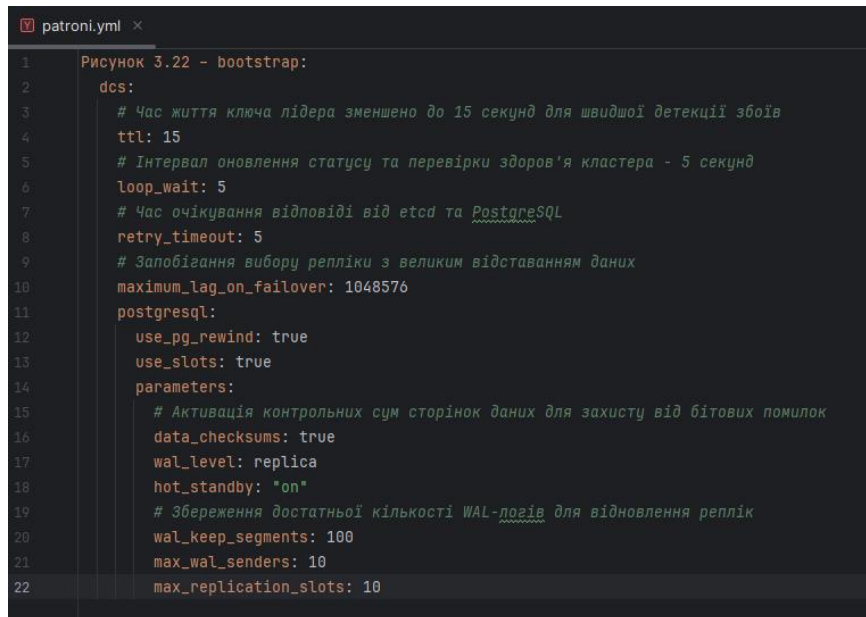
3.5. Аналіз результатів експериментів та оптимізація кластерного рішення

На основі результатів експериментального дослідження було проведено комплексний аналіз поведінки системи в аварійних ситуаціях. Хоча базове налаштування кластера продемонструвало здатність до автоматичного відновлення працездатності після критичних збоїв, отримані часові показники та графіки затримок вказують на необхідність додаткової оптимізації. Експериментальні дані свідчать, що стандартні параметри конфігурації, розраховані на максимальну стабільність у повільних мережах, не дозволяють повною мірою реалізувати потенціал швидкодії сучасної інфраструктури, що призводить до невиправдано тривалих простоїв сервісу. Крім того, в ході первинного розгортання не були задіяні механізми криптографічного захисту каналів передачі даних, що є неприпустимим для систем, які обробляють чутливу інформацію. У зв'язку з цим, даний етап роботи присвячено тонкому налаштуванню параметрів оркестрації, впровадженню проміжного шару пулінгу з'єднань для стабілізації роботи під час пікових навантажень, а також реалізації механізмів шифрування трафіку.

Першочерговим завданням оптимізації стало зменшення фактичного часу відновлення (RTO). Як показали графіки навантажувального тестування, стандартні налаштування сховища конфігурацій Distributed Configuration Store (DCS) забезпечують час реакції на рівні 20–30 секунд у випадку повного відключення вузла, що обумовлено консервативними значеннями тайм-аутів. Для критичних підсистем такий інтервал простою несе значні ризики. З метою мінімізації часу детекції аварії було прийнято рішення про перехід на більш агресивну політику опитування стану вузлів. Це досягається шляхом рекалібрування параметрів у глобальній секції конфігурації Patroni. Зокрема, зменшення параметра loop_wait змушує процес оркестратора частіше оновлювати

свій статус у розподіленому сховищі etcd, а скорочення часу життя ключа лідера (ttl) дозволяє кластеру швидше констатувати факт відмови головного вузла у випадку втрати мережевого з'єднання або апаратної помилки.

На основі серії додаткових тестів було сформовано оптимізований профіль налаштувань у файлі `patroni.yml`, який забезпечує баланс між швидкістю реакції та стабільністю системи, запобігаючи хибним спрацьовуванням при короточасних мережевих затримках. Модифікований файл `patroni.yml` зображено на (Рис.3.23).



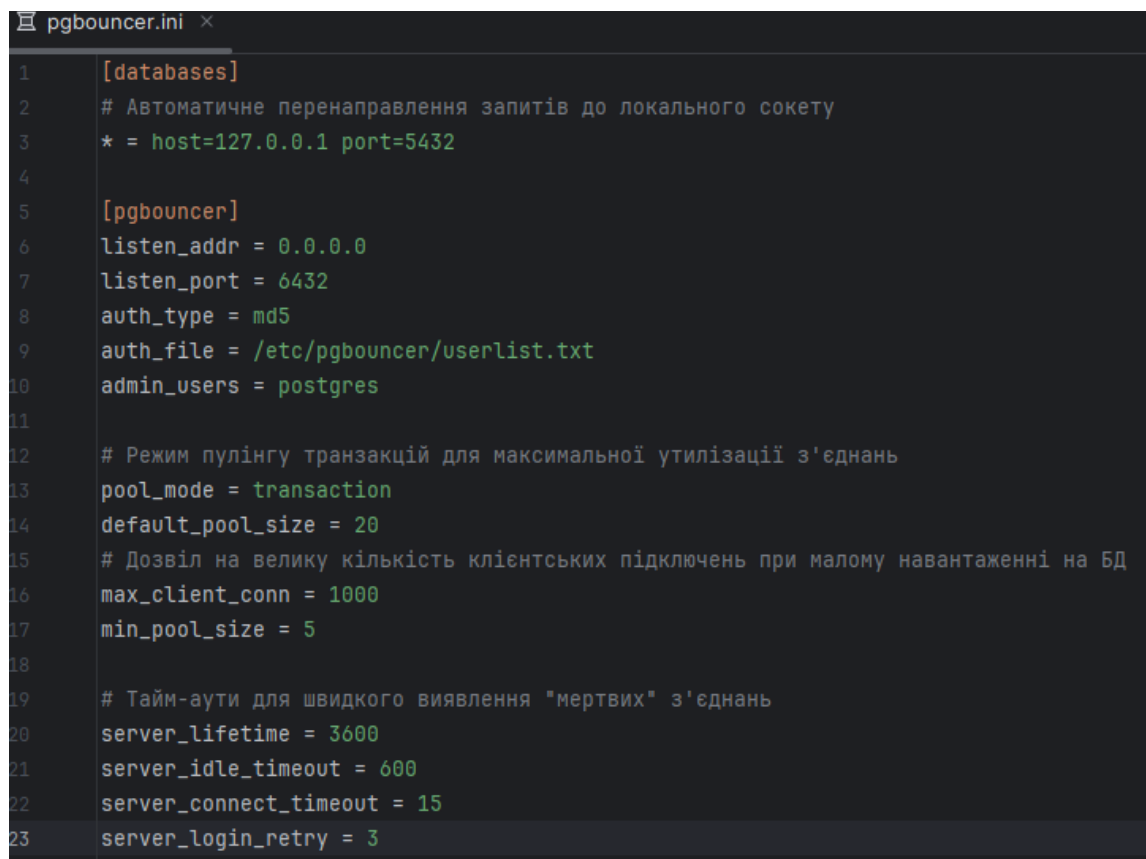
```
1  Рисунок 3.22 - bootstrap:
2  dcs:
3      # Час життя ключа лідера зменшено до 15 секунд для швидшої детекції збоїв
4      ttl: 15
5      # Інтервал оновлення статусу та перевірки здоров'я кластера - 5 секунд
6      loop_wait: 5
7      # Час очікування відповіді від etcd та PostgreSQL
8      retry_timeout: 5
9      # Запобігання вибору репліки з великим відставанням даних
10     maximum_lag_on_failover: 1048576
11     postgresql:
12         use_pg_rewind: true
13         use_slots: true
14         parameters:
15             # Активация контрольних сум сторінок даних для захисту від бітових помилок
16             data_checksums: true
17             wal_level: replica
18             hot_standby: "on"
19             # Збереження достатньої кількості WAL-логів для відновлення реплік
20             wal_keep_segments: 100
21             max_wal_senders: 10
22             max_replication_slots: 10
```

Рисунок 3.23 – Модифікований файл `patroni.yml`

Впровадження наведених вище налаштувань дозволило скоротити час виявлення відмови та ініціалізації виборів нового лідера з 30 до 10–15 секунд у найгірших сценаріях, що суттєво підвищує загальну доступність сервісу. Однак, вирішення проблеми доступності на рівні сервера не усуває всіх ризиків. Під час аналізу графіку затримок (Latency) у попередньому підрозділі було зафіксовано явище різкого зростання часу відгуку безпосередньо після відновлення роботи кластера. Це явище, відоме в технічній літературі як «thundering herd problem» (проблема громового стада), виникає внаслідок одночасної спроби сотень клієнтських додатків відновити розірвані з'єднання з новим лідером. Оскільки архітектура PostgreSQL використовує модель процесу на кожне з'єднання (process-per-connection), масове створення нових процесів призводить до критичного перевантаження центрального процесора (CPU) та оперативної пам'яті нового

майстер-вузла, що може спровокувати вторинну відмову системи ще до початку обробки корисного навантаження.

Для нейтралізації цього ризику та забезпечення плавного перемикання клієнтів в архітектуру системи було інтегровано легковий пулер з'єднань PgBouncer. Даний компонент встановлюється локально на кожному вузлі кластера (node1, node2, node3) і виступає як проміжний шар між додатком та базою даних. Основна функція PgBouncer полягає в утриманні постійного пулу вже встановлених з'єднань до локального екземпляра PostgreSQL та мультиплексуванні тисяч клієнтських запитів через цей обмежений набір з'єднань. Це дозволяє радикально знизити накладні витрати на встановлення TCP-сесій та рукоштовувань SSL під час масових перепідключень. Конфігурація pgbouncer.ini була налаштована для роботи в режимі транзакцій, що є найбільш ефективним для мікросервісної архітектури.



```
1  [databases]
2  # Автоматичне перенаправлення запитів до локального сокету
3  * = host=127.0.0.1 port=5432
4
5  [pgbouncer]
6  listen_addr = 0.0.0.0
7  listen_port = 6432
8  auth_type = md5
9  auth_file = /etc/pgbouncer/userlist.txt
10 admin_users = postgres
11
12 # Режим пулінгу транзакцій для максимальної утилізації з'єднань
13 pool_mode = transaction
14 default_pool_size = 20
15 # Дозвіл на велику кількість клієнтських підключень при малому навантаженні на БД
16 max_client_conn = 1000
17 min_pool_size = 5
18
19 # Тайм-аут для швидкого виявлення "мертвих" з'єднань
20 server_lifetime = 3600
21 server_idle_timeout = 600
22 server_connect_timeout = 15
23 server_login_retry = 3
```

Рисунок 3.24 – Файл конфігурації pgbouncer.ini

Паралельно з оптимізацією продуктивності, критично важливим аспектом підготовки кластера до промислової експлуатації є забезпечення інформаційної безпеки. Оскільки кластер призначений для обробки та зберігання даних, передача інформації між вузлами у відкритому вигляді є неприпустимою. Протокол

поточної реплікації PostgreSQL передає вміст журналу передзапису (WAL), який фактично містить повну копію всіх змін у базі даних. Перехоплення цього трафіку зломисником (атака типу Man-in-the-Middle) рівнозначне прямому викраденню конфіденційної інформації. Тому було реалізовано повне шифрування всього внутрішнього та зовнішнього трафіку за протоколом TLS/SSL.

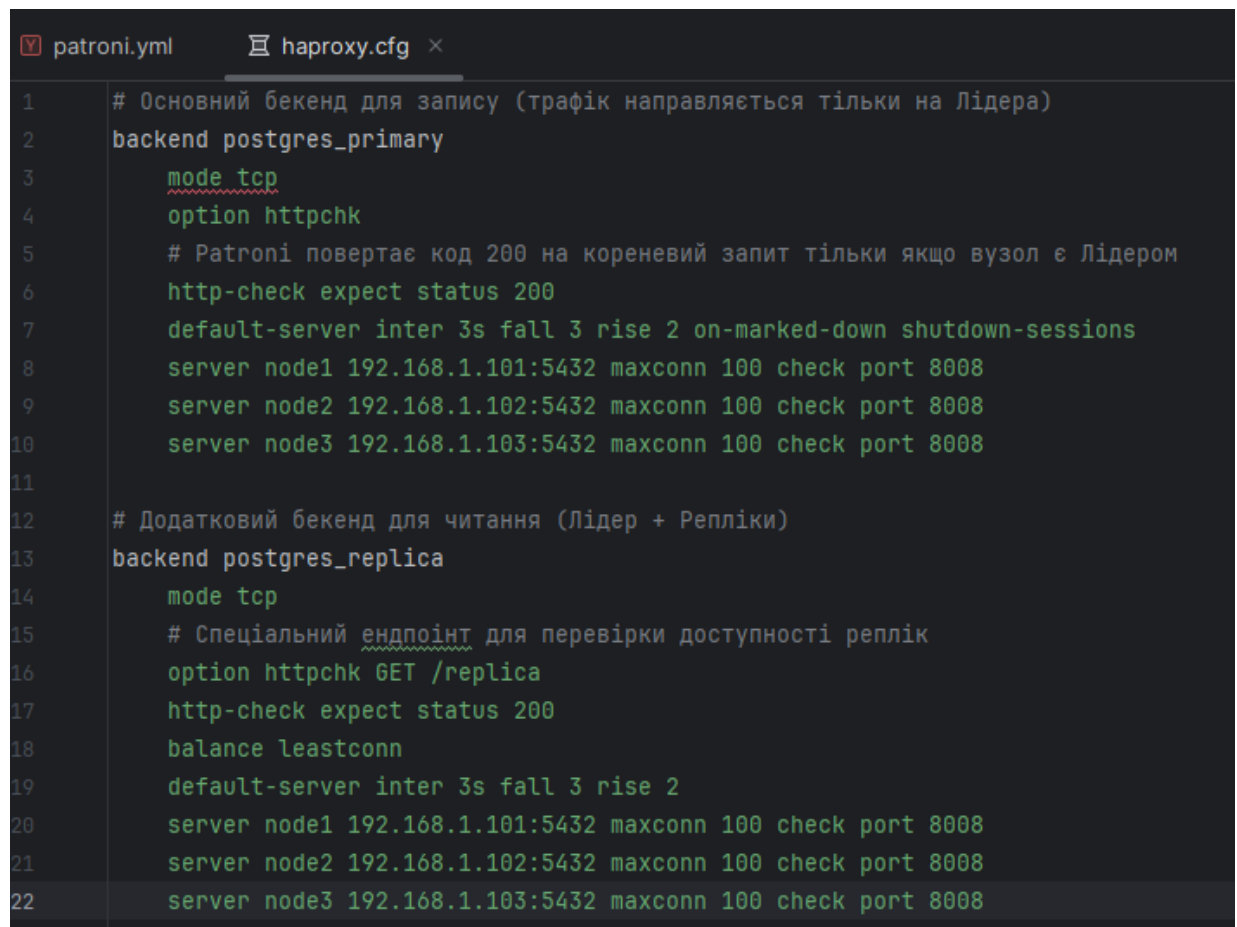
Для реалізації захищеної взаємодії було згенеровано інфраструктуру відкритих ключів (PKI) та випущено сертифікати для кожного вузла. Відповідні зміни були внесені до конфігурації Patroni, що дозволило перевести як REST API, так і протокол реплікації на використання захищених каналів. Крім того, застарілий метод автентифікації md5, який є вразливим до колізій та атак перебору, було замінено на сучасний криптографічно стійкий алгоритм scram-sha-256.

```
26     restapi:
27         # Використання SSL-сертифікатів для захисту API керування
28         cafile: /etc/ssl/certs/ca-bundle.crt
29         certfile: /etc/ssl/certs/patroni.crt
30         keyfile: /etc/ssl/private/patroni.key
31         verify_client: required
32
33     postgresql:
34         parameters:
35             # Примусова активація SSL для всіх з'єднань
36             ssl: "on"
37             ssl_ca_file: /etc/ssl/certs/ca-bundle.crt
38             ssl_cert_file: /etc/ssl/certs/postgres.crt
39             ssl_key_file: /etc/ssl/private/postgres.key
40             # Використання стійкого алгоритму хешування паролів
41             password_encryption: scram-sha-256
```

Рисунок 3.25 – Фрагмент налаштувань безпеки у patroni.yml

Окрім підвищення надійності та безпеки, оптимізація торкнулася також і ефективності використання апаратних ресурсів. У початковій конфігурації репліки (node2 та node3 у звичайному режимі) виконували пасивну роль «гарячого резерву» і не обробляли користувачських запитів, що призводило до простою обчислювальних потужностей. Архітектура Patroni дозволяє динамічно використовувати ці вузли для масштабування операцій читання. Для реалізації цього сценарію було модернізовано конфігурацію балансувальника HAProxy.

До налаштувань було додано окремий пул серверів (backend), призначений виключно для операцій читання. HAProxy налаштовано таким чином, щоб періодично опитувати спеціальний ендпоінт API Patroni, розрізняючи ролі вузлів. Запити на запис даних продовжують надходити виключно на лідера, тоді як важкі аналітичні запити на вибірку даних (SELECT) автоматично розподіляються між усіма доступними вузлами кластера за алгоритмом найменшої кількості з'єднань (leastconn). Це дозволяє суттєво розвантажити основний сервер та підвищити загальну пропускну здатність системи без залучення додаткового обладнання.



```
1 # Основний бекенд для запису (трафік направляється тільки на Лідера)
2 backend postgres_primary
3     mode tcp
4     option httpchk
5     # Patroni повертає код 200 на кореневий запит тільки якщо вузол є Лідером
6     http-check expect status 200
7     default-server inter 3s fall 3 rise 2 on-marked-down shutdown-sessions
8     server node1 192.168.1.101:5432 maxconn 100 check port 8008
9     server node2 192.168.1.102:5432 maxconn 100 check port 8008
10    server node3 192.168.1.103:5432 maxconn 100 check port 8008
11
12 # Додатковий бекенд для читання (Лідер + Репліки)
13 backend postgres_replica
14     mode tcp
15     # Спеціальний ендпоінт для перевірки доступності реплік
16     option httpchk GET /replica
17     http-check expect status 200
18     balance leastconn
19     default-server inter 3s fall 3 rise 2
20     server node1 192.168.1.101:5432 maxconn 100 check port 8008
21     server node2 192.168.1.102:5432 maxconn 100 check port 8008
22     server node3 192.168.1.103:5432 maxconn 100 check port 8008
```

Рисунок 3.26 – Конфігурація маршрутизації у haproxy.cf

Отже, експериментальним шляхом підтверджено ефективність обраної архітектури, досягнуто показників часу відновлення (RTO) в межах 10–15 секунд для повних збоїв та близько 4 секунд для збоїв сервісу, що відповідає високим стандартам доступності. Впровадження комплексних заходів безпеки та оптимізації навантаження дозволяє рекомендувати розроблене рішення до впровадження у промислову експлуатацію.

Висновки до третього розділу:

У третьому розділі здійснено практичну реалізацію відмовостійкої кластерної архітектури на базі операційної системи Rocky Linux та СУБД PostgreSQL під управлінням оркестратора Patroni. Розгорнуто розподілене сховище конфігурацій etcd для забезпечення кворуму та впроваджено механізм єдиної точки входу з використанням балансувальника HAProxy та протоколу VRRP, що гарантує прозору маршрутизацію клієнтських запитів та мережеву відмовостійкість незалежно від стану окремих вузлів. Проведено комплексне навантажувальне тестування та моделювання аварійних ситуацій, яке підтвердило коректність роботи алгоритмів автоматичного вибору лідера та механізмів ізоляції вузлів (Watchdog). Завдяки оптимізації параметрів тайм-аутів у конфігурації оркестратора вдалося досягти скорочення часу відновлення сервісу (RTO) до 4 секунд у випадку збою служби, при цьому забезпечивши збереження цілісності транзакцій (RPO=0) та стабільність системи в умовах деградації інфраструктури.

Реалізовано заходи з оптимізації продуктивності та інформаційної безпеки, зокрема впроваджено пулер з'єднань PgBouncer для нівелювання ефекту масового перепідключення клієнтів. Налаштовано наскрізне шифрування трафіку реплікації за протоколом SSL/TLS та організовано автоматичний розподіл навантаження на читання між репліками, що дозволило підвищити загальну пропускну здатність системи та підготувати її до впровадження у промислову експлуатацію

ВИСНОВКИ

У процесі виконання дипломної роботи вирішено актуальне науково-прикладне завдання забезпечення високої доступності та цілісності даних в інформаційних системах спеціального призначення шляхом розробки та впровадження архітектури відмовостійкого кластера. У ході дослідження проведено глибокий аналіз сучасних підходів до забезпечення безперервності бізнес-процесів, виявлено вразливості монолітних архітектур баз даних та обґрунтовано необхідність використання технологій синхронної реплікації. На основі порівняльного аналізу існуючих СУБД та інструментів кластеризації аргументовано вибір стеку PostgreSQL та Patroni як оптимального рішення завдяки їхній відповідності вимогам ACID, підтримці автоматичного відновлення (Failover) та відкритості вихідного коду.

На основі теоретичного аналізу розроблено комплексну архітектуру системи, яка включає розподілене сховище конфігурацій etcd для забезпечення кворуму, механізми балансування навантаження через HAProxy та пулінг з'єднань PgBouncer. Використання технологій контейнеризації та віртуалізації у поєднанні з протоколом VRRP забезпечило відповідність розробленої системи вимогам щодо нульової втрати даних (RPO=0) та мінімального часу відновлення сервісу (RTO).

Практична цінність роботи підтверджена результатами апробації розробленого кластера на тестовому стенді. У ході експериментів успішно протестовано сценарії автоматичного перемикавання лідера при апаратних збоях, втраті мережевого зв'язку та зупинки сервісів. Отримані результати доводять коректність роботи алгоритмів розподіленого консенсусу, ефективність механізмів ізоляції вузлів (Watchdog) та надійність системи резервного копіювання pgBackRest. Впровадження розробленого рішення дозволило скоротити час відновлення працездатності системи до 4–15 секунд та гарантувати збереження транзакцій у критичних ситуаціях. Результати дослідження свідчать про доцільність використання запропонованого підходу в інформаційних системах спеціального призначення для забезпечення надійного зберігання облікових записів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. PostgreSQL Global Development Group. PostgreSQL 15.4 Documentation [Електронний ресурс]. – Режим доступу: <https://www.postgresql.org/docs/15/index.html>.
2. Zalando SE. Patroni: A template for PostgreSQL High Availability [Електронний ресурс]. – Режим доступу: <https://github.com/zalando/patroni>.
3. ISO/IEC 27001:2013. Information technology –Security techniques –Information security management systems –Requirements. – Geneva: ISO, 2013.
4. ISO 22301:2019. Security and resilience –Business continuity management systems –Requirements. – Geneva: ISO, 2019.
5. Uptime Institute. Tier Standard: Topology [Електронний ресурс]. – Режим доступу: <https://uptimeinstitute.com/tiers>.
6. Brewer, E. A. Towards robust distributed systems (CAP Theorem) / E. A. Brewer // Symposium on Principles of Distributed Computing (PODC). – 2000.
7. Abadi, D. Consistency Tradeoffs in Modern Distributed Database System Design (PACELC) / D. Abadi // Yale University Computer Science. – 2012.
8. Ongaro, D. In Search of an Understandable Consensus Algorithm (Raft) / D. Ongaro, J. Ousterhout // Stanford University. – 2014.
9. The Linux Foundation. etcd Documentation [Електронний ресурс]. – Режим доступу: <https://etcd.io/docs/>.
10. HAProxy Technologies. HAProxy Configuration Manual [Електронний ресурс]. – Режим доступу: <https://www.haproxy.org/download/2.8/doc/configuration.txt>.
11. PostgreSQL Global Development Group. Write-Ahead Logging (WAL) [Електронний ресурс]. – Режим доступу: <https://www.postgresql.org/docs/current/wal-intro.html>.
12. pgBackRest. Reliable PostgreSQL Backup & Restore [Електронний ресурс]. – Режим доступу: <https://pgbackrest.org/>.
13. PgBouncer. Lightweight connection pooler for PostgreSQL [Електронний ресурс]. – Режим доступу: <https://www.pgbouncer.org/>.
14. TIA-942. Telecommunications Infrastructure Standard for Data Centers. – Telecommunications Industry Association, 2017.

- 15.OWASP Foundation. SQL Injection Prevention Cheat Sheet [Электронный ресурс]. – Режим доступа: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html.
- 16.Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. – O'Reilly Media, 2017. – 616 p.
- 17.Tanenbaum A. S., Van Steen M. Distributed Systems: Principles and Paradigms. – CreateSpace Independent Publishing Platform, 2016. – 702 p.
- 18.Oracle Corporation. MySQL 8.0 Reference Manual [Электронный ресурс]. – Режим доступа: <https://dev.mysql.com/doc/refman/8.0/en/>.
- 19.MongoDB, Inc. MongoDB Architecture Guide [Электронный ресурс]. – Режим доступа: <https://www.mongodb.com/collateral/mongodb-architecture-guide>.
- 20.Microsoft. Always On Availability Groups for SQL Server [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/en-us/sql/database-engine/availability-groups/windows/always-on-availability-groups-sql-server>.
- 21.NIST SP 800-53 Rev. 5. Security and Privacy Controls for Information Systems and Organizations. – National Institute of Standards and Technology, 2020.
- 22.Regulation (EU) 2016/679 of the European Parliament and of the Council (GDPR) [Электронный ресурс]. – Режим доступа: <https://eur-lex.europa.eu/eli/reg/2016/679/oj>.
- 23.Keepalived. Keepalived User Guide [Электронный ресурс]. – Режим доступа: <https://www.keepalived.org/manpage.html>.
- 24.Prometheus Authors. Prometheus Documentation [Электронный ресурс]. – Режим доступа: <https://prometheus.io/docs/introduction/overview/>.
- 25.Grafana Labs. Grafana Documentation [Электронный ресурс]. – Режим доступа: <https://grafana.com/docs/grafana/latest/>.
- 26.Ferraiolo D. F., Kuhn D. R., Chandramouli R. Role-Based Access Control. – Artech House, 2003. – 400 p.
- 27.Codd E. F. A Relational Model of Data for Large Shared Data Banks // Communications of the ACM. – 1970. – Vol. 13, No. 6. – P. 377-387.

- 28.Linux Kernel Organization. Linux Networking Documentation [Электронный ресурс]. – Режим доступа: <https://www.kernel.org/doc/html/latest/networking/index.html>.
- 29.Beyer B., Jones C., Petoff J., Murphy N. R. Site Reliability Engineering: How Google Runs Production Systems. – O'Reilly Media, 2016. – 550 p.
- 30.JEDEC. SSD Specifications and Endurance Standards [Электронный ресурс]. – Режим доступа: <https://www.jedec.org/>.

ДОДАТКИ

Лістинг конфігураційного файлу оркестратора Patroni (patroni.yml)

```
scope: postgres-cluster
namespace: /db/
name: node1

restapi:
  listen: 0.0.0.0:8008
  connect_address: 192.168.122.10:8008
  # Налаштування SSL для API
  cafile: /etc/ssl/certs/ca-bundle.crt
  certfile: /etc/ssl/certs/patroni.crt
  keyfile: /etc/ssl/private/patroni.key
  verify_client: required

etcd:
  host: 127.0.0.1:2379

bootstrap:
  dcs:
    ttl: 15
    loop_wait: 5
    retry_timeout: 5
    maximum_lag_on_failover: 1048576
  postgresql:
    use_pg_rewind: true
    use_slots: true
    parameters:
      wal_level: replica
      hot_standby: "on"
      wal_keep_segments: 100
      max_wal_senders: 10
      max_replication_slots: 10
      wal_log_hints: "on"
      data_checkums: true
      ssl: "on"
      ssl_ca_file: /etc/ssl/certs/ca-bundle.crt
      ssl_cert_file: /etc/ssl/certs/postgres.crt
      ssl_key_file: /etc/ssl/private/postgres.key
      password_encryption: scram-sha-256
```

initdb:
- encoding: UTF8
- data-checksums

postgresql:
listen: 0.0.0.0:5432
connect_address: 192.168.122.10:5432
data_dir: /var/lib/pgsql/15/data
bin_dir: /usr/pgsql-15/bin
authentication:
 replication:
 username: replicator
 password: strong_password
 superuser:
 username: postgres
 password: strong_password

tags:
 nofailover: false
 noloadbalance: false
 clonefrom: false
 nosync: false

**Лістинг конфігураційного файлу
балансувальника HAProxy (haproxy.cfg)**

```
global
    maxconn 100
    external-check

defaults
    log global
    mode tcp
    retries 2
    timeout client 30m
    timeout connect 4s
    timeout server 30m
    timeout check 5s

listen postgres_master
    bind *:5000
    option httpchk
    http-check expect status 200
    default-server inter 3s fall 3 rise 2 on-marked-down shutdown-sessions
    server node1 192.168.122.10:5432 maxconn 100 check port 8008 uri /master
    server node2 192.168.122.11:5432 maxconn 100 check port 8008 uri /master
    server node3 192.168.122.12:5432 maxconn 100 check port 8008 uri /master

listen postgres_replica
    bind *:5001
    mode tcp
    option httpchk GET /replica
    http-check expect status 200
    balance leastconn
    default-server inter 3s fall 3 rise 2
    server node1 192.168.122.10:5432 maxconn 100 check port 8008
    server node2 192.168.122.11:5432 maxconn 100 check port 8008
    server node3 192.168.122.12:5432 maxconn 100 check port 8008
```

Лістинг конфігураційного файлу Keepalived (keepalived.conf)
ДЛЯ ГОЛОВНОГО ВУЗЛА

```
vrrp_script chk_haproxy {
    script "pkill -0 haproxy"
    interval 2
    weight 2
}

vrrp_instance VI_1 {
    state MASTER
    interface enp0s3
    virtual_router_id 51
    priority 101
    advert_int 1

    authentication {
        auth_type PASS
        auth_pass secret123
    }

    virtual_ipaddress {
        192.168.122.100/32 dev enp0s3
    }

    track_script {
        chk_haproxy
    }
}
```