МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

І. Л. Красніков, І. Г. Лисаченко, А. К. Бабіченко

ІНТЕГРАЦІЯ MySQL У SCADA-СИСТЕМИ: ТЕОРІЯ І ПРАКТИКА

Навчальний посібник

для бакалаврів та магістрів напряму підготовки
G7 «Автоматизація, комп'ютерно-інтегровані
технології та робототехніка»
усіх форм навчання

Затверджено
редакційно-видавничою
радою НТУ «ХПІ»,
протокол № 1 від 19.02.2026 р.

Харків
НТУ «ХПІ»
2026

УДК 681.518
К 78

Рецензенти:
С. І. Планковський, д-р техн. наук, проф., Харківський національний університет радіоелектроніки;
В. О. Панасенко, д-р техн. наук, проф., Державна установа «Державний науково-дослідний і проєктний інститут основної хімії»

Красніков І. Л .

К 78 Інтеграція MySQL у SCADA-системи: теорія і практика : навчальний посібник для бакалаврів та магістрів напряму підготовки G7 «Автоматизація, комп'ютерно-інтегровані технології та робототехніка» усіх форм навчання / І. Л. Красніков, І. Г. Лисаченко, А. К. Бабіченко. Харків : НТУ «ХПІ», 2026. 235 с.

ISBN 978-617-05-0611-5

У навчальному посібнику викладено теоретичні положення та практичні підходи до побудови автоматизованих систем керування технологічними процесами із застосуванням SCADA-систем і реляційних баз даних. Розглянуто принципи організації зберігання, оброблення та передавання технологічних даних у промислових інформаційних системах із використанням СУБД MySQL, механізмів історизації та подієвих технологій обміну інформацією. Окрему увагу приділено інтеграції SCADA із базами даних у середовищі Windows, застосуванню технології Change Data Capture та використанню брокерів повідомлень. Наведено приклади практичної реалізації типових рішень у задачах промислової автоматизації.

Призначено для студентів спеціальності «Автоматизація, комп'ютерно-інтегровані технології та робототехніка» денної та заочної форм навчання під час вивчення фахових дисциплін і виконання навчальних проєктів. Матеріал також може бути корисним аспірантам, викладачам і фахівцям у галузі промислової автоматизації.

Іл. 45. Бібліогр. 18 назв.

УДК 681.518

ISBN 978-617-05-0611-5

ЗМІСТ

ВСТУП

Сучасні інформаційні системи дедалі частіше потребують оперативної обробки даних, що безпосередньо впливає на ефективність, стабільність і конкурентоспроможність цифрових рішень. Особливо актуальним це є у контексті великих даних, де своєчасна фіксація та аналітика інформації дозволяють швидко реагувати на зміни процесів. Це підвищує надійність, продуктивність і безпеку систем.

Цей навчальний посібник орієнтований на студентів та аспірантів, які готуються до професійної діяльності в галузі автоматизації, цифрових технологій, інформаційного та технічного управління. Він надає знання та практичні навички для інтеграції, обробки та збереження даних із різноманітних джерел у реляційних базах даних.

Особлива увага приділяється роботі з системою управління базами даних *MySQL*, яка забезпечує високу продуктивність та надійність при обробці великих обсягів інформації. Розглянуто базові операції, індексацію, реплікацію, тригери, збережені процедури, оптимізацію продуктивності та механізми потокової передачі даних. Приклади демонструють ефективне застосування *MySQL* у сучасних інформаційно-керуючих системах, включно з *SCADA*-системами (*Supervisory Control And Data Acquisition*) промислової автоматизації, платформами Інтернету речей (IoT) та системами аналітики.

Мета посібника полягає у формуванні у здобувачів освіти компетентностей для створення ефективних, масштабованих і безпечних баз даних, здатних інтегруватися у сучасні цифрові платформи керування, аналітики та моніторингу. Посібник забезпечує практичне застосування отриманих знань для вирішення реальних завдань у проєктуванні та експлуатації технологічних інформаційних систем.

РОЗДІЛ 1. ОСНОВИ БАЗ ДАНИХ ТА *MYSQL*

1.1. Загальні питання організації баз даних

База даних (БД) – це впорядкована сукупність даних, організована для ефективного зберігання, обробки, оновлення та надання доступу до інформації. Вона є ключовим компонентом сучасних інформаційних систем, оскільки дозволяє працювати з великими обсягами даних за допомогою спеціалізованого програмного забезпечення.

Структурована організація бази даних забезпечує швидкий пошук, фільтрацію та обробку інформації. При цьому зберігається цілісність та узгодженість даних, навіть у разі паралельного доступу кількох користувачів. Системи управління базами даних (СУБД) підтримують високу доступність, масштабованість та стабільну продуктивність під час роботи з великими інформаційними масивами.

Однією з ключових характеристик БД є безпека збереженої інформації. Сучасні СУБД реалізують механізми контролю доступу, автентифікації користувачів і шифрування, що забезпечує захист від несанкціонованого доступу та надійне збереження даних.

У сучасних IT-рішеннях бази даних активно застосовуються в аналітичних системах, машинному навчанні та штучному інтелекті. Вони використовуються для збирання, фільтрації, класифікації та прогнозування на основі великих обсягів інформації. Останнім часом зростає популярність хмарних баз даних, що дозволяють зменшити витрати на інфраструктуру, підвищити надійність та забезпечити гнучкий віддалений доступ до даних.

1.2. Типи баз даних: реляційні, нереляційні, бази реального часу

Тип бази даних визначається способом організації інформації та особливостями її обробки. Найпоширенішими є реляційні, нереляційні (*NoSQL*) та бази даних реального часу.

Реляційні бази даних організовують інформацію у вигляді таблиць відповідно до реляційної моделі. Кожен рядок у таблиці є окремим записом, а кожен стовпець - атрибутом. Такі системи підтримують формалізовану структуру даних, забезпечують транзакційність, цілісність і узгодженість інформації. Для виконання запитів використовується мова *SQL*. Реляційні СУБД широко застосовуються у фінансових, банківських, логістичних та облікових системах, а також у системах керування технологічними процесами, де потрібна надійність і точність обробки даних. Серед них: *MySQL*, *PostgreSQL*, *Oracle Database*.

Нереляційні бази даних (*NoSQL*) відрізняються гнучкою структурою зберігання, яка може базуватися на моделях «ключ–значення», документів, графів або колонок. Вони особливо ефективні для роботи з великими обсягами неструктурованих даних, що надходять у режимі реального часу, наприклад, від пристроїв Інтернету речей, сенсорів та систем моніторингу. Такі системи добре масштабуються, забезпечують високу швидкодію та стійкість до відмов. Прикладами є *MongoDB*, *Redis*, *Cassandra*.

Бази даних реального часу призначені для роботи з мінімальними затримками під час запису, читання та обробки інформації. Їх застосовують у критично важливих системах, де потрібна миттєва реакція, зокрема в автоматизованих системах керування технологічними процесами (АСУ ТП), енергетиці, транспорті та безпеці. Такі бази підтримують обробку потокових даних і забезпечують постійний доступ до актуальної інформації.

У системах керування технологічними процесами бази даних реального часу відіграють ключову роль, оскільки забезпечують оперативне зчитування сигналів від датчиків, вимірювальних приладів та виконавчих механізмів. Це дозволяє здійснювати контроль за параметрами процесу, тиском, температурою, витратою, рівнем тощо у режимі реального часу і негайно реагувати на відхилення, запобігаючи аварійним ситуаціям і підвищуючи безпеку виробництва.

Такі бази використовуються для збирання, аналізу та архівування технологічних даних, що дає змогу оптимізувати роботу обладнання, підвищувати якість продукції та забезпечувати адаптивне управління виробничими процесами.

Вибір типу бази даних у системах керування визначається специфікою технологічного процесу, вимогами до швидкодії, обсягами і структурою даних, а також критичністю часу реакції на зміни параметрів. Ефективність автоматизації значною мірою залежить від коректного вибору архітектури бази даних з урахуванням цих факторів.

1.3. Порівняльна характеристика популярних СУБД і обґрунтування вибору *MySQL* для систем реального часу

Вибір системи управління базами даних (СУБД) у сучасних інформаційних системах базується на технічних та функціональних вимогах, серед яких першочерговими є надійність, масштабованість, швидкодія, а також відповідність специфіці конкретного застосування. Особливо це актуально для систем реального часу, де критично важливі мінімальні затримки, стійкість до збоїв і стабільність роботи при змінних навантаженнях.

Серед найбільш поширених СУБД, які застосовуються у різних сферах, виділяють *PostgreSQL*, *MongoDB*, *Redis*, *Oracle Database* та *MySQL*. Кожна система має свої переваги та обмеження, які впливають на доцільність її використання в конкретних умовах.

PostgreSQL – це потужна об’єктно-реляційна СУБД з підтримкою розширених типів даних і складної аналітики. Вона ефективна для складних запитів та роботи зі структурованими і напівструктурованими даними. Втім, висока складність налаштування і адміністрування може ускладнити застосування у системах з жорсткими вимогами до швидкодії та простої підтримки, характерними для деяких рішень реального часу.

MongoDB, як документоорієнтована *NoSQL*-СУБД, забезпечує гнучкість у роботі з динамічною структурою даних. Вона добре підходить для проектів із

великими обсягами неструктурованої інформації. Однак обмежена підтримка транзакцій і відсутність повної консистентності обмежують використання у критичних виробничих та диспетчерських системах.

Redis характеризується дуже високою швидкодією за рахунок зберігання даних у оперативній пам’яті. Це робить її ефективною для кешування, організації черг і швидкої обробки подій. Проте Redis не призначена для довготривалого зберігання великих структурованих масивів даних або виконання складних запитів, тому її рідко використовують як основну СУБД у комплексних системах управління.

Oracle Database є корпоративним рішенням із широкими можливостями масштабування, обробки даних у реальному часі та вбудованими засобами безпеки. Незважаючи на це, висока вартість ліцензій і складність експлуатації обмежують її застосування у системах середнього рівня, навчальних та дослідницьких проєктах.

MySQL відзначається оптимальним балансом між функціональністю, швидкодією, простотою впровадження та адаптації під задачі реального часу. Завдяки підтримці транзакційної моделі *ACID*, механізму збереження InnoDB, реплікації, кластеризації та широкій сумісності з промисловими протоколами і *SCADA*-платформами (зокрема *Ignition*, *FactoryTalk*, *Proficy*), ця СУБД є ефективним вибором для комп’ютерно-інтегрованих систем управління.

MySQL забезпечує низькі затримки при обробці транзакцій, стабільну роботу при високих навантаженнях та легкість інтеграції з різними мовами програмування, що робить її придатною для використання як оперативне сховище сигналів, подій і технологічних параметрів. В системах автоматизації *MySQL* часто виконує роль буферу даних або центрального компонента інформаційної інфраструктури.

Хоча *MySQL* має певні обмеження щодо підтримки складної аналітики та спеціалізованих індексів, ці недоліки не критичні для задач, пов’язаних із короткотривалими транзакціями, обробкою подій і управлінням технологічними процесами в реальному часі.

Таким чином, серед розглянутих СУБД *MySQL* найбільш відповідає вимогам систем диспетчеризації, технологічного моніторингу та автоматизованого управління, поєднуючи стабільність, гнучкість інтеграції та прийнятну вартість впровадження й супроводу.

1.4. Основи роботи з таблицями у *MySQL*: створення, модифікація та видалення

SQL (*Structured Query Language*) є стандартною мовою для роботи з реляційними базами даних. Вона дозволяє як керувати структурою бази (створювати, змінювати таблиці), так і оперувати даними (вибирати, додавати, оновлювати, видаляти записи).

Таблиці є основними об’єктами баз даних, у яких зберігаються структуровані дані. У цьому розділі розглянуто базові операції з таблицями в *MySQL*: створення, зміну та видалення.

1.4.1. Створення таблиць

Для створення таблиці застосовується команда `CREATE TABLE`, у якій визначаються назва таблиці, імена стовпців, типи даних та додаткові атрибути. Загальний синтаксис команди:

```
CREATE TABLE table_name (
column1_name data_type constraints,
column2_name data_type constraints
);
```

Наприклад, у системі моніторингу параметрів виробничої лінії може створюватися таблиця для зберігання результатів вимірювань датчиків:

```
CREATE TABLE measurements (
measurement_id INT AUTO_INCREMENT PRIMARY KEY,
sensor_id INT NOT NULL,
value DECIMAL(10,2) NOT NULL,
```

```
measurement_time TIMESTAMP DEFAULT
CURRENT_TIMESTAMP
);
```

Тут створюється таблиця measurements, де `measurement_id` є унікальним ідентифікатором, `sensor_id` зберігає код датчика, `value` - виміряне значення, а `measurement_time` - час фіксації параметра.

1.4.2. Модифікація таблиць

Щоб змінити структуру таблиці, використовується команда `ALTER TABLE`. Вона дає змогу додавати, змінювати або видаляти стовпці.

У наведеному прикладі до таблиці measurements додається стовпець unit для збереження одиниць вимірювання:

```
ALTER TABLE measurements ADD unit VARCHAR(10);
```

Щоб змінити властивості цього стовпця, наприклад зробити його обов’язковим для заповнення та збільшити довжину, застосовується команда:

```
ALTER TABLE measurements MODIFY unit VARCHAR(20) NOT
NULL;
```

Для перейменування стовпця застосовується команда:

```
ALTER TABLE measurements CHANGE unit
measurement_unit VARCHAR(20);
```

Цей запит перейменовує стовпець `unit` на `measurement_unit`.

Видалення стовпця виконується за допомогою команди:

```
ALTER TABLE measurements DROP COLUMN
measurement_unit;
```

Вона повністю видаляє стовпець *measurement_unit* з таблиці.

1.4.3. Видалення таблиць

Щоб повністю видалити таблицю разом із її даними, використовується команда `DROP TABLE`:

```
DROP TABLE measurements;
```

Вона повністю прибирає таблицю `measurements` з бази даних.

1.4.4. Додаткові можливості

Щоб уникнути помилок, доцільно перевіряти існування таблиці перед її створенням або видаленням:

```
CREATE TABLE IF NOT EXISTS measurements (
measurement_id INT AUTO_INCREMENT PRIMARY KEY,
sensor_id INT NOT NULL
);
DROP TABLE IF EXISTS measurements;
```

Якщо потрібно швидко видалити всі дані з таблиці, залишивши її структуру, використовується команда `TRUNCATE`:

```
TRUNCATE TABLE measurements;
```

Для підтримання цілісності даних варто використовувати первинні ключі (*PRIMARY KEY*), унікальні обмеження (*UNIQUE*), значення за замовчуванням (*DEFAULT*) та індекси (*INDEX*) для прискорення пошуку. Перед виконанням *DROP* або *TRUNCATE* рекомендується створювати резервні копії, щоб уникнути втрати важливої інформації.

Знання основ роботи з таблицями у *MySQL* дає змогу будувати гнучкі та надійні бази даних, що легко адаптуються до змін у вимогах автоматизованих систем.

1.5. Базові *SQL* запити: *SELECT*, *INSERT*, *UPDATE*, *DELETE*

У цьому розділі розглядаються чотири базові типи *SQL*-запитів. Запит `SELECT` використовується для вибірки даних із таблиць бази даних. Команда `INSERT` дозволяє додавати нові записи. За допомогою `UPDATE` здійснюється оновлення існуючих даних, а `DELETE` застосовується для видалення записів за визначеними умовами.

Ці операції є фундаментальними для роботи з базами даних і становлять основу взаємодії з інформаційними системами, що використовують реляційну модель.

1.5.1. SELECT: отримання даних

Запит *SELECT* використовується для вибору даних із таблиць бази даних. Це один із найпоширеніших *SQL*-запитів, оскільки дає змогу отримувати необхідну інформацію з таблиць. Загальний синтаксис має вигляд:

```
SELECT column1, column2,…
FROM table_name
WHERE condition
ORDER BY column
LIMIT number;
```

Ця конструкція дозволяє вибирати конкретні стовпці, фільтрувати записи за умовами, сортувати результат і обмежувати кількість виведених рядків.

У наведеному прикладі вибираються всі записи з таблиці `sensor_data`, відсортовані за часом у порядку спадання, з обмеженням у 50 останніх вимірювань.

```
SELECT *
FROM sensor_data
ORDER BY timestamp DESC
LIMIT 50;
```

Інший приклад демонструє вибір даних температури та тиску для конкретного обладнання.

```
SELECT temperature, pressure
FROM sensor_data
WHERE equipment_id = 'BOILER-01';
```

Далі наведено запит, що відбирає лише ті записи, де рівень рідини перевищує 2.5 метра.

```
SELECT *
FROM tank_levels
WHERE water_level > 2.5;
```

Ще один варіант запиту здійснює вибір даних про викиди CO_2 за останні 24 години.

```
SELECT timestamp, co2_level
FROM sensor_data
WHERE timestamp >= NOW() - INTERVAL 1 DAY;
```

1.5.2. INSERT: додавання даних

Команда *INSERT* використовується для внесення нових записів у таблицю. Загальний синтаксис:

```
INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...);
```

У прикладі нижче у таблицю `sensor_data` вносяться дані про температуру, тиск та час вимірювання.

```
INSERT INTO sensor_data (temperature, pressure,
timestamp)
VALUES (85.2, 1.25, '2025-08-09 14:15:00');
```

Інший приклад показує додавання інформації про рівень води у резервуарі.

```
INSERT INTO tank_levels (tank_id, water_level,
timestamp)
```

```
VALUES ('TANK-03', 2.85, NOW());
```

У наступному випадку в таблицю `event_log` додається запис про аварійне відключення насоса.

```
INSERT INTO event_log (event_type, equipment_id,
timestamp)
VALUES ('EMERGENCY STOP', 'PUMP-01', NOW());
```

Ще один приклад демонструє внесення тестових даних під час пусконалагоджувальних робіт.

```
INSERT INTO sensor_data (temperature, pressure,
timestamp, description)
VALUES (90.0, 1.3, NOW(), 'TEST DATA');
```

1.5.3. UPDATE: оновлення даних

Команда *UPDATE* використовується для зміни існуючих значень у таблиці. Вона застосовується, коли необхідно виправити помилкові дані, відкоригувати параметри після калібрування датчиків або оновити статус обладнання. Загальний синтаксис:

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

Використання умови *WHERE* є обов'язковим у більшості випадків, щоб уникнути випадкового оновлення всіх записів.

У цьому прикладі оновлюється значення температури для конкретного запису за часом вимірювання.

```
UPDATE sensor_data
SET temperature = 83.5
WHERE timestamp = '2025-08-09 12:30:00';
```

Далі показано коригування рівня води після повторного калібрування датчика.

```
UPDATE sensor_data
SET water_level = 2.90
WHERE id = 105;
```

Ще один варіант використання цієї команди - зміна статусу обладнання в таблиці *equipment_status*.

```
UPDATE equipment_status
SET status = 'MAINTENANCE'
WHERE equipment_id = 'PUMP-02';
```

Останній приклад демонструє масове оновлення значень CO_2, якщо виявлено похибку калібрування на 5 одиниць.

```
UPDATE sensor_data
SET co2_level = co2_level - 5
WHERE timestamp BETWEEN '2025-08-09 00:00:00' AND
'2025-08-09 23:59:59';
```

1.5.4. DELETE: видалення даних

Команда *DELETE* використовується для видалення одного або кількох записів із таблиці. Найчастіше її застосовують для очищення журналів, видалення застарілих даних або помилкових записів. Загальний синтаксис:

```
DELETE FROM table_name
WHERE condition;
```

Як і з командою *UPDATE*, умова *WHERE* потрібна для запобігання видаленню всіх даних.

У прикладі нижче видаляється один запис із таблиці `sensor_data` за конкретним часом вимірювання.

```
DELETE FROM sensor_data
WHERE timestamp = '2025-08-09 12:30:00';
```

Далі показано видалення всіх записів, де температура перевищувала 150 °C, що вважається помилковим показом.

```
DELETE FROM sensor_data
WHERE temperature > 150;
```

Ще один приклад демонструє очищення журналу подій старше ніж 30 днів.

```
DELETE FROM event_log
WHERE timestamp < NOW() - INTERVAL 30 DAY;
```

Останній варіант демонструє видалення тестових даних, що були внесені під час налагодження системи.

```
DELETE FROM sensor_data
WHERE description = 'TEST DATA';
```

В роботі з базами даних у системах автоматизації важливо враховувати не лише базові операції створення, модифікації та видалення таблиць, а й аспекти, що забезпечують ефективність, надійність та цілісність збережених даних.

Для інтеграції баз даних із *SCADA*-системами особливо важливими є такі напрямки:

Індексація та оптимізація продуктивності запитів, що дозволяє обробляти великі обсяги даних із мінімальними затримками;

Забезпечення цілісності даних за допомогою зовнішніх ключів та обмежень, які гарантують коректність зв'язків між таблицями;

Автоматизація обробки даних за допомогою тригерів, що дозволяє реалізовувати реакції на зміну даних без додаткових втручань;

Підтримка транзакцій для збереження консистентності даних під час складних операцій;

Архівування та планування очищення бази даних з метою підтримання стабільної продуктивності системи;

Інтеграція баз даних з *SCADA*-системами, яка забезпечує ефективний обмін даними та їх обробку у реальному часі;

Забезпечення безпеки даних шляхом налаштування прав доступу, шифрування та аудиту операцій;

Резервне копіювання та відновлення для запобігання втраті критично важливої інформації.

Вказані питання розглядаються детально в наступних розділах посібника, що дає змогу глибше опанувати принципи інтеграції баз даних у комп’ютерно-інтегровані системи автоматизації технологічних процесів, забезпечуючи їх надійність і ефективність.

РОЗДІЛ 2. ОСНОВИ РОБОТИ В *MYSQL*

2.1. Архітектура *MySQL*

СУБД *MySQL* характеризується багаторівневою архітектурою, яка забезпечує високу продуктивність, надійність та масштабованість. Ця архітектура складається з трьох основних рівнів: клієнтського, серверного та рівня зберігання даних. Кожен рівень виконує унікальну функцію, забезпечуючи ефективну взаємодію користувачів з системою керування базами даних.

2.1.1. Клієнтський рівень

Клієнтський рівень відповідає за взаємодію між користувачем (або додатком) та серверним компонентом *MySQL*. Цей рівень реалізує протокол комунікації для передачі *SQL*-запитів та повернення результатів. Його ключові функції включають: приймання запитів від клієнта, інкапсуляцію їх у мережеві пакети (наприклад, через *TCP*/*IP*), передачу серверу, обробку відповідей та повернення даних у зрозумілому форматі. Популярні інструменти роботи з СУБД, такі як *MySQL Workbench*, *phpMyAdmin* чи командний інтерфейс *mysql*, функціонують у клієнтському середовищі.

2.1.2. Серверний рівень

Серверний рівень є ключовим компонентом архітектури *MySQL,* на якому здійснюється обробка *SQL*-запитів, керування транзакціями та оптимізація роботи системи. Цей рівень охоплює низку основних підсистем, що забезпечують повноцінне функціонування сервера баз даних.

Аналіз *SQL*-запитів виконується синтаксичним аналізатором, або парсером, який перевіряє правильність синтаксису та семантики. Він визначає, чи сформульовано запит відповідно до правил мови *SQL*. Парсер розбиває запит на окремі структурні складові: ключові слова, імена таблиць і стовпців, оператори та значення. Після цього він перевіряє їхню допустимість та

коректне поєднання. У разі виявлення помилок система формує відповідне повідомлення, що дозволяє користувачеві усунути недоліки. Після успішного аналізу запит передається оптимізатору, який формує найефективніший план його виконання з урахуванням наявних індексів, статистичних даних і внутрішньої структури таблиць.

Виконання запиту забезпечується спеціальним компонентом, що реалізує необхідні дії над таблицями, зокрема сортування, фільтрацію та інші операції з обробки даних. Для підвищення продуктивності *MySQL* використовує механізми кешування, які дають змогу зберігати результати раніше виконаних запитів і повертати їх без повторного виконання.

Серверний рівень також реалізує механізми транзакційності, що базуються на принципах моделі *ACID*: атомарності, узгодженості, ізольованості та надійності. Це гарантує цілісність даних і стійкість системи до збоїв. Крім того, передбачено можливість розширення функціоналу за допомогою плагінів, які додають нові можливості, зокрема засоби шифрування, автентифікації або ведення журналу доступу.

2.1.3. Рівень зберігання даних

MySQL пропонує різноманітні механізми зберігання даних, кожен з яких має свої особливості та найкраще підходить для певних завдань.

InnoDB є найпоширенішим механізмом, який забезпечує високу надійність завдяки підтримці транзакцій, зовнішніх ключів та блокування на рівні рядків. Він ідеально підходить для сучасних застосувань, що вимагають високої цілісності даних.

MyISAM є простішим і швидшим механізмом для читання даних, але не підтримує транзакцій і зовнішніх ключів. Він добре підходить для таблиць, які містять статичні дані або до яких рідко здійснюються запити на зміну.

Memory зберігає дані в оперативній пам’яті, що забезпечує надзвичайно швидкий доступ. Цей механізм використовується для кешування даних або тимчасових таблиць.

Archive оптимізований для зберігання великих обсягів даних, до яких рідко звертаються, і дозволяє лише додавати нові записи.

Серед інших механізмів можна виділити *CSV*, що використовується для зберігання даних у форматі *CSV*, та *Blackhole*, який поглинає дані без їх фактичного зберігання.

При виборі механізму зберігання слід враховувати тип запитів, розмір даних, частоту доступу, вимоги до продуктивності та надійності. *InnoDB* рекомендується для більшості застосувань, що потребують високої надійності та підтримки транзакцій. *MyISAM* підходить для таблиць переважно з операціями читання. Memory доцільно використовувати для кешування або тимчасових таблиць, а *Archive* – для архівування великих обсягів даних, до яких рідко звертаються.

Важливо: завжди створюйте резервну копію даних перед зміною механізму зберігання таблиці.

Взаємодія між рівнями архітектури *MySQL* відбувається за наступним принципом: клієнтський рівень надсилає *SQL*-запит серверу, де він проходить синтаксичний аналіз, оптимізацію та виконання з використанням відповідного механізму зберігання даних. Після обробки результати повертаються клієнту.

Багаторівнева архітектура *MySQL* забезпечує модульність, гнучкість, високу продуктивність, безпеку, масштабованість та надійність, що робить її потужним інструментом для ефективного управління системами баз даних.

2.2. Встановлення *MySQL*

MySQL є однією з найпопулярніших систем управління базами даних, що підтримує роботу на різних операційних системах, зокрема *Windows*, *Linux* та *macOS*. Процес її встановлення та початкового налаштування передбачає дотримання певних кроків.

Для встановлення *MySQL Community Server* на платформу *Windows* необхідно завантажити відповідний дистрибутив з офіційного сайту розробника

(https://www.*mysql.com*). Користувачеві пропонується два основних варіанти завантаження:

mysql-installer-web-community.msi, який вимагає наявності інтернет-з'єднання та автоматично завантажує й встановлює всі необхідні компоненти;

mysql-installer-community.msi, що дозволяє виконати інсталяцію без підключення до мережі інтернет, забезпечуючи автономний процес.

Безкоштовна версія має назву *MySQL Community Server.* Платна версія *MySQL Enterprise Edition* включає інструменти для гарячого резервного копіювання, моніторингу продуктивності, розширені функції безпеки, підтримку високої доступності, оптимізацію продуктивності, розширені можливості аудиту та професійну технічну підтримку 24/7.

Інсталятор *MySQL* пропонує кілька режимів встановлення:

Developer Default – встановлює компоненти, необхідні для розробників (режим за замовчуванням); -

Server only – встановлює лише серверну частину системи;

Client only – встановлює лише клієнтські інструменти;

Full – повна інсталяція всіх доступних компонентів;

Custom – вибіркове встановлення необхідних компонентів.

У режимі вибіркового встановлення доступні такі компоненти:

MySQL Server – основний сервер бази даних;

MySQL Workbench – інструмент із графічним інтерфейсом для адміністрування, розробки та проєктування баз даних;

MySQL Shell – багатофункціональний інструмент для управління базами через командний рядок із підтримкою *SQL, Python ma JavaScript*;

MySQL Router – компонент, що забезпечує маршрутизацію запитів до серверів *MySQL*, сприяючи високій доступності та балансуванню навантаження;

MySQL Connectors – набір драйверів для підключення до *MySQL* з різних мов програмування (*Java, .NET, Python, C++* та інші);

MySQL Documentation – локальна копія документації *MySQL* для доступу без підключення до інтернету;

MySQL Samples and *Examples* – приклади та зразки баз даних для навчання та тестування.

Окрім *Windows*, *MySQL* підтримує операційні системи *Linux* (*Ubuntu*, *CentOS*, *Debian* та інші дистрибутиви), *macOS*, а також може бути розгорнута на платформах *Docker* і в хмарних середовищах, таких як *AWS*, *Google Cloud* та *Microsoft Azure*.

2.3. Початкове налаштування *MySQL* та конфігурація *MySQL* у *Windows*

Після завершення установки *MySQL Server* автоматично запускається як служба *Windows*. Якщо цього не сталося, його можна запустити вручну через командний рядок або панель управління службами. Назва служби *MySQL* залежить від версії встановленого сервера: наприклад, для версії *MySQL* 8.0 вона має назву *MySQL* 80.

Служба *MySQL* на *Windows* працює як фонова програма, яка автоматично запускається під час завантаження операційної системи, якщо налаштована на автоматичний запуск. Вона відповідає за такі функції:

Прослуховування підключень до бази даних: служба запускає сервер *MySQL,* який чекає на підключення клієнтів (локальних або віддалених).

Обробку запитів: Коли клієнт підключається до *MySQL,* служба обробляє *SQL*-запити та повертає результати.

Управління базами даних: Вона здійснює доступ до баз даних, виконання транзакцій, резервне копіювання та відновлення даних, а також інші операції.

Виконання функцій управління: Служба може запускати або зупиняти окремі інструменти *MySQL,* такі як реплікація, резервне копіювання, моніторинг тощо.

Після того, як сервер *MySQL* було встановлено на ПК, потрібно виконати кілька кроків для його налаштування та підготовки до використання.

За замовчуванням *MySQL* використовує стандартні налаштування конфігурації, але їх можна змінити в конфігураційному файлі (*my.cnf* або *my.ini*, залежно від ОС). Шлях до цього файлу для *Windows*: *C:\ProgramData\MySQL\MySQL Server X.X\my.ini*

Файл конфігурації *my.ini* є основним джерелом налаштувань для сервера *MySQL*. Його можна використовувати для оптимізації продуктивності, особливо при роботі з великими обсягами даних та високими навантаженнями в реальному часі.

Далі наведено конфігураційний файл *my.ini* за замовчуванням у версії *MySQL* 8.0.

```
 [client]
# Порт для підключення клієнта до сервера MySQL
port=3306

[mysql]
# Вимикає звукові сигнали в клієнті mysql
no-beep

[mysqld]
# === Основні параметри сервера ===

# Порт, на якому сервер MySQL слухатиме підключення
port=3306

# Каталог для зберігання файлів баз даних
datadir=C:/ProgramData/MySQL/MySQL Server 8.0/Data

# Плагін аутентифікації за замовчуванням (для сумісності зі старими версіями)
default_authentication_plugin=mysql_native_password

# Механізм зберігання даних за замовчуванням
default-storage-engine=InnoDB

# Режими SQL
# STRICT_TRANS_TABLES: строгий режим, забороняє операції з невідповідними даними
# NO_ENGINE_SUBSTITUTION: запобігає автоматичній заміні механізмів зберігання
sql-mode="STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION"

# === Налаштування логування ===
# Тип виводу логів (FILE або TABLE)
```

```
log-output=FILE

# Загальний лог (0 - вимкнено, 1 - увімкнено)
general-log=0

# Файл для загального логу
general_log_file="IKLHOME.log"

# Лог повільних запитів (1 - увімкнено)
slow-query-log=1

# Файл для логу повільних запитів
slow_query_log_file="IKLHOME-slow.log"

# Поріг часу для повільних запитів (у секундах)
long_query_time=10

# Файл для логу помилок
log-error="IKLHOME.err"

# === Налаштування реплікації ===

# Бінарне логування (необхідне для реплікації)
log-bin="IKLHOME-bin"

# Унікальний ідентифікатор сервера
server-id=1

# === Загальні налаштування ===

# Регістронезалежні імена таблиць (1 - увімкнено)
lower_case_table_names=1

# Каталог для операцій LOAD DATA та SELECT INTO OUTFILE
secure-file-priv="C:/ProgramData/MySQL/MySQL Server 8.0/Uploads"

# Максимальна кількість одночасних підключень
max_connections=151

# Кількість таблиць, які можуть бути відкриті одночасно
table_open_cache=2000

# Максимальний розмір тимчасових таблиць у пам’яті
tmp_table_size=104M

# Кількість потоків у кеші для повторного використання
thread_cache_size=10

# === Налаштування MyISAM ===
```

```
# Максимальний розмір файлу для сортування таблиць MyISAM
myisam_max_sort_file_size=100G

# Розмір буферу для сортування MyISAM
myisam_sort_buffer_size=199M

# Розмір буферу для індексів MyISAM
key_buffer_size=8M

# === Налаштування буферів читання ===

# Буфер для послідовного читання (запити без індексів)
read_buffer_size=64K

# Буфер для випадкового читання (ORDER BY, сортування)
read_rnd_buffer_size=256K

# === Налаштування InnoDB ===

# Запис журналу на диск після кожної транзакції (максимальна надійність)
innodb_flush_log_at_trx_commit=1

# Розмір буферного пулу InnoDB для кешування даних та індексів
innodb_buffer_pool_size=4G

# Розмір журналу транзакцій InnoDB
innodb_log_file_size=48M
```

2.4. Інструменти для роботи з *MySQL*

Існують різні засоби для роботи з *MySQL*, що спрощують взаємодію з базою даних: управління даними, виконання запитів та конфігурування серверних параметрів. Наведено найпоширеніші з них.

2.4.1. Робота з MySQL через командний рядок

Командний клієнт *mysql* – текстовий інтерфейс для взаємодії з СУБД *MySQL* через командний рядок. Цей інструмент особливо ефективний для адміністраторів баз даних та розробників, які працюють із серверами в режимі реального часу або потребують автоматизації процесів.

Основні переваги роботи через командний рядок :

- прямий доступ до сервера без проміжних інтерфейсів;

- можливість використання в скриптах та автоматизованих процесах;

- мінімальні накладні витрати на обробку запитів;

- повний контроль над усіма аспектами роботи з СУБД;

Для початку роботи необхідно виконати підключення до сервера командою:

```
mysql -u <ім'я_користувача> -p
```

Після автентифікації доступні всі стандартні операції з базою даних. Робота через командний рядок дозволяє не лише виконувати *SQL*-операції, а й контролювати стан сервера. Наприклад, перевіряти активні підключення та загальну статистику за допомогою команд

```
SHOW STATUS LIKE 'Threads_connected';
SHOW PROCESSLIST;
```

Командний клієнт також дозволяє аналізувати продуктивність запитів за допомогою вбудованого профілювальника, який є механізмом детального збору статистики виконання кожного запиту.

Після активації профілювання командою `SET profiling = 1;` сервер збирає інформацію про тривалість різних етапів обробки *SQL*-команди: парсинг, оптимізацію та виконання. Команда `SHOW PROFILES;` виводить список профілів із зазначенням часу виконання кожного запиту, а `SHOW PROFILE FOR QUERY <n>;` дозволяє побачити розподіл часу за окремими етапами. Це дає змогу виявити вузькі місця та оптимізувати найповільніші запити.

Оптимізація таблиць здійснюється безпосередньо через команди:

```
OPTIMIZE TABLE sensor_data;
ANALYZE TABLE sensor_data;
```

Для роботи з метаданими можна отримувати детальну інформацію про структуру бази даних:

```
SHOW INDEX FROM sensor_data;
SHOW TABLE STATUS LIKE 'sensor_data';
```

Налаштування параметрів сервера можна здійснювати під час сесії. Наприклад, для перегляду та зміни параметрів буферного пулу InnoDB:

```
SHOW VARIABLES LIKE 'innodb_buffer_pool%'; SET
GLOBAL innodb_buffer_pool_size = 2147483648;
```

Для пакетного режиму роботи можна виконувати *SQL*-скрипти з файлів, наприклад:

```
mysql -u user -p database < script.sql
```

Інтерактивний режим з підказками включає автодоповнення команд та історію запитів, особливо при використанні *MySQL* з підтримкою *readline*.

Досвідчені користувачі можуть використовувати додаткові можливості:

! command – виконання системних команд без виходу з mysql;

\G – вертикальний формат виводу результатів;

\P – налаштування програми для перегляду результатів;

\R – зміна формату підказки;

\s – відображення статусу сервера.

Робота через командний рядок особливо ефективна під час виконання масових операцій з даними, налагодження продуктивності сервера, автоматизації адміністративних завдань, роботи з серверами через *SSH*-з'єднання та виконання операцій резервного копіювання та відновлення.

2.4.2. Графічний клієнт MySQL Workbench

MySQL Workbench є офіційним графічним інструментом для роботи з системою управління базами даних *MySQL,* що забезпечує зручний візуальний інтерфейс для адміністрування, проєктування, оптимізації та моніторингу. На відміну від командного рядка (*CLI*), де взаємодія відбувається виключно через текстові команди, *Workbench* орієнтований на користувачів, які надають перевагу графічному середовищу з меню, вкладками та інтерактивними панелями.

Workbench дозволяє створювати й редагувати бази даних, проєктувати структуру таблиць, виконувати *SQL*-запити та керувати обліковими записами користувачів. Також доступні засоби для резервного копіювання, імпорту й експорту даних, аналізу продуктивності та візуалізації запитів. Однією з ключових функцій є підтримка *Enhanced Entity-Relationship* (*EER*) *Diagram*, яка дає змогу створювати та редагувати логічну структуру бази даних у вигляді блок-схеми.

MySQL Workbench можна інсталювати як компонент *MySQL Installer*, що включає сервер, клієнтські утиліти та документацію, або як окремий програмний модуль через standalone-інсталятор, доступний на офіційному сайті. Після встановлення та запуску *Workbench* користувач має підключитися до сервера баз даних, вказавши необхідні параметри з’єднання — хост, порт, ім’я користувача та пароль. Після встановлення з’єднання відкривається доступ до всіх функцій адміністрування.

Основна робота з даними виконується у вкладці *SQL Editor*. Тут можна писати та запускати *SQL*-запити із підтримкою автодоповнення та підсвічування синтаксису. Результати запитів відображаються у табличному вигляді з можливістю сортування, фільтрації та експорту у формат *CSV*. Доступна багатовкладковість і можливість виконання лише виділеного фрагмента коду. У разі помилки редактор відображає повідомлення з описом проблеми, а нижня панель *Output* дозволяє простежити історію виконаних запитів.

Workbench також підтримує візуальне проєктування баз даних за допомогою *EER*-діаграм. Користувач може створювати нові таблиці, встановлювати між ними зв'язки, змінювати структуру шляхом перетягування елементів та генерувати відповідний *SQL*-код. Усі об'єкти можна документувати через додавання описів і коментарів.

Функції адміністрування сервера реалізовано через спеціальний розділ *Server Administration*. На вкладці *Server Status* відображаються основні показники – кількість активних підключень, обсяг використаної пам'яті, навантаження на процесор і загальний стан системи. Через розділ *Users* and *Privileges* керуються облікові записи користувачів і права доступу. У розділі *Options File* адміністратор має змогу змінювати системні параметри, зокрема обсяг кешу, обмеження кількості підключень та параметри логування. Перегляд журналів сервера реалізовано через вкладку *Server Logs*, яка дозволяє аналізувати помилки, попередження та історію подій. У розділі *Client Connections* надається інформація про активні сесії, виконувані запити й їхній поточний статус. За потреби адміністратор може завершити завислі процеси.

Для збереження даних і відновлення системи використовуються функції *Data Export* та *Data Import/Restore*. Вони дозволяють експортувати або імпортувати таблиці, схеми та цілі бази даних у форматах *SQL* або *CSV*. Підтримується також відновлення з дампів і можливість вибіркового перенесення даних.

Workbench включає засоби для моніторингу продуктивності, що є критично важливим для підтримання стабільної роботи сервера. У вкладці *Performance Dashboard* відображається завантаження процесора, обсяг використаної оперативної пам'яті, кількість з'єднань та швидкість виконання запитів. Розділ *Query Statistics* надає детальну статистику: частоту виконання запитів, час відповіді та ресурсоспоживання.

Для аналізу виконання *SQL*-запитів *Workbench* пропонує два інструменти: *EXPLAIN Plan* у вигляді таблиці та *Visual Explain* у формі графічної діаграми. Обидва дозволяють дослідити, які індекси використовуються, скільки рядків

обробляється, та виявити повільні або неефективні запити. Візуалізований режим особливо корисний для аналізу складних сценаріїв з багатьма таблицями. Якщо запит виконується повним скануванням таблиці, є змога створити індекс для оптимізації.

У вкладці *Performance Reports* доступні автоматично сформовані аналітичні звіти з рекомендаціями щодо усунення виявлених проблем. Інтерфейс *Client Connections* служить для моніторингу поточних з'єднань і швидкої діагностики процесів. Налаштування параметрів продуктивності здійснюється у *Options File*, де можна встановити обсяг кешу, обмеження на кількість підключень та інші параметри, що впливають на ефективність системи.

Для перенесення баз даних з інших систем *Workbench* надає інструмент *Database Migration Wizard.* Він підключається до джерела даних, проводить аналіз структури, конвертує типи даних, створює схему у форматі *MySQL* і переносить записи з перевіркою цілісності. За потреби користувач має змогу змінити правила конвертації або виключити окремі об'єкти зі схеми.

Завдяки широкому набору функцій *MySQL Workbench* є універсальним середовищем для адміністраторів і розробників. Він поєднує у собі інструменти для створення, тестування, оптимізації та супроводу баз даних. У великих інфраструктурах, де потрібна автоматизація та віддалений доступ, його доцільно комбінувати з командним рядком mysql. Такий підхід дозволяє ефективно використовувати переваги кожного з інструментів, підвищуючи надійність та контроль над системою.

2.4.3. Особливості роботи з великими обсягами даних

Під час обробки великих обсягів даних, а також у ситуаціях, що потребують роботи в реальному часі, важливо враховувати як продуктивність інструментів, так і зручність використання. Командний рядок (*CLI*) та графічне середовище *MySQL Workbench* мають суттєві відмінності у підходах до взаємодії з базою даних, кожен з яких має свої переваги та обмеження.

Командний рядок забезпечує мінімальне споживання ресурсів і максимальну швидкість виконання операцій. Він зручний для запуску скриптів, пакетної обробки, імпорту та експорту даних, а також для роботи зі службовими утилітами (*mysql, mysqldump*, *mysqladmin*).

У режимі реального часу *CLI* дозволяє оперативно виконувати діагностику стану сервера, переглядати активні з'єднання, блокування й навантаження через команди `SHOW PROCESSLIST`, `SHOW STATUS`, `EXPLAIN`, `SHOW ENGINE INNODB STATUS` тощо.

При роботі з великими обсягами даних *CLI* демонструє стабільність і дозволяє уникнути перевантаження буфера терміналу шляхом перенаправлення результатів у файл, наприклад за допомогою оператора `INTO OUTFILE`.

MySQL Workbench, своєю чергою, орієнтований на користувачів, яким важлива візуалізація та інтерактивна робота. Він має розвинені можливості моніторингу, зокрема вкладки *Query Monitor*, *Performance Reports* та *Query Statistics*, що відображають продуктивність запитів, обсяг оброблених даних і навантаження на систему в режимі реального часу.

Для аналізу ефективності *SQL*-запитів *Workbench* надає два інструменти: табличний *EXPLAIN Plan* і графічну діаграму *Visual Explain*, що допомагають виявити проблемні ділянки у складних запитах, таких як вкладення або об'єднання.

Однак графічний інтерфейс є більш вимогливим до ресурсів, особливо при обробці великих вибірок. У випадках, коли результат запиту перевищує сотні тисяч рядків, можуть виникати затримки, уповільнення або навіть зависання інтерфейсу. Для запобігання таким ситуаціям рекомендується обмежувати вибірку за допомогою параметра *LIMIT*, а результати експортувати у файл *CSV* безпосередньо з інтерфейсу *Workbench* для подальшої обробки в сторонніх застосунках, таких як *Excel*.

Таким чином, вибір між *CLI* і *MySQL Workbench* залежить від характеру завдань. Командний рядок доцільно використовувати для масових операцій,

роботи зі скриптами, обслуговування та аварійного реагування. *Workbench* краще підходить для візуального аналізу, оптимізації запитів і моделювання структури бази даних. У ситуаціях з високим навантаженням або великими обсягами даних рекомендовано комбінувати обидва підходи, забезпечуючи баланс між продуктивністю і зручністю керування.

2.4.4. Інші IDE для роботи з MySQL

Окрім стандартних інструментів на кшталт командного рядка *mysql* та *MySQL Workbench*, існує цілий ряд альтернативних середовищ розробки (*IDE*), які пропонують різні підходи до роботи з *MySQL*. Ці інструменти відрізняються функціоналом, інтерфейсом та спеціалізацією, що дозволяє підібрати оптимальне рішення під конкретні потреби.

Серед популярних *IDE* для роботи з *MySQL* можна виділити такі варіанти:

DBeaver – універсальний клієнт для роботи з різними СУБД, включаючи *MySQL*. Відмінною рисою є підтримка хмарних баз даних, інтуїтивний інтерфейс та потужні інструменти для експорту/імпорту даних. Особливо корисний для тих, хто працює з різними типами баз даних одночасно. Доступний у безкоштовній (*Community*) та комерційній (*Enterprise*) версіях.

HeidiSQL – легкий та швидкий клієнт для *Windows*, який пропонує зручний інтерфейс для адміністрування *MySQL*. Характеризується низьким споживанням ресурсів, простим експортом даних у різні формати та зручним редактором запитів. Ідеальний вибір для роботи на слабких машинах.

DataGrip від *JetBrains* – професійне комерційне середовище для роботи з базами даних. Пропонує розумне автодоповнення коду, рефакторинг *SQL*, інтеграцію з системами контролю версій та підтримку складних сценаріїв роботи. Особливо цінується розробниками за глибоку інтеграцію з іншими продуктами *JetBrains*.

phpMyAdmin – веб-інтерфейс для управління *MySQL*, який особливо популярний серед веб-розробників. Дозволяє керувати базами даних прямо з

браузера, що робить його зручним рішенням для швидких змін на віддалених серверах. Вимагає встановленого веб-сервера (*Apache* або *Nginx*) та *PHP*.

Navicat for *MySQL* – потужний комерційний інструмент з підтримкою візуального проектування баз даних, автоматизації завдань та розширених можливостей експорту. Добре підходить для команд, які працюють зі складними проектами. Доступний для Windows, macOS та Linux.

TablePlus – сучасний клієнт з мінімалістичним дизайном, який підтримує не лише *MySQL*, але й інші популярні СУБД. Відзначається високою швидкодією, зручним інтерфейсом для роботи з даними та підтримкою безпечних з'єднань. Доступний для macOS, Windows, Linux та iOS.

Кожен з цих інструментів має свої переваги: від простоти використання у випадку *HeidiSQL* до професійного підходу у *DataGrip*. Вибір конкретного *IDE* залежить від таких факторів, як складність проектів, необхідність спільної роботи, операційна система та індивідуальні уподобання розробника.

Для комплексної роботи з *MySQL* часто рекомендується освоїти кілька інструментів, використовуючи кожен для вирішення певного кола завдань. Наприклад, командний рядок *mysql* для автоматизації та скриптів, *MySQL Workbench* для проєктування та адміністрування, а один з альтернативних клієнтів (*DBeaver* або *DataGrip*) для повсякденної роботи з даними.

РОЗДІЛ 3. ОБРОБКА ДАНИХ У РЕАЛЬНОМУ ЧАСІ З *MySQL*

3.1. Підтримка баз даних реального часу

MySQL є потужною системою управління базами даних, що підтримує роботу з реальними даними завдяки своїм функціональним можливостям, які гарантують високу продуктивність, надійність та доступність інформації. Для забезпечення ефективного функціонування в режимі реального часу з великими обсягами даних у *MySQL* передбачено низку ключових механізмів.

Індексація дозволяє значно підвищити ефективність виконання запитів, створюючи індекси на важливих стовпцях. Реплікація допомагає розподілити навантаження між кількома серверами, підвищуючи доступність даних. Партіціонування дозволяє розділити великі таблиці на менші частини, що покращує управління даними. Оптимізація конфігурації, наприклад налаштування розміру буферного пулу *InnoDB*, допомагає збалансувати навантаження. Моніторинг та регулярне налаштування системи забезпечують стабільну роботу *MySQL*. Механізм *Change Data Capture* (*CDC*) дозволяє відслідковувати та захоплювати зміни даних у реальному часі.

Використання цих механізмів забезпечує стабільну та ефективну роботу *MySQL* з даними в реальному часі та великими обсягами інформації.

3.2. Індексація та оптимізація продуктивності запитів

Індекси є спеціальними структурами даних, які використовуються для прискорення пошуку та виконання запитів до бази даних. Вони дозволяють значно підвищити продуктивність, особливо при роботі з великими таблицями. У *MySQL* індекси створюються для одного або кількох стовпців таблиці з метою оптимізації операцій доступу до даних. За своєю логікою індекси нагадують зміст у книзі, що вказує, де можна швидко знайти необхідну інформацію. Хоча індекси фізично зберігаються окремо від основних таблиць, вони тісно пов’язані з їхніми даними. Завдяки цьому індекси скорочують час,

необхідний для виконання запитів, оскільки замість послідовного перегляду всіх записів забезпечується прямий доступ до потрібних значень.

Принцип роботи індексів у *MySQL* базується на використанні відсортованої структури. Наприклад, при створенні індексу для числового стовпця використовується деревоподібна структура, зазвичай це B-дерево, яка дозволяє швидко знайти відповідні значення. Завдяки цьому база даних під час виконання запитів не переглядає всі рядки таблиці, а звертається до індексу для визначення місцезнаходження необхідних записів.

У *MySQL* підтримується кілька типів індексів, які відрізняються своєю функціональністю. Первинні індекси (*Primary Key Indexes*) створюються автоматично для стовпців, що визначені як ключі таблиці, і забезпечують унікальність значень. Унікальні індекси (*Unique Indexes*) гарантують, що в стовпці не буде повторів, що особливо корисно, наприклад, для електронних адрес. Існують також звичайні індекси, які покращують швидкість пошуку, проте не забезпечують унікальність. Повнотекстові індекси застосовуються при роботі з великими обсягами текстових даних, зокрема в описах товарів або статтях. Крім того, можна створювати індекси на кілька стовпців, що дає змогу ефективно виконувати запити за кількома умовами одночасно.

Індекси створюються за допомогою команди `CREATE INDEX` або через параметри при створенні таблиці. Наприклад, команда

```
CREATE INDEX idx_email ON users (email);
```

створює індекс на стовпець `email` в таблиці `users`, що дозволяє пришвидшити пошук за цим стовпцем. Команда

```
CREATE INDEX idx_name_email ON users (username, email);
```

створює індекс на двох стовпцях *username* і *email*, що прискорює пошук за обома цими стовпцями.

Наступний код створює таблицю *users* з кількома полями:

`user_id` є унікальним ідентифікатором користувача, що автоматично збільшується і виконує роль первинного ключа (*Primary Key*);

`username` – це ім’я користувача у вигляді рядка довжиною до 50 символів;

`email` – адреса електронної пошти, ка повинна бути унікальною (*Unique*), щоб запобігти дублюванню;

`created_at` – часовий штамп створення запису з поточним часом за замовчуванням.

Крім того, під час створення таблиці одразу визначено індекс `idx_username` на стовпець `username`, що дозволяє пришвидшити пошук і сортування за цим полем.

```
CREATE TABLE users (
  user_id INT AUTO_INCREMENT PRIMARY KEY,
  username VARCHAR(50),
  email VARCHAR(100) UNIQUE,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  INDEX idx_username (username)
);
```

Використання індексів особливо виправдане у випадках, коли запити часто містять умови фільтрації, сортування або з’єднання. Вони є доцільними для полів, за якими здійснюється регулярний пошук, сортування або групування, а також у тих випадках, коли кілька полів часто використовуються спільно у запитах. Проте ефективність індексу може зменшуватись, якщо стовпець містить велику кількість однакових значень.

Незважаючи на переваги, індекси мають і певні недоліки. Зокрема, вони можуть знижувати швидкість запису та оновлення даних, оскільки при кожній зміні необхідно оновлювати і відповідні індекси. Крім того, індекси займають додаткову пам’ять, тому важливо зважено підходити до їх використання та уникати надмірного створення.

Щоб видалити індекс, використовується команда `DROP INDEX`. Наприклад, команда

```
DROP INDEX idx_email ON users;
```

видаляє індекс `idx_email` з таблиці `users`. Це може бути необхідно у випадках, коли індекс більше не використовується або негативно впливає на продуктивність при оновленні даних.

У цілому індекси є важливим інструментом підвищення продуктивності запитів у *MySQL*. Їх грамотне використання забезпечує ефективний доступ до даних і зменшує навантаження на систему, проте потребує ретельного аналізу доцільності впровадження у кожному конкретному випадку.

3.3. Кешування та прискорення обробки даних

Кешування в *MySQL* – це техніка зберігання результатів запитів або окремих фрагментів даних в оперативній пам’яті з метою підвищення швидкодії при повторному зверненні до тієї ж інформації. Завдяки кешуванню вдається суттєво зменшити час відповіді на запити, що особливо актуально при роботі з великими обсягами даних або в умовах високонавантажених сервісів.

Коли клієнт надсилає запит до бази даних, *MySQL* може зберегти результат виконання в оперативній пам’яті. При повторному надходженні подібного запиту система може отримати дані з пам’яті, що дозволяє значно зекономити ресурси та час.

У сучасних версіях *MySQL* (8.0+) основним механізмом кешування є *InnoDB Buffer Pool*, який зберігає як дані таблиць, так і індекси в оперативній пам’яті. Це пришвидшує обробку запитів на фізичному рівні зберігання.

Механізм *InnoDB Buffer Pool* налаштовується через конфігураційний файл *my.ini* (*Windows*) або *my.cnf* (*Linux*):

```
[mysqld]
innodb_buffer_pool_size = 4G
```

```
innodb_buffer_pool_instances = 4
```

Параметр `innodb_buffer_pool_size` визначає обсяг оперативної пам’яті, що виділяється для зберігання даних та індексів. Рекомендується виділяти 70-80% доступної оперативної пам’яті для виділених серверів баз даних.

Приклад: для сервера з 8 ГБ оперативної пам’яті:

`innodb_buffer_pool_size = 6G` (75% від 8 ГБ)

Для сервера з 32 ГБ оперативної пам’яті:

`innodb_buffer_pool_size = 24G` (75% від 32 ГБ)

Параметр `innodb_buffer_pool_instances` дозволяє розділити буферний пул на кілька екземплярів для зменшення конкуренції за ресурси у багатопоточних середовищах.

Після внесення змін необхідно перезапустити сервер бази даних.

Для перегляду налаштувань використовується команда:

```
SHOW VARIABLES LIKE 'innodb_buffer_pool%';
```

Для перегляду статистики ефективності:

```
SHOW STATUS LIKE 'Innodb_buffer_pool%';
```

Результат містить важливі метрики:

`Innodb_buffer_pool_read_requests` – загальна кількість логічних запитів на читання даних;

`Innodb_buffer_pool_reads` – кількість фізичних читань з диска (коли даних немає в кеші);

`Innodb_buffer_pool_pages_data` – кількість сторінок з даними у пулі;

`Innodb_buffer_pool_pages_free` – кількість вільних сторінок.

Співвідношення `Innodb_buffer_pool_reads` до `Innodb_buffer_pool_read_requests` показує ефективність кешування. Ідеально, коли це співвідношення менше 1%.

Для високонавантажених середовищ доречним є використання зовнішніх кешуючих систем, таких як *Redis* або *Memcached*.

Redis – високошвидкісна система типу «ключ-значення», яка ідеально підходить для кешування результатів частих запитів, сесійних даних та складних об'єктів. Підтримує структури даних (списки, множини, хеші) та автоматичне видалення застарілих даних (*TTL – Time To Live*).

Memcached – розподілена система кешування в пам'яті, ефективна для простих операцій кешування. Часто використовується для сесійних даних або повторюваних об'єктів у веб-додатках.

Інтеграція з *MySQL* здійснюється через проміжні шари на рівні прикладного коду або мікросервісів (*PHP*, *Python*, *Java*, *Node.js* тощо).

Кешування в *MySQL* має низку очевидних переваг. Воно підвищує швидкодію системи завдяки зменшенню часу виконання запитів, знижує навантаження на сервер, особливо в ситуаціях, коли запити дублюються, та сприяє масштабованості завдяки можливості інтеграції з зовнішніми кешуючими платформами.

Водночас воно має і свої обмеження. До них належить потреба у додаткових ресурсах оперативної пам'яті, необхідність правильного налаштування розміру буферного пулу для балансу між продуктивністю та використанням пам'яті, а також складність моніторингу ефективності кешування при використанні зовнішніх систем.

Таким чином, кешування є ефективним засобом підвищення продуктивності систем на базі *MySQL*. Його реалізація через *InnoDB Buffer Pool* та інтеграцію з зовнішніми рішеннями (*Redis*, *Memcached*) надає широкі можливості для адаптації до різних умов експлуатації.

3.4. Тригери та події для автоматизації завдань

Тригери та події є потужними інструментами в *MySQL,* які дозволяють автоматизувати певні завдання і реагувати на зміни в базі даних. Вони забезпечують автоматичне виконання операцій без необхідності втручання користувача чи програмного забезпечення.

3.4.1. Тригери в MySQL

Тригери є спеціальними збереженими процедурами, які автоматично виконуються при виконанні певних операцій над таблицею, таких як вставка, оновлення або видалення даних. Вони дозволяють реагувати на зміни в базі даних і виконувати додаткові дії, наприклад автоматичне оновлення інших таблиць, перевірку цілісності даних або логування змін.

Тригери в *MySQL* можуть бути прив’язані до таких операцій:

`BEFORE` – тригер виконується до виконання основної операції (вставка, оновлення або видалення).

`AFTER` – тригер виконується після виконання основної операції.

Синтаксис створення тригера:

```
CREATE TRIGGER trigger_name
BEFORE|AFTER INSERT|UPDATE|DELETE
ON table_name
FOR EACH ROW
BEGIN
    -- Дії, які потрібно виконати
END;
```

`trigger_name` – назва тригера.

{BEFORE | AFTER} – вказує, коли тригер має спрацьовувати : до (BEFORE) або після (AFTER) події.

{INSERT | UPDATE | DELETE} – вказує тип події, яка викликає тригер вставка (INSERT), оновлення (UPDATE) або видалення (DELETE).

table_name: Назва таблиці, до якої прив'язаний тригер.

FOR EACH ROW – вказує, що тригер спрацьовує для кожного рядка, який підпадає під дію події.

Уявімо, що у нас є таблиця products, і ми хочемо автоматично оновлювати поле updated_at кожного разу, коли запис змінюється.

```
CREATE TRIGGER update_product_timestamp
BEFORE UPDATE ON products
FOR EACH ROW
BEGIN
    SET NEW.updated_at = NOW();
END;
```

BEFORE UPDATE – тригер спрацьовує перед оновленням запису в таблиці products

NEW.updated_a – нове значення поля updated_at, яке буде записано після виконання оновлення.

NOW() – функція для отримання поточного часу.

Приклад тригера, що автоматично записує інформацію про видалені записи в окрему таблицю для аудиту.

```
CREATE TRIGGER log_deleted_user
AFTER DELETE ON users
FOR EACH ROW
BEGIN
    INSERT INTO deleted_users (user_id, username,
    deleted_at)
    VALUES (OLD.user_id, OLD.username, NOW());
END;
```

AFTER DELETE – тригер виконується після видалення запису з таблиці users.

OLD.user_id та OLD.username – старі начення запису перед видаленням.

У тригерах *MySQL* доступні спеціальні псевдонім и для доступу до даних:

NEW –доступ до нових значень рядка (використовується в INSERT та *UPDATE*);

OLD – доступ до старих значень рядка (використовується в UPDATE та DELETE)

Для перегляду існуючих тригерів використовується команда

```
SHOW TRIGGERS;
```

або для конкретної таблиці:

```
SHOW TRIGGERS WHERE `Table` = 'products';
```

Видалення тригера

```
DROP TRIGGER IF EXISTS trigger_name;
```

При роботі з тригерами важливо враховувати:

- Тригери можуть уповільнити операції вставки, оновлення та видалення;
- Занадто складна логіка в тригерах може ускладнити налагодження;
- Тригери не спрацьовують при операціях TRUNCATE TABLE;
- Рекурсивні тригери можуть призвести до безкінечного циклу

Тригери доцільно застосовувати насамперед для забезпечення цілісності даних. Слід уникати розміщення складної логіки в тригерах, оскільки це ускладнює налагодження та підтримку системи. Необхідно документувати призначення кожного тригера для полегшення подальшої роботи з базою даних. Важливо проводити тестування тригерів у тестовому середовищі перед

впровадженням у продуктивну систему. Потрібно враховувати вплив тригерів на продуктивність при роботі з великими обсягами даних, оскільки вони виконуються для кожного рядка і можуть суттєво уповільнити операції вставки, оновлення та видалення.

3.4.2. Події в MySQL

Події дозволяють автоматично виконувати *SQL*-запити через певні інтервали часу або в певні моменти. Це дозволяє виконувати періодичні завдання, такі як очищення тимчасових таблиць, агрегація даних або інші регулярні операції.

Синтаксис створення події**:**

```
CREATE EVENT event_name
ON SCHEDULE EVERY interval
{HOUR|MINUTE|DAY|WEEK| MONTH}
[STARTS timestamp] [ENDS timestamp]
[ON COMPLETION [NOT] PRESERVE]
DO
BEGIN
-- SQL –запити, які потрібно виконати
END;
```

`event_name` –назва події;

`EVERY interval` – інтервал виконання (наприклад, `EVERY 1 DAY`, `EVERY 10 MINUTE`) ;

`STARTS` –необов'язковий параметр, що вказує початок виконання події ;

`ENDS` –необов'язковий параметр, що вказує кінець виконання події ;

`ON COMPLETION PRESERVE` –зберігає подію після завершення (за замовчуванням події видаляються.

Припустимо, що в системі є таблиця *logs*, яка зберігає журнали подій. Щоденно необхідно автоматично очищати цю таблицю від записів старших за 30 днів.

```
CREATE EVENT cleanup_old_logs
ON SCHEDULE EVERY 1 DAY
STARTS '2025-01-11
DO
BEGIN
    DELETE FROM logs
    WHERE log_date<NOW()-INTERVAL 30 DAY;
    END;
```

`ON SCHEDULE EVERY 1 DAY` – подія виконується щодня;

`STARTS` – подія починає виконуватися о 2:00 ночі;

`DELETE FROM logs` видаляє записи з таблиці logs старші за 30 днів

Уявімо, що необхідно оновлювати статистику кожні 10 хвилин. Для цього створимо подію, яка оновлює відповідну таблицю.

```
CREATE EVENT update_statistics
ON SCHEDULE EVERY 10 MINUTE
DO
BEGIN
    UPDATE statistics SET last_updated = NOW();
    UPDATE statistics
    SET value = (SELECT COUNT(*)
    FROM users);
END;
```

Для використання подій в *MySQL* потрібно, щоб був увімкнений планувальник подій (*Event Scheduler*). За замовчуванням він може бути вимкнений.

Перевірка статусу планувальника:

```
SHOW VARIABLES LIKE 'event_scheduler';
```

Якщо результат буде *OFF*, потрібно увімкнути планувальник командою

```
SET GLOBAL event_scheduler = ON;
```

Для постійного увімкнення планувальника додайте до конфігураційного файлу my.ini (Windows) або my.cnf (Linux):

```
[mysqld] event_scheduler = ON
```

Перегляд подій:

```
SHOW EVENTS;
```

Перегляд детальної інформації:

```
SHOW CREATE EVENT cleanup_old_logs;
```

Тимчасове вимкнення події:

```
ALTER EVENT cleanup_old_logs DISABLE;
```

Увімкнення події:

```
ALTER EVENT cleanup_old_logs ENABLE;
```

Видалення події:

```
DROP EVENT IF EXISTS cleanup_old_logs;
```

При роботі з подіями важливо враховувати наступне:

- події виконуються на сервері бази даних, тому складні операції можуть вплинути на продуктивність;
- планувальник подій повинен бути постійно увімкнений для виконання подій;
- події не виконуються, якщо сервер вимкнений;
- рекомендується використовувати `STARTS` для планування подій на непікові години;
- для тривалих операцій краще використовувати зовнішні планувальники (*cron* у *Linux*, *Task Scheduler у Windows*).

Тригери і події є потужними інструментами для автоматизації завдань у *MySQL*, особливо для реалізації задач реального часу. Тригери дозволяють автоматично реагувати на зміни в таблицях, а події - виконувати періодичні

операції. Вони знижують навантаження на програму та забезпечують цілісність даних, однак важливо використовувати їх обережно, щоб не погіршити продуктивність бази даних.

3.5. Збережені процедури та функції

Збережені процедури та функції є важливими інструментами для обробки даних. Вони дозволяють організувати виконання складних операцій на стороні сервера, що допомагає оптимізувати продуктивність, зменшити навантаження на мережу та забезпечити більшу безпеку.

3.5.1. Збережені процедури

Збережені процедури в *MySQL* – це блоки *SQL*-коду, які виконуються на сервері бази даних за запитом. Вони можуть містити одну або кілька *SQL*-операцій і використовуються для виконання певних завдань, таких як вставка, оновлення, видалення даних, або навіть виконання складних обчислень.

Розглянемо загальний синтаксис збереженої процедури в *MySQL* :

```
DELIMITER //
CREATE PROCEDURE procedure_name (
  [IN|OUT|INOUT] parameter1 data_type,
  [IN|OUT|INOUT] parameter2 data_type
  ...
)
BEGIN
  -- Тіло процедури: один або декілька SQL-
операцій
END //
DELIMITER;
```

В списку параметрів можна вказувати напрямок передачі: `IN` (вхідний), `OUT` (вихідний) або `INOUT` (вхід-вихід).

Припустимо, необхідно створити процедуру, яка вставляє новий запис у таблицю `orders` з певними даними

```
DELIMITER //
CREATE PROCEDURE add_order(
  IN order_id INT,
  IN customer_id INT,
  IN order_date DATE
)
BEGIN
      INSERT INTO orders (order_id, customer_id,
      order_date)
      VALUES (order_id, customer_id, order_date);
END //

DELIMITER ;
```

`IN order_id INT, IN customer_id INT, IN order_date DATE` - це параметри, які приймаються процедурою;

`INSERT INTO orders` це основна *SQL*-операція, яка виконується;

`DELIMITER` використовується для зміни символу, який *MySQL* інтерпретує як кінець команди.

За замовчуванням цей символ крапка з комою (;). Однак, коли ви створюєте складні конструкції, такі як процедури або функції, які містять кілька команд *SQL*, вам потрібно тимчасово змінити цей символ, щоб *MySQL* не сприймав кожну крапку з комою всередині процедури як кінець команди.

Ця процедура приймає три вхідні параметри: `order_id`, `customer_id` та `order_date`, і вставляє новий запис у таблицю orders з цими значеннями.

Приклад виклику цієї процедури

```
CALL add_order(1, 123, '2025-01-11');
```

Цей виклик вставить новий запис у таблицю `orders` з `order_id` 1, `customer_id` 123 та `order_date` '2025-01-11'.

Створимо процедуру, яка розраховує середнє значення температури за певний період часу

```
DELIMITER //

CREATE PROCEDURE get_average_temp(
      IN start_time DATETIME,
      IN end_time DATETIME,
      OUT avg_temp FLOAT
)
BEGIN
  SELECT AVG(Value) INTO avg_temp
  FROM sensor_data
  WHERE timestamp BETWEEN start_time AND end_time
   AND sensor_type = 'Temperature';
END //

DELIMITER ;
```

Процедура називається get_average_temperature. Вона має три параметри: start_time (початковий час періоду), end_time (кінцевий час періоду) та avg_temp (вихідний параметр для збереження результату).

Процедура обчислює середнє значення температури (AVG(value)) з таблиці sensor_data. Враховуються лише записи, де timestamp знаходиться між start_time та end_time, і sensor_type дорівнює 'Temperature'.

Викликати зберігану процедуру get_average_temperature можна за допомогою команди CALL в *MySQL* . Ось приклад, як це зробити:

```
-- Встановлення значень для параметрів
SET @start_time = '2025-01-01 00:00:00';
SET @end_time = '2025-01-10 23:59:59'
SET @average_temperature =0;

-- Виклик процедури
CALL get_average_temperature (@start_time, @end_time,
@average_temperature);
```

```
-- Виведення результату
SELECT @average_temperature AS ‘Середня температура’;
```

Керування збереженими процедурами.

Перегляд списку процедур

```
SHOW PROCEDURE STATUS WHERE Db = ‘database_name’;
```

Перегляд коду процедури

```
SHOW CREATE PROCEDURE procedure_name;
```

Видалення процедури

```
DROP PROCEDURE IF EXISTS procedure_name;
```

Збережені процедури мають низку суттєвих переваг. Вони забезпечують покращену продуктивність завдяки одноразовій компіляції та багаторазовому виконанню. Використання процедур дозволяє зменшити мережевий трафік, оскільки замість багатьох окремих запитів виконується один виклик процедури. Процедури забезпечують централізовану логіку, яка зберігається на сервері бази даних, що полегшує її підтримку та оновлення. Важливою перевагою є підвищена безпека – можна надати доступ до виконання процедури без надання прямого доступу до таблиць. Крім того, процедури сприяють повторному використанню коду, оскільки їх можна викликати з різних додатків та модулів системи.

3.5.2. Збережені функції

Збережені функції схожі на процедури, але на відміну від процедур, функції повинні повертати значення. Вони використовуються для обчислень та виконання операцій, які можуть бути використані у виразах *SQL*.

Синтаксис створення зберіганої функції наступний**:**

```
DELIMITER //

CREATE FUNCTION function_name(param1 TYPE, param2
TYPE, ...)
RETURNS return_type
DETERMINISTIC
BEGIN
  -- SQL-операції
  RETURN value;
END //

DELIMITER ;
```

`RETURNS return_type` визначає тип даних, який повертає функція. `DETERMINISTIC` означає, що функція завжди повертає однаковий результат для однакових вхідних параметрів.

Наприклад, якщо необхцдно створити функцію, яка додає два числа:

```
DELIMITER //

CREATE FUNCTION add_numbers(a INT, b INT)
RETURNS INT
DETERMINISTIC
BEGIN
  RETURN a + b;
END //

DELIMITER ;
```

Ця функція `add_numbers` приймає два параметри a та b типу `INT` і повертає їх суму.

Приклад виклику цієї функції:

```
SELECT add_numbers(5, 10) AS result;
```

Цей виклик поверне результат 15.

Створимо процедуру, яка розраховує середнє значення температури за певний період часу

```
DELIMITER //

CREATE FUNCTION CalculateAverageTemperature(startTime
DATETIME, endTime DATETIME)
RETURNS FLOAT
DETERMINISTIC
BEGIN
  DECLARE avgTemp FLOAT;

  SELECT AVG(temperature) INTO avgTemp
  FROM SensorData
  WHERE ti*MES*tamp BETWEEN startTime AND endTime;

  RETURN avgTemp;
END //

DELIMITER ;
```

Функція `CalculateAverageTemperature` приймає два параметри startTime та endTime, які визначають період часу, за який потрібно розрахувати середнє значення температури. Функція повертає середнє значення температури за цей період.

Переваги використання збережених процедур і функцій включають покращення продуктивності, оскільки вони виконуються на сервері, що мінімізує необхідність передачі даних по мережі та зменшує час виконання складних операцій, що спрощує обслуговування та тестування; підвищення безпеки, оскільки дозволяє обмежити доступ до даних, даючи лише можливість виконувати конкретні операції через визначені процедури; та зменшення помилок, оскільки допомагають централізувати обробку даних, що зменшує ризик помилок у коді на рівні додатка. Недоліки та обмеження включають можливість погіршення продуктивності, якщо збережені процедури не оптимізовані, що може призвести до зниження продуктивності, особливо при

великій кількості викликів; підвищення складності підтримки, оскільки використання складних процедур та функцій може ускладнити підтримку та тестування системи, оскільки логіка зберігається безпосередньо в базі даних; та платформну залежність, оскільки збережені процедури і функції можуть бути специфічними для конкретної СУБД, що ускладнює переносимість додатка.

Підсумки: Збережені процедури та функції в *MySQL* є важливими інструментами для обробки даних і автоматизації бізнес-логіки. Вони дозволяють створювати ефективні та безпечні рішення для складних обчислень і операцій з базою даних. Важливо ретельно оптимізувати та тестувати ці об’єкти, щоб забезпечити високу продуктивність та безпеку системи.

3.6. Обробка великих обсягів даних у реальному часі

3.6.1. Розділення таблиць

Обробка великих обсягів даних у реальному часі, зокрема у *MySQL,* може значно виграти від використання розділення таблиць на логічні частини (*Partitioning*).

Partitioning – це спосіб фізичного розділення таблиці на менші частини (розділи), кожна з яких зберігається та обробляється окремо. Ці частини розподіляються на основі певного критерію (наприклад, діапазону дат, значень тощо), що дозволяє оптимізувати запити та покращити продуктивність роботи з великими обсягами даних. Ця технологія дозволяє працювати з великими обсягами даних більш ефективно, особливо в сценаріях, де таблиці містять мільйони або мільярди записів.

У *MySQL Partitioning* реалізовано як спосіб оптимізації зберігання та обробки великих обсягів даних. Ця технологія забезпечує прозорість для користувача, зберігаючи вигляд таблиці як єдиного цілого, незалежно від того, що запити фактично обробляються лише з відповідними розділами.

Партиціонування надає високу гнучкість, оскільки дані можуть бути розподілені за різними критеріями: діапазоном значень, списком можливих

значень або хеш-функцією. Також підтримується масштабованість, що дозволяє збільшувати кількість розділів відповідно до зростання обсягів даних.

MySQL підтримує чотири основні типи розділення даних.

Розділення за діапазоном значень (*RANGE Partitioning*) дозволяє розподіляти дані відповідно до вказаних діапазонів значень.

При розділенні за списком конкретних значень (*LIST Partitioning*) дані розподіляються згідно з визначеними списками.

Розділення за допомогою хеш-функції (*HASH Partitioning*) забезпечує рівномірний розподіл даних за допомогою хешування.

Розділення за ключами (*KEY Partitioning*) діє подібно до *HASH*, але використовує вбудовану хеш-функцію *MySQL*.

MySQL дозволяє створювати до 8192 розділів для однієї таблиці. *Partitioning* доступний для таблиць типу *InnoDB* та *NDB Cluster*. Однак не всі типи запитів, наприклад JOIN, можуть повністю використовувати переваги розділення

NDB (*NDB Cluster*) є розподіленою кластерною системою баз даних, що застосовується насамперед у системах із високими вимогами до доступності та масштабованості.

Partitioning у *MySQL* надає ряд переваг, зокрема підвищення швидкості виконання запитів, оскільки, якщо запит стосується лише певної підмножини даних (наприклад, за конкретний період часу), *MySQL* може пропустити нерелевантні розділи. Також розділення таблиць забезпечує кращу масштабованість, спрощуючи роботу з великими обсягами даних, і ефективне управління, завдяки чому архівування, резервне копіювання та видалення даних відбуваються швидше й простіше. Додатково, використання *HASH* і *KEY Partitioning* сприяє рівномірному розподілу даних між розділами, що допомагає збалансувати навантаження.

Partitioning у *MySQL* має свої обмеження та виклики, зокрема відсутність підтримки певних функцій: неможливість використання *FOREIGN KEY* у таблицях із розділенням та обмеження на використання деяких індексів, таких

як *FULLTEXT*. Управління розділеними таблицями є складним і вимагає ретельного планування, особливо якщо потрібно змінити критерії розподілу. Не всі запити, наприклад, складні *JOIN* або *UNION*, можуть ефективно використовувати переваги розділення. Також існує ризик збільшення накладних витрат, оскільки нерівномірний розподіл даних може негативно вплинути на продуктивність.

Partitioning у *MySQL* рекомендується використовувати, коли таблиці містять мільйони або мільярди записів, запити часто фільтрують дані за певними умовами (наприклад, часом, регіоном або типом пристрою), існує необхідність регулярно видаляти або архівувати старі дані, а також якщо потрібен рівномірний розподіл даних для балансування навантаження. Для роботи з даними, які надходять у реальному часі, наприклад, із датчиків через ПЛК, підключений до *SCADA*-системи, і записуються в *MySQL,* важливо організувати структуру таблиць та механізми їх розділення так, щоб забезпечити ефективну обробку великих обсягів інформації.

Далі наведено приклади створення таблиць із використанням Partitioning, які враховують особливості даних, що отримані від датчиків.

Для даних від датчиків, які надходять у реальному часі, зручно використовувати розділення таблиць за діапазоном часу, наприклад, по місяцях.

Приклад 1. Розділення за діапазоном часу (*RANGE Partitioning*).

Структура таблиці має вигляд:

```
CREATE TABLE sensor_data (
      sensor_id INT NOT NULL,
      reading_value DECIMAL(10, 2) NOT NULL,
      reading_time DATETIME NOT NULL)
) ENGINE=InnoDB
PARTITION BY RANGE (YEAR(reading_time) * 100 +
MONTH(reading_time)) (
      PARTITION p202301 VALUES LESS THAN (202302),
      PARTITION p202302 VALUES LESS THAN (202303),
      PARTITION p202303 VALUES LESS THAN (202304),
```

```
        PARTITION p_max VALUES LESS THAN MAXVALUE
);
```

У цьому прикладі дані розділяються за місяцями, використовуючи обчислення `YEAR(reading_time) * 100 + MONTH(reading_time)`, яке створює числовий код для кожного місяця (наприклад, 202301 для січня 2023 року).

Розділи організовані наступним чином:

`p202301` – записи за січень 2023 року,

`p202302` – за лютий 2023 року;

`p202303` – записи за березень 2023 року;

`p_max` – дані, які не підпадають під інші умови (наприклад, нові місяці).

Приклад 2. Розділення за хешем (*HASH Partitioning*).

Якщо дані від датчиків рівномірно розподілені за ідентифікаторами, можна використовувати розділення за хешем.

```
CREATE TABLE sensor_data (
      sensor_id INT NOT NULL,
      reading_value DECIMAL(10, 2) NOT NULL,
      reading_time DATETIME NOT NULL
) ENGINE=InnoDB
PARTITION BY HASH(sensor_id) PARTITIONS 8;
```

У цьому прикладі використовується хеш-функція для значення `sensor_id`, яка рівномірно розподіляє записи між 8 розділами. Дані кожного датчика потрапляють у певну партицію, залежно від результату хешування його `sensor_id`.

Приклад 3. Розділення за ключем (*KEY Partitioning*).

Розділення за ключем є подібним до *HASH*, але *MySQL* автоматично обчислює хеш

```
CREATE TABLE sensor_data (
      sensor_id INT NOT NULL,
      reading_value DECIMAL(10, 2) NOT NULL,
      reading_time DATETIME NOT NULL
) ENGINE=InnoDB
PARTITION BY KEY(sensor_id) PARTITIONS 4;
```

KEY Partitioning автоматично обчислює хеш для значення `sensor_id`, використовуючи внутрішню функцію *MySQL,* і розподіляє записи між 4 розділами. Відмінність від *HASH Partitioning* полягає в тому, що користувачеві не потрібно явно задавати хеш-функцію.

Приклад 4. Розділення за різними джерелами даних (*LIST Partitioning*).

Якщо необхідно обробляти дані окремо для кожного ПЛК, можна використовувати *LIST Partitioning*, де кожен розділ відповідає конкретному ПЛК.

```
CREATE TABLE sensor_data (
  id INT NOT NULL AUTO_INCREMENT,
  plc_id INT NOT NULL,      -- Ідентифікатор ПЛК
  sensor_id INT NOT NULL,  -- Ідентифікатор датчика
  signal_value DECIMAL(10, 2) NOT NULL,
  timestamp DATETIME NOT NULL,
  PRIMARY KEY (id, plc_id)
)
PARTITION BY LIST (plc_id) (
  PARTITION plc_1 VALUES IN (1),  -- Дані з ПЛК 1
  PARTITION plc_2 VALUES IN (2),  -- Дані з ПЛК 2
  PARTITION plc_3 VALUES IN (3),  -- Дані з ПЛК 3
  PARTITION plc_default VALUES IN (0) -- Решта ПЛК
);
```

Цей код створює таблицю `sensor_data`, яка зберігає дані, отримані від датчиків, підключених до програмованих логічних контролерів (ПЛК). Таблиця розділена на партиції залежно від ідентифікатора ПЛК (`plc_id`).

`id INT NOT NULL AUTO_INCREMENT` – унікальний ідентифікатор для кожного запису, який автоматично збільшується;

`plc_id INT NOT NULL` –ідентифікатор ПЛК, який вказує, з якого контролера надійшли дані;

`sensor_id INT NOT NULL` – ідентифікатор датчика, який надіслав дані;

`signal_value DECIMAL(10, 2) NOT NULL` – значення, яке було отримано з датчика;

`timestamp DATETIME NOT NULL` – час отримання даних від датчика;

`PRIMARY KEY (id, plc_id)` – первинний ключ включає унікальний ідентифікатор `id` і `plc_id`, що забезпечує швидкий доступ до даних у межах кожної партиції;

`PARTITION BY LIST (plc_id)` – таблиця розділена на партиції на основі значення `plc_id`, яке визначає, до якого ПЛК належать дані;

`PARTITION plc_1 VALUES IN (1)` – всі записи, у яких `plc_id = 1`, будуть зберігатися в партиції `plc_1`;

`PARTITION plc_2 VALUES IN (2)` – записи з `plc_id = 2` потрапляють у партицію `plc_2`;

`PARTITION plc_3 VALUES IN (3)` – відповідно, `plc_id = 3` зберігається в партиції `plc_3`;

`PARTITION plc_default VALUES IN (0)` – дані з іншими значеннями `plc_id`, які не відповідають жодному з перерахованих, зберігаються в партиції `plc_default`.

Приклад 4. Розділення за механізмом архівації (`ARCHIVE`).

Якщо виникає потреба зберігати історичні дані, які рідко використовуються, можна поєднати *Partitioning* із механізмом архівації *MySQL* Наприклад, активні дані можуть зберігатися в партиціях за останні кілька місяців, тоді як історичні дані доступні в архівних партиціях.

Наприклад основна таблиця може мати таку структуру

```
CREATE TABLE sensor_data (
        sensor_id INT NOT NULL,
        signal_value DECIMAL(10, 2) NOT NULL,
        timestamp DATETIME NOT NULL,
PRIMARY KEY (sensor_id, timestamp)
) ENGINE=InnoDB
PARTITION BY RANGE (YEAR(timestamp) * 100 +
MONTH(timestamp)) (
PARTITION active_202301
    VALUES LESS THAN (202302), -- Дані січень 2023
    PARTITION active_202302
    VALUES LESS THAN (202303), -- Дані лютий 2023
    PARTITION active_recent
    VALUES LESS THAN MAXVALUE -- Поточні дані
 );
```

Тоді структура архівної таблиці буде мати вигляд

```
CREATE TABLE sensor_data_archive (
    sensor_id INT NOT NULL,
    signal_value DECIMAL(10, 2) NOT NULL,
    timestamp DATETIME NOT NULL
) ENGINE=ARCHIVE;
```

`ENGINE=ARCHIVE` використовує механізм зберігання `ARCHIVE`, який автоматично стискає дані для зменшення займаного обсягу. Цей механізм оптимізований для зчитування. Таблиця ефективно обробляє запити `SELECT`, особливо якщо дані фільтруються за конкретними умовами (наприклад, діапазоном часу). Підтримує лише операції `INSERT` і `SELECT`. У таблицях типу `ARCHIVE` не можна виконувати `UPDATE` та `DELETE`, тому зміна або видалення даних неможливі. `ARCHIVE` не підтримує індекси, що може уповільнити виконання запитів для великих обсягів даних.

За допомогою наступного *SQL*-запиту можна переміщати старі дані в архив:

```
-- Переносимо дані за січень 2023 в архів
```

```
INSERT INTO sensor_data_archive (sensor_id,
    signal_value, timestamp)
SELECT sensor_id, signal_value, timestamp
FROM sensor_data
WHERE timestamp < '2023-02-01';

-- Видаляємо застарілі дані з основної таблиці
ALTER TABLE sensor_data DROP PARTITION active_202301;
```

Якщо дані надходять постійно, необхідно регулярно додавати нові партиції

```
ALTER TABLE sensor_data ADD PARTITION (
PARTITION active_202303 VALUES LESS THAN (202304)
);
```

Розглянемо декілька прикладів запитів, які демонструють, як отримати збережені дані з таблиць, що використовують *Partitioning* у *MySQL* :

Приклад 5. Отримання даних за певний період часу

```
RANGE Partitioning за місяцями:
SELECT *
FROM sensor_data
PARTITION(p_202301, p_202302) -- Січень та лютий 2023
WHERE timestamp BETWEEN '2023-01-01' AND '2023-02-
28';
```

Запит отримує дані тільки з певних партицій (`p_202301`, `p_202302`), які охоплюють січень і лютий 2023 року.

Приклад 6. Отримання даних за конкретним ПЛК якщо таблиця має *LIST Partitioning* за ПЛК

```
SELECT *
FROM sensor_data
PARTITION(plc_1, plc_2) -- Дані лише з ПЛК 1 та ПЛК 2
WHERE plc_id IN (1, 2);
```

Запит отримує дані лише з певних партицій (`plc_1`, `plc_2`), які відповідають ПЛК з ідентифікаторами 1 і 2, ігноруючи інші партиції.

Приклад 7. Отримання даних з усіх активних партицій якщо таблиця має *RANGE Partitioning* без обмеження за місяцем

```
SELECT *
FROM sensor_data
PARTITION(p_recent) -- Дані з актуальних партицій
WHERE timestamp > '2023-01-01';
```

Запит використовує тільки одну активну партицію (`p_recent`), яка включає всі нові дані, що надходять після 1 січня 2023 року.

3.6.2. Оптимальні SQL-оператори для SCADA

При роботі з *MySQL* у режимі реального часу особливо важливо ретельно обирати *SQL*-оператори. Невідповідні запити можуть значно знизити продуктивність, збільшити навантаження на сервер і спричинити проблеми з масштабованістю та доступністю бази даних. Розглянемо оператори, які рекомендується застосовувати у *SCADA*-системах, а також ті, яких слід уникати.

Використання `SELECT` з індексами значно підвищує швидкість вибірки даних. Наприклад, запит

```
SELECT * FROM sensor_data WHERE sensor_id = 102;
```

буде виконуватися ефективно, якщо поле `sensor_id` індексоване. Індекс дозволяє уникнути повного сканування таблиці, що особливо важливо при великих обсягах даних, характерних для систем моніторингу технологічних процесів.

Обмеження кількості рядків через оператор `LIMIT` допомагає знизити навантаження на сервер і пришвидшити обробку запитів. Так, запит

```
SELECT * FROM event_log ORDER BY event_time
```

```
DESC LIMIT 50;
```

повертає лише 50 останніх подій, що дозволяє швидко отримати актуальну інформацію без зайвих витрат ресурсів.

При необхідності виконувати об’єднання таблиць (*JOIN*) слід подбати про індексацію полів, за якими здійснюється з’єднання. Це суттєво прискорює обробку запитів у випадках, коли дані пов’язані між собою. Наприклад наступний запит

```
SELECT d.sensor_id, d.value, m.machine_name
FROM sensor_data d
JOIN machines m ON d.machine_id = m.machine_id
WHERE m.department = ‘Production’;
```

буде ефективним за умови індексації `machine_id` в обох таблицях.

Для операцій запису, що містять кілька змін одночасно, слід використовувати транзакції. Це гарантує цілісність даних і запобігає неконсистентності при паралельній обробці. Приклад транзакції:

```
START TRANSACTION;

INSERT INTO sensor_data (sensor_id, value, timestamp)
VALUES (102, 45.7, NOW());
UPDATE machine_status SET status = ‘Active’ WHERE
machine_id = 5;

COMMIT;
```

Оператори `UPDATE` мають виконуватися з використанням індексованих полів у `WHERE`, щоб уникнути повного блокування таблиці. Наприклад,

```
UPDATE sensor_data SET value = 50.3
WHERE
    sensor_id = 102
AND
    timestamp = ‘2025-08-09 10:00:00’;
```

спрацює значно швидше при наявності індексів на `sensor_id` та `timestamp`.

Використання агрегатних функцій (`COUNT`, `SUM`, `AVG`, `MIN`, `MAX`) корисне для підсумкової обробки даних, але вимагає індексації полів у `GROUP BY`. Так запит

```
SELECT sensor_id, AVG(value)
FROM sensor_data
WHERE timestamp BETWEEN NOW() - INTERVAL 1 HOUR AND
NOW()
GROUP BY sensor_id;
```

буде оптимальним, якщо `sensor_id` та `timestamp` індексовані.

3.6.3. Недоцільні оператори для режиму реального часу

У роботі з *MySQL* у режимі реального часу в *SCADA*-системах слід уникати певних операторів і типів запитів, які можуть суттєво знизити продуктивність і призвести до надмірного навантаження на сервер.

Використання `SELECT *` без фільтрації. Запити, що повертають всі поля без вибіркового обмеження, можуть генерувати великі обсяги даних, що призводить до зайвого навантаження на мережу і сервер. Наприклад, запит

```
SELECT * FROM sensor_data;
```

буде надмірно витратним, особливо якщо таблиця містить мільйони записів.

Рекомендується чітко вказувати необхідні поля, наприклад,

```
SELECT sensor_id, value, timestamp FROM sensor_data;
```

Запити без використання індексів, що призводять до повного сканування таблиці. Особливо це стосується фільтрації за полями без індексації. Наприклад,

```
SELECT * FROM alarms WHERE status = 'active';
```

буде повільним, якщо поле `status` не індексоване. У такому випадку рекомендується додати індекс або використовувати додаткові умови, що звужують вибірку.

Операції оновлення (`UPDATE`) або видалення (`DELETE`) без умови `WHERE`. Такі запити оновлюють або видаляють усі записи в таблиці, що створює великий обсяг роботи і часто не є необхідним. Приклад

```
UPDATE machine_status SET status = 'offline';
```

оновлює всі записи, що може викликати тривалі блокування таблиці та суттєве навантаження. Важливо завжди застосовувати умови для точкового впливу.

Транзакції, що блокують великі таблиці або обробляють великий обсяг даних. Масивні транзакції можуть викликати значні затримки та блокування інших операцій. Наприклад,

```
START TRANSACTION;
UPDATE production_data
SET parameter =  'new_value';
COMMIT;
```

без обмеження діапазону оновлення може призвести до блокування всієї таблиці. Слід оптимізувати транзакції, поділяти великі операції на менші.

Використання функцій у `WHERE`, які не підтримують індекси. Наприклад запит

```
SELECT * FROM sensor_data
WHERE YEAR(timestamp) = 2025;
```

не використовує індекси за `timestamp` і змушує виконувати повне сканування таблиці. Рекомендується переписувати умови так, щоб уникати застосування функцій до індексованих полів, наприклад,

```
SELECT * FROM sensor_data
WHERE timestamp
BETWEEN '2025-01-01' AND '2025-12-31';
```

Запити з надмірною агрегацією без індексів або з великими обсягами даних. Складні групування і підрахунки, такі як

```
SELECT sensor_id, COUNT(*) FROM sensor_data GROUP BY
sensor_id HAVING COUNT(*) > 1000;
```

можуть викликати суттєве навантаження, якщо таблиця велика і не оптимізована.

Підсумовуючи, для забезпечення ефективної роботи *MySQL* у режимі реального часу необхідно уникати повного сканування таблиць, вибірково опрацьовувати лише необхідні поля, завжди використовувати індекси, а також розбивати складні операції на оптимальні частини. Такі практики дозволять підтримувати стабільну продуктивність бази даних у системах автоматизації технологічних процесів.

РОЗДІЛ 4. РЕПЛІКАЦІЯ ТА КЛАСТЕРИЗАЦІЯ В *MYSQL*

4.1. Основи реплікації та кластеризації

У сучасних інформаційних системах, особливо під час роботи з великими обсягами даних і в режимі реального часу, ефективне керування даними є критично важливим завданням. Одними з ключових методів забезпечення надійності, продуктивності та масштабованості таких систем є реплікація та кластеризація.

Реплікація – це процес створення та підтримання копій даних на різних серверах або вузлах системи. Основна її мета полягає у підвищенні доступності даних, забезпеченні відмовостійкості та розподілі навантаження між серверами.

Кластеризація – це метод об’єднання серверів у групи (кластери) з метою підвищення продуктивності, відмовостійкості та ефективності розподілу навантаження.

Обидві технології застосовуються для підвищення продуктивності, масштабованості та надійності систем, що працюють із великими обсягами даних або в режимі реального часу. Зокрема, бази даних у системах *SCADA* активно використовують реплікацію та кластеризацію для забезпечення безперебійного збору, обробки й зберігання даних про технологічні процеси.

4.2. Реплікація *Master-Slave* і *Master-Master*

Реплікація в *MySQL* – це процес копіювання даних з однієї бази даних (*master*) до однієї або кількох реплік (*slaves*) з метою синхронізації та резервування даних у реальному часі. Вона забезпечує наявність актуальних копій даних на різних серверах, підвищує доступність і продуктивність системи та створює можливість резервного копіювання. Реплікація є основою для побудови розподілених баз даних, що дозволяє ефективно масштабувати ресурси та підвищувати стійкість до збоїв. *MySQL* підтримує різні типи реплікації, зокрема асинхронну та синхронну.

Асинхронна реплікація дає змогу головному серверу (*master*) продовжувати обробку транзакцій без очікування підтвердження від реплік (*slaves*). Це забезпечує високу продуктивність і пропускну здатність системи. Зміни на майстрі записуються до журналу (*binary log*), після чого репліки зчитують їх і застосовують. Передавання змін відбувається із затримкою, що може спричиняти тимчасову неузгодженість даних, особливо за високого навантаження або під час виконання великих транзакцій. Такий підхід доцільний для сценаріїв, де критичною є швидкодія, зокрема коли репліки використовуються для операцій читання, аналітики або збору даних із систем *SCADA*. У контексті *SCADA* асинхронна реплікація дає змогу швидко передавати дані про стан обладнання чи аварійні події до *MySQL* без суттєвого впливу на роботу основної системи.

Синхронна реплікація передбачає, що зміни на майстрі не вважаються підтвердженими, доки вони не будуть успішно записані на всіх репліках. Це забезпечує узгодженість даних, але збільшує затримки під час запису. Через це синхронна реплікація зазвичай менш придатна для систем із високими вимогами до швидкодії, зокрема для *SCADA*. Водночас вона є доречною для збереження критично важливих даних, наприклад архівів подій і конфігурацій обладнання, де пріоритетом є надійність.

Роль реплікації у *SCADA*-системах полягає в забезпеченні постійного оновлення бази даних для безперервного моніторингу та аналітики технологічних процесів. Реплікація допомагає масштабувати інфраструктуру та балансувати навантаження, дозволяючи додавати додаткові сервери для обробки запитів на читання. У разі відмови основного сервера один із вузлів-реплік може бути підвищений до ролі майстра, що мінімізує час простою. Крім того, реплікація дає змогу підтримувати резервні копії на віддалених серверах, забезпечуючи збереження інформації та швидше відновлення після збоїв.

Реплікація в *MySQL* може бути реалізована за різними схемами, зокрема *Master–Slave та Master–Master*.

Схема *Master–Slave* є поширеним варіантом, за якого один сервер (*master*) виконує операції запису, а інші сервери (*slaves*) отримують зміни та використовуються переважно для читання. Репліки синхронізуються з майстром шляхом відтворення змін із його журналу транзакцій. Така конфігурація підвищує масштабованість операцій читання, знижує навантаження на головний сервер і забезпечує вищу доступність у разі збою за умови налаштованого переключення ролей.

Схема *Master–Master* передбачає, що обидва сервери можуть виконувати операції читання і запису, синхронізуючи зміни між собою в обох напрямках. Це забезпечує рівномірніший розподіл навантаження та підвищену відмовостійкість, оскільки збій одного вузла не призводить до повної зупинки роботи системи. Водночас налаштування *Master–Master* потребує уваги до уникнення конфліктів під час запису, що зазвичай досягається розподілом зон відповідальності за дані або застосуванням механізмів узгодження змін у СУБД.

4.2.1. Налаштування реплікації Master-Slave у середовищі Windows

Для налаштування сервера *Master* необхідно внести зміни у конфігураційний файл *MySQL* (зазвичай він називається my.ini і знаходиться у папці *MySQL,* наприклад, *C:\ProgramData\MySQL \MySQL Server X.X\my.ini*).

У розділі [*mysql*d] необхідно додати наступні параметри:

```
server-id = 1
log-bin= mysql-bin
binlog-do-db= <ім'я_бази>
```

`server-id=1` – унікальний ідентифікатор сервера, який має бути різним для кожного сервера в системі реплікації;

`log-bin= mysql-bin` – увімкнення бінарного журналу (binary log) для фіксації змін у базі даних;

`binlog-do-db= <ім'я_бази>` – необов'язковий параметр, який дозволяє реплікувати лише вказану базу даних.

Після внесення змін необхідно перезапустити сервер *MySQL*.

Далі потрібно підключитися до *MySQL* і створити користувача для реплікації

```
CREATE USER 'replica_user'@'%' IDENTIFIED BY
'password';
GRANT REPLICATION SLAVE ON *.* TO 'replica_user'@'%';
FLUSH PRIVILEGES;
```

Перевірити стан майстра можна за допомогою команди:

```
SHOW MASTER STATUS;
```

MySQL > `SHOW MASTER STATUS;`

```
mysql> SHOW MASTER STATUS;
+------------------+----------+--------------+------------------+-------------------+
| File             | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |
+------------------+----------+--------------+------------------+-------------------+
| mysql-bin.000001 | 154      | my_database  |                  |                   |
+------------------+----------+--------------+------------------+-------------------+
1 row in set (0.00 sec)
```

Значення *File* і *Position*, отримані з цього запиту, будуть потрібні для налаштування зв'зку між серверами.

Для налаштування сервера *Slave* у цьому ж файлі конфігурації (`my.ini`) необхідно додати такі параметри в розділ `[mysqld]`:

```
server-id=2
relay-log=relay-bin
log-bin = mysql-bin
read-only=1
```

У цому файлі

`server-id = 2` – унікальний ідентифікатор сервера *Slave* (він має відрізнятися від `server-id` на сервері *Master*);

`relay-log = relay-bin` – увімкнення `relay log` для зберігання змін, отриманих від сервера *Master*;

`log-bin = mysql-bin` – необов'язкове увімкнення бінарного журналу на *Slave*, якщо він також буде джерелом реплікації для інших вузлів;

`read-only = 1` – забороняє запис на *Slave* (окрім користувачів із привілеями `SUPER`).

Після внесення змін також потрібно перезапустити сервер *MySQL*.

Для налаштування з'єднання між *Slave* і *Master* у *MySQL,* необхідно виконати команду:

```
CHANGE MASTER TO
 MASTER_HOST='ip_адреса_майстра',
 MASTER_USER='replica_user',
 MASTER_PASSWORD='password',
 MASTER_LOG_FILE='MySQL -bin.000001',
 MASTER_LOG_POS=154;
```

Запуск процесу реплікації виконується командою

```
START SLAVE;
```

Перевірка статусу реплікації

```
SHOW SLAVE STATUS\G;

       Slave_IO_State: Waiting for master to
send event
         Master_Host: 192.168.1.100
         Master_User: replica_user
         Master_Port: 3306
        Connect_Retry: 60
      Master_Log_File: MySQL -bin.000001
  Read_Master_Log_Pos: 154
       Relay_Log_File: relay-bin.000002
       Relay_Log_Pos: 325
Relay_Master_Log_File: MySQL -bin.000001
      Slave_IO_Running: Yes
```

```
      Slave_SQL_Running: Yes
       Replicate_Do_DB:
     Replicate_Ignore_DB:
      Replicate_Do_Table:
    Replicate_Ignore_Table:
   Replicate_Wild_Do_Table:
 Replicate_Wild_Ignore_Table:
         Last_Errno: 0
         Last_Error:
        Skip_Counter: 0
     Exec_Master_Log_Pos: 154
       Relay_Log_Space: 630
       Until_Condition: None
        Until_Log_File:
        Until_Log_Pos: 0
      Master_SSL_Allowed: No
      Master_SSL_CA_File:
      Master_SSL_CA_Path:
       Master_SSL_Cert:
      Master_SSL_Cipher:
        Master_SSL_Key:
    Seconds_Behind_Master: 0
Master_SSL_Verify_Server_Cert: No
        Last_IO_Errno: 0
        Last_IO_Error:
        Last_SQL_Errno: 0
        Last_SQL_Error:
 Replicate_Ignore_Server_Ids:
       Master_Server_Id: 1
         Master_UUID: 1b2c3d4e-5678-90ab-cdef-
1234567890ab
       Master_Info_File: MySQL
.slave_master_info
          SQL_Delay: 0
     SQL_Remaining_Delay: NULL
   Slave_SQL_Running_State: Reading event from
the relay log
      Master_Retry_Count: 86400
          Master_Bind:
   Last_IO_Error_timestamp:
   Last_SQL_Error_timestamp:
        Master_SSL_Crl:
      Master_SSL_Crlpath:
      Retrieved_Gtid_Set:
      Executed_Gtid_Set:
```

```
            Auto_Position: 0
        1 row in set (0.00 sec)
```

`Slave_IO_Running` та `Slave_SQL_Running` - якщо обидва мають значення *Yes*, реплікація працює;

`Master_Log_File` та `Read_Master_Log_Pos` – вказують, де Slave зчитує дані з Master.

`Seconds_Behind_Master` – відображає затримку між `Master` і `Slave`. Значення 0 означає, що слейв синхронізований.

4.2.2. Налаштування реплікації Master-Master

Реплікація *Master-Master* дозволяє обом серверам одночасно виконувати роль джерела та репліки, що забезпечує можливість запису на обидва сервери та синхронізацію даних між ними.

Для налаштування сервера *Maste r1* у файлі конфігурації *MySQL* (`my.ini`), розділ `[mysqld]`, необхідно додати:

```
server-id= 1
log-bin= mysql-bin
binlog-do-db= <ім'я_бази>
auto-increment-increment= 2
auto-increment-offset= 1
```

`server-id=1` – унікальний ідентифікатор сервера;

`log-bin=MySQL -bin` – увімкнення бінарного журналу;

`binlog-do-db=my_database` – (необов'язково) реплікувати тільки певну базу даних;

`auto-increment-increment= 2` – крок для автоінкременту, забезпечує уникнення конфліктів між серверами;

`auto-increment-offset=1` – зсув для автоінкременту, забезпечує унікальність значень автоінкременту на цьому сервері.

Після внесення змін необхідно перезапустити сервер *MySQL* через Служби *Windows*.

Для налаштування сервера *Master* 2 у файлі конфігурації (*my.ini*), розділ [mysqld], внести:

```
server-id=2
log-bin=mysql-bin
binlog-do-db=my_database
auto-increment-increment=2
auto-increment-offset=2
```

`server-id=2 -` унікальний ідентифікатор сервера;

`auto-increment-offset = 2 -` зсув автоінкременту, відмінний від `Master1.`

Після внесення змін перезапустити сервер *MySQL* через Служби *Windows*.

На обох серверах потрібно створити користувача для реплікації:

```
CREATE USER 'replica_user'@'%'
IDENTIFIED BY 'password';
GRANT REPLICATION SLAVE
ON *.* TO 'replica_user'@'%';
FLUSH PRIVILEGES;
```

Виконуємо команду `SHOW MASTER STATUS` та записуємо значення `File` і `Position` для налаштування взаємного зв'язку між серверами:

Наприклад, результат може бути таким:

```
File: mysql-bin.000001
Position: 154
```

Аналогічні дії виконуємо на сервері *Master* 2.

Наприклад, результат команди *SHOW MASTER STATUS* на сервері *Master*2:

```
File: mysql-bin.000001
Position: 200
```

Налаштування реплікації виконується так, щоб *Master* 1 отримував дані від *Master* 2:

```
CHANGE MASTER TO
    MASTER_HOST='IP_адреса_Master2',
    MASTER_USER='replica_user',
    MASTER_PASSWORD='password',
    MASTER_LOG_FILE='mysql-bin.000001',
    MASTER_LOG_POS=200;
START SLAVE;
```

А *Master* 2 отримував дані від *Maste r*1:

```
CHANGE MASTER TO
    MASTER_HOST='IP_адреса_Master 1',
    MASTER_USER='replica_user',
    MASTER_PASSWORD='password',
    MASTER_LOG_FILE='MySQL -bin.000001',
    MASTER_LOG_POS=154;
START SLAVE;
```

Перевірка стану реплікації на обох серверах виконується командою:

```
SHOW SLAVE STATUS\G;
```

За правильного налаштування маємо такі значення

```
Slave_IO_Running: Yes
Slave_SQL_Running: Yes
Seconds_Behind_Master: 0
```

Приклад виводу *Master* 1:

```
Slave_IO_State: Waiting for master to send event
     Master_Host: 192.168.1.102
     Master_User: replica_user
     Master_Port: 3306
    Connect_Retry: 60
  Master_Log_File: mysql-bin.000001
 Read_Master_Log_Pos: 200
    Relay_Log_File: relay-bin.000002
    Relay_Log_Pos: 325
```

```
Relay_Master_Log_File: mysql-bin.000001
   Slave_IO_Running: Yes
  Slave_SQL_Running: Yes
   Seconds_Behind_Master: 0
```

Приклад виводу *Master* 2

```
Slave_IO_State: Waiting for master to send event
     Master_Host: 192.168.1.101
     Master_User: replica_user
     Master_Port: 3306
    Connect_Retry: 60
   Master_Log_File: MySQL -bin.000001
 Read_Master_Log_Pos: 154
    Relay_Log_File: relay-bin.000003
    Relay_Log_Pos: 450
Relay_Master_Log_File: MySQL -bin.000001
   Slave_IO_Running: Yes
  Slave_SQL_Running: Yes
   Seconds_Behind_Master: 0
```

4.3. Обмеження та рекомендації при використанні реплікації

Реплікація є важливим механізмом забезпечення доступності даних, оскільки дозволяє зберігати їх копії на кількох вузлах. Проте сама по собі вона не гарантує високої доступності системи через низку обмежень. Однією з основних проблем є відсутність автоматичного перемикання у разі відмови головного сервера. Якщо основний вузол виходить з ладу, система повинна мати механізм перенаправлення запитів на інші доступні сервери, інакше робота додатків буде перервана. Кластеризація забезпечує таке перемикання автоматично, тому є необхідною складовою високодоступних систем.

Ще однією проблемою є навантаження на головний сервер. Незважаючи на наявність реплік, основний вузол залишається єдиною точкою прийому запитів на запис, що може створювати вузьке місце у продуктивності системи. Якщо навантаження перевищує обчислювальні можливості сервера, це може спричинити деградацію продуктивності або відмову.

Необхідно також враховувати ризик неузгодженості даних, особливо при використанні асинхронної реплікації. Оновлення можуть передаватися із затримкою, тому у разі відмови головного вузла останні зміни можуть не встигнути застосуватися на репліках. Це створює загрозу втрати частини даних і зниження їх актуальності.

Оновлення можуть передаватися із затримкою, тому у разі відмови головного вузла останні зміни можуть не встигнути застосуватися на репліках. Це створює загрозу втрати частини даних і зниження їх актуальності.

4.4. Кластери *MySQL*

У *MySQL* кластеризація для забезпечення високої доступності та масштабованості найчастіше реалізується двома підходами: *InnoDB Cluster* та *MySQL NDB Cluster* (часто його також називають *MySQL Cluster*). Обидва варіанти підвищують надійність і продуктивність, що є важливим для *SCADA*-систем з великим потоком даних

InnoDB Cluster базується на технології *Group Replication* і об’єднує кілька екземплярів *MySQL* у групу з автоматичним перемиканням у разі відмови вузла. Для прозорої роботи застосунків зазвичай використовується *MySQL Router*, який маршрутизує запити до доступних вузлів кластера та спрощує підключення клієнтів. Це забезпечує стабільний доступ до даних і узгоджену роботу кластера під час відмов. Масштабування можливе без повної зупинки системи, зокрема шляхом підключення додаткових вузлів і виконання покрокових (*rolling*) операцій обслуговування.

MySQL NDB Cluster оптимізований для навантажень із великою кількістю операцій у реальному часі. Він використовує розподілену архітектуру shared-nothing, у якій дані автоматично розподіляються між вузлами та дублюються між ними; за задумом *NDB Cluster* орієнтований на високу доступність, швидке перемикання та низькі затримки. На відміну від стандартної реплікації *MySQL*, *NDB Cluster* зазвичай описують як синхронний підхід усередині кластера, а також він підтримує онлайн-масштабування (додавання вузлів під час роботи).

Такий підхід може бути доречним для *SCADA*-систем із великою кількістю джерел даних (наприклад, сенсорів), де важливі стабільність і обробка значних потоків телеметрії

Таким чином, кластеризація в MySQL забезпечує безперервний доступ до даних, надійну реплікацію та масштабування відповідно до потреб SCADA-систем, поєднуючи високий рівень відмовостійкості й продуктивності.

4.4.1. Використання InnoDB Cluster для високої доступності MySQL

InnoDB Cluster у *MySQL* побудований на технології *Group Replication* і використовує рушій зберігання *InnoDB*. Для *SCADA*-систем це застосовують насамперед для підвищення відмовостійкості та забезпечення узгодженості даних між вузлами під час роботи в режимі реального часу.

InnoDB Cluster може працювати в режимі *single-primary*, коли операції запису виконуються на одному провідному вузлі (*Primary*), а інші вузли (*Secondary*) обслуговують переважно запити на читання. У режимі *multi-primary* запис дозволений на кількох вузлах, однак така конфігурація потребує контролю можливих конфліктів одночасних змін. Незалежно від обраного режиму, зміни поширюються між вузлами групи реплікації, що забезпечує актуальність даних у межах кластера.

Механізм *Group Replication* підтримує узгодження транзакцій усередині групи та працює на основі кворуму. Для практичної відмовостійкості доцільно мати щонайменше три вузли, щоб кластер залишався працездатним у разі відмови одного сервера. Далі наведено порядок налаштування вузлів *Group Replication* як основи подальшого використання *InnoDB Cluster*.

Крок 1. Конфігурація кожного сервера

Необхідно внести зміни у файл конфігурації *MySQL* (*my.ini*). Для кожного сервера задайте унікальний server-id, увімкніть параметри бінарного журналу та *GTID*, а також налаштуйте параметри *Group Replication* і мережеві адреси для

обміну даними між вузлами.Приклад секції [mysqld] для одного з вузлів (адреси та server-id змінюються на кожному сервері)

```
 [mysqld]
server-id = 1

# Бінарний журнал і режим реплікації
log-bin = mysql-bin
binlog-format = ROW

# GTID
gtid-mode = ON
enforce-gtid-consistency = ON

# Додаткові параметри для реплікації
log-slave-updates = ON
master-info-repository = TABLE
relay-log-info-repository = TABLE

# Group Replication
transaction-write-set-extraction = XXHASH64
group_replication_group_name = "aaaaaaaa-aaaa-aaaa-
aaaa-aaaaaaaaaaaa"
group_replication_start_on_boot = OFF
group_replication_local_address = "192.168.0.1:33061"
group_replication_group_seeds =
"192.168.0.1:33061,192.168.0.2:33061,192.168.0.3:33061"
group_replication_bootstrap_group = OFF
```

Примітка: значення group_replication_group_name має бути коректним UUID. Його можна згенерувати, наприклад, командою SELECT UUID().

На другому вузлі змініть server-id на 2 і group_replication_local_address на "192.168.0.2:33061". На третьому вузлі змініть server-id на 3 і group_replication_local_address на "192.168.0.3:33061". Параметр group_replication_group_seeds має містити адреси всіх вузлів групи.- server-id = 3 - group_replication_local_address = "192.168.0.3:33061"

Крок 2. Створення користувача для реплікації

Для того, щоб сервери могли обмінюватися даними, необхідно на кожному стервері створити спеціального користувача для групової реплікації.

```
CREATE USER 'replica_user'@'%' IDENTIFIED BY
'password';
GRANT REPLICATION SLAVE, RELOAD, SUPER ON *.* TO
'replica_user'@'%';
FLUSH PRIVILEGES;
```

Запуск `START GROUP_REPLICATION` вимагає підвищених прав, таких як `GROUP_REPLICATION_ADMIN`. У практиці ці дії виконуються під адміністраторським обліковим записом.

Крок 3. Ініціалізація групової реплікації

На першому вузлі (первинному) виконується початкова ініціалізація групи

```
SET GLOBAL group_replication_bootstrap_group = ON;
START GROUP_REPLICATION;
SET GLOBAL group_replication_bootstrap_group = OFF;
```

На інших вузлах, після запуску *MySQL*, достатньо виконати:

```
START GROUP_REPLICATION;
```

Для перевірки стану реплікації використовується команда:

```
SHOW STATUS LIKE 'group_replication%';
```

Переконайтеся, що `group_replication_primary_member` відображає сервер, який є головним (активним) у групі, а `group_replication_state` має значення *ONLINE*.

Крок 4. Перевірка роботи кластера

Для тестування коректності реплікації створюємо на одному з вузлів базу та таблицю

```
CREATE DATABASE testdb;
USE testdb;
CREATE TABLE test_table (
id INT PRIMARY KEY,
data VARCHAR(100));
INSERT INTO test_table VALUES (1, 'Test data');
```

На інших вузлах виконуємо запит:

```
SELECT * FROM testdb.test_table;
```

При правильній конфігурації запис має бути доступний на всіх учасниках групи.

Крок 5. Перевірка відмовостійкості та автоматичного перемикання

Механізм *Group Replication* забезпечує автоматичне призначення нового Primary у разі відмови активного вузла. Для цього необхідна наявність кворуму, тобто щонайменше трьох серверів у групі.

Щоб перевірити це вимикаємо головний один з сервер у кластері та перевіряємо який сервер став головним:

```
SHOW STATUS LIKE
'group_replication_primary_member';
```

Після відновлення зупиненого сервера, він автоматично приєднається до групи як новий слейв.

Крок 6: Додавання нового узла до кластеру

Для включення нового сервера необхідно втклнати наступні кроки:

- сконфігурувати `my.ini` за аналогією з іншими вузлами, але з унікальними `server-id` та `group_replication_local_address`;
- створити користувача для реплікації (можна заздалегідь, якщо використовується централізоване управління);
- запуститиь процес приєднання до групи

```
SHOW STATUS LIKE 'group_replication%';
```

Потім перевіряємо стан вузла командою:

```
SHOW STATUS LIKE 'group_replication%';
```

Якщо `group_replication_state = ONLINE` – приєднання відбулося успішно.

4.4.2. Балансування навантаження з використанням MySQL Router

За наявності кількох серверів баз даних доцільно передбачити розподіл запитів між вузлами залежно від типу операції, читання або запису. Для цього застосовують *MySQL Router*, офіційний проміжний програмний компонент від *Oracle*, який працює між клієнтом і серверами *MySQL*. *MySQL Router* приймає підключення від клієнтів, аналізує тип запиту та спрямовує їх на відповідні вузли згідно з налаштованою конфігурацією. Додатково *MySQL Router* підтримує автоматичне виявлення топології кластера, моніторинг доступності вузлів і перенаправлення трафіку в разі збою.

Типовий сценарій використання полягає в розділенні потоків читання і запису для конфігурацій з реплікацією або *InnoDB Cluster*. Запити на читання спрямовуються на вторинні (*Secondary*) вузли, а запити на запис на первинний (*Primary*) вузол. *MySQL Router* автоматично визначає роль кожного вузла і підтримує актуальну маршрутизацію навіть при зміні топології кластера.

MySQL Router має повну нативну підтримку Windows і поставляється як частина екосистеми MySQL. Інсталяція виконується через *MySQL Installer* або окремий MSI-пакет, а запуск можливий як у режимі консольного застосунку, так і у вигляді служби *Windows* (*Windows Service*). У виробничих умовах *MySQL Router* зазвичай розміщують на тому ж вузлі, що й прикладний сервіс, або на окремому проміжному сервері, що забезпечує ізоляцію рівня доступу до бази даних. Для підвищення доступності точки підключення можна розгортати кілька екземплярів *Router* на різних вузлах.

Під час експлуатації групової реплікації ролі *Primary* і *Secondary* можуть змінюватися. Для прикладних систем, зокрема *SCADA*, у такому разі важливо мати стабільну точку підключення, яка не змінюється при зміні провідного вузла. Це завдання розв'язується застосуванням *MySQL Router*, який виконує маршрутизацію запитів до актуальних вузлів кластера та дає змогу не змінювати налаштування підключення в *SCADA*-клієнтах і сервісах збору даних.

MySQL Router є проміжним мережевим компонентом, який приймає підключення від клієнтів і переспрямовує їх на актуальні вузли кластера відповідно до ролей *Primary* і *Secondary*. Для роботи *Router* використовує метадані *InnoDB Cluster*. Під час первинного налаштування *Router* підключається до одного з вузлів кластера, зчитує метадані, формує локальний конфігураційний файл і реєструє в кластері службовий обліковий запис, через який надалі отримує інформацію про топологію та ролі вузлів. У документації та в утиліті *mysqlrouter* ця процедура виконується через параметр `bootstrap`. У межах цієї процедури *Router* автоматично створює окремий обліковий запис *MySQL* із випадково згенерованим паролем для доступу до метаданих кластера.

Схема підключення прикладних клієнтів через MySQL Router показана на рис.4.1.

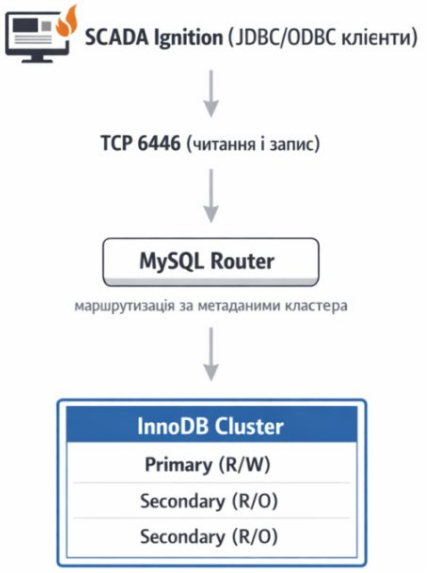


Рис.4.1. Схема підключення прикладних клієнтів через *MySQL Router*

Схема підвищення доступності точки підключення за рахунок кількох *Router* показана на рис. 4.2.

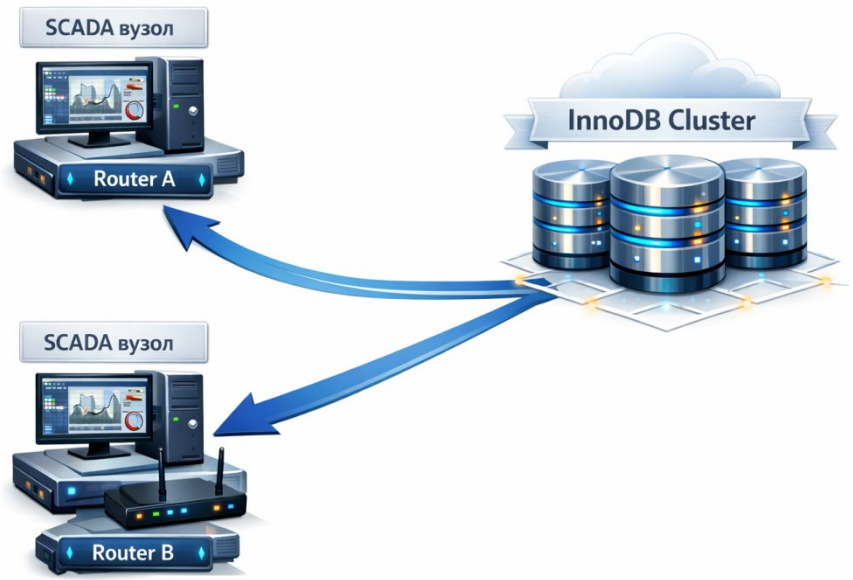


Рис.4.2. Схема підключення прикладних клієнтів через кілька *Router*

Після первинного налаштування *Router* створює окремі точки доступу для різних типів підключень. Для класичного протоколу *MySQL*, який використовують клієнти та драйвери на кшталт *mysql.exe* і *JDBC*, застосовуються порт 6446 для з'єднань читання і запису (*Read-Write*) та порт 6447 для з'єднань лише читання (*Read-Only*). Для протоколу *X* (*X Protocol*) *Router* використовує порти 6448 (*Read-Write*) та 6449 (*Read-Only*). *X Pro*tocol є клієнт-серверним протоколом *MySQL* для роботи з *X DevAPI* та *MySQL Shell*; на сервері він типово доступний через порт 33060, а *Router* надає відповідні порти-проксі 6448 і 6449.

Крок 1. Інсталяція MySQL Router у Windows

MySQL Router можна встановити двома типовими способами. Перший спосіб полягає в інсталяції через *MySQL Installer*, де *MySQL Router* обирають як окремий компонент під час встановлення або модифікації інсталяції. Другий

спосіб полягає у використанні окремого дистрибутива *Router*, який встановлюють у вибрану директорію та додають шлях до *mysqlrouter.exe* у змінну *PATH* або запускають *mysqlrouter.exe* з повного шляху. Після інсталяції перевірте, що *mysqlrouter.exe* запускається з командного рядка.

Для встановлення через окремий *MSI*-пакет завантажте *MySQL Router* з офіційного сайта *Oracle* (*dev.mysql.com/downloads/router*). Оберіть *MSI*-пакет для *Windows* відповідної розрядності (x64). Запустіть інсталятор і виконайте стандартну процедуру встановлення. За замовчуванням *MySQL Router* встановлюється в каталог *C:\Program Files\MySQL\MySQL Router 8.x*.

Крок 2. Підготовка облікового запису для первинного налаштування Router

Для первинного налаштування *Router* потрібні права на створення користувачів і доступ до службових даних кластера. Такі права доцільно надати окремому користувачу, наприклад router_init. Приклад команд для надання мінімальних прав користувачу `router_init`

```
GRANT CREATE USER ON *.* TO 'router_init'@'%' WITH
GRANT OPTION;
GRANT SELECT, INSERT, UPDATE, DELETE, EXECUTE
ON mysql_innodb_cluster_metadata.* TO
'router_init'@'%';
GRANT SELECT ON mysql.user TO 'router_init'@'%';
GRANT SELECT ON
performance_schema.replication_group_members
TO 'router_init'@'%';
GRANT SELECT ON
performance_schema.replication_group_member_stats
TO 'router_init'@'%';
GRANT SELECT ON
performance_schema.global_variables
TO 'router_init'@'%';
FLUSH PRIVILEGES;
```

Крок 3. Первинне налаштування MySQL Router за метаданими InnoDB Cluster

Первинне налаштування виконують командою mysqlrouter з параметром --*bootstrap*. У цьому кроці *Router* підключається до одного з вузлів кластера, зчитує метадані, формує локальні файли конфігурації та реєструє службовий обліковий запис для подальшого доступу до метаданих. Підключатися можна до будь-якого вузла кластера. Якщо вказаний вузол працює в режимі *Read-Only*, *Router* перепідключиться до вузла *Read-Write* для виконання реєстраційних дій.

Приклад команди у *Windows* з розміщенням окремої інсталяції *Router* у власній директорії:

```
mysqlrouter --bootstrap router_init@iklhome:3306 --
directory C:\mysqlrouter\myCluster --name router1 –force
```

Параметр `--directory` застосовують тоді, коли потрібно зберігати конфігурацію *Router* у власній директорії та мати можливість розміщувати кілька окремих екземплярів *Router* на одному сервері. Якщо *MySQL Router* встановлено як системний компонент, то за замовчуванням конфігурація створюється у системній директорії, і в межах такої інсталяції зазвичай використовується одна конфігурація *Router*.

Під час виконання `bootstrap` *Router* автоматично створює конфігураційний файл `mysqlrouter.conf` із налаштуваннями маршрутизації. Приклад типової структури згенерованого файлу:

```
[DEFAULT]
logging_folder = C:\mysqlrouter\myCluster\log
runtime_folder = C:\mysqlrouter\myCluster\run
config_folder = C:\mysqlrouter\myCluster

[logger]
level = INFO

[metadata_cache:myCluster]
cluster_type = gr
router_id = 1
```

```
user = mysql_router1_abc
metadata_cluster = myCluster
ttl = 0.5

[routing:myCluster_rw]
bind_address = 0.0.0.0
bind_port = 6446
destinations = metadata-
cache://myCluster/?role=PRIMARY
routing_strategy = first-available
protocol = classic

[routing:myCluster_ro]
bind_address = 0.0.0.0
bind_port = 6447
destinations = metadata-
cache://myCluster/?role=SECONDARY
routing_strategy = round-robin-with-fallback
protocol = classic

[routing:myCluster_x_rw]
bind_address = 0.0.0.0
bind_port = 6448
destinations = metadata-
cache://myCluster/?role=PRIMARY
routing_strategy = first-available
protocol = x

[routing:myCluster_x_ro]
bind_address = 0.0.0.0
bind_port = 6449
destinations = metadata-
cache://myCluster/?role=SECONDARY
routing_strategy = round-robin-with-fallback
protocol = x
```

Параметр `ttl` визначає інтервал оновлення метаданих кластера (у секундах). Значення 0.5 означає перевірку кожні 500 мілісекунд, що забезпечує швидку реакцію на зміни топології.

Крок 4. Запуск MySQL Router як служби Windows

Після первинного налаштування доцільно запускати *MySQL Router* як службу *Windows*. Це забезпечує автоматичний запуск *Router* після перезавантаження операційної системи. Для встановлення служби використовують `--install-service` або `--install-service-manual`, а для запуску встановленої служби застосовують `--service`.

Приклад встановлення служби з автоматичним запуском і подальший старт

```
mysqlrouter --install-service
mysqlrouter –service
```

Для встановлення служби з ручним запуском використовується команда

```
mysqlrouter --install-service-manual
```

Альтернативно, керування службою можна виконувати через графічний інтерфейс *Services* (*services.msc*) або за допомогою команди *sc*

```
sc query MySQLRouter
sc start MySQLRouter
sc stop MySQLRouter
```

Крок 5. Перевірка маршрутизації та режимів Read-Write і Read-Only

Для перевірки підключення з боку клієнта достатньо виконати два підключення до *Router* і визначити роль вузла, на який було встановлено з’єднання.

Приклад перевірки *Read-Write* через порт 6446

```
mysql -h 127.0.0.1 -P 6446 -u scada_user -p
```

Далі у середовищі *SQL* виконайте запит

```
SELECT   @@hostname AS server_host,
         @@read_only AS read_only_mode;
```

Для порту 6446 значення *@@read_only* має дорівнювати 0, а *@@hostname* відповідатиме поточному вузлу *Primary*. Для порту 6447 значення *@@read_only* має дорівнювати 1, а *@@hostname* відповідатиме одному з вузлів *Secondary*. Розділення портів *Read-Write* і *Read-Only* формується під час первинного налаштування *Router* за метаданими *InnoDB Cluster*.

Крок 6. Перевірка поведінки під час відмови Primary

Щоб перевірити, як *MySQL Router* поводиться у разі відмови провідного вузла, у тестовому середовищі тимчасово зупиніть службу *MySQL* на вузлі Primary або від'єднайте цей вузол від мережі. Після цього необхідно дочекатися, поки механізм *Group Replication* виконає перевибір провідного вузла і призначить новий *Primary*.

Далі необхідно повторити підключення до *MySQL Router* через порт 6446 та виконати перевірочний запит

```
mysql -h 127.0.0.1 -P 6446 -u scada_user -p
SELECT @@hostname AS server_host, @@read_only AS
read_only_mode;
```

Очікуваний результат полягає в тому, що підключення через той самий порт 6446 буде встановлено вже з новим вузлом *Primary*, при цьому зміна параметрів підключення на стороні клієнта не потрібна.

У типовому сценарії інтеграції *SCADA* через *JDBC* доцільно задавати підключення не до конкретного вузла *MySQL*, а до *MySQL Router*. У такому разі в налаштуваннях джерела даних фіксується одна точка доступу, а зміни ролей усередині кластера не потребують внесення змін у конфігурацію *SCADA*. Для сценаріїв, де переважають операції читання, можна передбачити окреме підключення до порту 6447 для аналітики або формування звітів, залишаючи порт 6446 для транзакційних операцій читання і запису.

Якщо *InnoDB Cluster* не використовується, а застосовується традиційна реплікація *master-slave*, *MySQL Router* можна налаштувати вручну через конфігураційний файл. У цьому сценарії **Router** не використовує метадані кластера, а керується статичними правилами маршрутизації.

Приклад базової конфігурації для сценарію з одним первинним вузлом (*master*) і вторинними вузлами (*slaves*). Конфігураційний файл зберігається в каталозі *C:\ProgramData\MySQL\MySQL Router\ з ім'ям mysqlrouter.conf:*

```
 [DEFAULT]
logging_folder = C:\ProgramData\MySQL\MySQL
Router\log
runtime_folder = C:\ProgramData\MySQL\MySQL
Router\run
config_folder = C:\ProgramData\MySQL\MySQL Router

[logger]
level = INFO

[routing:primary]
bind_address = 0.0.0.0
bind_port = 6446
destinations = 192.168.1.10:3306
routing_strategy = first-available
mode = read-write

[routing:secondary]
bind_address = 0.0.0.0
bind_port = 6447
destinations = 192.168.1.11:3306,192.168.1.12:3306
routing_strategy = round-robin
mode = read-only
```

`bind_address` і `bind_port` визначають адресу та порт, на якому Router приймає підключення;

`destinations` містить перелік MySQL-серверів для маршрутизації;

`routing_strategy` визначає алгоритм вибору вузла (*first-available, round-robin, round-robin-with-fallback*);

`mode` вказує тип операцій (*read-write* для запису, *read-only* для читання).

У цій конфігурації клієнти підключаються до порту 6446 для операцій запису та до порту 6447 для операцій читання.

MySQL Router має нативну підтримку *Windows* і інтегрується з екосистемою *MySQL*. Засіб дає змогу реалізувати балансування навантаження, автоматичне перемикання у разі збою та масштабування систем баз даних без внесення змін у клієнтських застосунках. Завдяки підтримці як сценаріїв реплікації, так і конфігурацій *InnoDB Clus*ter, *MySQL Router* може застосовуватися для розгортання в Windows-середовищі. Для прикладних систем, зокрема *SCADA*, використання *Router* дає змогу забезпечити стабільну точку підключення, яка автоматично враховує зміну ролей вузлів у кластері.

4.4.3. Використання NDB Cluster для забезпечення високої доступності

MySQL NDB Cluster є кластерною реалізацією *MySQL*, у якій зберігання даних виконується за допомогою рушія *NDB*. Архітектура *NDB Cluster* передбачає розподілення даних між вузлами та їх дублювання відповідно до заданого рівня реплікації. На одному комп'ютері можна встановити стандартний сервер *MySQL* і *MySQL NDB Cluster* паралельно, оскільки стандартний *MySQL* зазвичай працює з локальним дисковим сховищем, тоді як N*DB Cluster* використовує окремі компоненти кластера і спеціальний рушій ndbcluster для доступу до розподілених даних.

Для інтеграції зі *SCADA NDB Cluster* розглядають у випадках, коли потрібні масштабування за рахунок додавання вузлів і збереження доступності під час відмов окремих компонентів. При цьому необхідно враховувати, що робота *NDB Cluster* відрізняється від класичної реплікації *MySQL*: дані зберігаються у вузлах даних, а *SQL*-вузли забезпечують доступ до цих даних через звичні *SQL*-запити.

Склад *NDB Cluster* включає такі компоненти. Вузол керування (*Management Node*) виконує керування конфігурацією та моніторингом, запускається процесом *ndb_mgmd*. Вузли даних (*Data Nodes*) зберігають дані

кластера та виконують їх реплікацію, запускаються процесами *ndbd* або *ndbmtd*. *SQL*-вузли (*SQL Nodes*) надають доступ до даних через *SQL* і використовують рушій ndbcluster, зазвичай запускаються стандартним серверним процесом *mysqld* з увімкненими параметрами *NDB*.

Далі наведено приклад розгортання *MySQL NDB Cluster* у *Windows* на комп'ютері, де вже встановлено *MySQL Server* 8.0

Крок 1. Завантаження та підготовка дистрибутива

Необхідно завантажити інсталяційний пакет *MySQL NDB C*luster для *Windows* з офіційного сайту *MySQL* і розпакувати архів у вибрану директорію. Наприклад, розмістіть дистрибутив у *C:\MySQL-cluster*. Далі необхідно додати *C:\MySQL-cluster\bin* до системної змінної *PATH* або використовувати повні шляхи до виконуваних файлів.

Крок 2. Створення конфігураційного файла config.ini

Конфігурація NDB Cluster задається у файлі *config.ini* для вузла керування (*ndb_mgmd*). У конфігурації визначаються кількість реплік даних, вузол керування, вузли даних і *SQL*-вузли.

Параметр `NoOfReplicas` визначає кількість копій даних у кластері. Для забезпечення відмовостійкості значення `NoOfReplicas` має бути 2 або більше, а вузли даних необхідно розміщувати на різних хостах. Для навчального прикладу на одному комп'ютері часто задають `NoOfReplicas=1`, оскільки фізичне резервування на одному хості не забезпечує відмовостійкість на рівні обладнання.

Приклад файла *config.ini* для локального стенда у Windows:

```
# Загальні налаштування для вузлів даних
[ndbd default]
NoOfReplicas=1
DataDir=C:/MySQL-cluster/data
```

```
# Налаштування вузла керування
[ndb_mgmd]
HostName=localhost
DataDir=C:/MySQL-cluster/management

# Налаштування першого вузла даних
[ndbd]
HostName=localhost
NodeId=2

# Налаштування другого вузла даних
[ndbd]
HostName=localhost
NodeId=3

# Налаштування першого SQL-вузла
[mysqld]
NodeId=4

# Налаштування другого SQL-вузла
[mysqld]
NodeId=5
```

Крок 3. Запуск вузла керування (ndb_mgmd)

Необхідно відкрити командний рядок з правами адміністратора та запустити вузол керування, передавши шлях до *config.ini*

```
ndb_mgmd.exe -f C:\MySQL-cluster\config.ini
```

Приклад повідомлення під час запуску

```
MySQL Cluster Management Server MySQL-8.0.40 ndb-
8.0.40
```

Після запуску *ndb_mgmd* процес має залишатися активним, оскільки він забезпечує керування конфігурацією та обмін службовими повідомленнями в кластері. Якщо необхідно не тримати відкрите вікно командного рядка, можна запустити процес у фоновому режимі

```
start /B ndb_mgmd.exe -f C:\MySQL-cluster\config.ini
```

Або встановити вузол керування як службу Windows. Команди встановлення служби залежать від способу інсталяції та версії дистрибутива. Приклад варіанта, який використовують у практиці

```
ndb_mgmd.exe --install
net start ndb_mgmd
```

Крок 4. Запуск вузлів даних (ndbd)

Необхідно відкрити нове вікно командного рядка з правами адміністратора і запустити вузли даних. Команду ndbd.exe необхідно виконати для кожного вузла даних, визначеного у config.ini. Якщо всі вузли розгортаються на одному комп'ютері, запуск виконують у різних вікнах командного рядка.

```
ndbd.exe
```

Під час першого запуску вузол даних створює службові каталоги у *DataDir*. Якщо каталоги не створюються автоматично, їх необхідно створити вручну відповідно до структури, яку очікує *NDB*.

Типове повідомлення під час запуску *ndbd.exe*

```
2025-01-18 12:39:57 [ndbd] INFO -- Angel Connected
to 'localhost:1186'
2025-01-18 12:39:57 [ndbd] INFO -- Angel allocated
nodeid: 2
```

Це означає, що вузол даних підключився до вузла керування на порту 1186, а також отримав призначений ідентифікатор *NodeId*. Позначення *Angel* у цьому контексті стосується службового механізму взаємодії між вузлом даних і вузлом керування.

Крок 5. Перевірка стану кластера через ndb_mgm

Після запуску вузла керування і вузлів даних необхідно перевірити стан кластера утилітою *ndb_mgm*

```
ndb_mgm.exe -e show
```

Приклад виводу

```
Connected to Management Server at: localhost:1186
Cluster Configuration
---------------------
[ndbd(NDB)]   2 node(s)
id=2  @127.0.0.1 (MySQL-8.0.40 ndb-8.0.40,
Nodegroup: 0, *)
id=3  @127.0.0.1 (MySQL-8.0.40 ndb-8.0.40,
Nodegroup: 0)

[ndb_mgmd(MGM)] 1 node(s)
id=1  @127.0.0.1 (MySQL-8.0.40 ndb-8.0.40)

[mysqld(API)]  2 node(s)
id=4 (not Connected, accepting Connect from
127.0.0.1)
id=5 (not Connected, accepting Connect from
127.0.0.1)
```

У цьому прикладі видно, що вузли даних і вузол керування підключені, а *SQL*-вузли ще не підключені та очікують на з'єднання.

Крок 6. Запуск SQL-вузлів і підключення до NDB Cluster

Далі необхідно запустити *SQL*-вузли так, щоб сервер *MySQL* підключився до *NDB Cluster* і використовував рушій *ndbcluster*. Якщо використовується окремий *MySQL Server* як *SQL*-вузол, його запускають з параметрами *NDB*, зокрема із зазначенням рядка підключення до вузла керування

```
mysqld.exe --ndbcluster --ndb-
connectstring=localhost
```

Команду необхідно виконати для кожного *SQL*-вузла, визначеного у *config.ini*. Після запуску *SQL*-вузлів повторіть перевірку:

```
ndb_mgm.exe -e show
```

У секції [mysqld(API)] вузли мають перейти зі стану *not Connected* у стан підключення.

Після розгортання *NDB Cluster* інтеграція з прикладними системами виконується через *SQL*-вузли. При цьому вузол керування (*Management Node*), вузли даних (*Data Nodes*) і *SQL*-вузли (*SQL Nodes*) працюють спільно, а доступ до даних реалізується через стандартні *SQL*-запити з використанням рушія *ndbcluster*.

РОЗДІЛ 5. ОПТИМІЗАЦІЯ ПРОДУКТИВНОСТІ *MYSQL*

5.1. Конфігурування *MySQL* для роботи в режимі реального часу

Налаштування продуктивності *MySQL* є необхідним для стабільної роботи систем, що обробляють дані в режимі реального часу. За умов високого навантаження та вимог до мінімальних затримок конфігурацію серверної частини СУБД необхідно виконувати з урахуванням характеристик апаратного забезпечення та фактичного профілю запитів.

Основні параметри задаються у конфігураційному файлі *my.ini.* Вони впливають на використання оперативної пам’яті, операції введення і виведення, виконання транзакцій, обробку запитів і керування з’єднаннями. Наведені нижче значення є прикладами; їх необхідно уточнювати за результатами тестування та моніторингу на конкретному стенді.

Параметри пам’яті визначають обсяг ресурсів, виділених для внутрішніх структур бази даних:

Параметр `innodb_buffer_pool_size = 12G` задає розмір буфера для кешування сторінок даних та індексів *InnoDB*. Якщо сервер використовується переважно для *MySQL*, практичним орієнтиром є виділення для buffer pool близько 60-80 % доступної оперативної пам’яті з урахуванням потреб операційної системи та прикладних сервісів.

Параметр `innodb_log_buffer_size = 64M` визначає розмір буфера журналу; збільшення цього значення може бути доцільним у сценаріях з великими транзакціями, щоб зменшити частоту скидання журналу на диск.

Параметр `innodb_flush_log_at_trx_commit = 2` впливає на співвідношення між затримками та збереженням транзакцій під час аварій. За такого значення журнал не обов’язково синхронізується з диском на кожному COMMIT, що зменшує накладні витрати, але у разі відмови живлення можлива втрата частини останніх транзакцій у межах короткого інтервалу.

Параметр `innodb_io_capacity = 2000` задає орієнтир для планувальника фонових операцій введення і виведення InnoDB; значення

доцільно узгоджувати з реальними можливостями накопичувача, зокрема для *SSD* воно зазвичай вище, ніж для *HDD*.

Налаштування, пов’язані з обробкою запитів і тимчасових таблиць, впливають на швидкодію під час виконання складних операцій.

Параметри `tmp_table_size = 64M` та `max_heap_table_size = 64M` визначають максимальний розмір тимчасових таблиць у пам’яті. Фактичне обмеження визначається меншим із двох значень; якщо запит потребує більшого обсягу, тимчасова таблиця може бути перенесена на диск, що збільшує затримки.

Параметр `join_buffer_size = 64M` може прискорити окремі *JOIN*-операції, однак необхідно враховувати, що цей буфер виділяється на з’єднання і на операцію, тому надмірне збільшення `join_buffer_size` за великої кількості паралельних сесій може суттєво підвищити сумарне споживання пам’яті.

Параметри `query_cache_type = 0` і `query_cache_size = 0` у навчальних матеріалах часто наводять як рекомендацію для сценаріїв з частими оновленнями даних. Водночас необхідно враховувати версію *MySQL*: у *MySQL* 8.0 кеш запитів вилучено, тому параметри `query_cache_*` не застосовуються та можуть бути відсутні в конфігурації.

Параметри з’єднань визначають граничну кількість одночасних клієнтів та поведінку неактивних сесій. Параметр `max_connections = 1000` задає максимальну кількість паралельних підключень; перед його збільшенням необхідно оцінити достатність ресурсів, оскільки кожне підключення споживає пам’ять і створює додаткове навантаження на планувальник. Параметри `wait_timeout = 600` та `interactive_timeout = 600` задають час утримання неактивних з’єднань; їх значення необхідно узгоджувати з політикою роботи клієнтських застосунків і пулів з’єднань.

Приклад фрагмента my.ini, що відображає наведені параметри, може мати такий вигляд

```
[mysqld]
innodb_buffer_pool_size = 12G
innodb_log_buffer_size = 64M
innodb_flush_log_at_trx_commit = 2
innodb_io_capacity = 2000

tmp_table_size = 64M
max_heap_table_size = 64M
join_buffer_size = 64M

query_cache_type = 0
query_cache_size = 0

max_connections = 1000
wait_timeout = 600
interactive_timeout = 600
```

Коректно підібрані значення цих параметрів формують базову конфігурацію для роботи *MySQL* у режимі реального часу та створюють основу для подальшого моніторингу й уточнення налаштувань відповідно до фактичних характеристик навантаження.

5.2. Системи моніторингу продуктивності *MySQL* у *Windows*

Навіть за правильно налаштованої конфігурації MySQL продуктивність може знижуватися зі зростанням навантаження. Для виявлення причин уповільнення, контролю затримок і стабільності роботи сервера застосовують моніторинг у режимі реального часу.

У середовищі *Windows* у практиці найчастіше використовують *Percona Monitoring* and *Management* (*PMM*), *Zabbix* та *MySQL Enterprise Monitor*. Ці рішення дають змогу контролювати показники роботи *MySQL*, аналізувати запити та отримувати підказки щодо налаштувань і реагування на перевищення заданих порогів.

Percona Monitoring and Management є безоплатним інструментом компанії *Percona*. Він забезпечує збір і відображення показників використання

процесора та оперативної пам’яті, інтенсивності виконання запитів, а також ознак блокувань транзакцій. Інтерфейс *PMM* реалізовано у вигляді вебпанелей з графіками, а для *MySQL* доступні готові шаблони, що дає змогу швидко розпочати базовий моніторинг.

Zabbix є універсальною системою моніторингу для серверів, операційних систем, мережевого обладнання та прикладних сервісів. Для *MySQL* вона використовується для контролю ресурсів, кількості активних з’єднань і часових характеристик виконання запитів. *Zabbix* підтримує тригери та сповіщення щодо перевищення критичних значень. Для *Windows* застосовується офіційний агент, який встановлюється на вузол спостереження та передає дані на сервер *Zabbix*.

MySQL Enterprise Monitor є комерційним продуктом компанії *Oracle*. Він орієнтований на централізований контроль стану *MySQL* і містить засоби аналізу ефективності *SQL*-запитів та індексів, а також інструменти прогнозування навантаження. Окремим компонентом є модуль «*Advisor*», який формує рекомендації щодо усунення потенційних проблем і коригування налаштувань.

Поряд із зовнішніми системами моніторингу доцільно застосовувати вбудовані засоби *MySQL* для аналізу запитів. Команда *EXPLAIN* надає план виконання *SQL*-запиту з інформацією про вибір індексів, оцінку кількості рядків, що обробляються, та спосіб доступу до таблиць. Використання *EXPLAIN* дає змогу локалізувати причини повільного виконання запитів і обґрунтовано змінювати структуру запиту або індексацію.

5.3. Аналіз продуктивності *SQL*-запитів за допомогою *EXPLAIN*

Для підвищення ефективності обробки даних у *MySQL* необхідно розуміти, як саме сервер виконує S*Q*L-запит. Команда *EXPLAIN* дає змогу отримати план виконання запиту, тобто опис того, у якій послідовності та якими методами *MySQL* звертається до таблиць, які індекси може використати і які фактично застосовує, скільки рядків оцінно буде прочитано та який тип

доступу до таблиці обирається. Це дозволяє виявляти причини повільного виконання та обґрунтовано змінювати індекси або структуру запиту, що є актуальним для систем автоматизації з вимогами до затримок.

Команда *EXPLAIN* додається перед *SQL*-запитом. Наприклад:

```
EXPLAIN
SELECT *
FROM sensor_readings
WHERE machine_id = 3
AND reading_time > '2025-08-18 08:00:00';
```

У цьому прикладі використано таблицю `sensor_readings`, що містить дані датчиків технологічного обладнання. Припустимо, що поля `machine_id` і `reading_time` проіндексовано. Результатом *EXPLAIN* є таблиця з параметрами плану виконання, наприклад

id	select_type	table	type	possible_keys	key	key_len	rows	Extra
1	SIMPLE	sensor_readings	ref	machine_time_index	machine_time_index	8	50	Using where

Отримані значення слід інтерпретувати як характеристики обраного сервером способу доступу.

Поле `id` дорівнює 1, отже запит виконується одним кроком.

Значення `select_type = SIMPLE` означає, що запит не містить підзапитів і складних конструкцій об'єднання.

Поле `table` вказує таблицю, з якої читаються дані.

Поле `type` характеризує метод доступу, і значення ref означає, що *MySQL* виконує пошук за індексом за умовою рівності або відповідності ключу.

У `possible_keys` наведено індекси, які сервер може використати, а в key зазначено індекс, який фактично обрано.

Поле `key_len` відображає довжину використаної частини ключа в байтах.

Поле `rows` є оцінкою кількості рядків, які буде переглянуто під час виконання.

Поле `Extra` містить додаткові ознаки, і `Using where` означає, що після індексного доступу застосовується додаткова фільтрація умовою *WHERE*.

Такий аналіз дає змогу виявляти причини уповільнення. Наприклад, якщо в полі `type` відображається *ALL*, це означає повне сканування таблиці, що за великого обсягу телеметрії може збільшувати час виконання запитів. У таких ситуаціях зазвичай перевіряють наявність індексів за полями фільтрації та сортування, доцільність складеного індексу, а також структуру самого запиту.

Практичний приклад у контексті *SCADA*. Нехай система контролю обсмажування кави зчитує температуру кожної машини щосекунди та записує дані у `sensor_readings`. Для отримання останніх значень конкретної машини використовується запит

```
EXPLAIN
SELECT temperature, humidity
FROM sensor_readings
WHERE machine_id = 2
ORDER BY reading_time DESC
LIMIT 10;
```

Результат *EXPLAIN* дозволяє перевірити, чи задіяно індексацію, яка одночасно підтримує відбір за `machine_id` і сортування за `reading_time`. Якщо план вказує на невдалий тип доступу або суттєво завищену оцінку `rows`, це є підставою розглянути створення складеного індексу (`machine_id, reading_time`) або корекцію запиту та індексів відповідно до реального профілю звернень.

Таким чином, *EXPLAIN* стає ключовим інструментом для аналізу продуктивності *SQL*-запитів, дозволяючи адміністраторам і розробникам точно

визначати та усувати «вузькі місця» у роботі баз даних, особливо у промислових системах автоматизації.

5.4. Кешування в *MySQL* і *Redis*

Кешування зменшує кількість звернень до бази даних і може підвищувати продуктивність системи. У *MySQL* існують механізми, що зменшують витрати на читання даних, зокрема буферний пул *InnoDB*, а також історично застосовувався *Query Cache* для кешування результатів запитів. За умов частих змін у таблицях кешування результатів запитів може бути малоефективним, оскільки кеш швидко втрачає актуальність. У таких ситуаціях використовують зовнішній кеш, який працює окремо від *MySQL*.

Redis є системою зберігання даних у пам’яті типу «ключ значення» (in-memory *key-value store*), яку часто застосовують як зовнішній кеш для прикладних систем. Вона дає змогу зберігати результати *SELECT*-запитів або часто використовувані об’єкти з заданим часом життя (*TTL*). Під час повторних звернень прикладний сервіс спочатку перевіряє наявність даних у *Redis* і, за наявності актуального запису, отримує результат без виконання запиту в *MySQL*.

Типова схема взаємодії під час читання даних має такий порядок. Прикладний клієнт, наприклад *SCADA*-сервіс або проміжний сервіс збору даних, формує запит до прикладного рівня. Далі прикладний рівень виконує перевірку кеша в *Redis* за наперед визначеним ключем. Якщо запис знайдено і його час життя не минув, результат повертається клієнту з *Redis*. Якщо запису немає або він застарілий, прикладний рівень виконує запит до *MySQL*, отримує результат, зберігає його в *Redis* із заданим *TTL* і повертає клієнту.

У промислових системах автоматизації та *SCADA* такий підхід зазвичай застосовують для читання даних, які потрібні багатьом клієнтам одночасно і не вимагають миттєвого відображення кожної зміни. До них належать довідкові таблиці, параметри конфігурації, переліки обладнання, статуси з допустимою затримкою оновлення, а також агреговані показники для панелей моніторингу.

Для даних, що змінюються щосекундно і мають використовуватися як керувальні або аварійні сигнали, кешування потрібно застосовувати обережно, задаючи короткий *TTL* або відмовляючись від кеша на користь прямого читання з *MySQL* чи спеціалізованого сховища часових рядів.

Окремо слід враховувати узгодження кеша із *MySQL* під час запису даних. Типова практика полягає в тому, що після операцій *INSERT*, *UPDATE* або DELETE прикладний рівень або видаляє відповідні ключі в *Redis*, або перезаписує їх оновленим значенням. У разі відсутності цього кроку в *Redis* може залишатися застаріла інформація, що призводить до некоректних показів у *SCADA* або звітах.

Під час проєктування кешування доцільно зафіксувати правила формування ключів, значення T*T*L для різних класів даних і перелік операцій, після яких кеш необхідно інвалідовувати. Це забезпечує відтворювану поведінку системи та спрощує супровід під час зростання навантаження.

РОЗДІЛ 6. ІНТЕГРАЦІЯ *MYSQL* ЗІ *SCADA*-СИСТЕМАМИ

6.1. Особливості даних у *SCADA*

Інтеграція *SCADA*-систем із базами даних є важливою складовою автоматизованих систем керування технологічними процесами. *SCADA* забезпечує збір, первинну обробку та візуалізацію даних у режимі, близькому до реального часу, тоді як база даних виконує функції надійного довготривалого зберігання, підтримки історії змін і формування інформаційної основи для аналізу та звітності.

Потоки даних у *SCADA* формуються з різнорідних джерел, зокрема з вимірювальних каналів датчиків, сигналів стану виконавчих механізмів, технологічних подій, тривог та дій оператора. Відповідно дані можуть мати вигляд числових значень (аналогових і дискретних), станів, повідомлень подій і записів журналів. Така різноманітність безпосередньо впливає на вимоги до схеми даних, структури таблиць, часових атрибутів, механізмів запису та засобів подальшої обробки.

Для *SCADA*-характерні висока частота оновлення, великі обсяги часових рядів і потреба у відтворюваності історії. Тому під час проєктування збереження необхідно враховувати транзакційну цілісність, відновлюваність після збоїв, ефективність операцій запису й читання, а також політику життєвого циклу даних, включно з архівуванням і регламентним видаленням застарілих записів. Централізоване зберігання даних забезпечує можливість довгострокового аналізу, виявлення відхилень від нормативів, формування звітності та підтримки рішень щодо експлуатації обладнання.

Інтеграція *SCADA* з базами даних також полегшує взаємодію з корпоративними інформаційними системами, зокрема *ERP* (*Enterprise Resource Planning*), *MES* (*Manufacturing Execution System*) та B*I* (*Business Intelligence*). Це підсилює функції централізованого архівування, резервного копіювання і відновлення даних, а також створює підґрунтя для застосування інструментів аналітики, включно з прогнозними методами. У межах цього посібника

основною СУБД розглядається *MySQL*, оскільки вона широко застосовується у навчальних і прикладних проєктах та має розвинуту екосистему засобів інтеграції.

6.2. Технології обміну даними

У системах автоматизації технологічних процесів зв'язок між фізичним рівнем вимірювань, програмованими логічними контролерами (ПЛК) та диспетчерськими системами керування (*SCADA*) є важливим елементом архітектури. Обмін інформацією забезпечує безперервне отримання, обробку та візуалізацію даних про стан об'єкта керування в режимі реального часу.

Сучасні датчики передають вимірювані параметри (температура, тиск, витрата, концентрація тощо) до ПЛК двома основними способами: через аналогові входи з уніфікованими сигналами (4–20 мА або 0–10 В) або через цифрові шини та мережі. Вибір способу визначається типом датчика, вимогами до точності вимірювання, умовами експлуатації та доступною інфраструктурою зв'язку.

Для цифрового обміну використовуються стандартизовані протоколи, такі як *Modbus*, *HART*, 1-*Wire*, *I²C* або *CANopen*. Протокол *Modbus* застосовується у двох поширених реалізаціях: *Modbus RTU* для послідовних інтерфейсів (*RS*-485, *RS*-232) та *Modbus TCP/IP* для мереж *Ethernet*. Окрім передавання основних параметрів, окремі протоколи і польові шини дають змогу отримувати діагностичну інформацію про стан датчика або каналу зв'язку, що підвищує надійність експлуатації.

Після приймання сигналів ПЛК виконує алгоритмічну обробку, фільтрацію, формує керувальні впливи та готує дані для передачі на рівень *SCADA*.

Передача даних між ПЛК і *SCADA* може реалізовуватися різними протоколами.

Modbus TCP/IP широко використовується завдяки простоті й сумісності, однак працює за клієнт-серверною моделлю з періодичним опитуванням, що за

високих частот оновлення здатне створювати додаткове навантаження на мережу та джерело даних.

OPC UA (*Unified Architecture*) орієнтований на структуровані дані, підтримує модель інформаційних об'єктів, механізми безпеки, автентифікацію та контроль доступу, а також подієві механізми і підписки, що важливо для сучасних розподілених систем.

MQTT (*Message Queuing Telemetry Transport*) реалізує модель «публікація–підписка» через брокер і доцільний у розподілених системах та *IoT*-сценаріях, коли необхідно зменшити накладні витрати мережі й забезпечити ефективну доставку великої кількості коротких повідомлень.

REST API (*Representational State Transfer Application Programming Interface*) забезпечує обмін через *HTTP* і часто використовується для інтеграції *SCADA* з прикладними сервісами, вебкомпонентами та інструментами міжсистемної взаємодії.

Вибір протоколу визначається архітектурою системи, вимогами до затримок, обсягів і безпеки, а також доступністю відповідних драйверів і шлюзів у застосованій *SCADA*-платформі. На практиці протоколи часто поєднуються: один застосовується для збору даних із нижнього рівня, інший для подієвої доставки або міжсистемної інтеграції.

6.3. Інтеграція *SCADA*-системи із базами даних

Інтеграція *SCADA* із СУБД забезпечує історизацію технологічних параметрів, збереження подій і тривог, формування звітів, а також можливість ретроспективного аналізу для оптимізації режимів роботи й планування обслуговування. З технічної точки зору взаємодія зазвичай реалізується через стандартні інтерфейси доступу до реляційних баз даних, де дані передаються у вигляді *SQL*-операцій або викликів підготовлених запитів і процедур.

Запис даних з *SCADA* до бази може виконуватися безперервно або з визначеним інтервалом, залежно від критичності параметрів і допустимих затримок. Щоб уникати надмірного навантаження на мережу й сервер бази

даних, на практиці застосовуються буферизація, пакетний запис, збереження лише змінених значень, агрегування за часовими вікнами та нормалізація форматів даних. Для підвищення надійності необхідно передбачати поведінку системи у разі втрати з'єднання, включно з повторними спробами, локальним накопиченням і контрольованим відновленням потоку запису.

У частині зворотного обміну база даних може бути джерелом розрахункових показників, уставок, довідкових параметрів, планових значень і сигналів, що формуються прикладними сервісами. Такий обмін доцільно будувати так, щоб *SCADA* отримувала лише ті дані, які необхідні для відображення, сигналізації або керування, без дублювання функцій історизації і без перенесення на *SCADA* надмірних обчислювальних навантажень.

За наявності складної інфраструктури інтеграція може виконуватися через проміжне програмне забезпечення, яке діє як шлюз або конвертер протоколів. Це актуально, коли джерела даних не підтримують прямий доступ до СУБД, або коли необхідно централізувати перетворення форматів, маршрутизацію і політики безпеки.

6.4. Драйвери для підключення баз даних до *SCADA*

Для доступу *SCADA* до реляційних баз даних застосовуються драйвери й провайдери, які забезпечують узгодження форматів, керування з'єднаннями, виконання *SQL*-запитів і обробку результатів. Найпоширеними технологіями є *ODBC* (*Open Database Connectivity*) та *JDBC* (*Java Database Connectivity*). *ODBC* використовується як універсальний інтерфейс доступу до різних СУБД і часто застосовується в системах, де традиційно домінують *ODBC*-орієнтовані компоненти. *JDBC* є стандартним механізмом доступу для *Java*-платформ і застосовується у *SCADA*, реалізованих на *Java*-стеку; як приклад сучасної *SCADA* можна згадати Ignition, яка типово використовує *JDBC* для підключення до *MySQL*.

Окрім вибору технології доступу, необхідно враховувати сумісність версій драйверів із сервером СУБД, підтримку транзакцій, поведінку при

обривах з’єднання, режими тайм-аутів, можливості пулінгу з’єднань, а також вимоги безпеки, включно з шифруванням з’єднань і контролем привілеїв користувачів. Драйверний рівень фактично визначає характеристики взаємодії *SCADA* з *MySQL* у частині стабільності й продуктивності, тому його вибір і налаштування мають бути узгоджені з режимами роботи системи.

6.5. Інтеграція *MySQL* зі *SCADA*

Інтеграція *SCADA* з *MySQL* застосовується для організації історизації, збереження подій і тривог, підтримки звітності та реалізації контурів інформаційної взаємодії між диспетчеризацією й прикладними сервісами. У типовій архітектурі *SCADA* забезпечує надходження первинних даних від ПЛК і датчиків, а *MySQL* виконує функції централізованого сховища з можливістю структурованого збереження й подальшої обробки. *MySQL* у такому контексті зазвичай використовується з транзакційним механізмом зберігання, що забезпечує цілісність і відновлюваність даних при збоях.

Обмін даними може бути одностороннім, коли *SCADA* записує дані до *MySQL* для архівування та аналітики, або двостороннім, коли з *MySQL* до *SCADA* повертаються результати обчислень, уставки, діагностичні індикатори та події тривог, сформовані прикладною логікою. Принципово важливо розмежовувати ролі: *SCADA* відповідає за оперативне відображення, сигналізацію і керування, тоді як *MySQL* забезпечує збереження, вибірки, агрегування та передачу результатів обробки на рівень *SCADA*.

Технічно інтеграція може реалізовуватися прямим підключенням *SCADA* до *MySQL* через JDBC або ODBC, або через проміжні компоненти, які беруть на себе трансформацію форматів, буферизацію та маршрутизацію. Прямий підхід спрощує архітектуру, але потребує чітко визначених режимів запису, індексації та контролю частоти звернень. Підхід із проміжними компонентами підвищує гнучкість і може зменшувати навантаження на *SCADA* і *MySQL*, особливо коли необхідно виконувати попередню нормалізацію чи агрегацію даних перед записом або передачею назад у *SCADA*.

з корпоративними *IT*-системами та сприяє масштабуванню архітектури.

6.6. Проблеми та перспективи інтеграції

Інтеграція *SCADA* з *MySQL* супроводжується питаннями продуктивності, масштабованості, безпеки та експлуатаційної стійкості. За високої частоти оновлення технологічних параметрів основними джерелами проблем стають затримки операцій запису, пікові навантаження, конкуренція транзакцій і нераціональна структура запитів. Для забезпечення стабільної роботи необхідно передбачати механізми зменшення інтенсивності звернень, керування пакетним записом, оптимізацію схем даних, індексацію та регламенти життєвого циклу історичних даних.

Не менш важливою є інформаційна безпека, оскільки *SCADA*-системи працюють у критично важливих середовищах. Захист має включати обмеження доступу за принципом мінімальних привілеїв, сегментацію мережі, шифрування каналів зв’язку, аудит операцій та контроль облікових записів. Безпекові заходи мають узгоджуватися з вимогами до доступності, оскільки надмірно жорсткі політики можуть погіршувати експлуатаційну гнучкість.

Перспективи розвитку інтеграції пов’язані з переходом від суто транзакційної взаємодії до подієвих і потокових підходів, де зміни даних доставляються споживачам у вигляді подій із мінімальною затримкою. Це дозволяє зменшувати навантаження періодичного опитування і підвищувати оперативність реакції системи. У межах посібника такі підходи доцільно розглядати окремо як наступний крок розвитку інтеграційної архітектури, після засвоєння стандартних механізмів обміну даними між *SCADA* та *MySQL*.

РОЗДІЛ 7. ПОТОКОВА ОБРОБКА ДАНИХ

7.1. Вступ

У сучасних *SCADA*-системах, орієнтованих на роботу в реальному часі, класичні підходи до взаємодії з базами даних (наприклад, *JDBC* або *REST API*) виявляють суттєві обмеження. Головними проблемами є висока затримка запитів (латентність), надмірне навантаження на сервер БД через частий опит стану та відсутність механізму відстеження історії змін, що є критичним для технологічного моніторингу та діагностики.

Реальні технологічні об’єкти вимагають миттєвої реакції *SCADA*-системи на зміни в джерелі даних. Це передбачає наявність подій, що ініціюються не через періодичне опитування, а на момент фактичної зміни інформації. У цьому контексті виникає потреба переходу від класичної транзакційної моделі до потокової обробки даних.

Потокова передача даних (*data streaming*) – це підхід, при якому зміни передаються системі-споживачу негайно після виникнення у вигляді безперервного потоку подій. Ключовою технологічною концепцією тут є *Change Data Capture* (*CDC*), яка забезпечує виявлення та передачу лише змінених даних, без необхідності повторного зчитування всього стану бази.

CDC реалізується декількома способами: на основі *SQL*-запитів (query-based), з використанням тригерів (*trigger-based*) або шляхом аналізу журналів транзакцій БД (*log-based*). Саме l*og-based* підхід забезпечує найменшу затримку та найвищу точність. Він лежить в основі *Debezium* – інструменту для потокового зчитування змін з *MySQL* .

У результаті формується архітектура, де *MySQL* виступає джерелом подій, *Debezium* фіксує ці події, аналізуючи binlog, *Apache Kafka* виконує роль транспортного середовища для доставки подій, а *SCADA*-система виступає споживачем змін у реальному часі.

Для спрощення розгортання такої системи та уникнення залежностей від конкретного середовища всі її компоненти запускаються в контейнерах за

допомогою *Docker Desktop* для *Windows*. Цей підхід забезпечує стабільність, ізоляцію та гнучкість при налаштуванні та експлуатації потокової інфраструктури в інженерному середовищі.

7.2. Потокова передача змін через *binlog MySQL*

Бінарний журнал (*binlog*) у *MySQL* є критично важливим компонентом для реалізації потокової передачі змін у режимі реального часу. Він фіксує всі операції модифікації даних вставки, оновлення, видалення у вигляді послідовності подій низького рівня, забезпечуючи при цьому повну аудиторську слідковість і збереження цілісності інформації. Для застосування *binlog* у сценаріях *Change Data Capture* (*CDC*) необхідно належним чином налаштувати сервер *MySQL* .

Активація та конфігурація *binlog* здійснюється шляхом редагування конфігураційного файлу *MySQL my.ini*. В обов'язковому порядку потрібно встановити унікальний ідентифікатор сервера, задати шлях збереження бінарних логів, а також визначити формат *binlog* як *ROW*. Саме цей формат дозволяє фіксувати зміни на рівні окремих записів, що є ключовим для коректного відстеження змін. Параметр *binlog_row_image*, встановлений у *FULL*, забезпечує збереження повних станів даних до і після операції, що необхідно для точного відтворення змін у споживачів. Додатково налаштовується автоматичне очищення застарілих логів та синхронізація запису на диск після кожної транзакції, що підвищує надійність роботи системи.

Приклад файлу *my.ini*

```
[MySQL d]
# Обов'язкові параметри:
server-id = 1 # Унікальний ідентифікатор сервера
log_bin=/var/log/MySQL /MySQL -bin.log # Шлях до файлів логу
binlog_format = ROW # Єдиний формат, що фіксує зміни на рівні записів
```

```
    binlog_row_image = FULL # Записує повні "до" та
"після" стани даних
    expire_logs_days = 7 # Автоочищення логів старше 7
днів (опційно)

    # Додаткові налаштування для надійності:
    sync_binlog = 1 # Синхронізація логу на диск після
кожної транзакції
    gtid_mode = ON # Глобальні ідентифікатори транзакцій
(для кластерів)
    enforce_gtid_consistency = ON
```

Формат *ROW* є обов'язковим для ефективної роботи *CDC*, оскільки на відміну від форматів *STATEMENT* чи *MIXED* він фіксує фактичні зміни даних, гарантуючи точність навіть у складних випадках оновлень. Значення *FULL* для *binlog_row_image* дає можливість отримати повний знімок стану запису, що критично для аналітики та моніторингу технологічних процесів. Параметр *sync_binlog* із значенням 1 забезпечує мінімізацію ризику втрати даних при аваріях, що є пріоритетом у системах автоматизації, хоча може дещо впливати на продуктивність.

Для того щоб перевірити чи, увімкнений бінарний лог необхідно Підключіться до *MySQL* та виконати команду

```
SHOW VARIABLES LIKE 'log_bin';
```

7.3. Використання *Apache Kafka* для потокової обробки даних

Apache Kafka це розподілена платформа для обробки великих потоків даних у реальному часі, яка поєднує функції системи повідомлень, сховища даних та обробки потоків. Ключовою ідеєю *Kafka* є передача подій у вигляді послідовності повідомлень, що зберігаються у тематичних каналах, топіках (topics).

Кластер *Kafka* складається з кількох серверів, брокерів (*brokers*), які забезпечують зберігання, реплікацію та доставку повідомлень. Кожен топік (*topic*) ділиться на партиції (*partitions*), логічні сегменти, що дозволяють масштабувати навантаження та обробку паралельно між брокерами.

Повідомлення в межах кожної партиції мають унікальні ідентифікатори, офсети (offsets), які визначають порядок записів і дозволяють точно відстежувати прогрес читання.

Продюсери (*producers*) є джерелами даних у системі. Вони формують і надсилають повідомлення до топіків. Продюсер самостійно визначає, у який топік (topic) потрапить повідомлення, а також може впливати на вибір конкретної партиції для забезпечення контрольованого розподілу навантаження. Продюсерами можуть виступати як прості сенсори, що надсилають телеметричні дані, так і складні програмні компоненти, які формують події на основі бізнес-логіки.

З іншого боку, споживачі (*consumers*) підписуються на топіки та отримують звідти повідомлення для подальшої обробки, збереження чи візуалізації. Завдяки механізму офсетів (*offsets*) кожен споживач може зчитувати повідомлення детерміновано, навіть у разі збоїв або перезапуску. Для забезпечення масштабованої обробки *Kafka* дозволяє об’єднувати споживачів у групи (*consumer groups*), де кожен учасник обслуговує окрему партицію, що значно підвищує продуктивність.

Архітектурно *Kafka* забезпечує як масштабованість, так і відмовостійкість. Масштабованість досягається шляхом горизонтального розподілу партицій між кількома брокерами, а відмовостійкість через реплікацію партицій. Кожна партиція має одну основну (*leader*) копію та одну або кілька реплік (*replicas*), що зберігаються на інших брокерах. У разі відмови брокера з основною партицією *Kafka* автоматично призначає нову основну копію з наявних реплік, що забезпечує безперервність обробки подій.

Для координації компонентів *Kafka* застосовується допоміжний сервіс *Zookeeper*, який відповідає за керування метаданими, моніторинг стану брокерів і обрання провідних партицій. У новіших версіях *Kafka* поступово відмовляється від *Zookeeper* на користь внутрішньої системи керування метаданими, однак у більшості наявних розгортань *Zookeeper* залишається важливим елементом.

Загалом, злагоджена взаємодія брокерів, продюсерів, споживачів, партицій та координаційних служб утворює надійну інфраструктуру для обробки потокових подій у реальному часі. Це робить *Apache Kafka* надзвичайно ефективним інструментом для інтеграції в *SCADA* системи, де необхідна оперативна передача змін з баз даних до систем візуалізації та керування технологічними процесами.

7.4. *Debezium* у системах потокової передачі змін із *MySQL*

Щоб організувати автоматизовану обробку змін, зафіксованих у бінарному журналі *MySQL,* застосовується спеціалізований фреймворк з відкритим вихідним кодом, який інтегрується з *Apache Kafka* та реалізує підхід *Change Data Capture* (*CDC*). Йдеться про інструмент *Debezium*, що безперервно відстежує зміни у вказаній базі даних і транслює їх у вигляді подій *Kafka*.

Debezium використовує бінарний журнал змін *MySQL* (*binary log* або *binlog*) як головне джерело інформації. Цей журнал містить усі транзакційні події, що змінюють дані в таблицях: вставки, оновлення та видалення. Щоб увімкнути таке відстеження, сервер *MySQL* має бути налаштований на запис змін до бінарного журналу (binlog) з відповідними параметрами, як описано в розділі 7.2.

Debezium працює як *Kafka Connector*, тобто компонент, який підключається до зовнішньої бази *MySQL* і зчитує зміни безпосередньо з її журналу. Після обробки кожна зміна конвертується у повідомлення *Kafka*, яке транслюється у відповідну тему. Структура такої події чітко розділяє старий і новий стани запису, а також надає супровідну інформацію щодо джерела, часу та типу операції. Це дає змогу споживачам *Kafka*, зокрема *SCADA*, *MES* або аналітичним модулям, оперативно реагувати на зміни в базі даних.

Формат повідомлень, що створюються *Debezium*, є структурованим і заснованим на *JSON*. Кожне повідомлення включає секції *before* та *after*, які містять відповідно попередній і поточний стан зміненого запису. Окрім цього, подія містить метадані: ідентифікатор транзакції, часову мітку, тип операції

(CREATE, UPDATE, DELETE), назву таблиці та бази даних, а також точну позицію події у binlog. Такий формат забезпечує уніфікованість і гнучкість при обробці, дозволяючи *SCADA*-системам здійснювати селекцію подій, будувати історію змін або формувати алерти.

На відміну від тригерів у БД або ручного опитування, *Debezium* є є рішенням, яке не порушує роботу системи. Він не змінює структуру таблиць, не навантажує транзакції та дозволяє реалізувати асинхронну обробку подій з високою продуктивністю. Його архітектура забезпечує високу надійність та відмовостійкість, оскільки у разі збоїв або перезапуску компонент автоматично продовжує роботу з останньої зафіксованої позиції в журналі змін, зберігаючи гарантії доставки без втрат.

У типовій конфігурації *Kafka Connect* з *Debezium* слід забезпечити активований *binlog* на сервері *MySQL,* створити окремого користувача з правами REPLICATION SLAVE, REPLICATION *CLI*ENT та SELECT, а також надати параметри підключення. Обов'язковим є визначення *topic.prefix*, тобто префіксу іменування *Kafka*-тем, у які здійснюватиметься трансляція змін.

Події, згенеровані *Debezium* у форматі *JSON*, містять як самі змінені поля, так і метадані події. До таких метаданих належать ідентифікатор транзакції, час, тип операції (Create, Update, Delete), назва таблиці та бази даних, а також позиція у *binlog*. Завдяки цьому споживачі можуть здійснювати фільтрацію, агрегацію, маршрутизацію та інші операції, які є критичними для *SCADA*-систем або хмарних платформ обробки.

Однією з ключових переваг використання *Debezium* є можливість масштабування. Один потік *binlog* може обслуговувати кілька споживачів одночасно, наприклад, підсистему історичного архівування, модуль тривожної сигналізації та сервіс бізнес-аналітики. Окрім цього, оскільки кожна подія містить як старий, так і новий стан запису, *SCADA*-система може не лише обробляти подію в реальному часі, а й здійснювати ретроспективне відновлення станів об'єктів.

Таким чином, використання *Debezium* у комбінації з *Apache Kafka* є надійним і масштабованим підходом для побудови потокової архітектури збору та передачі змін з баз даних до технологічних систем. Такий підхід особливо ефективний у середовищах, де критично важлива оперативність, достовірність та цілісність даних, що характерно для *SCADA*-платформ у промисловій автоматизації.

7.5. Інтеграції *Kafka* і *MySQL* у *Windows*

Існує кілька підходів до інтеграції *MySQL* з *Apache Kafka* через *Debezium* в операційній системі *Windows*. Вони можуть відрізнятися залежно від вимог до масштабованості, простоти налаштування та підтримки контейнеризації.

Перший спосіб полягає у встановленні кожного компонента окремо, використовуючи стандартні інсталяційні пакети. *Apache Kafka* можна запустити, завантаживши дистрибутив з офіційного сайту та налаштувавши всі необхідні залежності, зокрема *Zookeeper*, який є важливим компонентом для керування *Kafka*. Для використання *Debezium*, що функціонує як частина *Kafka Connect*, необхідно його налаштувати та встановити відповідний плагін для *MySQL*. Цей варіант надає повний контроль над процесом конфігурації, що дозволяє здійснити тонке налаштування параметрів кожного модуля відповідно до потреб проєкту.

Другий підхід базується на використанні контейнерів, зокрема за допомогою *Docker*, для розгортання всієї системи, включаючи *MySQL, Apache Kafka* та *Debezium*.

Технологія контейнеризації є підходом до розгортання програмного забезпечення, за якого кожна програма разом з усіма необхідними для її роботи компонентами (бібліотеками, конфігураціями, залежностями) пакується в окрему ізольовану оболонку, тобто контейнер. Контейнери дозволяють запускати програму в однаковому середовищі незалежно від того, на якій операційній системі або апаратному забезпеченні вона розгортається. Завдяки

цьому розробники та адміністратори можуть уникати помилок, спричинених «несумісністю середовища».

Одним з найпоширеніших інструментів для роботи з контейнерами є *Docker*. Це відкрите програмне забезпечення, яке дає змогу створювати, запускати та управляти контейнерами. У *Docker* кожен сервіс, наприклад *MySQL* чи *Kafka*, може бути представлений як окремий контейнер, який легко запускати за допомогою коротких команд або описів конфігурацій у файлах *docker-compose.yml*.

Контейнеризація дозволяє ізолювати кожен компонент в окремому середовищі, забезпечуючи його незалежну роботу без потреби налаштовувати систему вручну для кожного сервісу. Контейнери включають усі необхідні залежності: операційну систему, бібліотеки та допоміжні компоненти. Завдяки цьому програму можна запускати в будь-якому середовищі, де працює *Docker*, незалежно від базової операційної системи (наприклад, *Windows* чи *Linux*).

Такий підхід суттєво спрощує налаштування та адміністрування, забезпечує однакову поведінку компонентів у різних середовищах, полегшує масштабування, оновлення та міграцію системи. Контейнери також усувають проблеми сумісності програмного забезпечення: кожен з них інкапсулює власні бібліотеки й залежності, запобігаючи конфліктам на хост-системі.

Це особливо важливо для складних архітектур, що включають *MySQL, Apache Kafka* та *Debezium*, оскільки кожен із компонентів має специфічні вимоги до версій та конфігурацій. Контейнеризоване середовище дає змогу швидко розгортати та підтримувати такі сервіси, забезпечуючи гнучкість, стабільність та високу доступність.

У технологічному середовищі *Debezium* підключається до *MySQL* як реплікаційний клієнт, аналогічно до вторинного сервера. Він постійно читає події binlog та перетворює їх у структуровані повідомлення, які зберігаються у брокері (*broker*) *Apache Kafka*. Затримка таких повідомлень зазвичай не перевищує 100 мілісекунд, що відповідає вимогам оперативного технологічного моніторингу.

7.6. Передача даних із *MySQL* до *SCADA*

Сучасні *SCADA*-системи активно інтегруються з потоковими платформами типу *Apache Kafka*, що відкриває можливості обробки даних у реальному часі, оперативного реагування на зміни в базах даних, формування аналітичних панелей та архівування параметрів. Для інтеграції з *Kafka* зазвичай застосовується проміжний шар, наприклад *Node-RED*. Цей візуальний інструмент на базі *Node.js* спеціалізований для швидкої організації потоків даних у промислових системах. Він дозволяє інженерам без глибокого програмування створювати логіку трансформації складних *JSON*-повідомлень з *Kafka* у спрощений формат, зрозумілий *SCADA*-системам, з подальшою передачею через *MQTT* або *REST API*. *Node-RED* виступає універсальним інтеграційним інструментом між компонентами, забезпечуючи не лише сумісність форматів, а й гнучке керування маршрутизацією, фільтрацією та відмовостійкістю доставки.

Деякі *SCADA*-платформи, зокрема *Siemens WinCC OA* та *Ignition* (з модулем *MQTT Engine Transmission*), підтримують прямий зв'язок з *Kafka* через вбудованих клієнтів. Однак такий підхід має суттєві обмеження: складність фільтрації подій, відсутність інструментів для попередньої агрегації чи нормалізації даних, а також ризики при зміні структури повідомлень. Саме тому для складних промислових систем перевага надається проміжним компонентам типу *Node-RED*, мікросервісам або скриптовим рушіям *SCADA*.

Ключовим аспектом є вибір протоколу передачі даних, *MQTT* чи *REST*. Протокол *MQTT* добре підходить для систем реального часу з потоками дрібних повідомлень, тоді як *REST* краще слугує для періодичної синхронізації чи менш інтенсивного трафіку. Більшість сучасних *SCADA*-платформ (*Ignition, Siemens WinCC OA, Schneider EcoStruxure),* підтримують обидва варіанти, а остаточний вибір залежить від конкретного сценарію.

Також критично важливо враховувати політику безпеки, зокрема автентифікацію *MQTT*-клієнтів за допомогою таких методів як *TLS*-сертифікати або пари логін/пароль, разом із обмеженням доступу до *Kafka* через механізми

ACL (*Access Control List*) для контролю операцій читання та запису у топики. Для гарантії надійності системи необхідно забезпечити буферизацію даних на проміжних компонентах, наприклад у *Node-RED*, на випадок тимчасової недоступності *SCADA*-системи, а також реалізувати механізми автоматичної повторної передачі при втраті мережевого з'єднання між *Kafka*, шлюзами та *SCADA*-платформою. Ці заходи є обов'язковими для підтримки цілісності даних у умовах промислової експлуатації.

Протокол *MQTT* (*Message Queuing Telemetry Transport*) розроблений з урахуванням потреб ефективної роботи в умовах обмежених обчислювальних ресурсів, мінімальних затримок та нестабільного мережевого з'єднання. Завдяки компактності, високій швидкодії та подієво орієнтованій моделі публікації/підписки, *MQTT* став стандартом у системах диспетчеризації, моніторингу та інтернету речей.

У середовищі *Windows MQTT* повноцінно реалізується на базі платформи *SCADA Ignition*, що забезпечує вбудовану підтримку цього протоколу. Завдяки цьому *Ignition* може безпосередньо взаємодіяти з розподіленими сенсорами, контролерами або мікроконтролерами, які передають дані у визначені теми. Отримані значення оновлюються у вигляді тегів у реальному часі без необхідності періодичного опитування, що суттєво знижує навантаження на мережу і підвищує ефективність обробки інформації.

Після отримання даних через *MQTT* платформа *Ignition* виконує їх обробку, перевірку достовірності та, за потреби, фільтрацію у режимі реального часу. Для передачі оброблених значень до зовнішньої бази даних *MySQL* існують два основні підходи.

Перший підхід передбачає використання вбудованого сценарного середовища на основі *Python* (*Jython*), яке автоматично записує значення тегів у таблиці бази даних з фіксацією часу події, вимірюваного значення, ідентифікатора сенсора чи технологічної зони.

Другий метод передбачає застосування зовнішнього середовища *Node.js* у поєднанні з графічним інструментом *Node-RED*, що забезпечує потокову

передачу даних з *MySQL* до *SCADA* Ignition без необхідності програмування в самій системі.

Архітектура передбачає, що польові пристрої, мікроконтролери або ПЛК публікують телеметричні дані у визначені теми *MQTT*. Повідомлення приймаються через *MQTT*-брокер, який може бути розгорнутий у *Windows* (наприклад, *Mosquitto*) або інтегрований як модуль у платформу *Ignition* через *MQTT Engine*. Платформа автоматично оновлює теги, виконує сценарії або подієву логіку і записує дані до бази *MySQL* . Збережені у структурованому вигляді дані використовуються для аналітики, формування звітів, візуалізації та експорту.

7.6.1. Формат даних і обробка в SCADA

Як було зазначено у попередніх розділах, *Debezium* формує повідомлення у форматі *JSON*. Такий формат має стандартну структуру зі службовими секціями, де зберігається інформація про подію, таблицю, тип операції і змінені поля. Нижче наведено типовий приклад повідомлення:

```
{
 "before": null,
 "after": {
  "id": 1,
  "temperature": 78.2,
  "timestamp": "2025-07-22T14:45:00Z"
 },
 "op": "c",
 "ts_ms": 1753454700000,
 "source": {
  "table": "sensors",
  "db": "monitoring"
 }
}
```

SCADA-система у більшості випадків не потребує всієї цієї інформації. Основний інтерес становить секція `after`, яка містить безпосередньо значення полів після внесення змін. З цією метою у *Node-RED* використовується *JSON*-

парсер і функціональний блок для виділення ключових параметрів. Потім формується нове повідомлення у спрощеному вигляді:

```
{
 "id": 1,
 "temperature": 78.2,
 "timestamp": "2025-07-22T14:45:00Z"
}
```

Це повідомлення публікується через *MQTT* у відповідний топік, який уже підписаний *SCADA*-додатком.

7.6.2. Приклад передачі даних у SCADA Ignition

У *SCADA*-середовищі *Ignition* існує вбудований драйвер *MQTT Transmission/Engine*, який дозволяє підписуватись на *MQTT*-топіки і автоматично створювати теги на основі отриманих *JSON*-повідомлень. У середовищі *Tag Browser* створюється з'єднання з брокером, наприклад *ECLIpse Mosquitto*, і вказується шаблон структури тега. Якщо повідомлення має вигляд як у прикладі вище, то тег *temperature* створюється автоматично і отримує значення 78.2.

Ignition також підтримує обробку *timestamp* та *ID* як окремих тегів або використання цих параметрів у скриптах для логування в базу даних, побудови історичних графіків чи активації тривог. Завдяки цьому можна реалізувати повноцінний канал обміну між базою даних *MySQL* та *SCADA* через *Kafka* у режимі, близькому до реального часу.

7.7. Налаштування *Docker* для інтеграції *Apache Kafka*, *MySQL* та *SCADA*

Інтеграція *Apache Kafka*, *MySQL* та *SCADA* у середовищі *Docker* дозволяє створити ефективну, ізольовану та гнучку інфраструктуру для потокової обробки даних. Основу цієї системи становить *Docker Compose* – інструмент для керування багатоконтейнерними застосунками, який відрізняється від

базового *Docker* тим, що дозволяє одночасно запускати та координувати кілька взаємопов'язаних контейнерів через єдину конфігурацію.

Без *Docker Compose* інженеру довелося б вручну запускати окремі контейнери для *MySQL, Kafka* та *Zookeeper*, складним чином налаштовуючи параметри їхньої взаємодії. З Docker Compose процес спрощується до двох кроків: по-перше, створення файлу *docker-compose.yml* з централізованим описом усіх сервісів та їхніх залежностей; по-друге, виконання єдиної команди docker-compose up, яка автоматично створює спільну мережу для контейнерів, запускає компоненти у необхідній послідовності (наприклад, *Zookeeper* перед *Kafka*) та конфігурує зв'язок між ними. Це нагадує використання креслення для автоматизації будівництва складного об'єкта замість ручного складання кожного елемента окремо.

У такій конфігурації *MySQL* виступає в ролі бази даних, де зберігаються структуровані технологічні дані з об'єктів автоматизації. *Apache Kafka*, як система обробки повідомлень, приймає зміни в базі даних за допомогою механізму *Change Data Capture* (*CDC*), реалізованого через *Debezium*. Зі свого боку, *SCADA*-система, яка працює на іншому сервері або робочій станції, може підключатися до брокера повідомлень *Kafka* для читання потоків у реальному часі, а також здійснювати запис у *MySQL* для подальшого розповсюдження змін.

Нижче наведено приклад конфігураційного файлу *docker-compose.yml*, який забезпечує одночасний запуск служб *Zookeeper*, *Kafka*, *Kafka Connect* (з підтримкою *Debezium*) та *MySQL*. Усі сервіси взаємодіють у спільній мережі *kafka-net*, що забезпечує надійну маршрутизацію запитів між контейнерами.

```
services:

 # Координаційний сервіс для Kafka (керування метаданими, лідерами партицій)
 zookeeper:
  image: confluentinc/cp-zookeeper:latest
  environment:
   ZOOKEEPER_CLIENT_PORT: 2181
```

```
 # Основний брокер потокових повідомлень
 kafka:
  image: confluentinc/cp-kafka:latest
  depends_on:
   - zookeeper
  environment:
   KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181 # Підключення
до Zookeeper
   KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092 #
Адреса для клієнтів
   KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1 # Фактор
реплікації системних топиків

 # Джерело даних з активованим binlog (ROW-формат) для
CDC
 MySQL :
--server-id=1
 --log-bin=/var/lib/MySQL /MySQL -bin.log
 --binlog-format=ROW
 --binlog_row_image=FULL
 --gtid-mode=ON
 --enforce-gtid-consistency=ON
  environment:

 MYSQL _ROOT_PASSWORD: rootpass # Обов’язково змінити для
продакшену!
   MYSQL _DATABASE: testdb # Тестова БД
  volumes:
   -         ./MySQL         /init.SQL:/docker-entrypoint-
initdb.d/init.SQL # Ініціалізаційні скрипти

 # Сервіс для інтеграції MySQL ->Kafka через Debezium
 connect:
  image: debezium/connect:latest
  environment:
   BOOTSTRAP_SERVERS: kafka:9092 # Підключення до Kafka
   # Конвертери для формату повідомлень (JSON)
   KEY_CONVERTER:
org.apache.kafka.connect.json.JsonConverter
   VALUE_CONVERTER:
org.apache.kafka.connect.json.JsonConverter
  volumes:
   - ./connect/debezium... # Плагін Debezium (можна
використати вбудований)
```

У цьому прикладі контейнери *zookeeper*, *kafka* та *kafka-connect* запускаються з використанням офіційних образів *Confluent* та *Debezium*. Контейнер *Zookeeper* забезпечує координацію *Kafka*-брокерів. *Kafka* виконує роль брокера повідомлень. Контейнер *kafka-connect* запускає службу *Kafka Connect* із *Debezium*-плагіном, яка дозволяє здійснювати захоплення змін у *MySQL* у вигляді потокових подій, що публікуються в *Kafka*.

Зовнішня *SCADA*-система, наприклад *Ignition*, може підключатися до *Kafka* через *MQTT*-шлюз або *REST API*, якщо її можливості це підтримують. Альтернативно, вона може працювати безпосередньо з *MySQL,* яку *Kafka*-інфраструктура оновлює через *CDC*. Таким чином, досягається гнучка інтеграція системи моніторингу й управління з потоковими джерелами даних.

7.8. Типова архітектура потокової обробки даних

Розглянута в попередніх розділах архітектура потокової обробки даних інтегрує низку технологій для ефективного функціонування систем реального часу. Наведена на рис.7.1 схема демонструє взаємозв'язок між основними компонентами, де кожен елемент виконує чітко визначену роль у загальному ланцюжку обробки інформації.

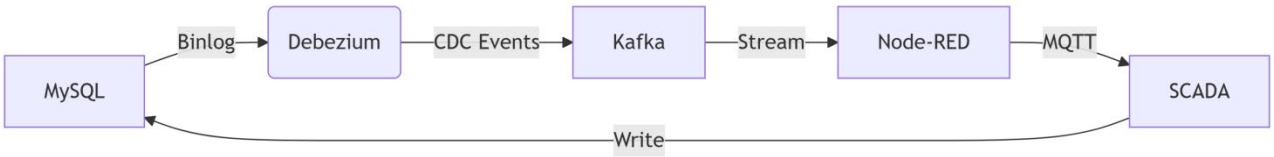


Рисунок 7.1. Архітектура потокової обробки даних

Зображена схема дозволяє краще усвідомити функціональний розподіл ролей у системі.

MySQL виступає основним джерелом даних, де активовано бінарний журнал змін у форматі *ROW*. Це дозволяє фіксувати всі операції модифікації даних на рівні окремих записів, що є критично важливим для подальшої потокової обробки.

Debezium, працюючи як інструмент *Change Data Capture*, постійно моніторить зміни в бінарному журналі *MySQL*. Він перетворює ці зміни у структуровані події у форматі *JSON*, які потім передаються до *Apache Kafka*, яка є ядроми всієї потокової інфраструктури. *Kafka* забезпечує надійне зберігання та доставку повідомлень, дозволяючи обробляти великі обсяги даних у реальному часі з мінімальними затримками.

Для подальшої інтеграції з *SCADA*-системами використовується *Node-RED*, який виконує роль проміжного шару. Він відповідає за трансформацію складних *JSON*-повідомлень у спрощений формат, зрозумілий для *SCADA*-систем, а також за маршрутизацію даних через протокол *MQTT*, оптимізований для роботи в умовах обмежених ресурсів та нестабільних мережевих з'єднань.

Фінальним інструментом є *SCADA*-система, така як Ignition або *WinCC OA*, яка отримує дані у реальному часі, візуалізує їх та ініціює відповідні події управління. Використання *MQTT* дозволяє *SCADA*-системі оперативно реагувати на зміни без необхідності періодичного опитування, що суттєво знижує навантаження на мережу та підвищує ефективність роботи.

Ключовою перевагою такої архітектури є її масштабованість та гнучкість. Завдяки використанню *Kafka* та *Docker*-контейнерів система може легко адаптуватися до зростаючого навантаження, а ізольованість компонентів забезпечує стабільність навіть у разі збоїв окремих служб.

РОЗДІЛ 8. БЕЗПЕКА ТА НАДІЙНІСТЬ *MYSQL*

8.1. Загальні положення

Забезпечення безпеки та надійності бази даних *MySQL* є ключовим завданням для будь-якої *SCADA*-системи. Бази даних у таких системах зберігають важливі технологічні параметри, історію процесів та команди керування. Тому система керування базами даних повинна гарантувати цілісність, конфіденційність та доступність даних навіть за умов збоїв обладнання, несанкціонованого доступу чи навмисних атак. *MySQL* підтримує розвинений механізм автентифікації, систему прав користувачів, шифрування даних, а також засоби резервного копіювання та реплікації, що безпосередньо впливають на стабільну роботу *SCADA*.

8.2. Автентифікація та керування доступом

Система управління користувачами та правами доступу в *MySQL* є основним механізмом забезпечення безпеки бази даних. Вона дозволяє адміністраторам контролювати, хто і які дії може виконувати з даними, дотримуючись принципу мінімально необхідного доступу.

Важливо правильно налаштувати облікові записи *MySQL* та їхні права. Для кожного користувача визначаються ім’я, хост підключення, пароль та набір привілеїв. Привілеї поділяються на глобальні, які діють для всіх баз даних сервера, та специфічні, що застосовуються до окремих баз, таблиць або стовпців. У *SCADA*-системах це дозволяє, наприклад, надавати оператору лише права перегляду поточних даних, аналітику права на читання історичних даних, а адміністратору повний доступ до керування базою.

Створення нового користувача виконується командою `CREATE USER`. Починаючи з сучасних версій *MySQL*, можна вказати алгоритм автентифікації, політику пароля та додаткові параметри безпеки, наприклад обмеження доступу за IP, термін дії пароля та захист від перебору:

```
    CREATE USER ‘app_user’@’192.168.1.%’
    IDENTIFIED     WITH     caching_sha2_password     BY
‘Strong_Passw0rd!’
    PASSWORD EXPIRE INTERVAL 180 DAY
    FAILED_LOGIN_ATTEMPTS 5
    PASSWORD_LOCK_TIME 2;
```

Надання прав доступу здійснюється командою `GRANT`. Наприклад:

```
    GRANT   SELECT,   INSERT,   UPDATE   ON   db_prod.*   TO
‘app_user’@’192.168.1.%’;
```

Для спрощення адміністрування доступу застосовуються ролі, які дозволяють групувати привілеї та призначати їх користувачам без дублювання:

```
    CREATE ROLE ‘report_role’;
    GRANT SELECT ON db_prod.* TO ‘report_role’;
    GRANT ‘report_role’ TO ‘analyst’@’%’;
    SET DEFAULT ROLE ‘report_role’ TO ‘analyst’@’%’;
```

Перевірити надані права можна командою:

```
    SHOW GRANTS FOR ‘app_user’@’192.168.1.%’;
```

Іноді адміністратор хоче дозволити користувачу делегувати свої привілеї іншим. Для цього в команді `GRANT` використовується параметр `WITH GRANT OPTION`:

```
    GRANT ALL PRIVILEGES ON db_admin.* TO
‘dba’@’localhost’ WITH GRANT OPTION;
```

Тут користувач `dba` отримує всі привілеї на базу `db_admin` і може самостійно надавати частину своїх прав іншим користувачам. У *SCADA*-системах така можливість застосовується рідко, зазвичай для ролей адміністраторів бази даних, тому її слід використовувати обережно.

Важливою можливістю є завершення небажаних сесій або запитів. Для цього застосовують дві команди:

```
KILL CONNECTION 12345; - завершує з'єднання
KILL QUERY 12345;   - перериває лише поточний запит
```

Ідентифікатор з'єднання можна отримати за допомогою команди `SHOW PROCESSLIST;`.

MySQL підтримує кілька рівнів надання прав: глобальний, рівень бази даних, таблиці та стовпців. *MySQL* підтримує кілька рівнів надання прав: глобальний, рівень бази даних, таблиці та стовпців.

Контроль доступу до окремих рядків у *MySQL* реалізується кількома способами. Один із найпоширеніших серед них, використання уявлень **(***VIEW***)** з параметром `SQL SECURITY INVOKER`, що дозволяє обмежити видимі дані за допомогою умов `WHERE`. Наприклад, оператору *SCADA* можна надати доступ лише до даних певної технологічної лінії:

```
CREATE VIEW operator_view AS
SELECT *
FROM sensor_data
WHERE line_id = 1
SQL SECURITY INVOKER;
```

У цьому випадку запити до `operator_view` виконуються під обліковим записом користувача, який робить запит. Це означає, що користувач бачить лише ті рядки, на які він має права доступу, навіть якщо VIEW створив інший користувач.

Для порівняння існує ще один режим: `SQL SECURITY DEFINER`. У ньому запити виконуються під обліковим записом користувача, який створив `VIEW` або збережену процедуру. Таким чином, всі користувачі, які виконують запити, отримують доступ до даних відповідно до прав автора об'єкта, а не своїх власних.

Вибір між `INVOKER` і `DEFINER` залежить від того, чи потрібно реалізувати контроль доступу на рівні кожного користувача (`INVOKER`), чи забезпечити однаковий доступ усім користувачам, незалежно від їхніх прав (`DEFINER`).

Додатково контроль доступу можна реалізувати через збережені процедури**,** які повертають лише дозволені дані, надаючи користувачам доступ тільки до виклику процедури без прямого доступу до базових таблиць.

Ще один рівень контролю забезпечує логіка прикладного рівня *SCADA*. Інтерфейс або сервер *SCADA* обмежує доступ до даних відповідно до прав користувача, навіть якщо база даних містить більше інформації.

8.3. Шифрування даних

Захист даних у *SCADA*-системах потребує застосування механізмів шифрування як під час передавання інформації мережею, так і на етапі її зберігання в базі даних. Шифрування є важливим елементом забезпечення конфіденційності та цілісності технологічної інформації, особливо в системах реального часу, де дані безперервно передаються між диспетчерським рівнем і сервером бази даних. Навіть у разі несанкціонованого доступу до мережі або сховища даних застосування шифрування унеможливлює їх коректну інтерпретацію без відповідних ключів.

Передавання даних між *SCADA*-клієнтами та сервером *MySQL* захищається за допомогою протоколу *TLS* (*Transport Layer Security*). *TLS* формує зашифрований канал зв’язку, через який передаються *SQL*-запити, відповіді сервера, облікові дані користувачів і значення технологічних параметрів. Це дозволяє запобігти перехопленню або модифікації даних у процесі обміну. Для використання *TLS* необхідно налаштувати сертифікати сервера та, за потреби, центру сертифікації. Сертифікати можуть бути створені як внутрішніми засобами підприємства, так і отримані від комерційних або корпоративних центрів сертифікації.

У середовищі *Windows* для роботи *TLS* достатньо вказати відповідні сертифікати у конфігураційному файлі *MySQL* та примусово дозволити використання захищених з'єднань. Приклад мінімальної конфігурації наведено нижче:

```
[mysqld]
ssl-ca=C:/mysql/certs/ca-cert.pem
ssl-cert=C:/mysql/certs/server-cert.pem
ssl-key=C:/mysql/certs/server-key.pem
require_secure_transport=ON
```

На стороні *SCADA*-клієнта або іншого клієнтського програмного забезпечення *MySQL* задаються параметри перевірки сертифіката сервера:

```
[client]
ssl-ca=C:/mysql/certs/ca-cert.pem
```

Факт використання захищеного з'єднання може бути перевірений за допомогою стандартних системних змінних *MySQL*, що відображають стан *SSL/TLS*.

Окрім захисту каналу передавання, *MySQL* підтримує шифрування даних на рівні зберігання. Для таблиць механізму InnoDB передбачено вбудоване шифрування з використанням стандарту *AES* (*Advanced Encryption Standard*). *AES* є симетричним алгоритмом блочного шифрування з фіксованою довжиною блоку 128 біт і підтримкою ключів різної довжини, що забезпечує належний рівень криптографічної стійкості при помірних обчислювальних витратах.

Шифрування може бути активоване як для всіх нових таблиць *InnoDB*, так і вибірково для окремих таблиць. Наприклад, для конкретної таблиці режим шифрування вмикається або вимикається стандартними засобами керування схемою бази даних. Такий підхід доцільний у *SCADA*-системах для захисту критично важливих архівних даних або конфіденційної технологічної інформації без необхідності шифрування всього сховища.

Додатково *MySQL* надає можливість шифрування окремих полів таблиць за допомогою функцій `AES_ENCRYPT` та `AES_DECRYPT`. Це дозволяє

захищати вибрані атрибути, наприклад ідентифікаційні або персональні дані, не змінюючи структуру всієї таблиці. Такий підхід може застосовуватися у допоміжних підсистемах *SCADA*, де разом із технологічними параметрами зберігається службова або облікова інформація.

Паролі користувачів *MySQL* зберігаються у вигляді криптографічних хешів і не зберігаються у відкритому вигляді. Для сучасних інсталяцій рекомендовано використовувати алгоритм *caching_sha2_password* та дотримуватися політик регулярної зміни паролів і обмеження доступу.

Таким чином, поєднання *TLS* для захисту каналу зв'язку та механізмів шифрування на рівні зберігання забезпечує базовий, але достатній рівень безпеки даних *MySQL* у *SCADA*-орієнтованих системах автоматизації без надмірного ускладнення їх архітектури.

8.4. Резервне копіювання

Надійна робота *SCADA*-системи неможлива без систематичного створення резервних копій бази даних. База даних у таких системах зберігає критично важливу технологічну інформацію, включаючи поточні значення параметрів, історію процесів та службові дані, відновлення яких після збоїв є обов'язковою умовою безперервності керування. Тому резервне копіювання розглядається як складова загальної стратегії забезпечення надійності та відмовостійкості *SCADA*.

У практиці експлуатації *MySQL* застосовують декілька підходів до резервного копіювання, що відрізняються обсягом даних і швидкістю відновлення. Повне резервне копіювання передбачає збереження всього вмісту бази даних, включаючи таблиці, індекси, схеми та допоміжні об'єкти. Такий тип копії є базовим і використовується як вихідна точка для подальшого відновлення системи після серйозних відмов.

Інкрементальне резервне копіювання охоплює лише ті зміни, що відбулися після останнього повного резервного копіювання. Цей підхід зменшує обсяг даних, що зберігаються, і скорочує час створення копій, що є

важливим для *SCADA*-систем із великим обсягом історичних даних. Водночас відновлення з інкрементальних копій потребує послідовного застосування повного бекапу та всіх наступних інкрементів.

Диференціальне резервне копіювання зберігає всі зміни, що накопичилися з моменту останнього повного бекапу. Такий підхід займає більше дискового простору порівняно з інкрементальним, але дозволяє швидше відновити базу даних, оскільки для цього достатньо лише повної та останньої диференціальної копії.

У середовищі *Windows* для логічного резервного копіювання *MySQL* зазвичай застосовуються стандартні засоби, які формують дампи бази даних у вигляді *SQL*-скриптів. Ці копії можуть бути відновлені на будь-якому сумісному сервері *MySQL* і не залежать від конкретної апаратної платформи. Приклад створення повної резервної копії подано нижче:

```
mysqldump -u username -p database_name >
C:\backups\backup_full.sql
```

Для *SCADA*-систем принципово важливо не лише створювати резервні копії, але й забезпечувати регулярність цього процесу та контроль його виконання. Автоматизація резервного копіювання в *Windows* зазвичай реалізується за допомогою системних засобів планування завдань, що дозволяє виконувати бекапи у визначений час без участі оператора. Окрему увагу приділяють веденню журналів виконання та перевірці можливості відновлення даних із резервних копій.

Таким чином, резервне копіювання в *SCADA*-орієнтованих системах на базі *MySQL* виконує не допоміжну, а критично важливу функцію. Воно забезпечує збереження технологічної інформації, скорочує час відновлення після відмов і є необхідною умовою безпечної та безперервної експлуатації автоматизованих систем керування.

8.5. Реплікація і відмовостійкість

Для забезпечення безперервності роботи *SCADA MySQL* застосовує механізми реплікації. Класична схема Master-Slave дозволяє дублювати дані основного сервера на підлеглі вузли, що дає можливість балансування навантаження та швидкого відновлення після відмови. Сучасні можливості, такі як Group Replication і InnoDB Cluster, детально розглянуті у Розділі 5. Вони забезпечують багатовузлову синхронізацію з автоматичним виявленням збоїв і самовідновленням, що гарантує неперервність збору та обробки технологічних даних *SCADA* навіть при виході з ладу одного з серверів.

8.6. Журнали та моніторинг у *SCADA*-орієнтованих системах на базі *MySQL*

Журнали та засоби моніторингу відіграють ключову роль у забезпеченні безпеки, надійності та стабільності роботи *SCADA*-систем, що використовують *MySQL* як центральне сховище технологічних даних. Вони дозволяють відстежувати коректність обміну інформацією між *SCADA*-клієнтами та сервером бази даних, своєчасно виявляти помилки, затримки в обробці запитів і потенційні порушення політик доступу.

MySQL підтримує декілька типів журналів, кожен з яких виконує окрему функцію в інфраструктурі автоматизованих систем керування. Журнал помилок (*error log*) використовується для реєстрації критичних подій роботи сервера, збоїв запуску, помилок доступу та внутрішніх відмов. Загальний журнал запитів (*general query log*) фіксує всі *SQL*-запити, що надходять до сервера, і може застосовуватися для аналізу поведінки клієнтів *SCADA*, зокрема у процесі налагодження або аудиту. Журнал повільних запитів (slow query log) дозволяє виявляти неефективні операції читання або запису, які можуть негативно впливати на оперативність відображення технологічних параметрів. Бінарний журнал (binary log) є основою механізмів реплікації, відновлення даних і реалізації подієвих архітектур, зокрема *Change Data Capture*.

У контексті *SCADA* використання журналів *MySQL* дозволяє аналізувати, які запити формуються для читання поточних значень тегів, отримання історичних даних або запису команд керування. Це є важливим для виявлення вузьких місць у взаємодії між диспетчерським рівнем і сервером бази даних, а також для підтвердження коректності роботи прикладного програмного забезпечення.

Окрім журналювання, важливим компонентом експлуатації є постійний моніторинг стану сервера *MySQL*. Вбудований механізм performance_schema забезпечує збір статистики щодо часу виконання запитів, використання ресурсів, активності з'єднань і поведінки користувачів. На основі цих даних можуть будуватися системи зовнішнього моніторингу, які дозволяють у реальному часі контролювати продуктивність бази даних, виявляти аномальні навантаження та своєчасно реагувати на деградацію сервісу.

Інтеграція журналів і засобів моніторингу з *SCADA*-системою забезпечує комплексний підхід до контролю надійності зберігання та обробки технологічної інформації. Вона дозволяє не лише фіксувати події та помилки, але й формувати основу для аналізу причин аварійних ситуацій, оптимізації роботи запитів і підвищення загальної стійкості автоматизованої системи керування. Саме тому журнали та моніторинг розглядаються як обов'язкові елементи інфраструктури *MySQL* у промислових *SCADA*-рішеннях.

РОЗДІЛ 9. ПРАКТИЧНІ ПРИЙОМИ РОБОТИ З *MYSQL* ДЛЯ МОНІТОРИНГУ ТА ІНТЕГРАЦІЇ ДАНИХ

У цьому розділі наведено практичні підходи до використання *MySQL* у середовищі *Windows* для моніторингу, обробки та передавання технологічних даних у режимі, наближеному до реального часу. Матеріал спирається на поняття й моделі, розглянуті раніше, та показує, як засобами *MySQL* забезпечують роботу з частими оновленнями даних, контроль стану сервера, а також інтеграцію з прикладними інженерними системами.

Спочатку розглянуто налаштування сервера для спостереження за змінами та навантаженням. Для цього використано двійковий журнал *MySQL (binary log) як джерело інформації про операції зміни даних і підсистему Performance Schema* як стандартний механізм збору статистики щодо виконання запитів. Окремо розглянуто тригери як приклад інструмента прикладного рівня, що дає змогу автоматизувати журналювання змін у таблицях у межах самої бази даних.

Далі подано базові підходи до організації реплікації *Master-Slave*, а також приклади прийомів підвищення ефективності *SQL*-запитів у задачах, де важливими є затримка відповіді та стабільність роботи під навантаженням. У цьому ж контексті показано типові напрями інтеграції *MySQL* з інженерними програмними середовищами та *SCADA*-системами.

Як приклад сучасної *SCADA*-платформи в розділі обрано *Ignition*. Це зроблено з методичних міркувань, оскільки Ignition має виразну шлюзову архітектуру, підтримує стандартне підключення до СУБД через *JDBC* та широко застосовується у практиці промислової автоматизації. Такі властивості дають змогу на одному прикладі продемонструвати типові сценарії взаємодії з *MySQL*, зокрема читання для візуалізації, запис для архівування та організацію обміну даними через повідомлення. Водночас у викладенні збережено інженерну узагальненість: описані підходи можуть бути відтворені в інших *SCADA*-системах за наявності аналогічних механізмів підключення та

інтеграційних модулів, тоді як *MySQL* у межах посібника залишається основною СУБД і центральним сховищем технологічних даних.

Завершальні підрозділи присвячено передаванню даних між *MySQL* та зовнішніми компонентами у двох логіках. Перша логіка базується на опитуванні, коли прикладна система періодично виконує запити до *MySQL* і отримує актуальний стан. Друга логіка базується на передаванні змін, тобто на захопленні змін даних (*Change Data Capture, CDC*), коли до споживачів доставляються події про внесені зміни без потреби постійного опитування. У цьому зв'язку розглянуто місце *Node-RED* як проміжного інтеграційного середовища та *MQTT* як протоколу обміну повідомленнями через брокер (*broker*). Для подієвих сценаріїв окреслено роль *Apache Kafka* як транспортного середовища подій і *Debezium* як інструмента зчитування змін із двійкового журналу *MySQL* та публікації їх для подальшої обробки.

9.1. Моніторинг у *MySQL*: *binary log* і *Performance Schema*

У цьому підрозділі розглянуто налаштування *MySQL* у середовищі Windows для моніторингу даних у режимі, близькому до реального часу, з використанням стандартних засобів *MySQL*: двійкового журналу (*binary log, binlog*), підсистеми *Performance Schema* та механізмів реплікації *Master-Slave*.

Для виконання налаштувань має бути встановлено *MySQL* Server для Windows. Для адміністрування та виконання *SQL*-запитів доцільно використовувати графічний клієнт *MySQL Workbench* або консольний клієнт mysql.exe у командному рядку. Потрібен обліковий запис з правами адміністратора, а також доступ до конфігураційного файлу my.ini, який зберігається в каталозі *C:\ProgramData\MySQL\MySQL Server* 8.0.

Інсталятор *MySQL* завантажується з офіційного сайту https://dev.mysql.com/downloads/installer. Під час інсталяції слід обрати компоненти *MySQL* Server і *MySQL Workbench*, задати пароль для користувача root та налаштувати автоматичний запуск служби *MySQL* у *Windows*.

Після завершення інсталяції потрібно перевірити, що служба *MySQL* запущена у вікні «Служби *Windows*». Підключення до сервера виконується або через *MySQL* Workbench, або через командний рядок командою

```
mysql -u root -p
```

Для увімкнення *Binary Log* необхідно відкрити файл *my.ini* у каталозі *C:\ProgramData\MySQL\MySQL Server* 8.0. У секції [mysqld] додаються або розкоментуються такі параметри:

```
[mysqld]
log-bin = mysql-bin
server-id = 1
expire_logs_days = 7
```

Після збереження змін необхідно перезапустити службу *MySQL* у вікні «Служби *Windows*». Стан увімкнення двійкового журналу перевіряють запитом:

```
SHOW VARIABLES LIKE 'log_bin';
```

Якщо значення дорівнює ON, журналювання змін увімкнено.

Для використання *Performance Schema* необхідно переконатися, що у файлі my.ini встановлено параметр performance_schema = ON. Перевірка виконується запитом:

```
SHOW VARIABLES LIKE 'performance_schema';
```

Дані про роботу запитів накопичуються в системній базі *performance_schema*. Для перегляду узагальненої статистики виконання запитів можна застосувати запит:

```
SELECT *
FROM
performance_schema.events_statements_summary_by_digest;
```

Для локальної реплікації *Master-Slave* на одному комп'ютері необхідно створити дві інсталяції *MySQL* з різними портами або використати *Docker*. У конфігурації кожного вузла задається унікальний *server-id*. Для вузла *Master* задають, зокрема, увімкнення *binary log* та *server-id*, наприклад:

```
[mysqld]
log-bin = mysql-bin
server-id = 1

Для вузла Slave задають власний server-id та relay-log,
наприклад:

[mysqld]
server-id = 2
relay-log = relay-log
```

Стан реплікації на вузлі *Slave* перевіряють командою:

```
SHOW SLAVE STATUS\G
```

Перегляд двійкових журналів змін виконується утилітою *mysqlbinlog*, яка розміщується в каталозі *bin* інсталяції *MySQL*. Приклад виклику:

```
mysqlbinlog "C:\ProgramData\MySQL\MySQL Server
8.0\Data\mysql-bin.000001"
```

Для автоматизації обробки змін у навчальних або дослідницьких задачах можуть застосовуватися скрипти на *Python* з бібліотекою mysql-connector-python, які дозволяють відстежувати операції *INSERT, UPDATE, DELETE* та виконувати додаткову обробку отриманих подій.

9.2. Тригери *MySQL* та журналювання змін технологічних параметрів

У цьому розділі розглянуто створення та використання тригерів у *MySQL* для автоматизації обробки даних і журналізації змін у технологічних процесах.

На прикладі контролю параметрів парового котла демонструється автоматичний аудит змін, включаючи фіксацію температури, тиску та витрати пари у спеціальній таблиці аудиту для подальшого аналізу.

Тригер у *MySQL* є спеціальним об'єктом, який автоматично виконує *SQL*-інструкцію у відповідь на події вставки, оновлення або видалення в таблиці. Вони застосовуються для автоматизації внутрішніх процедур бази даних: ведення журналів змін, перевірок даних, виконання обмежень і контролю станів у реальному часі. У цьому прикладі тригери реалізують аудит параметрів котла та зберігають усі зміни у таблиці аудиту.

Для збереження параметрів котла та їх подальшого аналізу створюється база даних `boiler_monitoring` і дві таблиці: `boiler_data` для робочих значень та `boiler_audit` для історії змін. Для цього виконуються наступні команди

```
CREATE DATABASE boiler_monitoring;

USE boiler_monitoring;

CREATE TABLE boiler_data (
  record_id INT AUTO_INCREMENT PRIMARY KEY,
  temperature DOUBLE NOT NULL,
  pressure DOUBLE NOT NULL,
  steam_flow DOUBLE NOT NULL,
  measured_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE boiler_audit (
  audit_id INT AUTO_INCREMENT PRIMARY KEY,
  action VARCHAR(50),
  old_temperature DOUBLE,
  new_temperature DOUBLE,
  old_pressure DOUBLE,
  new_pressure DOUBLE,
  old_steam_flow DOUBLE,
  new_steam_flow DOUBLE,
  changed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Для автоматичного фіксування змін параметрів котла створюється тригер `before_boiler_update`, який перед оновленням запису перевіряє значень і у разі необхідності записує інформацію в таблицю аудиту:

```
DELIMITER $$
CREATE TRIGGER before_boiler_update
BEFORE UPDATE ON boiler_data
FOR EACH ROW
BEGIN
  IF OLD.temperature != NEW.temperature
    OR OLD.pressure != NEW.pressure
    OR OLD.steam_flow != NEW.steam_flow THEN
    INSERT INTO boiler_audit (
      action, old_temperature, new_temperature,
      old_pressure, new_pressure,
      old_steam_flow, new_steam_flow
    ) VALUES (
      'UPDATE',
      OLD.temperature, NEW.temperature,
      OLD.pressure, NEW.pressure,
      OLD.steam_flow, NEW.steam_flow
    );
  END IF;
END $$
DELIMITER ;
```

Перевірка тригера здійснюється шляхом додавання початкового запису та його оновлення:

```
    INSERT INTO boiler_data (temperature, pressure,
steam_flow) VALUES (184.5, 1.2, 3.5);
    UPDATE boiler_data SET temperature = 188.0, pressure
= 1.3, steam_flow = 3.6 WHERE record_id = 1;
    SELECT * FROM boiler_audit;
```

audit_id	action	old_temperature	new_temperature	old_pressure	new_pressure	old_steam_flow	new_steam_flow	changed_at
1	UPDATE	184.5	188	1.2	1.3	3.5	3.6	2025-07-16 22:11:48

Для автоматичного фіксування додавання нових записів створюється тригер `after_boiler_insert`:

```
DELIMITER $$
CREATE TRIGGER after_boiler_insert
AFTER INSERT ON boiler_data
FOR EACH ROW
BEGIN
  INSERT INTO boiler_audit (
    action, old_temperature, new_temperature,
    old_pressure, new_pressure,
    old_steam_flow, new_steam_flow
  ) VALUES (
    'INSERT',
    0, NEW.temperature,
    0, NEW.pressure,
    0, NEW.steam_flow
  );
END $$
DELIMITER ;
```

Тестування тригера проводиться додаванням нового запису:

```
INSERT INTO boiler_data (temperature, pressure, steam_flow) VALUES (190.0, 1.4, 3.7);
SELECT * FROM boiler_audit;
```

audit_id	action	old_temperature	new_temperature	old_pressure	new_pressure	old_steam_flow	new_steam_flow	changed_at
1	UPDATE	184.5	188	1.2	1.3	3.5	3.6	2025-07-16 22:11:48
2	INSERT	0	190	0	1.4	0	3.7	2025-07-16 22:13:31

Для автоматичного запису інформації про видалені записи створюється тригер `after_boiler_delete`:

```
DELIMITER $$
CREATE TRIGGER after_boiler_delete
AFTER DELETE ON boiler_data
FOR EACH ROW
BEGIN
  INSERT INTO boiler_audit (
    action, old_temperature, new_temperature,
    old_pressure, new_pressure,
    old_steam_flow, new_steam_flow
  ) VALUES (
    'DELETE',
    OLD.temperature, 0,
```

```
    OLD.pressure, 0,
    OLD.steam_flow, 0
  );
END $$
DELIMITER ;
```

Видалення запису і перевірка роботи тригера здійснюється командами:

```
DELETE FROM boiler_data WHERE record_id = 1;
SELECT * FROM boiler_audit;
```

audit_id	action	old_temperature	new_temperature	old_pressure	new_pressure	old_steam_flow	new_steam_flow	changed_at
1	UPDATE	184.5	188	1.2	1.3	3.5	3.6	2025-07-16 22:11:48
2	INSERT	0	190	0	1.4	0	3.7	2025-07-16 22:13:31
3	DELETE	188	0	1.3	0	3.6	0	2025-07-16 22:14:41

Інформацію про наявні тригери у базі можна отримати командою:

```
SHOW TRIGGERS FROM boiler_monitoring;
```

Trigger	Event	Table	Statement	Timing	Created	sql_mode
after_boiler_insert	INSERT	boiler_data	BEGIN INSERT INTO boiler_audit (actio...	AFTER	2025-07-16 22:13:13.44	ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLE...
before_boiler_update	UPDATE	boiler_data	BEGIN IF OLD.temperature != NEW.tempera...	BEFORE	2025-07-16 22:10:57.00	ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLE...
after_boiler_delete	DELETE	boiler_data	BEGIN INSERT INTO boiler_audit (actio...	AFTER	2025-07-16 22:14:24.77	ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLE...

Видалення тригера виконується за допомогою команди:

```
DROP TRIGGER IF EXISTS before_boiler_update;
```

9.3. Налаштування реплікації *Master-Slave* у *MySQL*

У цьому розділі розглядається порядок налаштування реплікації *Master-Slave* у *MySQL* для підвищення масштабованості та надійності роботи системи. Реплікація дозволяє автоматично копіювати зміни з головного сервера (*Master*) на допоміжний сервер (*Slave*), що забезпечує безперервність роботи, відмовостійкість та можливість горизонтального масштабування бази даних. Для реалізації сценарію використовуються два сервери *MySQL*, які можуть бути розгорнуті на локальних машинах або віртуальних машинах, а також у контейнерах *Docker*.

Підготовка *Master*-сервера починається з редагування конфігураційного файлу `my.ini`, у секції `[mysqld]` задаються параметри:

[*mysqld*]

```
server-id = 1
log-bin = mysql-bin
Binlog-do-db = testdb
```

`server-id = 1` - унікальний ідентифікатор сервера *Master*, який відрізняється від інших серверів мережі. `log-bin` активує ведення бінарного журналу, а `binlog-do-db` визначає базу даних для реплікації.

Після внесення змін служба *MySQL* перезапускається через «Служби *Windows*» або командним рядком:

```
net stop MySQL 80
net start MySQL 80
```

Поточний стан журналу перевіряється командою:

```
SHOW MASTER STATUS;
```

```
+------------------+----------+--------------+------------------+
| File             | Position | Binlog_Do_DB | Binlog_Ignore_DB |
+------------------+----------+--------------+------------------+
| mysql-bin.000001 |      120 | testdb       |                  |
+------------------+----------+--------------+------------------+
```

Результат включає назву поточного бінарного лог-файлу (`File`), позицію початку реплікації (`Position`) та бази, що включені чи виключені з реплікації (`Binlog_Do_DB`, `Binlog_Ignore_DB`). Значення `File` і `Position` використовуються для налаштування *Slave*.

Створення користувача для реплікації виконується на *Master*-сервері. Для цього створюється окремий обліковий запис із правами доступу до бінарного журналу:

```
CREATE USER 'replica_user'@'%' IDENTIFIED BY
'password';
GRANT REPLICATION SLAVE ON *.* TO
'replica_user'@'%';
FLUSH PRIVILEGES;
```

`replica_user` дозволяє *Slave* підключатися до *Master*, `%` вказує на дозвіл підключень з будь-якої *IP*-адреси. Права `REPLICATION SLAVE` забезпечують доступ до бінарного журналу, а `FLUSH PRIVILEGES` застосовує зміни без перезапуску сервера.

Налаштування сервера *Slave* виконується через редагування файлу *my.ini*. У секції `[mysqld]` необхідно вказати параметри для активації реплікації:

```
server-id = 2
relay-log = mysql-relay-bin
log-bin = mysql-bin
read-only = 1
```

`server-id = 2` - унікальний ідентифікатор Slave.

`relay-log` визначає файл для збереження транзакцій із Master.

`log-bin` активує журнал змін для можливого майбутнього використання Slave як Master.

`read-only = 1` обмежує запис даних на сервері Slave. Після внесення змін служба *MySQL* перезапускається.

Налаштування з'єднання *Slave* з *Master* виконується на сервері *Slave* шляхом вказання параметрів підключення до *Master*-сервера:

```
CHANGE MASTER TO
MASTER_HOST='127.0.0.1',
MASTER_PORT=3306,
MASTER_USER='replica_user',
MASTER_PASSWORD='password',
MASTER_LOG_FILE='MySQL -bin.000001',
MASTER_LOG_POS=120;
```

`MASTER_HOST` - *IP*-адреса або хостнейм *Master*-сервера (у прикладі це локальна *IP*-адреса).

`MASTER_PORT` - порт, на якому працює *MySQL* (за замовчуванням 3306).

`MASTER_USER / MASTER_PASSWORD` - облікові дані користувача реплікації, створеного раніше.

`MASTER_LOG_FILE` - назва бінарного журналу, отриманого за допомогою `SHOW MASTER STATUS` на Master-сервері.

`MASTER_LOG_POS` - позиція журналу, з якої потрібно почати реплікацію.

Запуск та перевірка реплікації проводяться шляхом тестування обміну даними між *Master* та *Slave*. На *Master*-сервері у тестовій базі *testdb* створюється таблиця та додається запис:

```
 USE testdb;

CREATE TABLE test_replica (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(50)
);
INSERT INTO test_replica (name) VALUES
('Test1');
```

На сервері *Slave* перевіряється наявність запису:

```
USE testdb;
SELECT * FROM test_replica;
```

Якщо запис `'Test1'` з'явився у таблиці на сервері *Slave*, це свідчить про успішну та коректно налаштовану реплікацію.

9.4. Налаштування *Group Replication* у *MySQL*

Розділ описує принципи роботи *MySQL Group Replication*, налаштування двох вузлів *MySQL* 8.* у режимі групової реплікації, перевірку обміну даними між вузлами та процедури відновлення підключення вузла до групи. Групова реплікація забезпечує синхронне оновлення даних на всіх вузлах, дозволяє кожному вузлу виконувати роль *Primary*, що підвищує відмовостійкість та доступність системи.

Налаштування *Group Replication* проводиться на прикладі об'єднання двох серверів *MySQL* у єдину групу (*cluster*) з автоматичним обміном даними. Усі вузли підтримують роль *Primary*, що забезпечує синхронізацію даних у реальному часі та високий рівень доступності. Для конфігурації використовуються два сервери з різними *IP*-адресами: 192.168.0.192 та 192.168.0.109.

На сервері 192.168.0.192 конфігурація файлу `my.ini` у секції `[mysqld]` містить:

```
[mysqld]
server_id=1
log-bin=mysq -bin
binlog-format=ROW
gtid-mode=ON
enforce-gtid-consistency=ON
log-slave-updates=ON
transaction-write-set-extraction=XXHASH64
 # Налаштування групової реплікації
loose-group_replication_group_name="aaaaaaaa-bbbb-cccc-
dddd-eeeeeeeeeeee"
loose-
group_replication_local_address="192.168.0.192:33061"
loose-
group_replication_group_seeds="192.168.0.192:33061,192.16
8.0.109:33061"
group_replication_ip_whitelist="192.168.0.192,192.168.0.1
09"
loose-group_replication_start_on_boot=ON
```

```
loose-group_replication_bootstrap_group=OFF
```

На сервері 192.168.0.109 конфігурація `my.ini` у секції `[mysqld]`:

```
server-id=2
log-bin= mysq-bin
binlog-format=ROW
gtid-mode=ON
enforce-gtid-consistency=ON
log-slave-updates=ON
transaction-write-set-extraction=XXHASH64
# Налаштування групової реплікації
loose-group_replication_group_name="aaaaaaaa-bbbb-
cccc-dddd-eeeeeeeeeeee"
loose-
group_replication_local_address="192.168.0.109:33061"
loose-
group_replication_group_seeds="192.168.0.192:33061,19
2.168.0.109:33061"
loose-group_replication_start_on_boot=ON
loose-group_replication_bootstrap_group=OFF
```

Після внесення змін служба *MySQL* перезапускається:

```
net stop MySQL 80
net start MySQL 80
```

Порти 3306 і 33061 відкриті у *Windows Firewall*:

```
netsh advfirewall firewall add rule name="MySQL "
dir=in action=allow protocol=TCP localport=3306

netsh advfirewall firewall add rule name="MySQL GR"
dir=in action=allow protocol=TCP localport=33061
```

Перевірка відкриття порту 33061:

```
netstat -an | findstr 33061
```

На обох серверах створюється користувач для реплікації `repl` із необхідними привілеями:

```
CREATE USER 'repl'@'%' IDENTIFIED BY 'password';
GRANT REPLICATION SLAVE, REPLICATION CLIENT ON *.*
TO 'repl'@'%';
```

```
FLUSH PRIVILEGES;
```

Користувачеві `repl` надаються два спеціальні привілеї, необхідні для роботи реплікації в *MySQL* .

Привілей `REPLICATION SLAVE` дозволяє користувачеві отримувати бінарні логи з головного сервера (*Primary*) та застосовувати їх на своєму сервері. Без цього привілею сервер не зможе підключитися до майстра та отримувати оновлення.

Привілей `REPLICATION CLIENT` дозволяє користувачеві переглядати статус реплікації, але без можливості керування нею.

Привілеї застосовуються до всіх баз даних і таблиць (але вони не дають доступу до самих даних, лише до інформації про реплікацію).

Після цього необхідно ініціаліалюзувати *MySQL Group Replication*. Повине бути завантажений плагін `Group Replication, що` перевіряється командою `SHOW PLUGINS`.

Якщо `group_replication` немає у списку, його потрібно завантажити вручну

```
INSTALL PLUGIN group_replication
 SONAME 'group_replication.dll';
```

Перевірити стан плагіну можна командою

```
SELECT PLUGIN_NAME, PLUGIN_STATUS
FROM INFORMATION_SCHEMA.PLUGINS
WHERE PLUGIN_NAME = 'group_replication';
```

Для запуску *Group Replication* на першому сервері (192.192.168.192) спочатку активується режим початкового налаштування. Цю дію потрібно виконати лише один раз і тільки на одному з вузлів при створенні групи:

```
SET GLOBAL group_replication_bootstrap_group=ON;
```

Далі налаштовується внутрішній канал відновлення, що використовується для синхронізації даних між серверами. У параметрах вказуються користувач `repl` та його пароль. Команда виконується на кожному сервері перед запуском реплікації і потребує надання користувачу прав `REPLICATION SLAVE` та `REPLICATION CLIENT`:

```
CHANGE MASTER TO MASTER_USER='repl',
MASTER_PASSWORD='Password'
FOR CHANNEL 'group_replication_recovery';
```

Запускається процес групової реплікації. Сервер, на якому команда виконується першим, стає ініціатором та першим учасником групи:

```
START GROUP_REPLICATION;
```

Режим початкового налаштування групи вимикається, щоб уникнути випадкового повторного створення групи при перезапуску сервера:

```
SET GLOBAL GROUP_REPLICATION_BOOTSTRAP_GROUP=OFF;
```

Для перевірки стану реплікації використовується команда:

```
SELECT * FROM performance_schema.replication_group_members;
```

У результаті статус повинен бути *ONLINE*

До групи додається другий сервер (192.168.0.109). Для цього на ньому виконується налаштування каналу відновлення та запуск реплікації:

```
CHANGE MASTER TO MASTER_USER='repl',
MASTER_PASSWORD='Password'
FOR CHANNEL 'group_replication_recovery';

START GROUP_REPLICATION;
```

Стан групи перевіряється повторно. У таблиці повинні відображатися обидва сервери у статусі ONLINE:

```
SELECT * FROM
performance_schema.replication_group_members;
```

```
+---------------------------+--------------------------------------+---------------+-------------+--------------+-------------+
| CHANNEL_NAME              | MEMBER_ID                            | MEMBER_HOST   | MEMBER_PORT | MEMBER_STATE | MEMBER_ROLE |
+---------------------------+--------------------------------------+---------------+-------------+--------------+-------------+
| group_replication_applier | 093e0028-dd6f-11ef-b8f6-08002761ab19 | DESKTOP-68FUOEC |        3306 | ONLINE       | SECONDARY   |
| group_replication_applier | 9ab18f1f-4e83-11eb-b097-0a0027000011 | iklhome       |        3306 | ONLINE       | PRIMARY     |
+---------------------------+--------------------------------------+---------------+-------------+--------------+-------------+
```

Тестування Group Replication проводиться на головному сервері шляхом створення нової бази даних і тестової таблиці, після чого додається запис:

```
CREATE DATABASE testdb;
USE testdb;
CREATE TABLE test_table (id INT PRIMARY KEY, name
VARCHAR(50));
INSERT INTO test_table
VALUES (22, 'Hello Group Replication');
```

Це дозволяє перевірити, що зміни на головному сервері автоматично поширюються на інші вузли групи, а синхронізація працює коректно.

```
USE testdb;
SELECT * FROM test_table;
```

Зупинка групової реплікації виконується командою:

```
STOP GROUP_REPLICATION;
```

У випадку необхідності повного видалення інформації про попередню групову реплікацію на другому сервері (*Slave*) та повторного підключення його до групи послідовність дій передбачає кілька кроків. Спочатку зупиняється активна реплікація:

```
STOP GROUP_REPLICATION.
```

Далі скидаються всі попередні налаштування, що стосуються зв'язку з групою. Це робиться за допомогою команд:

```
RESET SLAVE ALL;
RESET MASTER;
```

На наступному етапі у файловій системі сервера видаляється файл `auto.cnf`, який зберігає унікальний ідентифікатор вузла (*UUID*). При наступному запуску служби *MySQL* буде створено новий *UUID*, що усуває можливі конфлікти в кластері. Після цього виконується перезапуск служби *MySQL*.

Після відновлення роботи сервера налаштовується канал відновлення групової реплікації, де задаються параметри користувача та пароль:

```
CHANGE MASTER TO MASTER_USER='repl',
MASTER_PASSWORD='Password'
FOR CHANNEL 'group_replication_recovery';
```

Завершальним кроком є повторний запуск реплікації, у результаті чого сервер приєднується до групи як новий вузол:

```
START GROUP_REPLICATION
```

9.5. Оптимізація складного *SQL*-запиту

У цьому підрозділі розгляуто підходи до оптимізації *SQL*-запитів, спрямовані на підвищення продуктивності систем обробки даних у реальному часі. Основна увага приділяється використанню індексів, аналізу плану виконання запиту, вибору типів з'єднань таблиць та оптимізації агрегатних функцій.

9.5.1. Створення бази даних та прикладу складного запиту

Правильна оптимізація запитів дозволяє значно знизити час їх виконання, зменшити навантаження на сервер бази даних і забезпечити своєчасну обробку

інформації в системах, критичних до затримок, таких як *SCADA*, моніторинг сенсорів або керування обладнанням. Для прикладу використовується база даних, що моделює структуру та зв'язки із *SCADA*-системою, і містить таблиці для збереження інформації про датчики, їхні вимірювання та аварійні події

```
CREATE DATABASE IF NOT EXISTS SCADA_db;
USE SCADA_db;

-- Таблиця з інформацією про датчики
CREATE TABLE sensors (
  sensor_id INT AUTO_INCREMENT PRIMARY KEY,
  sensor_name VARCHAR(255) NOT NULL,
  sensor_type VARCHAR(100),
  location VARCHAR(255)
);

-- Таблиця з вимірюваннями від датчиків
CREATE TABLE IF NOT EXISTS measurements (
  measurement_id INT AUTO_INCREMENT PRIMARY KEY,
  sensor_id INT,
  measurement_time DATETIME,
  value DECIMAL(10, 3),
  FOREIGN KEY (sensor_id) REFERENCES
sensors(sensor_id)
);

-- Таблиця для аварійних подій
CREATE TABLE IF NOT EXISTS alarms (
  alarm_id INT AUTO_INCREMENT PRIMARY KEY,
  sensor_id INT,
  alarm_time DATETIME,
  alarm_type VARCHAR(100),
  description TEXT,
  FOREIGN KEY (sensor_id) REFERENCES
sensors(sensor_id)
);
```

Ця структура моделює реальні процеси збору та зберігання даних у *SCADA*-системах, де датчики відправляють періодичні вимірювання, а у разі перевищення порогів генеруються аварійні сповіщення.

Формування запиту проводиться на прикладі складного *SQL*-запиту з об’єднанням таблиць, агрегатними функціями та фільтрацією, що можуть бути ресурсомісткими, наприклад, підрахунок кількості вимірювань і середнього значення параметра для кожного датчика за останній місяць:

```
    SELECT
    s.sensor_name,
    COUNT(m.measurement_id) AS measurement_count,
    AVG(m.value) AS average_value
FROM
    measurements m
JOIN
    sensors s ON m.sensor_id = s.sensor_id
WHERE
    m.measurement_time > NOW() - INTERVAL 1 MONTH
GROUP BY
    s.sensor_name
ORDER BY
    average_value DESC;
```

Запит об’єднує таблиці, застосовує `COUNT`, `AVG` та фільтрацію за датою. За великого обсягу даних він може створювати значне навантаження на сервер.

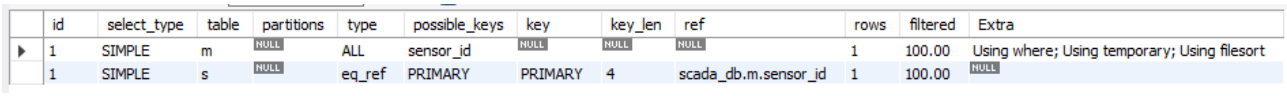
9.5.2. Аналіз плану виконання запиту

Аналіз виконання запиту здійснюється за допомогою команди *EXPLAIN*, яка показує, як *MySQL* планує виконати *SQL*-запит і дозволяє виявити можливі проблеми та неоптимальні операції.

```
EXPLAIN
SELECT
  s.sensor_name,
  COUNT(m.measurement_id) AS measurement_count,
  AVG(m.value) AS average_value
FROM
  measurements m
JOIN
  sensors s ON m.sensor_id = s.sensor_id
WHERE
  m.measurement_time > NOW() - INTERVAL 1 MONTH
GROUP BY
```

```
  s.sensor_name
ORDER BY
  average_value DESC;
```

В результаті отримуємо наступні дані



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	m	NULL	ALL	sensor_id	NULL	NULL	NULL	1	100.00	Using where; Using temporary; Using filesort
1	SIMPLE	s	NULL	eq_ref	PRIMARY	PRIMARY	4	scada_db.m.sensor_id	1	100.00	NULL

`id: 1 #` Ідентифікатор запиту. Основний запит без підзапитів

`select_type: SIMPLE #` Простий *SELECT*-запит без *UNION* чи підзапитів

`table: m #`Таблиця `measurements`, яка першою обробляється

`type: ALL #` Повне сканування таблиці (неоптимально)

`possible_keys: sensor_id #` Потенційний індекс, який міг би бути використаний.

`key: NULL #` Індекс не використовується

`key_len: NULL #` Довжина ключа не вказана, оскільки індекс не використано

`ref: NULL #` Немає прив’язки до значення (повне сканування)

`rows: 1 #` Прогнозована кількість оброблюваних рядків

`filtered: 100.00 #` 100% рядків таблиці проходять фільтрацію

`Extra: Using where; #` Застосовується фільтр `WHERE`

`Using temporary; #` створюється тимчасова таблиця

`Using filesort #` використовується зовнішнє сортування

`id: 1 #` Запис є частиною того ж самого запиту

`select_type: SIMPLE`

`table: s #` Таблиця `sensors`, яка приєднується

`type: eq_ref #` Ефективний тип з’єднання по унікальному ключу

`possible_keys: PRIMARY #` У таблиці доступний первинний ключ

key: PRIMARY # Застосовується первинний ключ (найшвидше з’єднання)

key_len: 4 #Довжина ключа (INT) 4 байти

ref: *SCADA*_db.m.sensor_id # Поле з’єднання зовнішній ключ із таблиці measurements

rows: 1

filtered: 100.00

Extra: NULL

9.5.3. Оптимізація запиту

Оптимізація запиту починається з аналізу плану виконання, що дозволяє визначити основні фактори, які впливають на продуктивність. Повне сканування таблиці measurements свідчить про те, що *MySQL* не використовує індекси для фільтрації за датою (*measurement_time*), що призводить до високого навантаження на сервер при великому обсязі даних. Створення тимчасової таблиці та зовнішнє сортування під час групування значно уповільнюють виконання запиту.

Для оптимізації запиту варто додати індекс на колонку measurement_time для швидкої фільтрації даних за датою:

```
CREATE INDEX idx_measurement_time ON
measurements(measurement_time);
```

Переконатися, що зовнішній ключ sensor_id у таблиці measurements має індекс. Якщо запит часто звертається до sensor_id, measurement_time та value, можна створити комбінований індекс, що охоплює всі ці колонки:

```
CREATE INDEX idx_measurements_covering ON
measurements(sensor_id, measurement_time, value);
```

Це дозволяє *MySQL* отримувати необхідні дані без звернення до основної таблиці, зменшуючи *I/O* операції. Якщо запит часто виконується з `GROUP BY sensor_name ORDER BY average_value DESC`, можна розглянути створення додаткового індексу на колонку `sensor_id` у таблиці `sensors` разом з полем `sensor_name` для прискорення з'єднання та групування. Можна також переписати запит так, щоб фільтрувати дані на ранньому етапі за допомогою підзапиту або *CTE*, щоб скоротити кількість рядків, що обробляються агрегатними функціями.

Ось приклад повністю оптимізованого запиту для вашого випадку з урахуванням індексів та ранньої фільтрації даних. Запит використовує підзапит для відбору лише необхідних вимірювань за останній місяць, що зменшує кількість оброблюваних рядків у JO*I*N та агрегатних функціях:

```
-- Створення індексів для оптимізації
CREATE INDEX idx_measurement_time ON
measurements(measurement_time);
CREATE INDEX idx_measurements_covering ON
measurements(sensor_id, measurement_time, value);

-- Оптимізований запит
SELECT
  s.sensor_name,
  COUNT(m.measurement_id) AS measurement_count,
  AVG(m.value) AS average_value
FROM
  sensors s
JOIN (
  SELECT measurement_id, sensor_id, value
  FROM measurements
  WHERE measurement_time > NOW() - INTERVAL 1 MONTH
) m ON s.sensor_id = m.sensor_id
GROUP BY
  s.sensor_name
ORDER BY
  average_value DESC;
```

У цьому запиті підзапит `m` відбирає лише дані за останній місяць, що дозволяє *JOIN* і агрегатним функціям працювати з меншою кількістю рядків. Комбінований індекс `idx_measurements_covering` дозволяє *MySQL* обійтись без звернення до основної таблиці, що значно прискорює обчислення `COUNT` та `AVG`.

9.6. Інтеграція *MySQL* і *MATLAB*

У цьому підрозділі розглядається використання системи керування базами даних *MySQL* спільно з програмним середовищем *MATLAB* для аналізу та візуалізації даних. Така інтеграція є типовою для інженерних і науково-дослідних задач, у яких результати вимірювань або обчислень накопичуються в базі даних, а подальша обробка виконується засобами математичного моделювання.

MATLAB використовується як клієнт до *MySQL*, який виконує *SQL*-запити, отримує результати у табличному вигляді та застосовує до них інструменти чисельного аналізу і графічного подання. Обмін даними реалізується через стандартний *JDBC*-драйвер *MySQL* і функціональні засоби *Database Toolbox*.

Підключення *MATLAB* до *MySQL* забезпечує прямий доступ до даних без проміжного експорту у файли та дозволяє працювати з єдиним джерелом інформації. Це зручно під час аналізу технологічних параметрів, експериментальних результатів або даних, що надходять із зовнішніх прикладних систем.

Для встановлення з’єднання використовується JDBC-драйвер *MySQL* Connector/J. *MATLAB* звертається до сервера *MySQL* як до зовнішнього джерела даних, виконуючи *SQL*-інструкції безпосередньо з програмного коду. Результати запитів повертаються у вигляді клітинкових масивів, які за потреби перетворюються у табличний формат *MATLAB*.

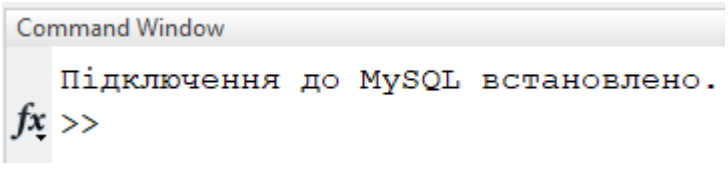
9.6.1. Підключення до MySQL з MATLAB

Перед початком роботи має бути встановлено *MySQL* Server і створено обліковий запис з правами доступу до бази даних. Також необхідно мати *JDBC*-драйвер *MySQL*, шлях до якого додається до середовища виконання *MATLAB*.

Нижче наведено приклад встановлення з'єднання *MATLAB R2014a*.

```
clear; clc;

javaaddpath('C:\MySQL-connector\mysql-connector-java-
5.1.4-bin.jar');
dbname   = 'matlab_db';
username = 'root';
password = 'root';

driver = 'com.mysql.jdbc.Driver';
url    =
'jdbc:mysql://localhost:3306/matlab_db?useUnicode=true&ch
aracterEncoding=utf8';

conn = database('', username, password, driver, url);

if isempty(conn.Message)
    disp('Підключення до MySQL встановлено.');
else
    error(conn.Message);
end
```

Приклад результату виконання в *MATLAB*



9.6.2. Робота з таблицями та даними

Після встановлення з'єднання можна створювати таблиці, додавати записи та виконувати запити на вибірку. Далі наведено приклад створення таблиці та перевірки її структури.

Для збереження тестових даних створюється таблиця *test_table*, яка містить ідентифікатор запису, текстове поле, числове значення та часову мітку створення запису. Створення таблиці виконується стандартною *SQL*-інструкцією *CREATE TABLE*, яка передається до *MySQL* через активне з'єднання.

```
SQL-запит для створення таблиці
SQLquery = ['CREATE TABLE IF NOT EXISTS test_table (' ...
            'id INT AUTO_INCREMENT PRIMARY KEY, ' ...
            'name VARCHAR(100) NOT NULL, ' ...
            'value DOUBLE, ' ...
            'created_at TIMESTAMP DEFAULT
CURRENT_TIMESTAMP)'];

exec(conn, SQLquery);

% Перевірка структури таблиці
SQLquery = 'DESCRIBE test_table';
data = fetch(conn, SQLquery);

disp('Структура створеної таблиці:');
disp(data);

disp('Структура таблиці:');
for i = 1:size(data, 1)
    disp(['Колонка: ', data{i,1}, ...
          ', Тип: ', data{i,2}, ...
          ', Null дозволено: ', data{i,3}]);
end
```

Приклад результату виконання в *MATLAB*

```
Структура створеної таблиці:
    'id'            'int'            'NO'     'PRI'    'null'                'auto_increment'
    'name'          'varchar(100)'   'NO'     ''       'null'                ''
    'value'         'double'         'YES'    ''       'null'                ''
    'created_at'    'timestamp'      'YES'    ''       'CURRENT_TIMESTAMP'   'DEFAULT_GENERATED'

Структура таблиці:
Колонка: id, Тип: int, Null дозволено: NO
Колонка: name, Тип: varchar(100), Null дозволено: NO
Колонка: value, Тип: double, Null дозволено: YES
Колонка: created_at, Тип: timestamp, Null дозволено: YES
```

Для перевірки роботи з’єднання виконаємо вставку кількох записів і вибірку даних з таблиці.

```
% Вставка даних
SQLquery = ['INSERT INTO test_table (name, value)
VALUES ' ...
            '(''Sample A'', 123.45), ' ...
            '(''Sample B'', 678.90), ' ...
            '(''Sample C'', 345.67)'];
exec(conn, SQLquery);

disp('Дані успішно вставлено');

% Вибірка всіх записів
SQLquery = 'SELECT * FROM test_table';
data = fetch(conn, SQLquery);

disp('Дані з таблиці test_table:');
disp(data);
```

Приклад результату виконання

Дані успішно вставлено

Дані з таблиці *test_table*:

[1] ‘*Sample* A’ [123.4500] ‘2026-01-16 14:12:08.0’

[2] ‘*Sample* B’ [678.9000] ‘2026-01-16 14:12:08.0’

[3] ‘*Sample* C’ [345.6700] ‘2026-01-16 14:12:08.0’

За потреби отримані дані можуть бути перетворені у табличний формат *MATLAB* для подальшого аналізу.

```
data_table = cell2table(data, ...
    'VariableNames',
{'id','name','value','created_at'});

disp(data_table);
```

id	name	value	created_at
1	'Sample A'	123.45	'2026-01-21 21:01:44.0'
2	'Sample B'	678.9	'2026-01-21 21:01:44.0'
3	'Sample C'	345.67	'2026-01-21 21:01:44.0'

9.7. Налаштування симулятора промислового контролера за протоколом *MODBUS* та інтеграція зі *SCADA Ignition*

У цьому підрозділі показано, як організувати обмін даними за протоколом *Modbus* між симулятором промислового пристрою та *SCADA Ignition*. Роль підлеглого пристрою виконує програмний симулятор, а Ignition здійснює зчитування та запис технологічних значень через стандартні засоби *Modbus*. Такий підхід дає змогу відпрацювати адресацію регістрів, формати даних і логіку взаємодії *SCADA* з контролером без залучення реального ПЛК, зберігаючи типову для автоматизації схему обміну запитами та відповідями.

Як симулятор підлеглого пристрою *Modbus* використовується *ModSim*, який працює як *Slave* (*server*) і приймає запити від головного пристрою. У даному сценарії головним пристроєм є *SCADA Ignition*, яка виступає *Master* (*client*) і ініціює обмін, періодично опитуючи регістри та, за потреби, записуючи керувальні значення. *ModSim* можна завантажити зі сторінки розробника за адресою https://www.win-tech.com/html/demos.htm. Програма підтримує роботу через послідовний інтерфейс і через *Ethernet*, а для навчальних задач зазвичай обирають *Modbus TCP*.

ModSim є умовно безкоштовним програмним забезпеченням. У демонстраційному режимі сеанс роботи обмежено приблизно десятьма хвилинами, після чого програму необхідно перезапустити. На зміст і повноту налаштувань це не впливає, оскільки параметри пристрою та структура регістрів задаються без обмежень, а повторний запуск потрібен лише для продовження обміну.

Для спрощення конфігурації *ModSim* та *Ignition* можна розміщувати на одному комп’ютері. У цьому разі як адреса хоста використовується *localhost*, а з’єднання встановлюється через порт 502, який є стандартним для *Modbus TCP*. Слід враховувати, що в окремих конфігураціях *Windows* порт 502 може бути зайнятий іншою службою або вимагати підвищених прав доступу. Якщо з’єднання не встановлюється саме з цієї причини, доцільно обрати альтернативний порт у *ModSim*, наприклад 1502, і вказати той самий порт у налаштуваннях пристрою *Modbus* в *Ignition*.

Після запуску *ModSim* під час першого звернення до ліцензії можна відмовитися від реєстрації та продовжити роботу в демо-режимі. Далі створюється нова конфігурація підлеглого пристрою через меню *File → New* або комбінацією *Ctrl+N*. У початковому вікні налаштувань задаються основні параметри регістрів, які будуть доступні для читання та запису з боку *SCADA*. Типове вікно налаштування *Modbus* у *ModSim* наведено на рис. 9.1.

Параметри підлеглого пристрою задаються через поля *Device Id, Address, Length* та *Modbus Point Type*. Поле Device Id визначає адресу Modbus-пристрою на рівні протоколу. Поле Address задає початкове зміщення. Поле Length задає кількість елементів, що надаються для обміну. Поле Modbus Point Type визначає тип елементів, з якими працює симулятор, наприклад Holding Register для 16-бітних регістрів з можливістю читання і запису.

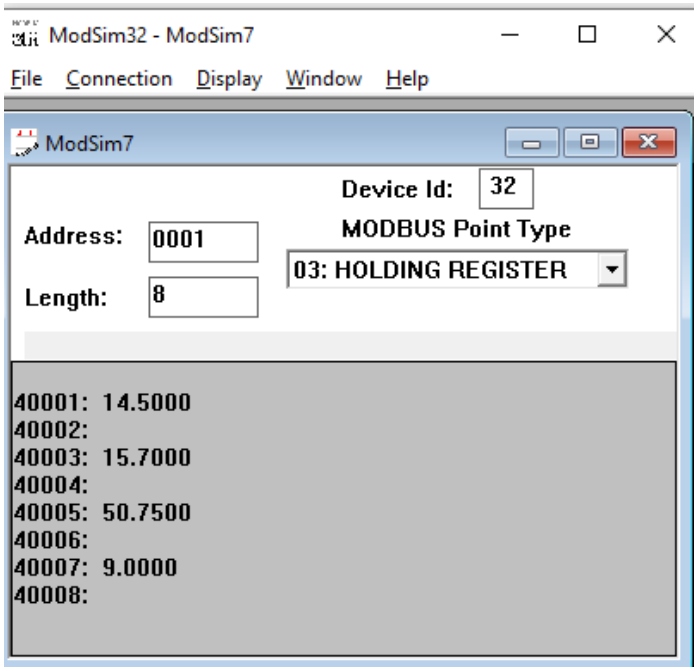


Рис.9.1. Вікно налаштування *ModSim*

Для зручного узгодження з типовими налаштуваннями клієнтів *Modbus* у *SCADA* доцільно використати наступні значення: Device Id 32, Address 1, Length 8, а *Modbus Point Type* залишити як *Holding Register*. Після задання параметрів конфігурацію слід зберегти, оскільки надалі вона використовується при запуску сервера *Modbus TCP*.

У межах прикладу обміну даними симулятор формує значення двох аналогових сигналів, які інтерпретуються як вимірювання датчиків у діапазоні 4…20 мА, а *SCADA* виконує запис керувальних команд для двох виконавчих механізмів, наприклад у відсотках відкриття клапана 0–100%. Такий набір сигналів є типовим для демонстраційних схем, оскільки одночасно ілюструє читання вимірювань і запис керування, а також дозволяє в подальшому пов’язати ці значення з тегами Ignition та елементами *HMI*.

Щоб коректно відображати аналогові параметри, у *ModSim* необхідно узгодити формат подання чисел з очікуваннями *SCADA*. Якщо значення зчитуються як числа з плаваючою комою одинарної точності, необхідно в меню *Display* встановити формат *Float* (*Swapped*). Це пов’язано з тим, що значення *IEEE* 754 для типу float кодується у два суміжні 16-бітні регістри, а порядок

слів у *Modbus*-пристроях може бути переставленим. Саме режим *Swapped* дозволяє без додаткових перетворень отримати коректні значення у клієнті.

Після підготовки структури регістрів налаштовується автоматичне формування змінних значень. Для цього обирається регістр з адресою 40001, відкривається вікно *Write Floating Pt.*, після чого активується *Auto Simulation*. У параметрах симуляції вмикається *Enabled*, тип симуляції змінюється з *Random* на Increment, інтервал зміни *Change Interval* (*secs*) задається 2 с, крок *Step Value* встановлюється 0,25, а межі Simulation Range задаються в діапазоні від 4,0 до 20,0. Подібні налаштування застосовуються для другого каналу з адресою 40003, наприклад з кроком 0,3, щоб зробити поведінку двох датчиків відмінною та зручною для спостереження.

Для імітації керувальних впливів виконавчих механізмів використовуються регістри 40005 та 40007, для яких доцільно встановити діапазон від 0 до 100 та крок зміни 5. Загальний вигляд вікна *Auto Simulation* наведено на рис. 9.2.

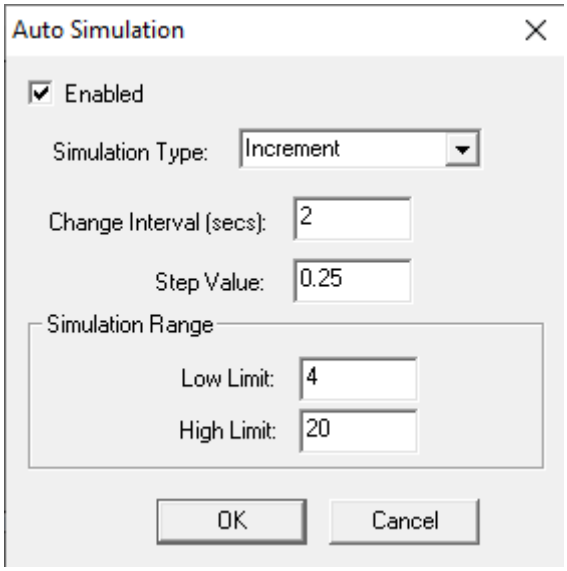


Рис. 9.2. Налаштування *Auto Simulation* у *ModSim* для аналогових величин

Після завершення налаштувань запускають режим роботи *Modbus/TCP* server у *ModSim*. Для цього в меню *Connection* обирають *Connect* і канал *Modbus/TCP Svr*, у вікні *Select Service Port* задають порт 502 і підтверджують

запуск. Стан каналу зв’язку перевіряють через команду *Status* у меню *Connection*. Зупинка симулятора виконується через *DisConnect*. На цьому етапі Mo*d*Sim готовий до приймання запитів від *Ignition*.

9.7.1. Встановлення та налаштування SCADA-системи Ignition для опитування симулятора роботи Slave-пристрою за протоколом MODBUS TCP

SCADA Ignition є сучасною платформою автоматизації, яка підтримує операційні системи *Windows*, *Linux* та *macOS* і широко застосовується для побудови диспетчерських систем, орієнтованих на інтеграцію з базами даних і зовнішніми сервісами.

Для встановлення *SCADA Ignition* у середовищі *Windows* використовується інсталяційний файл, який завантажується з офіційного сайту розробника *https://inductiveautomation.com/downloads*. Як приклад може застосовуватися файл типу *ignition-8.1.45-Windows-64-installer.exe*, при цьому конкретна версія визначається моментом встановлення. Під час інсталяції система може працювати у двох режимах. *Trial Mode* забезпечує доступ до повного функціоналу протягом обмеженого часу з можливістю повторного перезапуску сесії, що є достатнім для навчальних і демонстраційних сценаріїв. *Licensed Mode* передбачає використання постійної.

У процесі інсталяції обирається каталог встановлення, який доцільно залишити за замовчуванням, наприклад *C:\Program Files\Inductive Automation\Ignition*, а також тип інсталяції. Режим *Typical* забезпечує встановлення стандартного набору модулів і є достатнім для подальшої роботи з проєктами *SCADA* та підключенням до бази даних *MySQL*. Після завершення копіювання файлів виконується початкове налаштування *Ignition Gateway*, під час якого обирається редакція *Standard Edition* і створюється обліковий запис адміністратора.

На етапі конфігурації мережевих параметрів задаються порти доступу до основних компонентів системи. У типовій конфігурації використовується *HTTP*-порт 8088 для доступу до вебінтерфейсу *Gateway*, *HTTPS*-порт 8043 для захищеного з'єднання та порт *Gateway Network* 8060 для внутрішньої взаємодії компонентів платформи. Для навчальних сценаріїв ці значення доцільно залишати без змін, оскільки вони забезпечують коректну роботу системи без додаткового мережевого налаштування (рис. 9.3).

Після завершення інсталяції основним інтерфейсом адміністрування системи є вебінтерфейс *Ignition Gateway*. Доступ до нього здійснюється через браузер за адресою *http://localhost:8088*. Через Gateway виконуються керування модулями, налаштування підключень до баз даних, а також завантаження допоміжних інструментів. Одним із таких інструментів є *Ignition Designer*, який використовується для створення проєктів, роботи з тегами, виконання *SQL*-запитів і конфігурації взаємодії з *MySQL*.

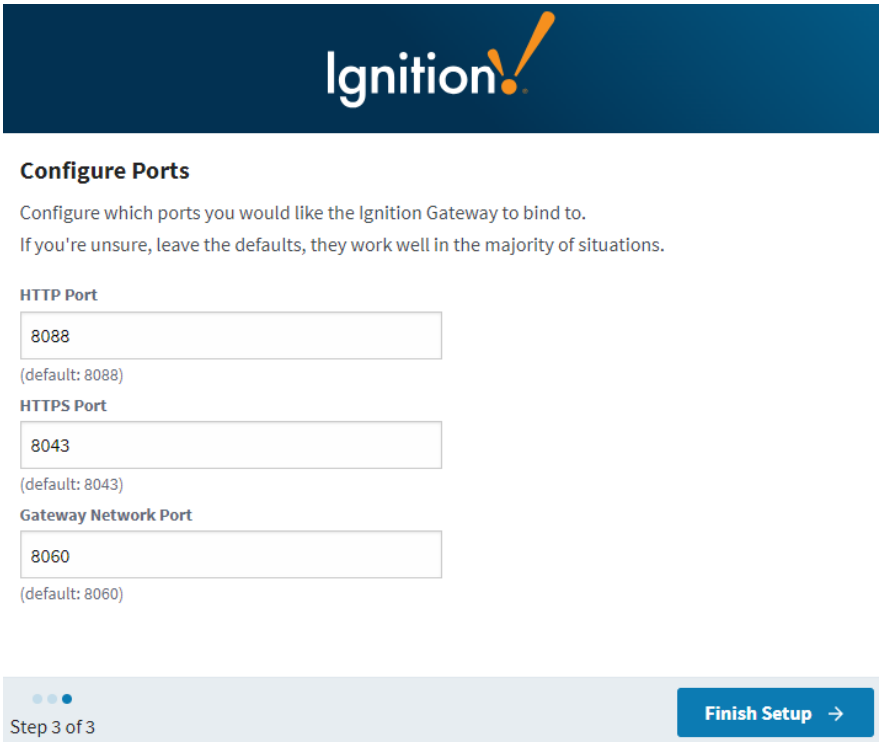


Рис. 9.3. Порти *SCADA* Ignition під час встановлення

Сторінка завантаження *Ignition Designer Launcher* автоматично визначає операційну систему та пропонує відповідний інсталяційний файл. Після встановлення *Launcher* використовується для запуску *Ignition Designer* та підключення до *Gateway* (рис. 9.4).

Під час першого запуску Ignition Designer необхідно вказати адресу Ignition Gateway, наприклад http://localhost:8088, а також облікові дані адміністратора, створені на етапі інсталяції *Gateway*. Після успішного підключення створюється новий проєкт, наприклад з назвою *Ignition_MySQL*, який надалі використовується для розгляду прикладів роботи *SCADA Ignition* з базою даних *MySQL* у наступних підрозділах.

Для підключення до *ModSim* використовується вебінтерфейс *Ignition Gateway*, доступний за адресою http://localhost:8088. Після входу під обліковим записом адміністратора переходять до конфігурації *OPC UA* і створення підключення до пристрою через вбудований драйвер *Modbus*, оскільки цей шлях не потребує залучення зовнішнього *OPC server* і є найбільш прямим для навчальних сценаріїв.

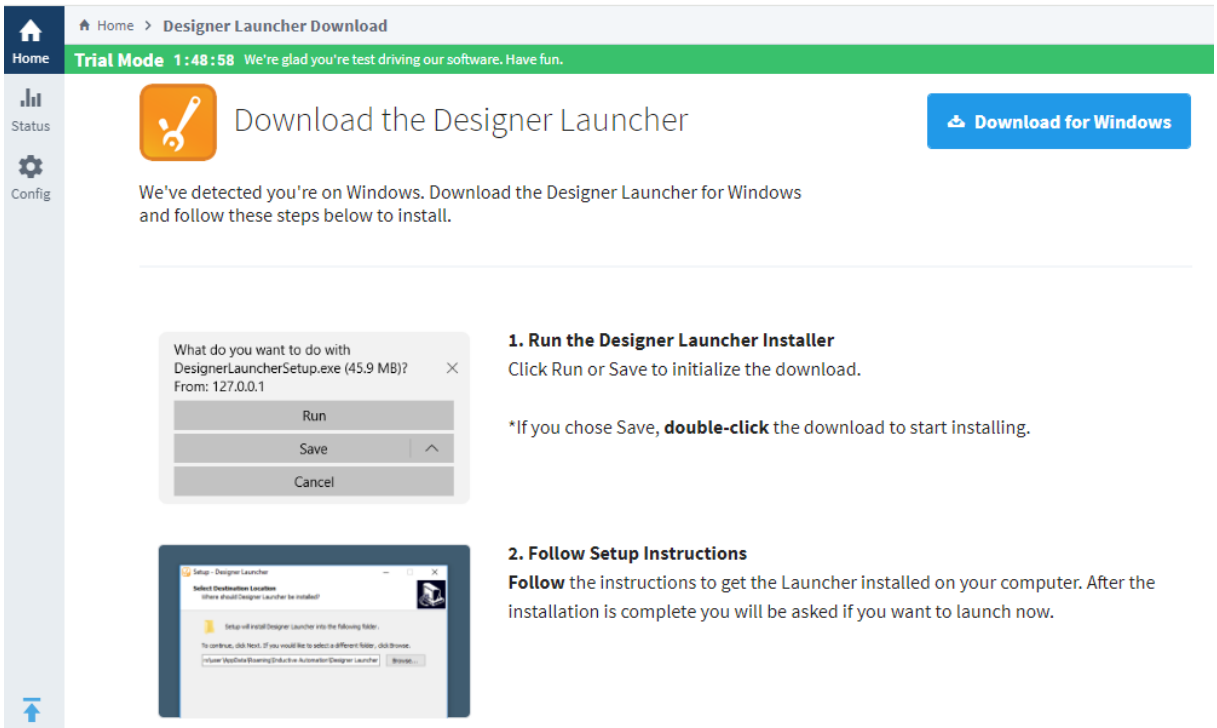


Рис. 9.4. Сторінка завантаження *Ignition Designer*

Після входу під обліковим записом адміністратора у розділі *Config* відкривають пункт *Devices* і створюють новий пристрій командою *Create new Device*. У переліку драйверів обирають *Modbus TCP*. У формі налаштувань задають ім’я підключення *Modsim_TCP*_1, а також параметри з’єднання. Якщо симулятор *ModSim* і *SCADA* система працюють на одному комп’ютері, як адресу вузла вказують localhost, а порт залишають стандартним для *Modbus TCP*, тобто 502. Додаткові параметри доступні через *Show advanced properties*, однак для базової перевірки обміну даними їх доцільно залишити без змін. Приклад типового вікна конфігурації пристрою наведено на рис. 9.5.

Після натискання *Create New Device* створене з’єднання має перейти у стан *Connected*, що відображається в переліку *Devices*, наприклад у вигляді рядка *Modsim_TCP_1, Type Modbus TCP, Enabled true, Status Connected*.

Після створення з’єднання необхідно налаштувати карту адрес *Modbus*, яку використовуватиме драйвер для формування змінних у вбудованому адресному просторі *OPC UA*. Для цього у рядку *Modsim_TCP_1* натискають кнопку *More* і обирають команду *Addresses*. Відкривається вікно *Address Configuration*, у якому задаються групи змінних шляхом заповнення рядків таблиці. У даному прикладі створюються дві групи змінних, що відповідають аналоговим входам і виходам. Параметри першої групи вводять так: *Prefix an_in, Start 1, End 4, Unit ID 32, Modbus Type Holding Register (Float), Modbus Address 1*. Параметри другої групи вводять так: *Prefix an_out, Start 6, End 8, Unit ID 32, Modbus Type Holding Register (Float), Modbus Address 5*. Значення *Radix* встановлюють 12, що визначає формат імен змінних, які генерує драйвер. Приклад заповнення параметрів у вікні *Address Configuration* наведено на рис. 9.6.

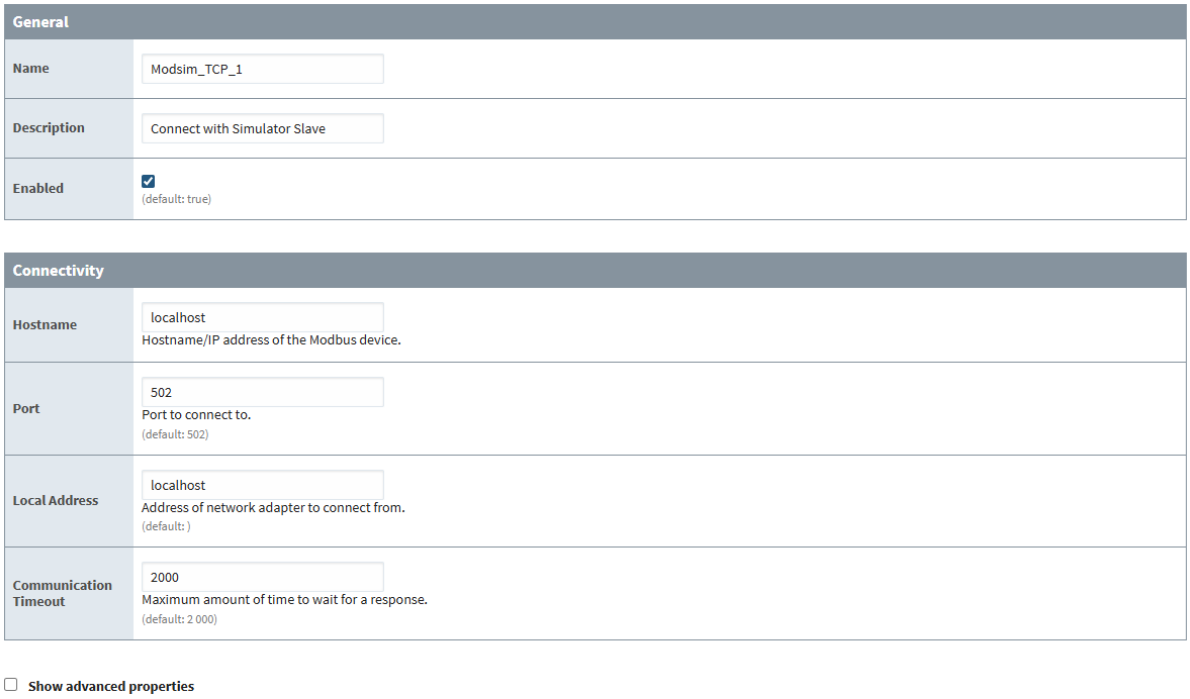


Рис. 9.5. Вікно конфігурації підключення *Modbus TCP* у *Ignition Gateway*

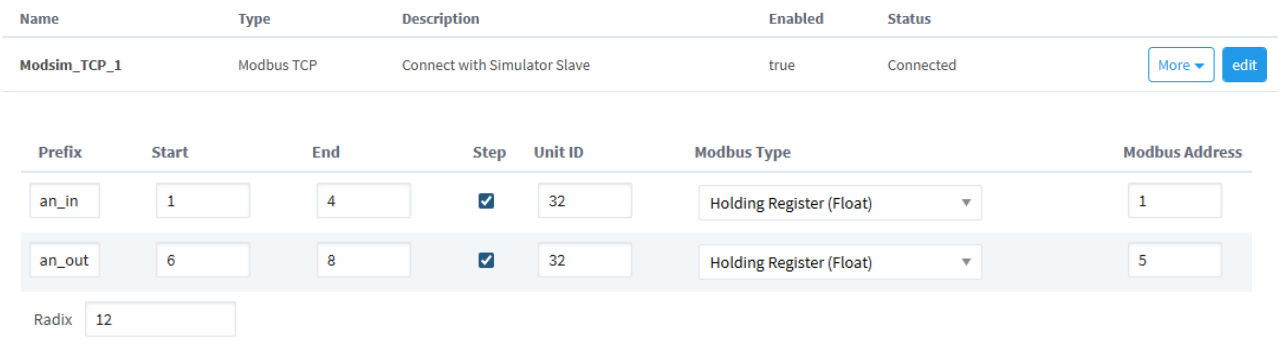
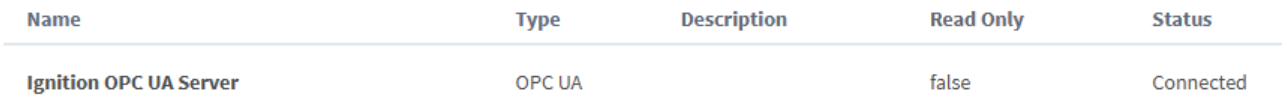


Рис. 9.6. Налаштування карти адрес *Modbus* у вікні *Address Configuration* для пристрою *Modsim_TCP*_1

Окремо слід налаштувати *OPC UA* підключення, яке використовуватиме *Ignition Designer* для перегляду адресного простору і додавання тегів через функцію *Browse Devices*. У *Ignition* 8 для цього створюється *OPC Connection* до внутрішнього *OPC UA сервера*. У вебінтерфейсі *Gateway* відкривають розділ *Config*, далі переходять у *Connections*, після чого обирають *OPC Connections* і натискають *Create New*. Створюють підключення з назвою *Ignition OPC UA Server*, типом *OPC UA*, параметром *Read Only false* і перевіряють, що *Status* має значення *Connected*. У полі *Endpoint URL* вказують адресу внутрішнього

сервера у форматі *opc.tcp://localhost:62541/discovery*. Приклад створеного *OPC Connection* до внутрішнього *OPC UA* сервера наведено на рис. 9.7. Після збереження підключення рекомендується перезапустити *Ignition Designer*, щоб нове OPC підключення стало доступним під час конфігурування *OPC* тегів.



Name	Type	Description	Read Only	Status
Ignition OPC UA Server	OPC UA		false	Connected

Рис.9.7. Створення *OPC Connection* до внутрішнього *OPC UA* сервера *Ignition у розділі Config → Connections → OPC Connections*

Після виконання вказаних налаштувань переходять до перевірки стану з'єднання. Для цього у вебінтерфейсі *Ignition Gateway* відкривають розділ *Status* і далі *Status → Connections → Devices*. Для пристрою *Modsim_TCP*_1 статус має бути *Connected*. Якщо відображається *Disconnected*, необхідно повернутися до *Config → Opcua → D*evices, відкрити налаштування *Modsim_TCP*_1 і перевірити коректність *host*, *port*, а також відповідність *Unit ID* значенню *Device ID у ModSim*. Додатково слід перевірити, що симулятор *ModSim* запущений і налаштований на роботу з *Holding Registers*, а також що в брандмауері *Windows* дозволені локальні з'єднання на порт 502.

9.7.2. Приклад взаємодії SCADA-системи Ignition з програмою-симулятором за протоколом Modbus

Середовище розробки проєктів *Ignition Designer* надає інструменти для перевірки взаємодії з фізичними пристроями та програмами-симуляторами. У навчальній постановці доцільно розрізняти два рівні перевірки. Перший рівень полягає у контролі стану тегів і якості даних у *Tag Browser*, тобто у підтвердженні, що *SCADA*-система коректно читає та, за потреби, записує значення. Другий рівень полягає у створенні екрана людино машиної взаємодії у модулі *Vision*, де значення тегів відображаються в графічному вигляді, а оператор може вводити керувальні впливи.

Для перевірки працездатності з’єднання необхідно запустити програму-симулятор *ModSim* та *SCADA* систему *Ignition*, після чого відкрити *Ignition Designer* через програму *Designer Launcher*. У вікні *Launcher* відображається перелік доступних *Gateway* серверів (див. рис.9.8).

Обирають потрібний сервер і виконують запуск середовища розробки командою *Open Designer*. Під час входу до системи вводять облікові дані користувача та підтверджують авторизацію командою *Login*.

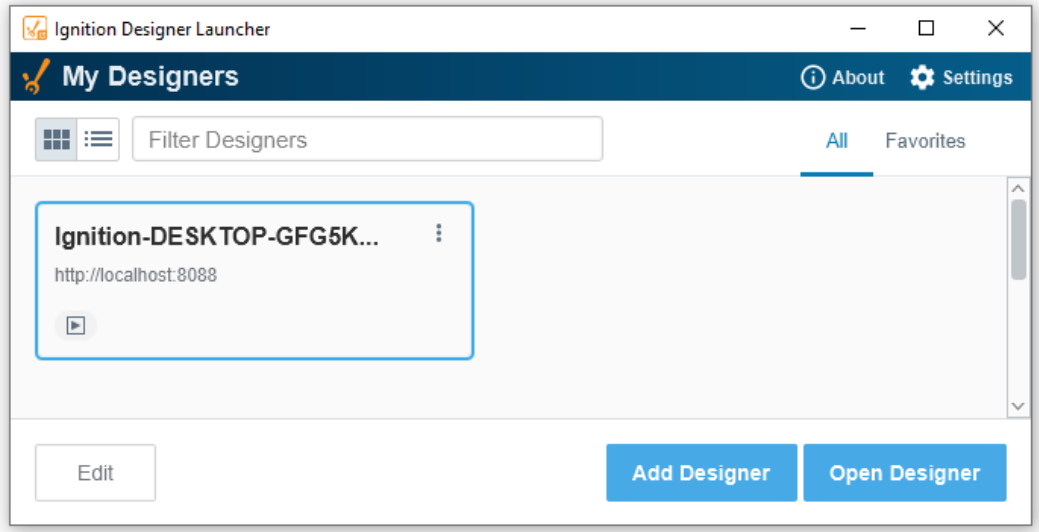


Рис.9.8. Вікно *Ignnition Designer Launcher*

Після успішного входу відкривається менеджер проєктів *Ignition Designer*. Якщо проєкт ще не створено, обирають команду *New Project*, уводять назву, наприклад *Modsim_TCP*, залишають інші параметри без змін і натискають *Create New Project*. Після завантаження проєкту відкривається робочий простір *Ignition Designer* з панеллю *Project Browser*, вікном *Tag Browser* та інструментальними панелями для розроблення.

На першому етапі необхідно додати до проєкту теги, що відповідають змінним підключеного *Modbus* пристрою. Теги в *Ignition* є внутрішніми об’єктами, які зберігають поточні значення та забезпечують доступ до них із графічних екранів, скриптів і модулів історизації. Для роботи з тегами використовується *Tag Browser*. У ньому створюють папку, наприклад *Modsim*,

після чого виконують додавання тегів із підключеного пристрою через команду *Browse Devices*. У результаті відкривається вікно *Connected Devices* з *OPC Browser*, у якому доступний адресний простір внутрішнього *OPC UA* сервера та підключені пристрої. Приклад вікна *Connected Devices* під час *відкриття Browse Device*s наведено на рис. 9.9.

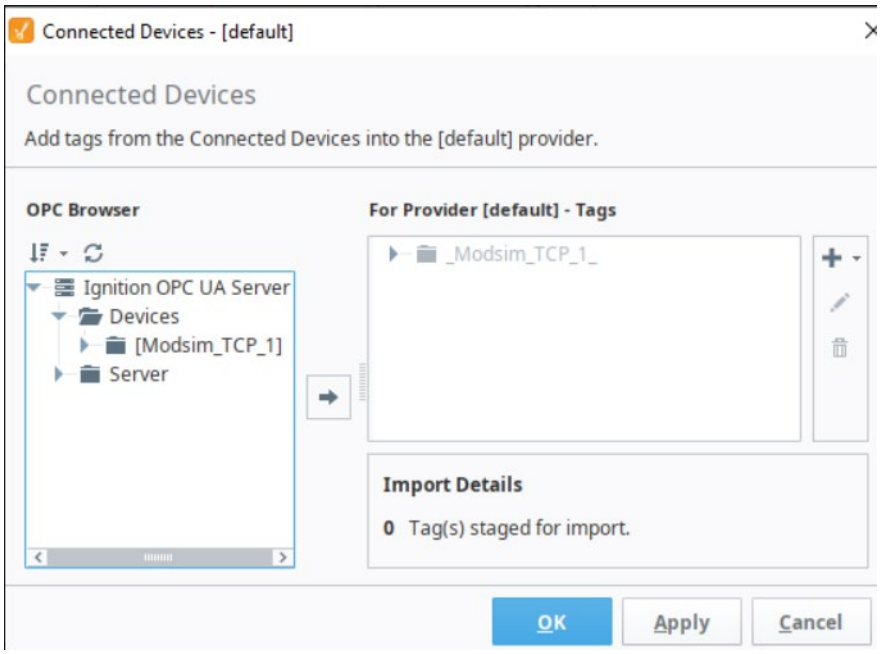


Рис. 9.9. Вікно *Connected Devices* після виконання команди *Browse Devices*

У дереві адресного простору послідовно відкривають *Ignition OPC UA Server*, далі *Devices*, після чого обирають пристрій *Modsim_TCP*_1. Додавання тегів виконують перетягуванням потрібних вузлів до створеної папки в *Tag Browser*. Для прискорення роботи доцільно перетягнути весь вузол *Modsim_TCP*_1, після чого *Ignition* автоматично створить набір *OPC* тегів для змінних, сформованих драйвером на основі карти адрес. Після додавання необхідно перевірити, що для тегів встановлено коректний стан якості, зокрема *Quality* має значення *Good*, а в колонці *Value* відображаються поточні числові значення. Приклад структури створених тегів у *Tag Browser* наведено на рис. 9.10.

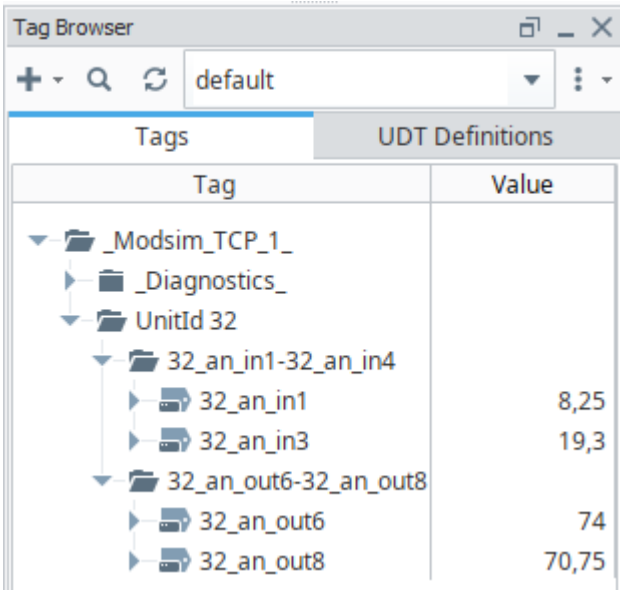


Рис. 9.10. Вікно *Tag Browser* у *Ignition Designer*

Після підтвердження обміну даними переходять до створення екрана людино машиної взаємодії у модулі *Vision*. У *Project Browser* відкривають розділ *Vision*, розгортають вузол *Windows* і створюють головне вікно проєкту командою *New Main Window*. У діалоговому вікні *New Window* залишають ім’я за умовчанням або задають власне, після чого підтверджують створення командою *Create Window*. У результаті у робочому просторі з’являється шаблон вікна, що слугуватиме основою для екрана оператора.

Далі додають графічні компоненти для відображення значень тегів і введення керувальних впливів. Усі компоненти доступні на панелі *Component Palette*, яка розміщена праворуч у середовищі *Ignition Designer* та згрупована за функціональним призначенням. Для відображення аналогових входів доцільно використати числові індикатори або текстові поля лише для читання, пов’язані з тегами групи *an_in*. Для задання значень виходів доцільно використати *Numeric Text Field*, *Slider* або інший елемент введення, пов’язаний із тегами групи *an_out*. Прив’язка компонентів до тегів здійснюється через властивість *data binding*, у якій обирається відповідний тег у дереві *Tag Browser*. Після налаштування прив’язок виконують попередній перегляд екрана в режимі

Preview Mode та перевіряють, що зміна значень у *ModSim* відображається на екрані, а введені оператором значення передаються у змінні виходів.

Після налаштування підключення до програми-симулятора та додавання тегів до проєкту можна переходити до створення людино-машинного інтерфейсу, який забезпечує наочне відображення стану аналогових входів і керування аналоговими виходами. Приклад такого інтерфейсу наведено на рис. 9.11.

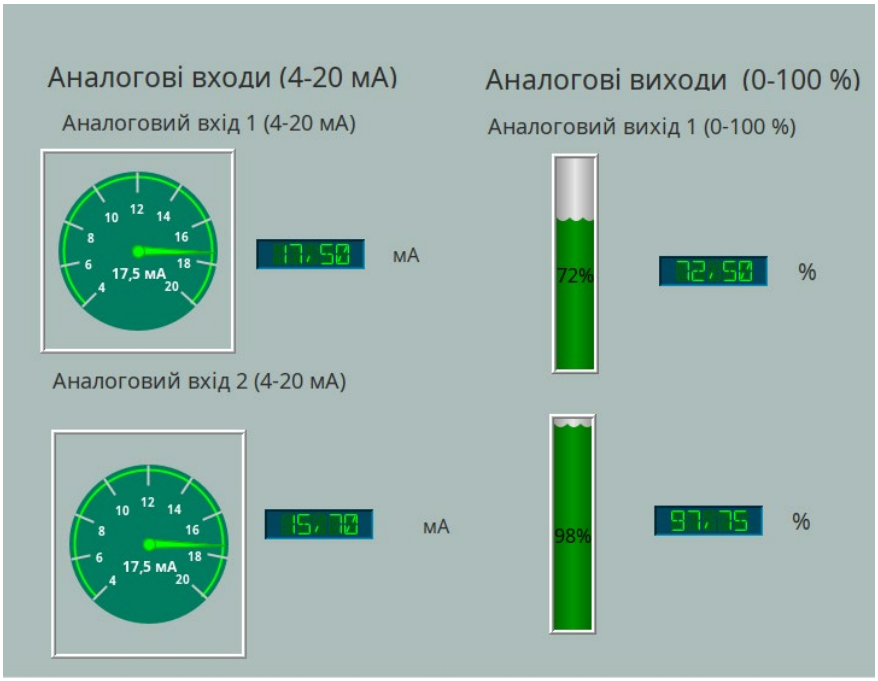


Рис. 9.11. Вікно людино-машинного інтерфейсу

Інтерфейс умовно поділяється на дві частини. Ліва частина призначена для відображення аналогових входів зі стандартним струмовим сигналом 4–20 мА, права використовується для задання та індикації аналогових виходів у відсотках у діапазоні 0–100 %.

Для кожного аналогового входу використовується комбінація графічного та числового компонентів.

Як графічний індикатор доцільно застосувати компонент *Meter*. У його властивостях задаються мінімальне значення 4 та максимальне значення 20, що відповідає діапазону струмового сигналу. Значення властивості *Value*

прив’язується до відповідного вхідного тега, наприклад *32_an_in1*. Прив’язка виконується через механізм binding шляхом вибору Tag binding.

Для зручності сприйняття доцільно одразу додати текстові написи, що позначають функціональні зони інтерфейсу, наприклад «Аналогові входи (4–20 мА)» та «Аналогові виходи (0–100 %)». Для цього використовується компонент *Label* із панелі *Component Palette*.

Поруч із круговим індикатором розміщується компонент *Numeric Label*, який відображає поточне числове значення струму. Його властивість *Value* також прив’язується до того самого тега. Для підвищення наочності у властивості *FormatString* задається формат із фіксованою кількістю знаків після коми, а поряд додається статичний напис «мА», що позначає одиниці вимірювання.

Аналогічні дії виконуються для другого вхідного сигналу. Таким чином оператор одночасно бачить як графічну індикацію рівня сигналу, так і точне числове значення.

Для аналогових виходів використовується вертикальний індикатор рівня та числовий елемент введення.

Як основний графічний елемент можливо застосувати компонент *Level Indicator* або *Cylindrical Tank*. У його властивостях встановлюється діапазон від 0 до 100, що відповідає відсотковому представленню вихідного сигналу. Властивість *Value* прив’язується до тега аналогового виходу, наприклад 32_an_out6.

Для задання значення оператором використовується компонент *Numeric Text Field* або *Spinner*. Його властивість *Value* прив’язується до того самого вихідного тега з можливістю запису. Це дозволяє змінювати значення виходу безпосередньо з інтерфейсу. Поруч розміщується статичний напис «%» для позначення одиниць вимірювання.

Другий аналоговий вихід налаштовується аналогічно, що забезпечує єдину логіку та візуальну симетрію інтерфейсу.

9.8. Інтеграція *MySQL* зі *SCADA Ignition*

У цьому підрозділі розглядаються принципи інтеграції *SCADA*-системи Ignition з системою керування базами даних *MySQL* для організації зберігання та використання технологічних даних. Така інтеграція є типовою для сучасних автоматизованих систем, у яких *SCADA* виконує функції збору, первинної обробки та візуалізації інформації, а база даних забезпечує її довготривале зберігання, аналіз і подальше використання.

SCADA Ignition у межах посібника обрано як репрезентативний приклад сучасної *SCADA*-платформи, що широко застосовується в промисловій автоматизації. Це зумовлено наявністю вбудованих засобів роботи з реляційними базами даних, підтримкою стандартних механізмів доступу через *JDBC*, а також орієнтацією на відкриті протоколи та кросплатформені рішення. При цьому розглянуті підходи до інтеграції не є унікальними лише для Ignition і можуть бути узагальнені для інших *SCADA*-систем, що підтримують роботу з *MySQL*.

У типовій архітектурі *SCADA*-системи *Ignition* база даних *MySQL* використовується як центральне сховище технологічних параметрів, журналів подій, архівів вимірювань і результатів обчислень. Запис даних до бази здійснюється з боку *SCADA* у процесі виконання сценаріїв керування або опитування обладнання, тоді як читання використовується для відображення історичних значень, формування трендів, звітів і аналітичних представлень у *HMI*.

Інтеграція *Ignition* з *MySQL* реалізується через стандартний механізм підключення до баз даних на основі *JDBC*. Це дозволяє *SCADA*-системі виконувати *SQL*-запити на вставку, оновлення та вибірку даних безпосередньо на сервері *MySQL*, зберігаючи чітке розмежування між рівнем збору даних і рівнем їх зберігання. Такий підхід відповідає класичній багаторівневій архітектурі автоматизованих систем керування.

У контексті роботи в режимі, наближеному до реального часу, інтеграція *SCADA* Ignition з *MySQL* забезпечує два основні напрями взаємодії. Перший

напрям пов’язаний із регулярним записом актуальних технологічних параметрів до бази даних для подальшого аналізу та архівації. Другий напрям стосується зчитування з бази результатів обробки, розрахункових показників або подій, які формуються на рівні СУБД чи зовнішніх сервісів і мають бути відображені оператору або використані в логіці керування.

Таким чином, використання *MySQL* разом зі *SCADA Ignition* дозволяє побудувати узгоджену інформаційну модель технологічного процесу, у якій база даних виконує роль надійного та масштабованого сховища, а *SCADA* забезпечує оперативний доступ до даних і їх візуальне представлення. У наступних підрозділах ці принципи застосовуються для розгляду конкретних сценаріїв обміну даними та організації потокової передачі інформації.

9.8.1. Підключення SCADA Ignition до MySQL

Для роботи *SCADA Ignition* із *MySQL* використовується *JDBC*-драйвер, який забезпечує стандартний механізм з’єднання та виконання *SQL*-запитів з боку *Gateway*. У навчальних сценаріях доцільно застосовувати *MySQL Connector/J*, вибираючи версію, сумісну з установленим *MySQL Server*. Файл драйвера має вигляд *JAR*, наприклад *mysql-connector-j-8.1.0.jar*.

Підключення драйвера виконується через вебінтерфейс *Ignition Gateway* у розділі *Config → Databases → Drivers*. Після додавання або перевірки наявного драйвера в таблиці *JDBC Drivers* має відображатися запис для *MySQL* зі статусом *Installed*, що означає коректне завантаження драйвера середовищем *Gateway*. Статус підключених драйверів показано на рис.9.12.

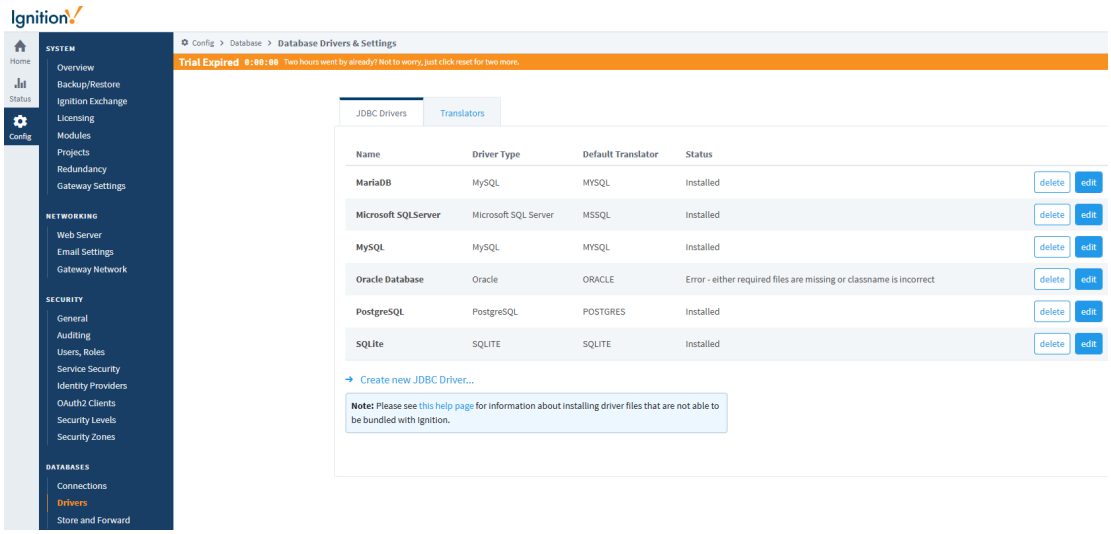


Рис. 9.12. Список підключених драйверів баз даних у *SCADA* Ignition Gateway

Після того як драйвер *MySQL* доступний у системі, створюється з'єднання до конкретної бази даних у розділі *Config* → *Databases* → *Connections*. У параметрах з'єднання задаються ім'я, наприклад *MySQL_SCADA_DB*, адреса у форматі *JDBC URL*, наприклад *jdbc:mysql://localhost:3306/SCADA_db*, а також облікові дані користувача *MySQL*. Якщо *MySQL* розгорнуто на іншому комп'ютері, замість *localhost* вказується *IP*-адреса або домен сервера, наприклад *jdbc:mysql://192.168.1.100:3306/SCADA_db*. У такій постановці *host* є адресою сервера, port є портом *MySQL*, зазвичай 3306, *database* є назвою бази даних.

Коректність налаштованого з'єднання контролюється в розділі *Status* → *Connections* → *Databases*. Для працездатного підключення стан має бути *Valid*, як показано на рис. 9.13. На цій сторінці також відображаються узагальнені показники активних з'єднань і поточної інтенсивності запитів, що є зручним для первинної перевірки в навчальних прикладах.

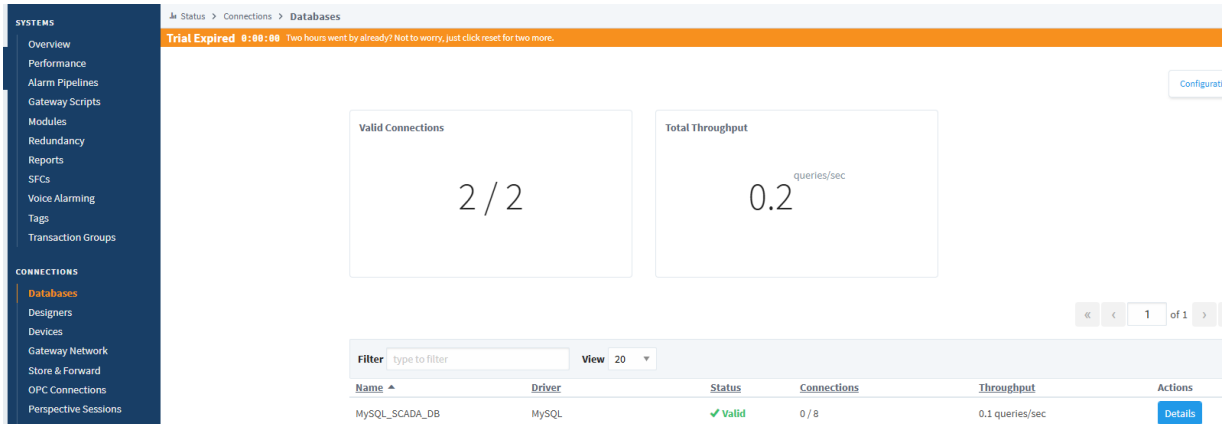


Рис. 9.13. Стан підключень до баз даних у *SCADA Ignition Gateway*

Якщо з'єднання не переходить у стан *Valid*, діагностика виконується через журнал *Gateway* у розділі *Status* або *Config* → *Logs*. Типові причини пов'язані з невірними обліковими даними *MySQL*, недоступністю сервера, некоректним *URL* або помилкою завантаження *JDBC*-драйвера.

Ignition Designer містить вбудовані засоби, які дають змогу працювати з *MySQL* без залучення зовнішніх утиліт у процесі налагодження.

Для виконання запитів безпосередньо з *Designer* використовується інструмент *Database Query Browser*. Його застосовують, коли потрібно швидко виконати `SELECT`, `INSERT` або іншу SQL-інструкцію і одразу побачити результат. Типовим тестом є запит `SELECT NOW();`, оскільки він не залежить від наявності таблиць і повертає поточні дату та час сервера *MySQL*. У вікні *Database Query Browser* вводиться текст запиту у верхньому полі, а в правій частині обирається підключення до бази даних, створене в Gateway, наприклад *MySQL_SCADA_DB*. Після натискання *Execute* результат з'являється в нижній панелі у вигляді таблиці з одним стовпцем `NOW()` і одним рядком значення часу. Приклад інтерфейсу та результату показано на рис. 9.14.

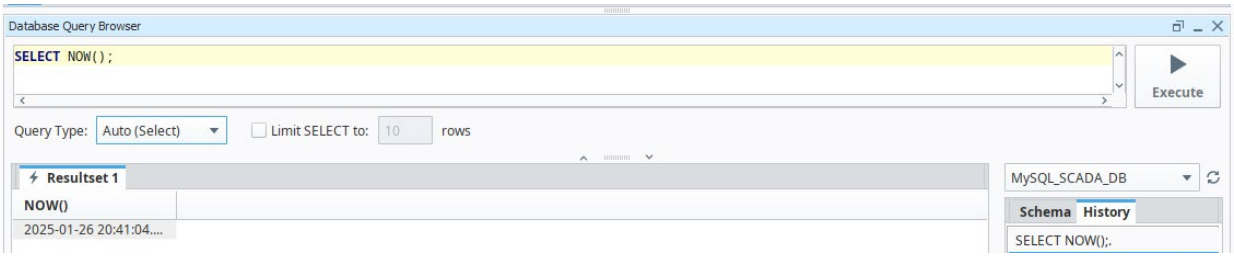


Рис. 9.14. Вікно *Database Query Browser* у *Ignition Designer*

Додатковим інструментом перевірки є *Script Console*. У *Ignition* вона виконує скрипти на *Jython*, тому приклади варто подавати саме в цьому форматі. *Script Console* зручна тоді, коли потрібно не лише виконати запит, а й одразу обробити результат, перевірити обробку винятків або вивести дані у потрібному вигляді. Для тесту з'єднання достатньо виконати той самий `SELECT NOW()`, але вже через *system.db.runQuery* із явним зазначенням імені підключення. Рекомендовано використовувати точну назву підключення без зайвих пробілів, наприклад *MySQL_SCADA_DB*.

```
    connection = "MySQL_SCADA_DB"
    query = "SELECT NOW()"
    try:
        results = system.db.runQuery(query, connection)
        print "Connection is successful. Current time
from the database:", results[0][0]
    except Exception as e:
        print "Connection error:", str(e)
```

У разі коректного підключення в консолі з'являється повідомлення про успіх і значення часу, яке часто відображається як об'єкт типу *Java Date*, наприклад: *Connection is successful. Current time from the database: Sun Jan 26 19:08:17 EET 2025*. Якщо підключення задано неправильно або відсутні права доступу, виводиться повідомлення про помилку, що дає змогу швидко відрізнити проблеми мережі, автентифікації та доступу до бази.

Після того як підтверджено працездатність з'єднання, переходять до технологічної частини, де джерелом даних для запису у *MySQL* є теги. У практичних системах теги формуються на основі підключення до обладнання

через *OPC* (наприклад *OPC UA*) або через інший драйвер/шлюз, а в навчальних прикладах джерелом можуть бути як реальні канали, так і імітаційні значення. Важливо, щоб у *Tag Brow*ser були доступні потрібні теги, а їх значення змінювалися або принаймні оновлювалися. На рис. 9.15 наведено приклад, де в гілці тегів відображено *Channel_1*, *Channel_2*, *Channel_3*, *Channel_*4 та їх поточні значення. Саме наявність цих тегів у проєкті визначає, які поля доцільно закладати в структуру таблиці *MySQL* для історизації.

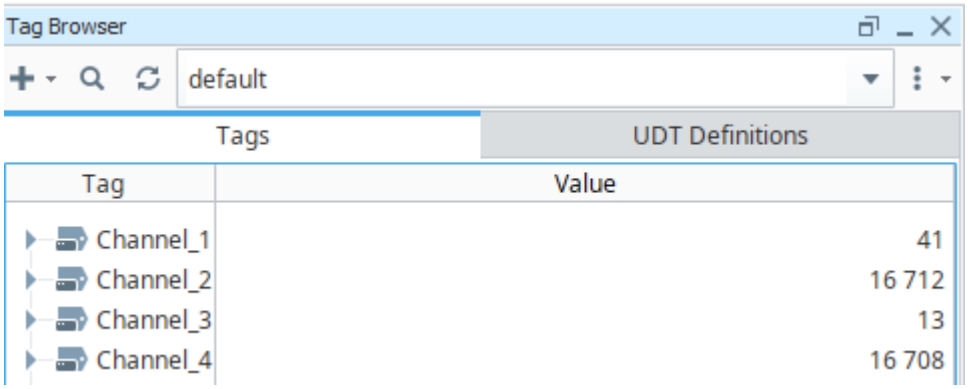


Рис. 9.15. Вікно *Tag Browser* у *Ignition Designer*

Щоб значення тегів автоматично записувалися у *MySQL* з заданою періодичністю, у *Ignition* застосовують *Transaction Groups*. Їхня роль полягає в узгодженні трьох складових: списку тегів, підключення до бази даних і режиму запису. У типових задачах історизації вибирають режим вставки нового рядка під час кожного циклу, оскільки це формує послідовність вимірювань, прив’язаних до часу. Для цього доцільно заздалегідь створити таблицю, яка містить колонку часу та колонки під кожен технологічний параметр. Якщо база даних *SCADA_db* вже створена на попередніх етапах, достатньо створити таблицю в цій базі, і надалі використовувати її як цільову.

```
CREATE TABLE IF NOT EXISTS scada_data (
  id INT AUTO_INCREMENT PRIMARY KEY,
  t_stamp DATETIME DEFAULT CURRENT_TIMESTAMP,
  Channel_1 INT,
  Channel_2 INT,
```

```
  Channel_3 INT,
  Channel_4 INT
);
```

У цій структурі *id* використовується як технічний ідентифікатор рядка, *t_stamp* зберігає мітку часу запису, а поля *Channel_1…Channel_4* містять значення відповідних тегів. Якщо теги мають інший тип, наприклад *REAL* або *Double*, типи колонок у таблиці слід узгодити з фактичними даними, наприклад використовувати *FLOAT* або *DOUBLE*. У навчальному прикладі допустимо залишати INT, якщо значення каналів цілі або приводяться до цілих.

Налаштування *Transaction Group* виконують у *Designer* у розділі *Transaction Groups*, де створюється група стандартного типу і до її складу додаються вибрані теги. Під час додавання важливо, щоб зіставлення між тегом і колонкою таблиці було однозначним: для кожного тегу задається цільове ім’я, яке збігається з назвою колонки в *MySQL*. Якщо передбачено запис мітки часу, у параметрах групи задають використання колонки *t_stamp* як поля часу, щоб у кожному вставленому рядку фіксувався момент запису. У частині налаштувань, пов’язаній із базою даних, вибирають *Data Source*, тобто підключення *MySQL_SCADA_DB*, і вказують *Table name*, тобто *scada_data*. Приклад загального вигляду налаштувань і параметрів, пов’язаних із джерелом даних, таблицею, режимом запису та періодом, показано на рис. 9.16. Для демонстраційного режиму зручно задавати інтервал, наприклад 30 секунд, щоб записи з’являлися у таблиці з помітним кроком і їх було легко перевіряти.

Після ввімкнення групи дані починають накопичуватися в таблиці *MySQL*. Перевірку доцільно виконувати або через *MySQL* Workbench, або через Database Query Browser у Designer, оскільки це забезпечує однакову картину з точки зору SQL. Практичним є запит на перегляд останніх записів, відсортованих за ідентифікатором або часом.

```
SELECT id, t_stamp, Channel_1,
Channel_2, Channel_3, Channel_4
FROM scada_data
ORDER BY id DESC
LIMIT 10;
```

Якщо таблиця заповнюється, у результаті відображаються рядки з послідовними значеннями id, мітками часу t_stamp та значеннями каналів, як показано на рис.9.17.

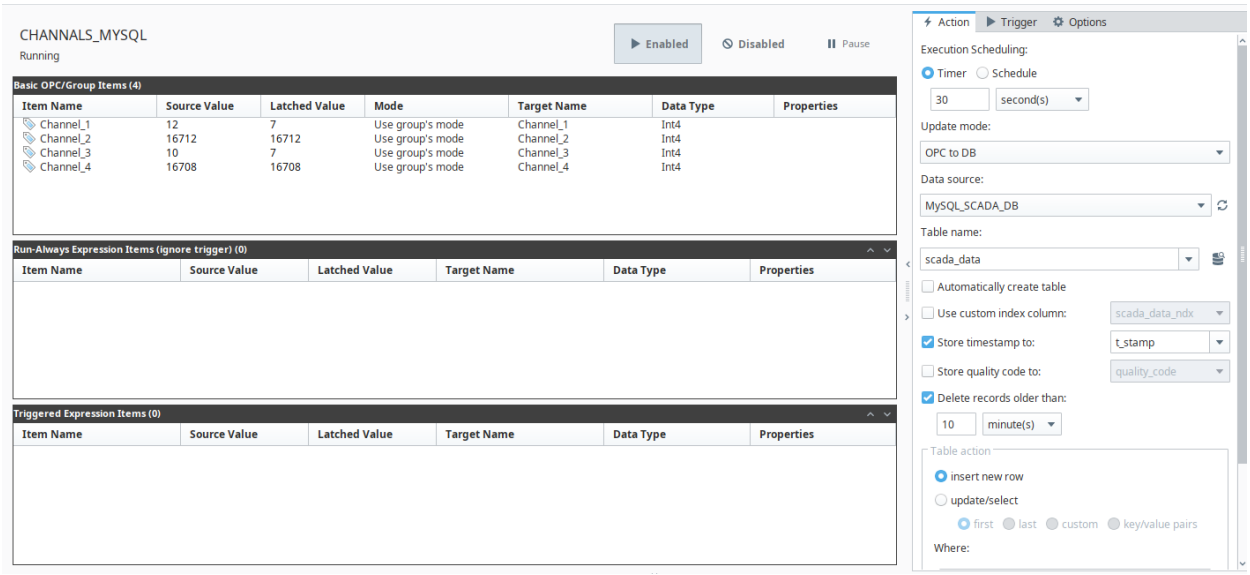


Рис. 9.16. Налаштування підключення до *MySQL* у *Ignition Designer*

```
mysql> SELECT * FROM scada_db.scada_data;
+-----+---------------------+-----------+-----------+-----------+-----------+
| id  | t_stamp             | Channel_1 | Channel_2 | Channel_3 | Channel_4 |
+-----+---------------------+-----------+-----------+-----------+-----------+
| 617 | 2025-02-13 21:40:41 |        55 |     16712 |        20 |     16708 |
| 618 | 2025-02-13 21:41:11 |        70 |     16712 |        13 |     16708 |
| 619 | 2025-02-13 21:41:41 |        85 |     16712 |         6 |     16708 |
| 620 | 2025-02-13 21:42:11 |       100 |     16712 |        16 |     16708 |
| 621 | 2025-02-13 21:42:41 |        14 |     16712 |         9 |     16708 |
| 622 | 2025-02-13 21:43:11 |        29 |     16712 |        19 |     16708 |
| 623 | 2025-02-13 21:43:41 |        44 |     16712 |        12 |     16708 |
| 624 | 2025-02-13 21:44:11 |        59 |     16712 |         5 |     16708 |
| 625 | 2025-02-13 21:44:41 |        74 |     16712 |        15 |     16708 |
| 626 | 2025-02-13 21:45:11 |        89 |     16712 |         8 |     16708 |
| 627 | 2025-02-13 21:45:41 |         3 |     16712 |        18 |     16708 |
| 628 | 2025-02-13 21:46:11 |        18 |     16712 |        11 |     16708 |
| 629 | 2025-02-13 21:46:41 |        33 |     16712 |         4 |     16708 |
| 630 | 2025-02-13 21:47:11 |        48 |     16712 |        14 |     16708 |
| 631 | 2025-02-13 21:47:41 |        63 |     16712 |         7 |     16708 |
+-----+---------------------+-----------+-----------+-----------+-----------+
```

Рис. 9.17. Таблиця з результатами в *MySQL*

Такий результат означає, що повний ланцюжок “теги *Ignition* → *Transaction Groups* → *MySQL*” працює коректно, а база даних може

використовуватися як джерело історичних значень для трендів і звітів. Якщо записи не з’являються, причину зазвичай встановлюють у двох площинах: або теги не мають валідних значень, або група не може виконати запис у *MySQL* через некоректно вибране підключення, відсутність таблиці чи права доступу. У таких випадках зручно повернутися до тесту `SELECT NOW()` у *Database Query Browser* та до перегляду актуальних значень тегів у *Tag Browser*, оскільки саме ці два спостереження швидко відсікають значну частину типових помилок.

Описаний підхід відповідає поширеній практиці використання *MySQL* як сховища історичних технологічних даних у зв’язці зі *SCADA*. У наступних підрозділах ці самі дані розглядаються як основа для зворотних сценаріїв, коли результати обробки в *MySQL* або у зовнішніх сервісах потрібно повернути в *SCADA* для відображення оператору або використання в прикладній логіці.

9.9. Передача даних з *MySQL* до *SCADA Ignition через Node-RED* і *MQTT*

У цьому підрозділі розглядається передача даних з бази даних *MySQL* до *SCADA*-системи *Ignition* із використанням середовища *Node-RED* та протоколу *MQTT*. Джерелом даних є таблиця *MySQL*, що наповнюється в режимі, наближеному до реального часу, засобами *SCADA Ignition* у підрозділі 9.8. У межах наведеного прикладу під передачею змін розуміється публікація лише нових записів, які з’являються в таблиці, без повторного надсилання раніше переданих значень.

Обмін даними організовано за такою узагальненою схемою: база даних *MySQL* формує нові рядки вимірювань, середовище *Node-RED* виконує інкрементальне зчитування таблиці, перетворює отримані значення у повідомлення *MQTT* та публікує їх на брокер повідомлень *Mosquitto* (*broker*). Далі *SCADA Ignition* отримує ці повідомлення через модуль *MQTT Engine* і відображає їх у вигляді тегів. Така організація дозволяє відпрацювати повний

ланцюжок «база даних → проміжна маршрутизація → брокер повідомлень → *SCADA*» у середовищі *Windows*, зберігаючи характерну для промислової автоматизації модель publish/subscribe (публікація/підписка) та забезпечуючи оперативне оновлення даних у *SCADA*-системі.

9.9.1. Архітектура передачі даних та ролі компонентів

Архітектура передачі даних ґрунтується на поєднанні реляційної бази даних, проміжного середовища потокової обробки та *SCADA*-системи, що взаємодіють за моделлю асинхронного обміну повідомленнями. Загальна схема побудована за принципом односторонньої передачі змін від рівня зберігання даних до рівня диспетчерського відображення без прямого з'єднання між *MySQL* і *SCADA Ignition*.

База даних *MySQL* у цій архітектурі виконує роль джерела даних. Вона накопичує результати вимірювань і станів технологічних змінних у вигляді рядків таблиці, які формуються *SCADA Ignition* за допомогою механізму Transaction Groups. Таблиця містить часову мітку та значення технологічних параметрів і використовується як первинне сховище історичних даних. У межах цього підрозділу *MySQL* не ініціює передачу інформації, а лише надає доступ до нових записів для зовнішніх компонентів.

Середовище *Node-RED* виступає як проміжний рівень маршрутизації та обробки даних. Його функція полягає у періодичному опитуванні таблиці *MySQL*, виявленні нових записів і формуванні повідомлень для подальшої передачі. На відміну від прямого підключення *SCADA* до бази даних, *Node-RED* дозволяє реалізувати логіку інкрементального читання, перетворення форматів і керування частотою передачі без втручання у роботу *SCADA* або структуру бази даних. Таким чином *Node-RED* виконує роль програмного шлюзу між реляційною моделлю даних і повідомлювальною архітектурою.

Передача даних між *Node-RED* і *SCADA Ignition* здійснюється за допомогою протоколу *MQTT*, який реалізує модель *publish/subscribe*. У цій схемі *Node-RED* є публікатором повідомлень, що надсилає сформовані дані у визначені теми, а

SCADA Ignition виступає підписником. Посередником між ними є *MQTT*-брокер Mosquitto, який забезпечує приймання, зберігання та доставку повідомлень клієнтам відповідно до налаштованих підписок. Брокер не аналізує вміст повідомлень і не зберігає стан технологічних змінних, а лише гарантує коректний обмін повідомленнями між компонентами.

На стороні *SCADA Ignition* за приймання *MQTT*-повідомлень відповідає модуль *MQTT Engine*. Він виконує підписку на задані теми брокера, перетворює структуру повідомлень у внутрішній адресний простір *SCADA* та автоматично формує теги відповідно до ієрархії тем. У розглянутому прикладі використовується режим Custom namespaces, у якому шлях *MQTT*-теми безпосередньо відображається у шлях тегів Ignition. Це дозволяє приймати дані без використання специфічних промислових надбудов і зосередитися на загальній логіці потокової передачі.

Загалом описана архітектура забезпечує зв’язування між компонентами, масштабованість та чітке розмежування функцій між рівнями збору, зберігання, маршрутизації та візуалізації даних. Такий підхід є характерним для сучасних інформаційних систем автоматизації та створює основу для подальшого переходу до більш складних потокових рішень, зокрема з використанням механізмів відстеження змін і стрімінгових платформ.

9.9.2. MQTT broker Mosquitto у Windows

Для реалізації обміну повідомленнями між *Node-RED* та *SCADA Ignition* у даному прикладі використовується *MQTT*-брокер *Mosquitto*. Він виконує функцію центрального вузла, який приймає повідомлення від публікаторів та передає їх усім підписникам відповідно до заданих тем. Завданням *Mosquitto* є забезпечення надійного приймання та розповсюдження повідомлень *MQTT* у локальному середовищі *Windows*.

Інсталяція *Mosquitto* у *Windows* здійснюється за допомогою офіційного інсталятора з розширенням *.exe*, який доступний на сайті проєкту *Eclipse Mosquitto*. Під час встановлення доцільно обрати варіант інсталяції брокера як

служби *Windows*, що забезпечує його автоматичний запуск разом з операційною системою та стабільну роботу без ручного втручання користувача. Типовим каталогом встановлення є *C:\Program Files\mosquitto*, у якому розміщуються виконувані файли та конфігураційні параметри брокера.

Після завершення інсталяції необхідно переконатися, що служба *Mosquitto Broker* запущена. Це виконується через стандартний інструмент керування службами *Windows*, який відкривається командою services.msc. У переліку служб має бути присутній запис *Mosquitto Broker* зі станом *Running*. Якщо служба зупинена, її запуск здійснюється вручну через контекстне меню. Запущений стан служби свідчить про готовність брокера до приймання клієнтських підключень.

Додаткова перевірка доступності брокера виконується за допомогою команди `netstat -an | findstr 1883` у командному рядку. У результаті повинен відображатися відкритий порт 1883 у стані прослуховування (*LISTENING*), що свідчить про коректну роботу служби.

Файл конфігурації *mosquitto.conf* за замовчуванням знаходиться в директорії встановлення за шляхом *C:\Program Files\mosquitto\mosquitto.conf*. У разі відсутності файлу його потрібно створити вручну, дотримуючись мінімального необхідного формату для коректного запуску брокера.

Зміст базового конфігураційного файлу *mosquitto.conf* має такий вигляд:

```
listener 1883
allow_anonymous true
max_connections 100
```

Параметр `listener` визначає номер порту, який використовує брокер для приймання вхідних з’єднань. Параметр *allow_anonymous* вмикає або вимикає автентифікацію клієнтів; у навчальному середовищі доцільно дозволити анонімні з’єднання. Параметр *max_connections* задає граничну кількість одночасно підключених клієнтів.

Після застосування або перевірки конфігурації служба *Mosquitto Broker* має бути перезапущена для гарантованого використання актуальних параметрів.

9.9.3. Node-RED як публікатор даних з MySQL у MQTT

Node-RED у даному прикладі використовується як проміжний програмний компонент, який виконує інкрементальне читання нових записів із таблиці *MySQL* та публікує їх у вигляді повідомлень *MQTT* у заданій темі. Такий підхід дозволяє забезпечити регулярну синхронізацію даних без повторного надсилання вже переданих рядків, що узгоджується з прийнятим у цьому підрозділі трактуванням «передачі змін» як передачі лише нових записів.

Для роботи *Node-RED* у *Windows* необхідно встановити платформу *Node.js*, яка є середовищем виконання *JavaScript* поза браузером. Інсталяційний пакет *Node.js* завантажують з офіційного сайту https://nodejs.org/en/download/, після чого виконують встановлення зі стандартними параметрами. Важливо перевірити, щоб під час інсталяції була увімкнена опція додавання до системної змінної *PATH*, оскільки без цього виконання npm-команд у терміналі буде ускладненим.

Після інсталяції *Node.js* встановлюють *Node-RED* як глобальний пакет npm. Для цього відкривають *PowerShell* або *CMD*, за потреби з правами адміністратора, і виконують команду встановлення.

```
npm install -g --unsafe-perm node-red
```

Після завершення інсталяції Node-RED запускають командою

```
node-red.
```

У консолі має з’явитися повідомлення про запуск сервера, а вебінтерфейс *Node-RED* стає доступним за адресою http://127.0.0.1:1880

Для реалізації потоку «*MySQL* → *MQT*T» потрібні два типи функціональних вузлів. Перший тип забезпечує підключення до *MySQL* і

виконання *SQL*-запитів, для чого застосовують пакет *node-red-node-mysql*. Другий тип забезпечує публікацію в *MQTT*-брокер, для чого достатньо стандартного вузла *mqtt out*, який у більшості інсталяцій *Node-RED* доступний за замовчуванням. Окремі пакети для «вбудованого брокера» в цій роботі не є обов'язковими, оскільки брокер *Mosquitto* вже розгорнуто як окрему службу *Windows* і виступає зовнішнім сервером *MQTT*.

Перевірка наявності необхідних вузлів виконується у вебінтерфейсі *Node-RED* через меню, розділ *Manage palette*, вкладку *Nodes* або *Installed*. У списку встановлених пакетів має бути присутній node-red-node-mysql. Якщо його немає, пакет встановлюють через вкладку *Install* у тому ж розділі, після чого *Node-RED*, як правило, не потребує повного перевстановлення, але інколи доцільно перезапустити *Node-RE*D для коректного підхоплення нових вузлів.

Потік у *Node-RED* будується так, щоб з заданою періодичністю ініціювати *SQL*-запит до *MySQL*, отримати нові рядки таблиці, підготувати повідомлення у погодженому форматі та опублікувати його у *MQTT*-тему. Типова послідовність вузлів має вигляд: *inject → function (формування SQL) → mysql → function (підготовка payload і topic) → mqtt out.*

Періодичний запуск реалізується вузлом *inject*. У його параметрах задають інтервал спрацювання, наприклад кожні 5 секунд, та, за потреби, увімкнення запуску одразу після старту *Node-RED*. Вузол *inject* не формує корисного навантаження сам по собі, а лише служить тригером циклічного опитування бази даних.

Далі у вузлі *function* формується *SQL*-запит. Саме на цьому етапі закріплюється вимога «лише нові записи», оскільки запит має повертати тільки ті рядки, що ще не були передані в *MQTT*. Для навчального прикладу доцільно використати одну з двох простих стратегій.

Перша стратегія базується на інкрементальному первинному ключі *id*. У цьому разі у *Node-RED* зберігається останній переданий ідентифікатор *last_id*. Під час кожного циклу формується запит виду

```
SELECT * FROM sensor_data WHERE id > last_id ORDER BY id ASC;
```

Після отримання результатів *last_id* оновлюється на найбільше значення *id* з оброблених рядків. Такий підхід є наочним, технічно простим і добре підходить для навчальних задач, якщо таблиця має монотонно зростаючий *id*.

Друга стратегія базується на часовій мітці *t_stamp*. У цьому разі зберігається останній опрацьований час *last_ts*, а запит має вигляд

```
SELECT * FROM sensor_data WHERE t_stamp > last_ts ORDER BY t_stamp ASC;.
```

Після обробки *last_ts* оновлюється до максимальної *t_stamp* із результатів. Цей варіант доцільний, коли логіка формування даних прив’язана до часу або коли id може бути несуцільним. Водночас у навчальному сценарії він потребує уважності до формату часу та часової зони, тому для першого проходження зазвичай простіше використовувати саме *id*.

Щоб *Node-RED* пам’ятав *last_id* або *last_ts* між спрацюваннями *inject*, використовується механізм контексту, тобто *context*. Найпростіший варіант для навчального прикладу полягає у використанні контексту вузла або контексту потоку, наприклад *flow context*. У цьому випадку значення *last_id* читається на початку циклу, використовується для формування *SQL*-запиту, а після успішної обробки результатів оновлюється. Такий механізм зберігає значення в оперативній пам’яті між циклами роботи потоку. За потреби зберігати *last_id* навіть після перезапуску *Node-RED*, застосовують *persistent context*, але для навчальної постановки це не є обов’язковим, якщо явно не вимагається відновлення стану після рестарту.

Після виконання *SQL*-запиту вузол *MySQL* повертає масив рядків. Далі у вузлі *function* виконується перетворення кожного рядка у повідомлення *MQTT*. У цьому ж вузлі задається тема публікації, наприклад *SCADA/sensors/data* або інший узгоджений варіант. Формат *payload* доцільно робити однозначним для подальшого перетворення в теги *Ignition*, наприклад *JSON* із полями

temperature, pressure, flow_rate, t_stamp та *id*. Після підготовки повідомлення дані передаються на вузол mqtt out, у якому задаються параметри підключення до брокера *Mosquitto*. Як сервер вказують loc*a*lhost, порт 1883, після чого підтверджують налаштування та виконують *Deploy* потоку.

У результаті після запуску потік циклічно виконує інкрементальне читання таблиці, формує повідомлення лише для нових рядків та публікує їх у *MQTT*-тему. Це забезпечує узгоджений обмін даними з брокером *Mosquitto* та підготовлює основу для подальшого підключення *MQTT Engine в Ignition* і побудови тегів на стороні *SCADA*.

9.9.4. Формат повідомлення і правила іменування тем

У цьому підпункті фіксуються вимоги до формату повідомлень *MQTT* і правил формування тем (*topic*), яких необхідно дотримуватися на *стороні Node-RED* для коректної інтеграції з *SCADA Ignition* через модуль *MQTT Engine* у режимі *Custom namespaces*. Чітке задання цих правил є важливим, оскільки саме вони визначають структуру тегів, що автоматично формуються в *Ignition*, та передбачуваність адресного простору *SCADA*.

В проекті передбачається використання *MQTT Engine* у режимі *Custom* namespaces з типом *Namespace JSON*. За такої конфігурації *MQTT Engine* інтерпретує тіло повідомлення як структуровані дані у форматі *JSON* і автоматично перетворює вкладені поля *JSON*-об’єкта у відповідні теги *Ignition*.

З цієї причини payload повідомлення, яке публікує *Node-RED*, має бути саме *JSON*-об’єктом, а не текстовим рядком виду ключ=значення. Використання рядкового формату ускладнює подальшу обробку, потребує додаткового парсингу на стороні *SCADA* і не відповідає логіці роботи *Custom namespaces.*

Типовий приклад *payload* у форматі *JSON* може мати такий вигляд:

```
{
"id": 125,
"temperature": 23.8,
```

```
"pressure": 1.03,
"flow_rate": 10.4,
"t_stamp": "2026-01-28T14:32:05"
}
```

У цій структурі кожне поле *JSON* відповідає окремому технологічному параметру або службовому атрибуту. *MQTT Engine* при отриманні такого повідомлення створює або оновлює відповідні теги *Ignition*, значення яких автоматично синхронізуються з вхідними повідомленнями. Такий підхід є прозорим, легко масштабованим і зручним для подальшого використання даних у *HMI*, скриптах або модулях історизації.

Не менш важливим є уніфікований підхід до іменування *MQTT*-тем. У даному прикладі використовується ієрархічна структура topic, наприклад:

```
SCADA/sensors/data
```

Така форма обрана з кількох міркувань. По-перше, вона логічно відображає роль повідомлень у системі: *SCADA* позначає приналежність до диспетчерського рівня, sensors вказує на джерело або семантику даних, а *data* узагальнює тип переданого вмісту. По-друге, ієрархічна структура дозволяє застосовувати фільтрацію тем у *MQTT Engine* за допомогою шаблонів, наприклад *SCADA*/#, що спрощує налаштування підписки на групи повідомлень.

За умови використання *Custom* namespaces і *topic filter* виду *SCADA*/# у *Ignition* формується дерево тегів із передбачуваною структурою:

Tags → MQTT Engine → Custom → SCADA → sensors → data

У цьому вузлі кожне поле *JSON*-повідомлення перетворюється на окремий тег. Завдяки цьому шлях до тегів є стабільним і зрозумілим, а додавання нових параметрів у *JSON* не потребує ручного створення тегів у *SCADA*.

На стороні *Node-RED* у вузлі *function*, який формує повідомлення перед передачею в *mqtt out*, необхідно явно створювати *payload* як об'єкт *JavaScript*, а не як рядок. *Node-RED* автоматично серіалізує об'єкт у *JSON* під час публікації в *MQTT*, що повністю узгоджується з очікуваннями *MQTT Engine* у режимі *JSON*.

Таким чином, розглядаються два принципові положення. По-перше, для інтеграції з *MQTT Engine* у режимі *Custom namespaces pa*yload має бути *JSON*-структурою. По-друге, *topic* формується у вигляді ієрархічного шляху, наприклад *SCADA/sensors/data*, що забезпечує передбачуване формування тегів у *SCADA Ignition*. Дотримання цих правил дозволяє уникнути неоднозначностей під час приймання даних і спрощує подальше розширення системи.

9.9.5. Налаштування MQTT Engine в Ignition: Sparkplug B і Custom namespaces

Модуль *MQTT Engine* у *SCADA Ignition* підтримує дві концептуально різні моделі роботи з *MQTT*-повідомленнями: *Sparkplug B* та *Custom namespaces*. Обидві моделі ґрунтуються на протоколі *MQTT*, проте відрізняються підходом до структурування повідомлень, правилами іменування тем і способом формування тегів у *SCADA*.

Модель *Sparkplug B* орієнтована на промислові системи з чітко визначеною ієрархією пристроїв і вузлів. Вона передбачає використання стандартизованого формату повідомлень, жорстко регламентовану структуру topic та наявність спеціалізованих *Sparkplug*-публікаторів на стороні джерел даних. У такій моделі *SCADA* отримує не лише значення параметрів, а й службову інформацію про стан вузлів, їхню доступність і життєвий цикл. *Sparkplug B* доцільно застосовувати у промислових проєктах із великою кількістю фізичних пристроїв і вимогами до формалізованої взаємодії.

Модель *Custom namespaces* є більш універсальною та гнучкою. Вона не накладає жорстких вимог на формат повідомлень і дозволяє використовувати

довільну ієрархію *MQTT*-тем. Формування тегів у *Ignition* у цьому випадку здійснюється на основі структури *topic* та вмісту p*a*yload. Такий підхід зручний для інтеграції з програмними джерелами даних, зокрема *Node-RED*, які не реалізують *Sparkplug*-протокол, а також для навчальних і демонстраційних задач.

У межах даного підрозділу використовується саме модель *Custom namespaces*. Це зумовлено тим, що джерелом повідомлень є *Node-RED*, який публікує дані у форматі *JSON* без підтримки *Sparkplug B*, а метою прикладу є наочне опрацювання базового механізму передачі змін з *MySQL* у *SCADA Ignition* через *MQTT* без ускладнення архітектури.

Налаштування *Custom namespaces* виконується у вебінтерфейсі *Ignition* Gateway. Після входу під обліковим записом адміністратора необхідно перейти до розділу *Config → MQTT Engine → Settings* і відкрити секцію *Namespaces*. У цій секції створюється або редагується запис типу *Custom*.

Для даного прикладу параметри *Custom namespace* задаються таким чином. Як тип namespace обирається *JSON*, що означає інтерпретацію тіла *MQTT*-повідомлення як *JSON*-структури. Параметр *Sparkplug B Enabled* має бути вимкнений. У полі *Topic Filter* задається шаблон, наприклад *SCADA*/#, який дозволяє приймати всі повідомлення, теми яких починаються з префікса *SCADA*.

Після збереження налаштувань *MQTT Engine* починає обробляти всі вхідні повідомлення, що відповідають заданому фільтру. Важливо розуміти відповідність між topic і шляхом тегів у *Ignition*. Якщо *Node-R*ED публікує повідомлення з темою *SCADA/sensors/data*, то в *Tag Browser* формується ієрархія виду:

Tags → MQTT Engine → Custom → SCADA → sensors → data

У цьому вузлі кожне поле *JSON*-повідомлення автоматично відображається як окремий тег. Наприклад, поле *temperature* у *payload* відповідає тегу *temperature*, поле *pressure* - тегу *pressure* тощо. Таким чином,

структура *topic* безпосередньо визначає шлях тегів у *SCADA*, а структура *JSON* - їх внутрішній склад.

Такий механізм забезпечує прозору та передбачувану інтеграцію. Зміна імені теми або додавання нових полів у *JSON* призводить до автоматичного розширення адресного простору тегів без ручного втручання. Саме тому модель *Custom namespaces* є доцільною для навчального прикладу, у якому важливо показати загальний принцип взаємодії *MySQL, Node-RED, MQTT* і *SCADA Ignition* у режимі передачі змін.

9.9.6. Приклад створення потоку передачі даних із MySQL у SCADA Ignition через Node-RED та MQTT

Розглянемо приклад створення потоку в *Node-RED*, який виконує інкрементальне читання таблиці *MySQL* та публікує лише нові записи у *MQTT*. Надалі *SCADA Ignition* отримує ці повідомлення через модуль *MQTT Engine* і відображає їх у вигляді тегів. Приклад розраховано на роботу в середовищі *Windows* і передбачає, що брокер *Mosquitto* вже встановлено та підтверджено його працездатність, а в *Ignition* налаштовано *MQTT Engine* у режимі *Custom namespaces*. Такий підхід має практичну цінність у промислових системах автоматизації, оскільки дозволяє зменшити навантаження на мережу та сервери, забезпечуючи передавання лише актуальних змін технологічних даних, а також спрощує інтеграцію між різнорідними підсистемами без жорсткої прив’язки між ними.

Необхідно мати створене та працездатне підключення до *MySQL*, а також базу даних *SCADA_db*, у якій таблиця наповнюється даними в режимі реального часу засобами *Ignition*. Для коректної реалізації підходу «лише нові записи» таблиця має містити монотонно зростаючий ідентифікатор *id* або часову мітку *t_stamp*, за якими можна визначити появу нових рядків.

Крок 1. Підготовка таблиці в MySQL для демонстрації інкрементального читання

У *MySQL* створюють таблицю, яка містить технічний ідентифікатор та часову мітку вставки. Нижче наведено приклад структури таблиці sensor_data.

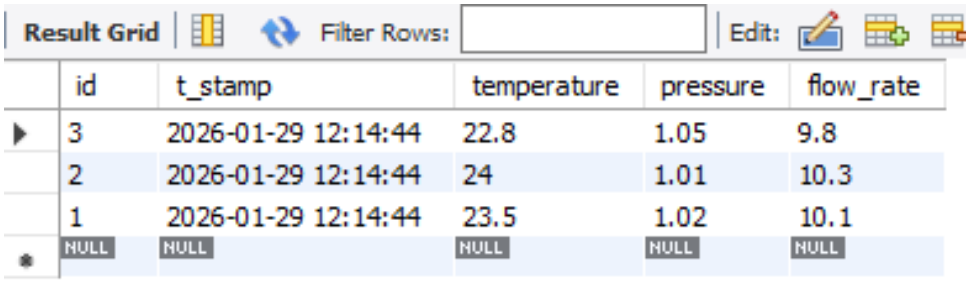
```
CREATE DATABASE IF NOT EXISTS SCADA_db;
USE SCADA_db;
CREATE TABLE IF NOT EXISTS sensor_data (
id INT AUTO_INCREMENT PRIMARY KEY,
t_stamp DATETIME DEFAULT CURRENT_TIMESTAMP,
temperature FLOAT NOT NULL,
pressure FLOAT NOT NULL,
flow_rate FLOAT NOT NULL
);
```

За потреби додають декілька записів для первинної перевірки.

```
INSERT INTO sensor_data (temperature, pressure,
flow_rate) VALUES
(23.5, 1.02, 10.1),
(24.0, 1.01, 10.3),
(22.8, 1.05, 9.8);
```

Контроль правильності виконують запитом:

```
SELECT * FROM sensor_data ORDER BY id DESC LIMIT 5;
```



id	t_stamp	temperature	pressure	flow_rate
3	2026-01-29 12:14:44	22.8	1.05	9.8
2	2026-01-29 12:14:44	24	1.01	10.3
1	2026-01-29 12:14:44	23.5	1.02	10.1
NULL	NULL	NULL	NULL	NULL

Рис. 9.18. Таблиця *sensor_data* у *MySQL*

Крок 2. Запуск Node-RED і підготовка необхідних вузлів

Node-RED запускають командою:

```
node-red
```

Після запуску у консолі відображається адреса вебінтерфейсу, зазвичай http://127.0.0.1:1880/. Далі у браузері відкривають інтерфейс *Node-RED*.

Перед побудовою потоку необхідно переконатися, що встановлено вузли для доступу до *MySQL* та для публікації повідомлень у *MQTT*. У *Node-RED* відкривають меню, переходять у *Manage palette* і перевіряють наявність пакета *node-red-node-mysql* (див.рис.9.19).

Якщо вузол відсутній, його встановлюють через вкладку *Install*.

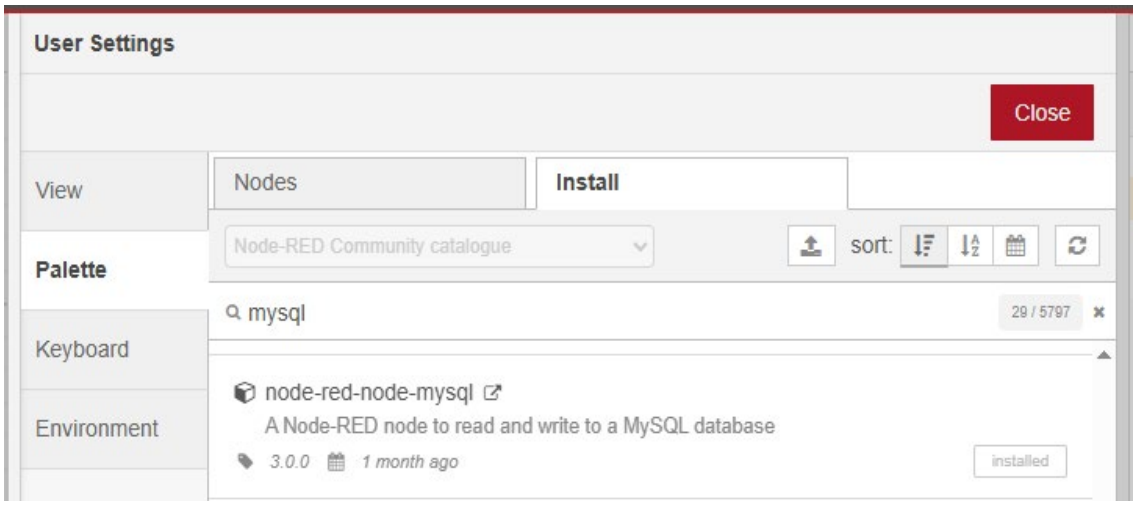


Рис. 9.19. Перевірка встановлення пакета *node-red-node-mysql* у *Manage palette.*

Крок 3. Формування потоку Node-RED для інкрементального читання MySQL

Потік будується так, щоб кожен цикл опитування обирав лише ті рядки, які мають *id* більший за останній опрацьований. Значення «останнього *id*» зберігається між циклами у контексті потоку *Node-RED* (*flow context*).

У робочій області *Node-RED* розміщують вузли у такій послідовності: *inject* → *function* (формування інкрементального *SQL*) → *MySQL* → function (підготовка *JSON* і *topic*) → *mqtt out* (див.рис.9.20).

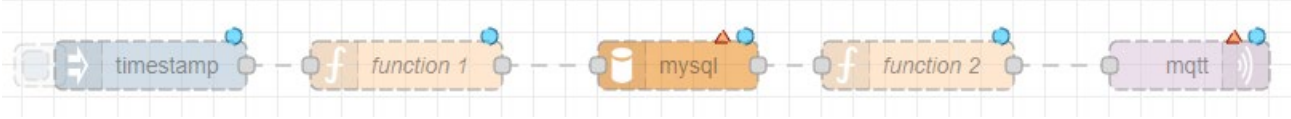


Рис. 9.20. Загальний вигляд потоку Node-RED для інкрементального читання MySQL та публікації в MQTT

Крок 3.1. Налаштування вузла inject

Вузол *inject* налаштовують на періодичне спрацювання, наприклад кожні 5 секунд. Додатково вмикають опцію спрацювання одразу після запуску потоку (див. рис.9.21)

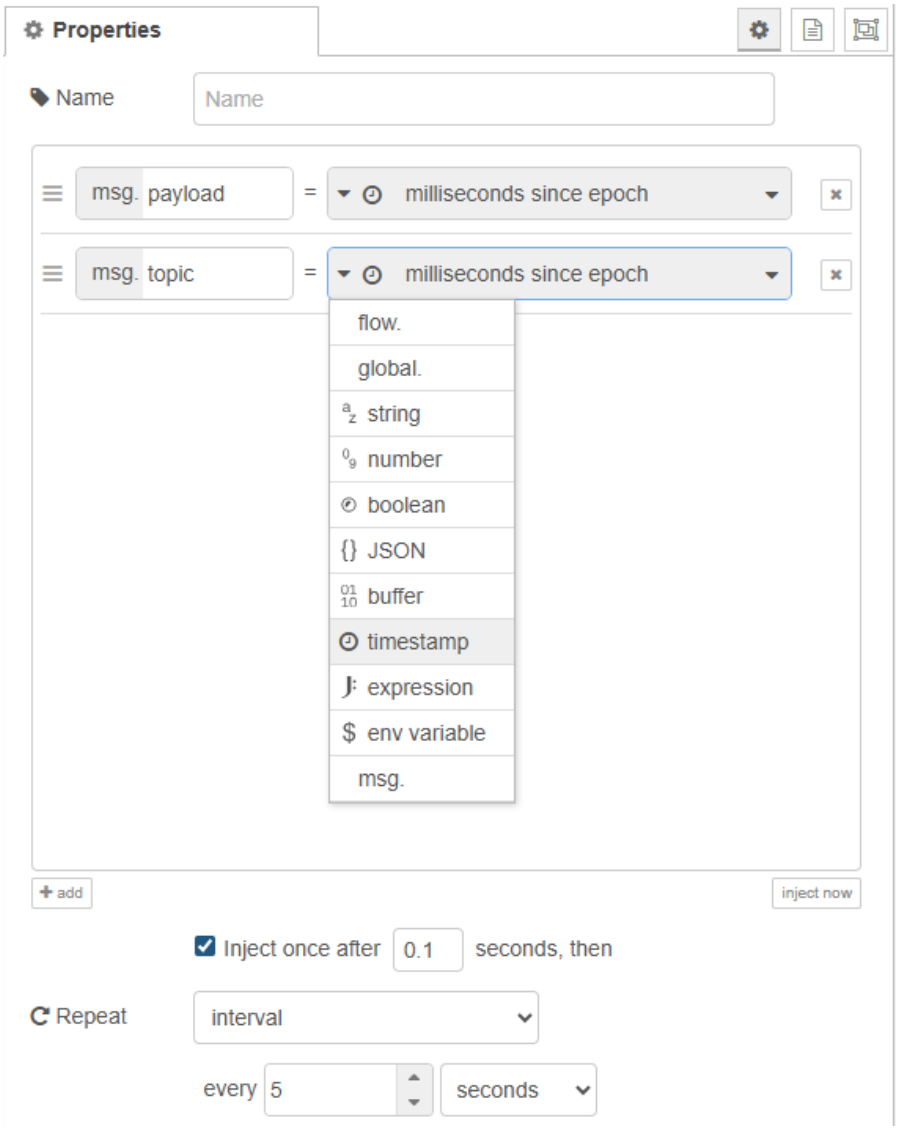


Рис. 9.21 Налаштування вузла *inject* для періодичного запуску опитування *MySQL*

Крок 3.2. Налаштування вузла function для інкрементального SQL-запиту

Додають вузол *function*, який формує *SQL*-запит на вибірку лише нових рядків. Логіка роботи така: зчитується *last_id* з *flow context*, формується *SELECT* із умовою *id* > *last_id*, а також задається сортування за *id* за зростанням.

Приклад коду вузла *function*:

```
let lastId = flow.get('last_id') || 0;
msg.topic = "SELECT id, t_stamp, temperature,
pressure, flow_rate FROM sensor_data WHERE id > " +
lastId + " ORDER BY id ASC";
return msg;
```

Рис. 9.22. Код вузла *function* для формування інкрементального *SQL*-запиту за полем *id*

Крок 3.3. Налаштування вузла MySQL

Додають вузол *MySQL* та задають параметри підключення. Як сервер вказують *localhost*, порт 3306, базу даних *SCADA_db* і облікові дані користувача *MySQL* (див.рис.9.23). Після збереження налаштувань вузол має виконувати запит, сформований попереднім вузлом *function*.

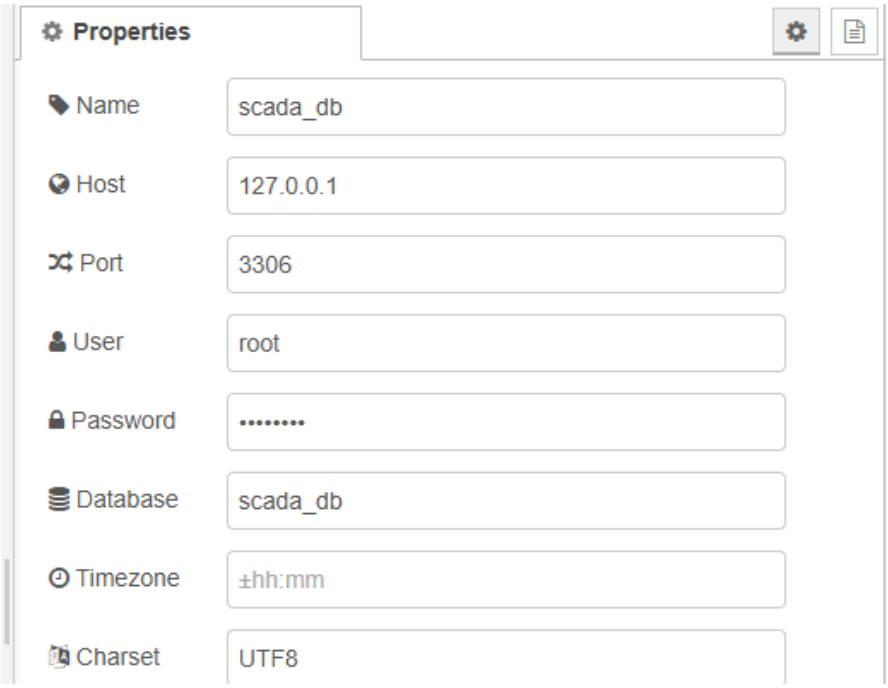


Рис. 9.23. Налаштування вузла *MySQL* у *Node-RED* для підключення до *SCADA_db*

Крок 3.4. Підготовка MQTT-повідомлення у форматі JSON і оновлення last_id

Додають другий вузол *function*, який виконує дві дії. По-перше, він відбирає отримані рядки та формує *MQTT*-повідомлення у форматі *JSON*. По-друге, він оновлює *last_id у flow context* так, щоб наступний цикл обробляв лише нові записи.

Оскільки *MQTT Engine* у цьому прикладі налаштовано на *Custom namespaces* типу *JSON*, повідомлення формують саме як *JSON*. Рекомендовано надсилати один запис як одне повідомлення. Якщо *MySQL* повернув кілька рядків, вузол формує кілька вихідних повідомлень (див.рис.9.24).

Приклад коду вузла *function*:

```
if (!Array.isArray(msg.payload) || msg.payload.length
=== 0) {
return null;
}
let out = [];
for (let i = 0; i < msg.payload.length; i++) {
let row = msg.payload[i];
flow.set('last_id', row.id);
out.push({
topic: "SCADA/sensors/data",
payload: {
id: row.id,
t_stamp: row.t_stamp,
temperature: row.temperature,
pressure: row.pressure,
flow_rate: row.flow_rate
}
});
}
return [out];
```

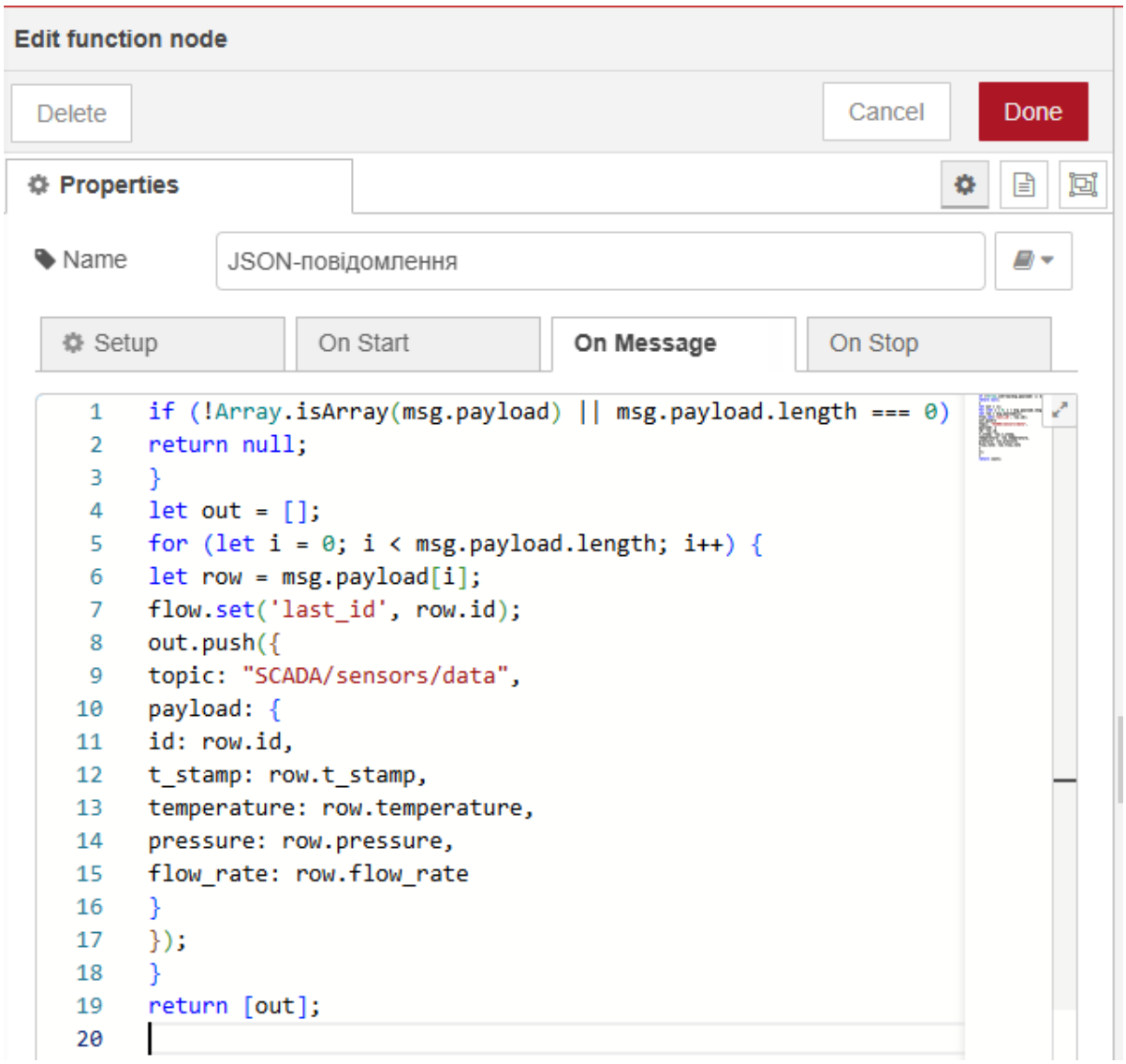


Рис. 9.24. Код вузла *function* для формування *JSON*-повідомлень і оновлення *last_id* у *flow context*.

Крок 3.5. Налаштування вузла mqtt out

Додають вузол *mqtt out* та налаштовують підключення до брокера *Mosquitto*. У полі *Server* задають localhost, порт 1883. *Topic* у цьому прикладі використовується з msg.topic, тому в параметрах вузла mqtt out залишають порожнє значення Topic або встановлюють опцію використання topic із повідомлення (див.рис.9.25). Після побудови потоку натискають Deploy.

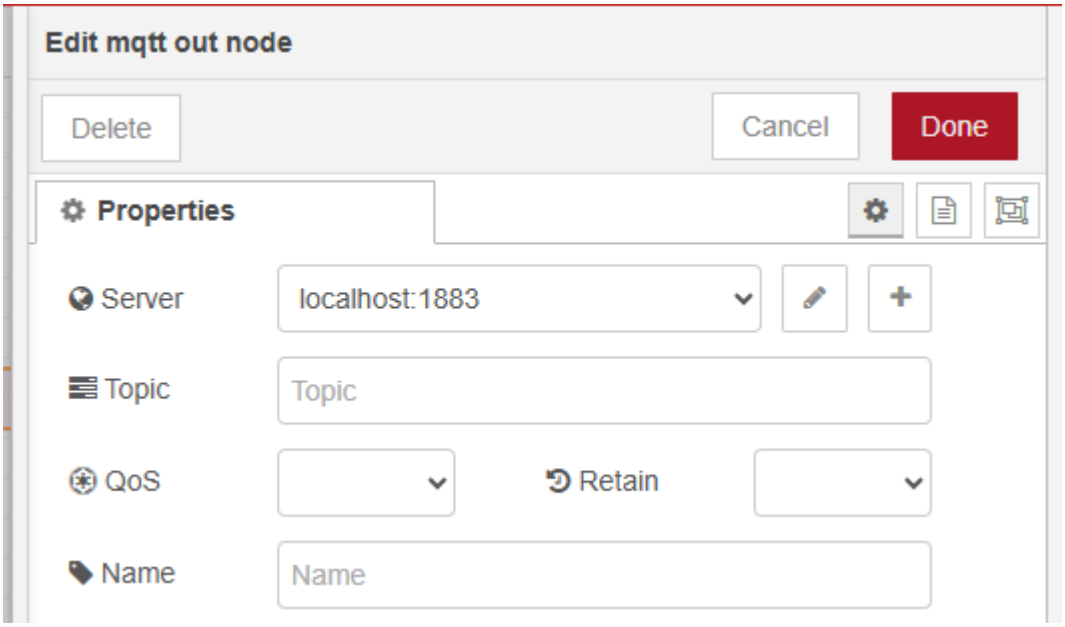


Рис. 9.25. Налаштування вузла mqtt out для публікації повідомлень на *Mosquitto* (*localhost*:1883)

Крок 4. Контроль публікації повідомлень у Node-RED

Для перевірки працездатності сформованого потоку в середовищі *Node-RED* використовують вузол *Debug*. Після додавання вузла до схеми його під'єднують до виходу вузла, який формує *MQTT*-повідомлення (див.рис.9.26).

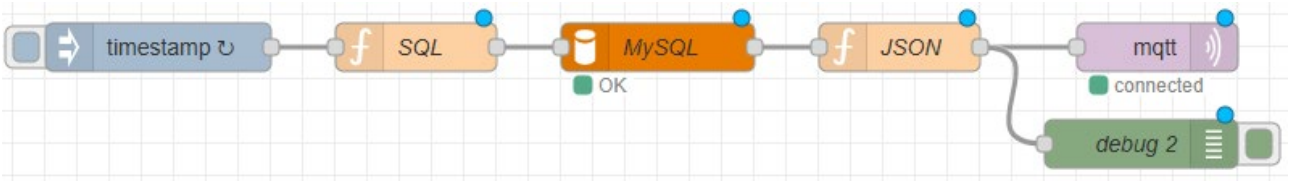


Рис. 9.26. Загальний вигляд потоку *Node-RED* із вузлом *Debug*

Далі виконують налаштування вузла *Debug*. Для цього двічі клацають на відповідному вузлі та в параметрі *Output* обирають режим *msg.payload*. За такого налаштування у вікні налагодження відображаються безпосередньо дані повідомлення без службових полів об'єкта msg.

Після завершення конфігурування потоку застосовують зміни натисканням кнопки *Deploy* у правій верхній частині інтерфейсу *Node-RED*.

Потік переходить у робочий режим. Для перегляду результатів у правій бічній панелі відкривають вкладку *Debug*. У цій області мають з'являтися повідомлення у форматі *JSON*, що містять поля *id, t_stamp, temperature, pressure* та *flow_rate*. Кожне повідомлення відповідає одному новому запису, зчитаному з таблиці *MySQL*.

У разі додавання нових рядків до таблиці *MySQL* у вікні *Debug* повинні відображатися лише повідомлення, що відповідають цим новим записам. Повторна поява повідомлень для вже оброблених рядків свідчить про помилки в реалізації інкрементального читання, зокрема про некоректне збереження або оновлення значення останнього обробленого ідентифікатора або часової мітки у контексті *Node-RED*.

Крок 5. Перевірка появи тегів у SCADA Ignition

На рівні *Ignition Gateway* спочатку перевіряють стан модуля *MQTT Engine* та з'єднання з брокером. Для цього у вебінтерфейсі *Ignition Gateway* переходять до розділу *Config → MQTT Engine → Servers* і переконуються, що відповідне підключення до брокера *Mosquitto* має стан *Connected* (див.рис.9.27). Стан *Connected* означає, що модуль *MQTT Engine* успішно встановив з'єднання з брокером і готовий приймати повідомлення.

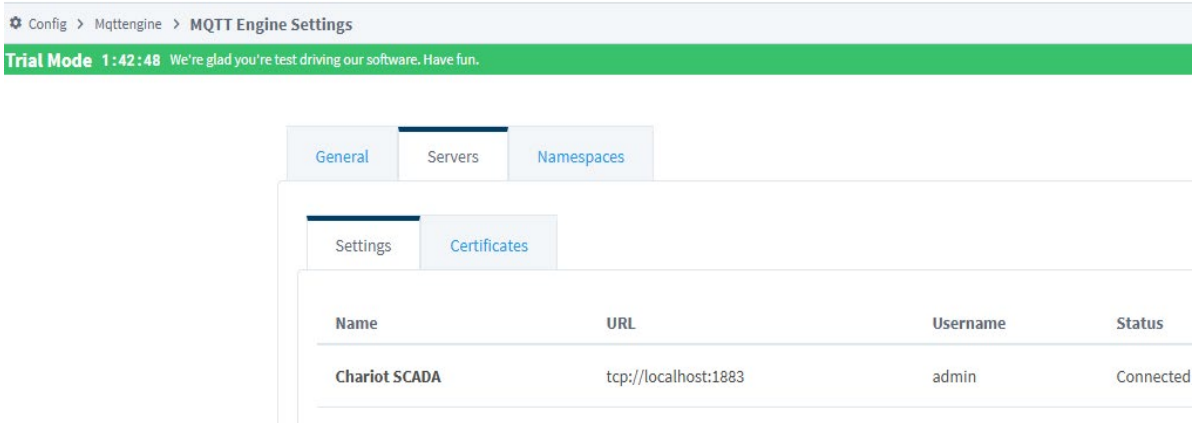


Рис. 9.27. Перевірка стану модуля *MQTT Engine*

Далі у вкладці *Namespaces* обирають режим *Custom* і задають підписку на *topic SCADA/sensors/data* з обробкою повідомлень у форматі *JSON*. Приклад відповідних налаштувань наведено на рис. 9.28.

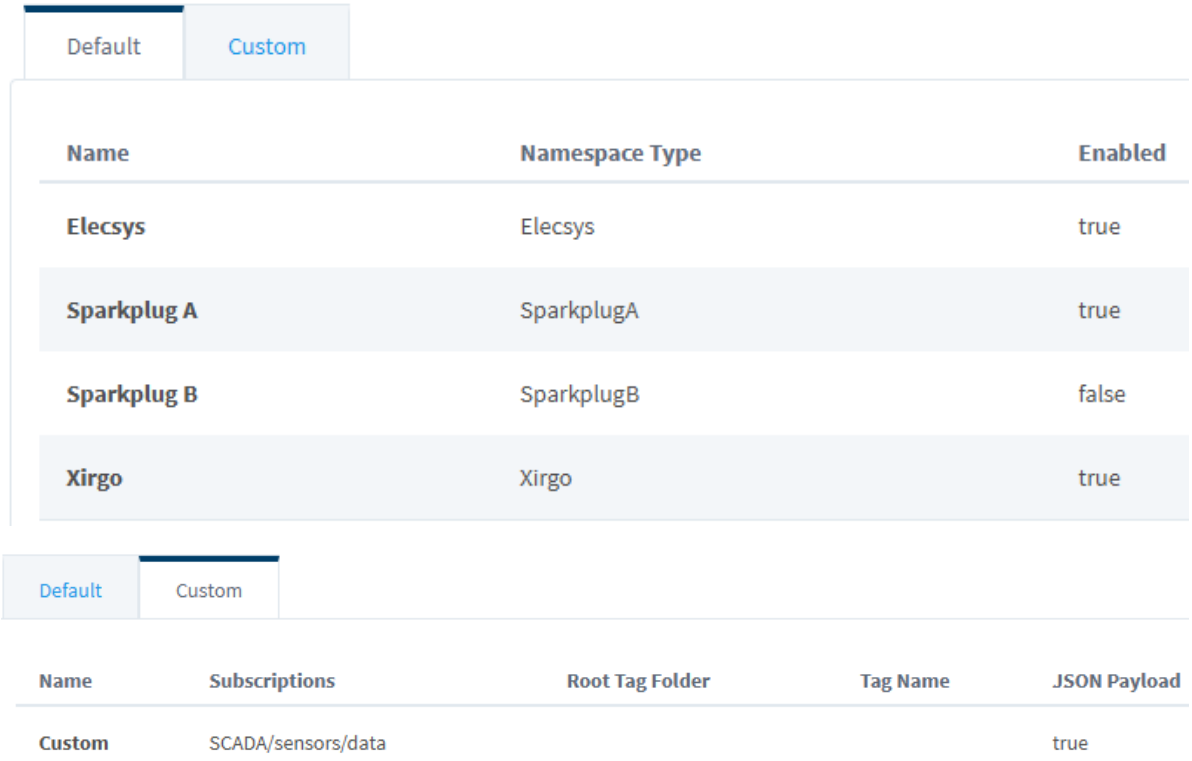


Рис. 9.28. Приклад налаштування модуля *MQTT Engine*

Після цього запускають середовище розробки *Ignition Designer*. У вікні запуску обирають потрібний *Gateway* та відкривають проєкт. У робочому середовищі *Ignition Designer* переходять до панелі *Tag Browser*, яка зазвичай розташована зліва. У дереві тегів розкривають гілку *MQTT Engine*, а далі – *Custom*.

У гілці *Custom* має з'явитися ієрархія тегів, що відповідає структурі *MQTT-topic*. Для topic виду *SCADA/sensors/data* типовим є шлях *Custom* → *SCADA* → *sensors* → *data*. Кожне поле *JSON*-повідомлення, яке публікує *Node-RED*, автоматично відображається у вигляді дочірнього тегу, наприклад *id, t_stamp, temperature, pressure ma flow_rate*. Значення цих тегів оновлюються в режимі реального часу під час надходження нових *MQTT*-повідомлень.

На цьому етапі доцільно переконатися, що теги мають коректний тип даних і змінюють свої значення синхронно з додаванням нових записів у таблицю *MySQL*. Це підтверджує правильну роботу зв'язки *Node-RED → MQTT broker → MQTT Engine → Tag Browser*.

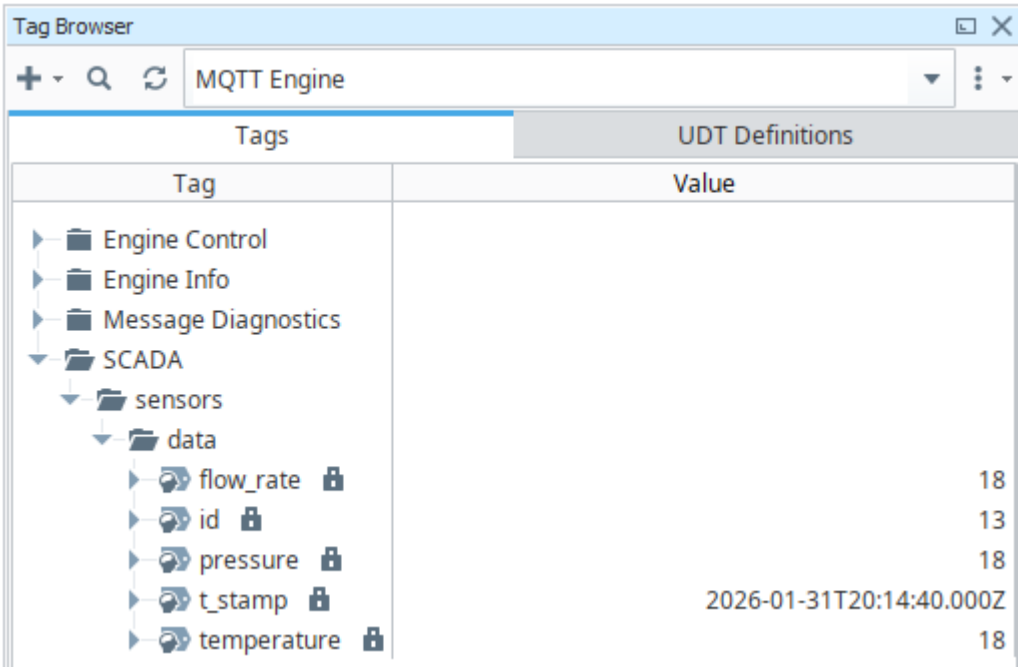


Рис. 9.29. Відображення тегів у *Ignition Designer*

Крок 6. Перевірка інкрементальності передачі

Для перевірки принципу «лише нові записи» у *MySQL* додають новий рядок у таблицю *sensor_data* та спостерігають, що *Node-RED* публікує лише цей новий рядок, а в *Ignition* оновлюються відповідні теги (дивю рис. 9.30).

Result Grid | Filter Rows: | Edit:

id	t_stamp	temperature	pressure	flow_rate
1	2026-01-31 21:38:18	23.5	1.02	10.1
2	2026-01-31 21:38:18	24	1.01	10.3
3	2026-01-31 21:38:18	22.8	1.05	9.8
4	2026-01-31 21:39:43	22	1.1	10
5	2026-01-31 21:52:06	20	1	10
12	2026-01-31 21:58:08	22	11	12
13	2026-01-31 22:14:40	18	18	18
14	2026-01-31 22:20:05	22.22	1.11	10.1
15	2026-01-31 22:22:11	23	1	11

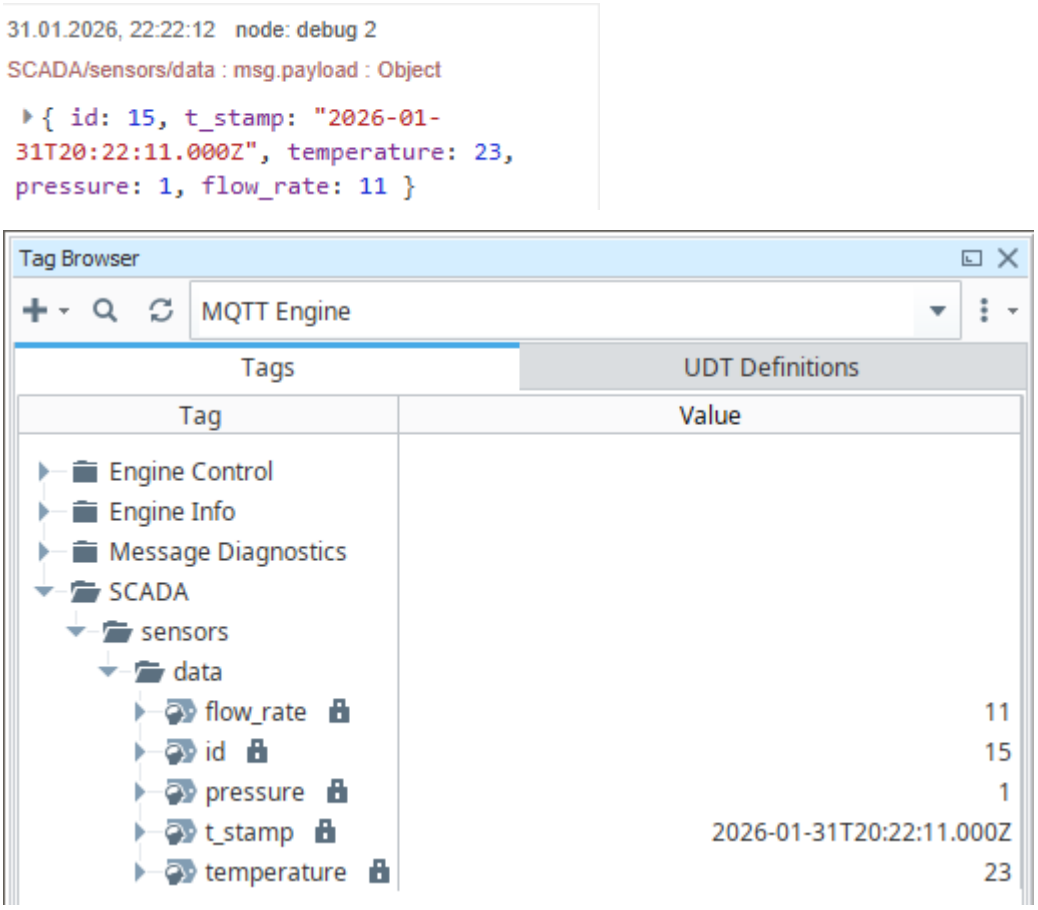


Рис. 9.30. Додавання нового запису в *MySQL* і підтвердження інкрементальної передачі за *id*.

Примітка. Якщо Node-RED перезапускається, значення *last_id* у *flow context* може бути скинуте, якщо не налаштовано збереження контексту на диск. Для навчального прикладу це допустимо. У разі потреби стабільної роботи між перезапусками застосовують *file-based context Node-RED* або зберігають останній опрацьований ідентифікатор у допоміжній таблиці *MySQL*.

9.10. Інтеграції *MySQL* з *Apache Kafka* та *Debezium*

У традиційних *SCADA*-архітектурах обмін даними між компонентами часто будується за опитувальним принципом. *SCADA*-система або допоміжні сервіси з певною періодичністю виконують запити до бази даних, зчитуючи нові або змінені записи. Такий підхід є простим у реалізації, проте має низку обмежень: надмірне навантаження на СУБД, затримки між фактичною подією та її відображенням, а також складність масштабування.

Механізм *Change Data Capture* (*CDC*) дозволяє перейти до подієвої моделі, у якій кожна зміна в базі даних перетворюється на подію та передається далі незалежним транспортним рівнем. У розглянутій архітектурі цю роль виконує *Apache Kafka*, а *Debezium* забезпечує зчитування змін із журналу бінарних логів *MySQL* і перетворення їх у структуровані повідомлення, як це показано на рис.9.31.

Такий підхід добре узгоджується з вимогами промислових систем автоматизації, де важливо не лише отримати поточне значення параметра, а й мати доступ до історії змін, можливість повторного відтворення подій і незалежність між джерелами та споживачами даних.

Рис. 9.31. Загальна схема подієвої інтеграції *MySQL* з *Kafka* через *Debezium*

9.10.1. Загальні положення та принципи подієвої інтеграції

У межах *CDC* зміни в таблицях *MySQL* фіксуються не шляхом періодичних *SELECT*-запитів, а через бінарний журнал (*binlog*), який є

внутрішнім механізмом транзакційної СУБД. *Debezium* працює як конектор (*connector*). *Kafka Connect*, підключається до *MySQL* як реплікаційний клієнт і зчитує події *binlog* у форматі, придатному для трансляції. *Apache Kafka* виступає як розподілений журнал подій, у якому кожна зміна має позицію читання (*offset*) і може бути повторно прочитана споживачами (див. рис.3.32).

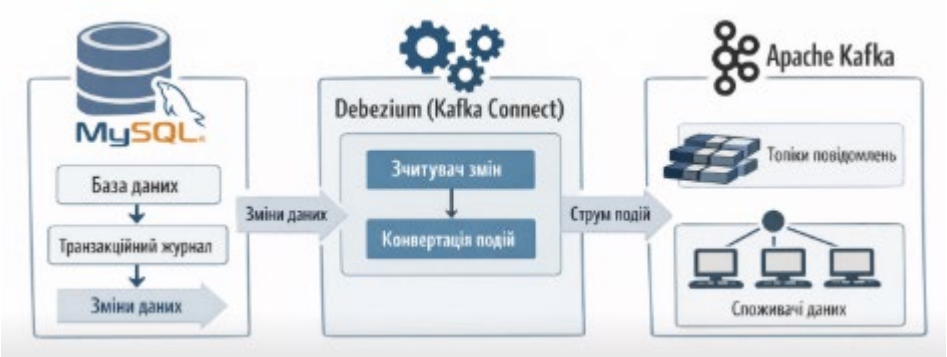


Рис. 9.32. Ролі *MySQL*, *Debezium* та *Apache Kafka* в архітектурі *CDC*

У системах автоматизації це дає змогу передавати лише фактичні зміни технологічних параметрів без періодичного опитування бази даних. Такий підхід доцільний для ведення архівів змін, аналізу й реконструкції аварійних режимів, побудови цифрових двійників, а також для підключення кількох незалежних споживачів даних без додаткового навантаження на *MySQL*.

9.10.2. Підключення бази даних MySQL до Apache Kafka за допомогою Debezium

Для виконання наведеного прикладу необхідно встановити *Docker Desktop* у середовищі *Windows*. У версіях *Windows* 10/11 *Pro*, *Enterprise* або *Education Docker Desktop* зазвичай працює без додаткових налаштувань. У разі використання *Windows Home* обов’язковою є наявність підсистеми *WSL* 2. Також у налаштуваннях *BIOS/UEFI* має бути активована апаратна віртуалізація процесора (*Intel VT-x* або *AMD-V*).

Після інсталяції *Docker Desktop* виконують первинну перевірку в *PowerShell* або *Command Prompt*. Команду запускають у звичайному режимі користувача.

У разі успішного виконання команди буде виведено таке повідомлення

```
Hello from Docker!
This message shows that your installation appears to
be working correctly.
```

Для роботи з *Apache Kafka* та *Debezium* створюють окрему робочу директорію, наприклад *D:\Kafka-Docker*. У цій директорії зберігаються всі файли, пов’язані з проєктом інтеграції, зокрема конфігурація контейнерів і допоміжні матеріали. Саме з цієї директорії виконуються команди керування *Docker Compose*.

У створеній директорії формують файл *docker-compose.yml*, у якому описують конфігурацію трьох служб: *Zookeeper*, *Kafka* та *Kafka Connect* із *Debezium*. У навчальному прикладі використовується один брокер *Kafka* і один екземпляр *Kafka Connect*, що є достатнім для демонстрації механізму *Change Data Capture*. Мережеві порти контейнерів виводяться на хостову систему *Windows*, що дає змогу керувати *Kafka Connect* через *REST API* та спостерігати за роботою сервісів у процесі виконання прикладу.

Рекомендований вміст *docker-compose.yml* наведено нижче.

```
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:7.6.0
    container_name: zookeeper
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
    ports:
      - "2181:2181"

  kafka:
    image: confluentinc/cp-kafka:7.6.0
    container_name: kafka
    depends_on:
      - zookeeper
```

```
        ports:
          - "9092:9092"
        environment:
          KAFKA_BROKER_ID: 1
          KAFKA_ZOOKEEPER_CONNECT: "zookeeper:2181"
          KAFKA_LISTENERS: "PLAINTEXT://0.0.0.0:9092"
          KAFKA_ADVERTISED_LISTENERS:
"PLAINTEXT://kafka:9092"
          KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1

      connect:
        image: debezium/connect:2.6
        container_name: connect
        depends_on:
          - kafka
        ports:
          - "8083:8083"
        environment:
          BOOTSTRAP_SERVERS: "kafka:9092"
          GROUP_ID: "1"
          CONFIG_STORAGE_TOPIC: "connect_configs"
          OFFSET_STORAGE_TOPIC: "connect_offsets"
          STATUS_STORAGE_TOPIC: "connect_statuses"
          KEY_CONVERTER:
"org.apache.kafka.connect.json.JsonConverter"
          VALUE_CONVERTER:
"org.apache.kafka.connect.json.JsonConverter"
          CONNECT_KEY_CONVERTER_SCHEMAS_ENABLE: "false"
          CONNECT_VALUE_CONVERTER_SCHEMAS_ENABLE: "false"
          CONNECT_REST_ADVERTISED_HOST_NAME: "connect"
```

Після збереження файлу *docker-compose.yml* у командному рядку переходять до директорії проєкту та запускають контейнери у фоновому режимі.

```
cd /d D:\Kafka-Docker
docker compose up -d
```

```
C:\Users\ikl>cd /d D:\Kafka-Docker

D:\kafka-docker>docker compose up -d
[+] Running 3/3
 ⠿ Container zookeeper  Started
 ⠿ Container kafka      Started
 ⠿ Container connect    Started
```

Примітка: зупинити та видалити запущені контейнери можна командою

```
Docker-compose down
```

Далі перевіряємо, чи працюють контейнери командою *Docker ps*

```
D:\Kafka-Docker>Docker ps
```

```
D:\kafka-docker>docker ps
CONTAINER ID   IMAGE                              COMMAND                  CREATED
067ab99e5e1c   debezium/connect:2.6               "/docker-entrypoint.…"   23 seconds ago
f234322a39e4   confluentinc/cp-kafka:7.6.0        "/etc/confluent/dock…"   23 seconds ago
fa7aaa2127c7   confluentinc/cp-zookeeper:7.6.0    "/etc/confluent/dock…"   24 seconds ago
```

```
STATUS          PORTS                                              NAMES
Up 22 seconds   8778/tcp, 0.0.0.0:8083->8083/tcp, 9092/tcp         connect
Up 22 seconds   0.0.0.0:9092->9092/tcp                             kafka
Up 23 seconds   2888/tcp, 0.0.0.0:2181->2181/tcp, 3888/tcp         zookeeper
```

У списку мають бути *zookeeper*, *kafka*, *connect* зі статусом *Up*.

Debezium зчитує події не з таблиць безпосередньо, а з *binlog MySQL*. Тому в *MySQL* має бути активовано журналювання змін і встановлено формат *ROW*. Для Windows інсталяції *MySQL Server* типово використовують конфігураційний файл *my.ini*, який часто розміщується за шляхом *C:\ProgramData\MySQL\MySQLServer8.0\my.ini*. Конкретний шлях може відрізнятися залежно від інсталятора.

У секції [*mysqld*] додають або розкоментовують такі параметри.

```
[mysqld]
server-id=1
log_bin=mysql-bin
binlog_format=ROW
binlog_row_image=FULL
```

Після внесення змін перезапускають службу *MySQL* у *Windows*. Назва служби може відрізнятися, тому її коректну назву доцільно перевірити в services.msc або командою *sc query*. Для типового випадку *MySQL*80 використовують:

```
net stop MySQL80
net start MySQL80
```

Після перезапуску перевіряють активність binlog у *MySQL* CLI або в *MySQL* Workbench.

```
SHOW VARIABLES LIKE 'log_bin';
SHOW VARIABLES LIKE 'binlog_format';
```

У результаті `log_bin` має бути `ON`, а `binlog_format` має бути `ROW`.

Далі створюють користувача, який використовуватиме Debezium для реплікаційного доступу. У навчальному прикладі дозволяється простий пароль, але в реальних системах використовують політику складності паролів і обмеження доступу за *IP*.

```
CREATE USER 'debezium'@'%' IDENTIFIED BY 'dbz';
GRANT SELECT, RELOAD, SHOW DATABASES, REPLICATION
SLAVE, REPLICATION CLIENT
ON *.* TO 'debezium'@'%';
FLUSH PRIVILEGES;
```

Для демонстрації роботи механізму *Change Data Capture* створюють окрему базу даних, що імітує накопичення технологічних вимірювань у системі автоматизації. Як джерело подій використовується таблиця, до якої в режимі реального часу записуються значення основних параметрів процесу.

```
CREATE DATABASE IF NOT EXISTS scada_process;
USE scada_process;

CREATE TABLE IF NOT EXISTS process_measurements (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  t_stamp DATETIME NOT NULL DEFAULT
CURRENT_TIMESTAMP,
  temperature FLOAT NOT NULL,
  pressure FLOAT NOT NULL,
  flow_rate FLOAT NOT NULL,
  valve_position FLOAT NOT NULL
);
```

Таблиця `process_measurements` моделює типовий журнал вимірювань у *SCADA*-системі. Поле `id` виконує роль технічного ідентифікатора запису та забезпечує однозначне впорядкування подій, що є важливим для

механізмів *CDC*. Поле `t_stamp` зберігає часову мітку формування вимірювання. Поля `temperature`, `pressure`, `flow_rate` та `valve_position` відповідають основним аналоговим параметрам технологічного процесу та можуть безпосередньо відображатися у вигляді тегів у *SCADA*.

Для первинної перевірки роботи *CDC* у таблицю додають кілька записів вручну.

```
INSERT INTO process_measurements
(temperature, pressure, flow_rate, valve_position)
VALUES
(180.5, 1.25, 12.8, 45.0),
(181.2, 1.27, 13.1, 47.5),
(182.0, 1.30, 13.4, 50.0);
```

Kafka Connect надає *REST API* для керування конекторами. У цьому прикладі *Debezium Connect* уже запущений як контейнер *connect* і слухає порт 8083 на хості *Windows*.

Спочатку перевіряють доступність Kafka Connect.

```
curl http://localhost:8083/
```

У разі успішного з’єднання повертається службова інформація про версію Kafka Connect, що підтверджує його працездатність.

```
D:\kafka-docker>curl http://localhost:8083/
{"version":"3.7.0","commit":"2ae524ed625438c5","kafka
_cluster_id":"rsboJvBUS7S8SY8HzYHImg"}
```

Далі готують конфігурацію *Debezium*-конектора у вигляді *JSON*-файлу. У робочій директорії створюють файл `mysql-scada-connector.json`.

Важливо. Для підключення з контейнера до *MySQL*, що працює на хості Windows, використовують *host.docker.internal*.

```
{
```

```
      "name": "mysql-scada-connector",
      "config": {
        "connector.class":
"io.debezium.connector.mysql.MySqlConnector",
        "tasks.max": "1",
        "database.hostname": "host.docker.internal",
        "database.port": "3306",
        "database.user": "debezium",
        "database.password": "dbz",
        "database.server.id": "184055",
        "database.server.name": "mysql_server",
        "topic.prefix": "mysql_server",
        "database.include.list": "scada_process",
        "database.connectionTimeZone": "UTC",

"schema.history.internal.kafka.bootstrap.servers":
"kafka:9092",
        "schema.history.internal.kafka.topic":   "schema-
changes.scada_process",
        "include.schema.changes": "true",
        "snapshot.mode": "initial"
      }
    }
```

Реєстрацію конектора виконують через *POST*-запит до *Kafka Connect*:

```
    cd /d D:\Kafka-Docker
    curl  -i  -X  POST  -H  "Accept:application/json"  -H
"Content-Type:application/json" ^
    http://localhost:8083/connectors   -d   @mysql-scada-
connector.json
```

У разі успіху сервер повертає код `201 Created`. Після цього перевіряють список конекторів і статус.

```
    curl http://localhost:8083/connectors
    curl http://localhost:8083/connectors/mysql-scada-
connector/status
```

Запит повинен повернути статус *RUNNING*

```
    {"name":"mysql-
connector","connector":{"state":"RUNNING","worker_id":"17
```

2.18.0.4:8083"},"tasks":[{"id":0,"state":"RUNNING","worke
r_id":"172.18.0.4:8083"}],"type":"source"}

За потреби зупинити та видалити конектор використовують команду:

```
curl -X DELETE http://localhost:8083/connectors/mysql-
scada-connector
```

Після успішної реєстрації *Debezium*-конектор автоматично починає публікувати події змін даних у теми *Apache Kafka*. Імена тем формуються за правилом:

```
topic.prefix + назва бази даних + назва таблиці
```

Для розглянутого прикладу з базою даних *scada_process* та таблицею *process_measurements* очікуваною є тема:

```
mysql_server.scada_process.process_measurements
```

Після успішної реєстрації *Debezium*-конектора необхідно переконатися, що події змін у *MySQL* дійсно надходять до *Apache Kafka*. Для цього виконують перевірку на рівні брокера *Kafka* за допомогою стандартних консольних утиліт.

Спочатку підключаються до контейнера з брокером *Kafka*. У командному рядку *Windows* виконують команду:

```
docker exec -it kafka bash
```

Після виконання цієї команди відкривається командний інтерпретатор усередині контейнера *Kafka*. Саме в цьому середовищі доступні службові утиліти *Kafka*.

```
D:\kafka-docker>docker exec -it kafka bash
[appuser@fe6598ae610f ~]$
```

Далі отримують перелік усіх тем, які наразі існують у брокері. Для цього виконують команду:

```
    kafka-topics --bootstrap-server kafka:9092 --list

    C:\Users\ikl>docker exec -it kafka bash
    [appuser@fe6598ae610f ~]$ kafka-topics --bootstrap-
server kafka:9092 --list

    __consumer_offsets
    connect_configs
    connect_offsets
    connect_statuses
    mysql_server
    mysql_server.scada_process.process_measurements
    schema-changes.automation_db
    schema-changes.scada_process
    [appuser@fe6598ae610f ~]$
```

Наявність нашої теми підтверджує, що *Debezium*-конектор зареєстрований коректно і *Kafka* готова приймати події змін.

Після цього запускають консольного споживача *Kafka*, який дозволяє безпосередньо переглядати повідомлення з обраної теми. У контейнері *Kafka* виконують команду:

```
  kafka-console-consumer --bootstrap-server kafka:9092 \
    --topic
mysql_server.scada_process.process_measurements --from-
beginning
```

Параметр `--from-beginning` означає, що споживач зчитує всі наявні події з початку журналу теми. Після виконання цієї команди термінал переходить у режим очікування повідомлень.

На цьому етапі нічого вводити більше не потрібно, вікно споживача залишається відкритим і очікує появи нових подій.

Далі, у окремому вікні (*MySQL CLI* або *MySQL Workbench*) виконують операцію вставки нового запису в таблицю `process_measurements`:

```
  INSERT INTO scada_process.process_measurements
  (temperature, pressure, flow_rate, valve_position)
  VALUES (185.4, 1.18, 12.3, 37.5);
```

Після виконання цієї *SQL*-команди повертатися до *Kafka* не потрібно. Якщо *Debezium*-конектор працює коректно, у вікні *kafka-console-consumer* автоматично з'являється нове повідомлення у форматі *JSON*.


```
{"before":null,"after":{"id":9,"t_stamp":1769977357000,"temperature":185.4,"pressure":1.18,"flow_rate":12.3,"valve_position":37.5},"source":{"version":"2.6.2.Final","connector":"mysql","name":"mysql_server","ts_ms":1769970157439,"snapshot":"false","db":"scada_process","sequence":null,"ts_us":1769970157439323,"ts_ns":1769970157439323000,"table":"process_measurements","server_id":1,"gtid":"92091 2d4-64be-11f0-b78c-18c04d096464:87","file":"mysql-bin.000052","pos":11245,"row":0,"thread":20,"query":null},"op":"c","ts_ms":1769970157441,"ts_us":1769970157441577,"ts_ns":1769970157441577381,"transaction":null}
```


Повідомлення містить:

секцію `after` з новими значеннями полів таблиці;

службову інформацію про джерело події;

поле `op`, яке вказує тип операції (*c* - вставка, *u* - оновлення, *d* - видалення).

Поява такого повідомлення означає, що:

MySQL зафіксував зміну у *binlog*;

Debezium зчитав зміну з журналу транзакцій;

подія була передана до *Apache Kafka*;

брокер *Kafka* доставив її споживачу.

Отримане повідомлення у вікні консольного споживача підтверджує коректну роботу всього ланцюга *Change Data Capture* на рівні бази даних і потокової платформи. На цьому етапі *Apache Kafka* фактично виконує роль центрального журналу подій, у якому кожна зміна таблиці *MySQL* зафіксована у впорядкованому вигляді та доступна для подальшого споживання. Використання *kafka-console-consumer* дозволяє наочно переконатися у наявності подій, однак цей інструмент має суто діагностичне призначення і не

застосовується у промислових системах автоматизації для безпосередньої інтеграції з *SCADA*.

9.10.3. Передавання подій з Apache Kafka до MQTT через Node-RED

Наступним кроком є передача зафіксованих подій змін у диспетчерський рівень системи керування. Для цього необхідно організувати підключення *SCADA*-системи до потоку даних *Kafka* таким чином, щоб кожна подія вставки або оновлення запису в *MySQL* автоматично відображалася у вигляді актуальних значень технологічних параметрів. Оскільки *SCADA Ignition* не має вбудованих засобів прямої роботи з *Apache Kafka*, між потоковою платформою та *SCADA* вводиться проміжний рівень перетворення і транспортування даних. У навчальному прикладі роль такого проміжного рівня виконує *Node-RED*, а передавання даних у бік диспетчерського рівня реалізується через *MQTT*-брокер *Eclipse Mosquitto*.

Щоб *Node-RED*, який працює локально у *Windows*, міг підключатися до *Apache Kafka*, розгорнутої у *Docker*, необхідно забезпечити доступ до брокера *Kafka* через окремий зовнішній порт хостової системи. Якщо використати лише внутрішню адресу *kafka*:9092, вона буде доступною виключно всередині *docker*-мережі, і клієнт у *Windows* не зможе встановити коректне з’єднання. Тому в конфігурації контейнера *Kafka* задають два канали доступу. Перший призначений для взаємодії всередині *docker*-мережі через *kafka*:9092. Другий призначений для зовнішніх клієнтів з *Windows* через *localhost*:29092. Приклад налаштування сервісу *kafka* у файлі *docker-compose.yml* наведено нижче.

```
kafka:
    image: confluentinc/cp-kafka:7.6.0
    container_name: kafka
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
      - "29092:29092"
```

```
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: "zookeeper:2181"
      KAFKA_LISTENERS:
PLAINTEXT://0.0.0.0:9092,PLAINTEXT_HOST://0.0.0.0:29092
      KAFKA_ADVERTISED_LISTENERS:
PLAINTEXT://kafka:9092,PLAINTEXT_HOST://localhost:29092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
```

Після внесення змін у *docker-compose.yml* контейнери перезапускають, щоб нові порти та параметри *listeners* були застосовані. Далі перевіряють, що *Kafka* доступна з Windows саме через *localhost*:29092, оскільки цей порт використовується вузлом *kafkajs-client* у *Node-RED*.

Node-RED інстальовано локально в середовищі *Windows* і він запускається як окремий процес. Для споживання (*consumer*) подій з *Apache Kafka* у навчальному прикладі застосовується вузол *kafkajs-consumer* (*KafkaJS*), який забезпечує підключення до брокера *Kafka* через порт, доступний з хостової системи. Щоб додати необхідні вузли *KafkaJS* у *Node-RED*, у вебінтерфейсі редактора відкривають меню у правому верхньому куті та переходять до розділу *Manage Palette*. Далі відкривають вкладку *Install*, у полі пошуку вводять *node-red-contrib-kafkajs* і виконують інсталяцію пакета. Після завершення інсталяції у палітрі вузлів стають доступними компоненти *KafkaJS*, зокрема *kafkajs-client* та *kafkajs-consumer*, які використовуються для налаштування з’єднання з *Apache Kafka* та читання повідомлень (message) із заданої теми (topic). На рис. 9.33 показано приклад відображення встановленого пакета і доступних вузлів *KafkaJS* у *Node-RED*.

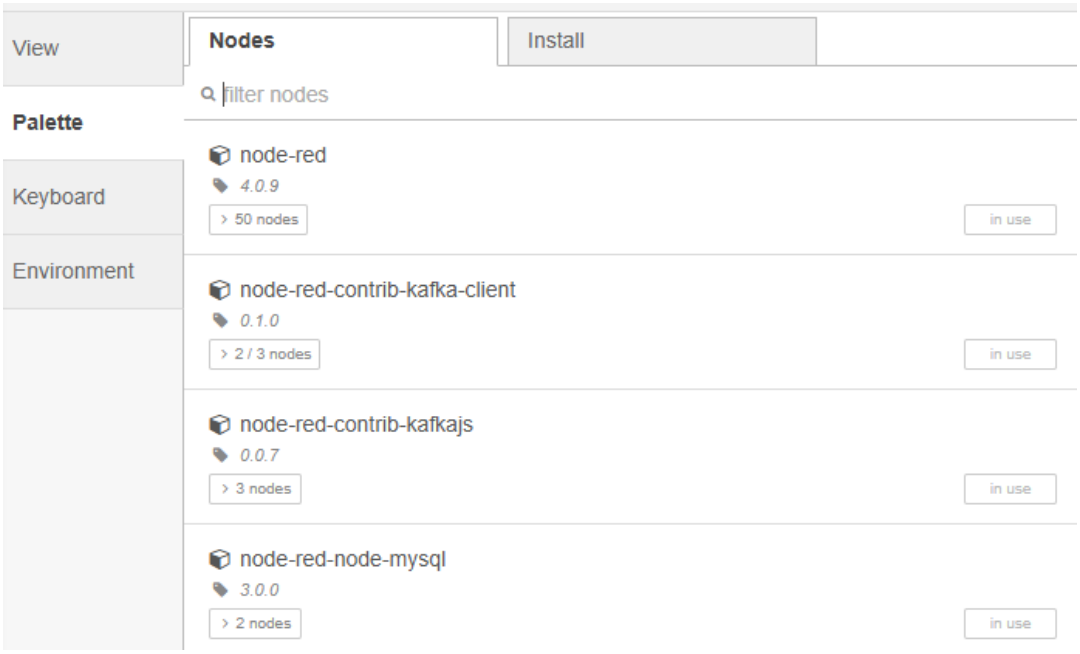


Рис. 9.33. Набір вузлів *KafkaJS* у палітрі *Node-RED* після інсталяції *node-red-contrib-kafkajs*

Після підготовки палітри створюють новий потік обробки подій. Для підключення до *Kafka* використовують вузол *kafkajs-consumer*, у властивостях якого спочатку створюють або вибирають *kafkajs-client*. У параметрах *kafkajs-client* задають адресу брокера (*broker*) та порт, які доступні з *Windows*. Якщо *Apache Kafka* розгорнуто в *Docker*, доцільно передбачити окремий зовнішній порт для клієнтів з хостової системи, наприклад *localhost*:29092, тоді як внутрішня взаємодія контейнерів може здійснюватися через *kafka*:9092. Далі у властивостях *kafkajs-consumer* задають ідентифікатор групи (*group id*) споживача (*consumer group*) та назву теми (*topic*), з якої потрібно читати події *Debezium*.

У поточному стенді тема має назву *mysql_server.scada_process.process_measurements*. Вікно налаштування *kafkajs-consumer* наведено на рис. 9.34. У вікні Advance Options потрібно увімкнути опцію *From Beginning*.

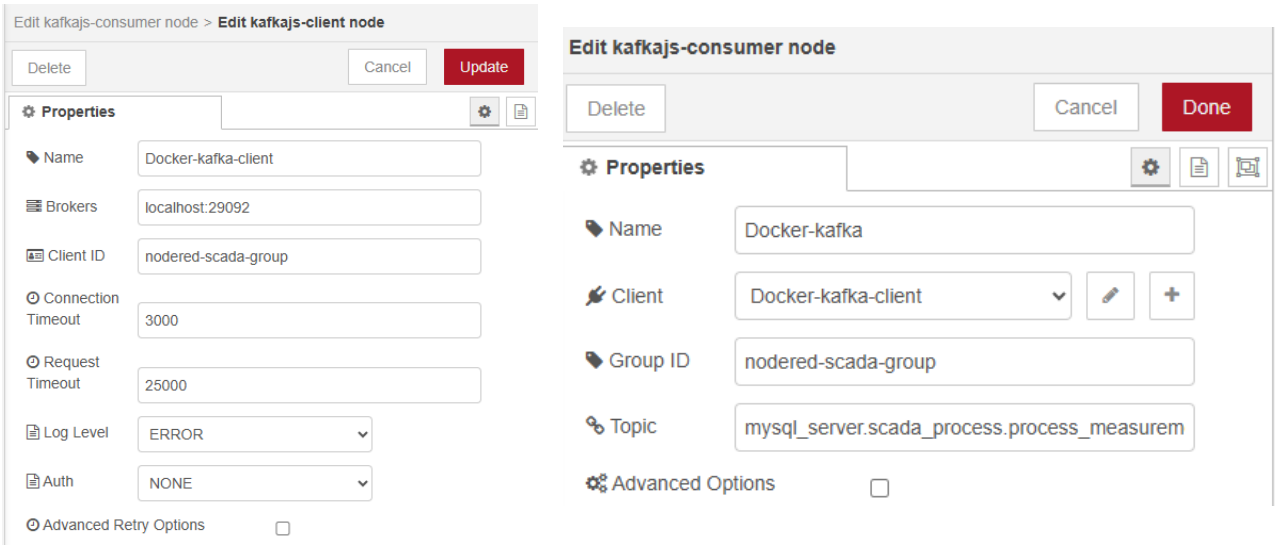


Рис. 9.34. Налаштування вузла *kafkajs-consumer* у *Node-RED* для читання теми *Debezium* з *Apache Kafka*

Після застосування налаштувань і виконання *Deploy* працездатність підключення контролюють через вузол *debug*, під'єднаний безпосередньо до виходу *kafkajs-consumer*. У вікні *Debug* відображається службова структура повідомлення *Kafka*, яка містить метадані доставлення, зокрема часову мітку (*timestamp*), зміщення (*offset*) та ключ (*key*). На цьому етапі важливо зафіксувати, що корисні дані *Debezium* містяться у полі *msg.payload.value*, тоді як інші поля належать транспортному рівню *Kafka*. Приклад такого виводу наведено на рис. 9.35.

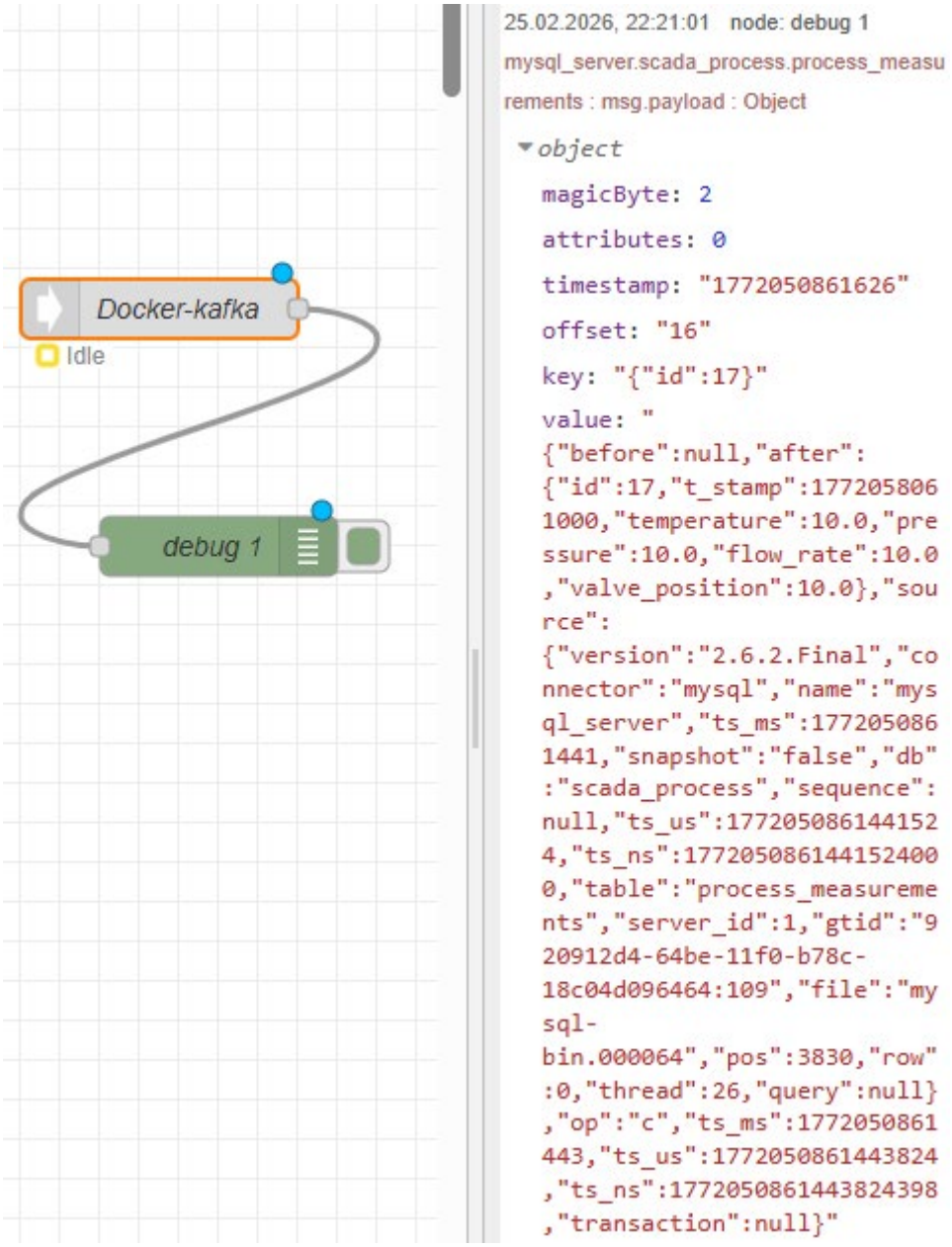


Рис. 9.35. Налаштування вузла *kafkajs-consumer* у *Node-RED* для читання теми *Debezium* з *Apache Kafka*

Оскільки поле *msg.payload.value* містить корисні дані повідомлення *Debezium*, перед прикладною обробкою його необхідно привести до вигляду *JavaScript*-об’єкта. У навчальному стенді це виконується стандартним вузлом *JSON*, який у потоці доцільно підписати як *Parsing Value*, щоб було зрозуміло його призначення. Вузол *JSON* додають із палітри *Node-RED* та розміщують безпосередньо після *kafkajs-consumer*. У налаштуваннях вузла задають обробку властивості *msg.payload.value* з перетворенням у *JavaScript Object*. Після цього

у наступному вузлі *function* стає можливим звернення до *msg.payload.value.after*, що містить актуальні значення полів таблиці, та до *msg.payload.value*, яке визначає тип операції. Загальний вигляд пайплайна обробки даних у *Node-RED* наведено на рис. 9.36.

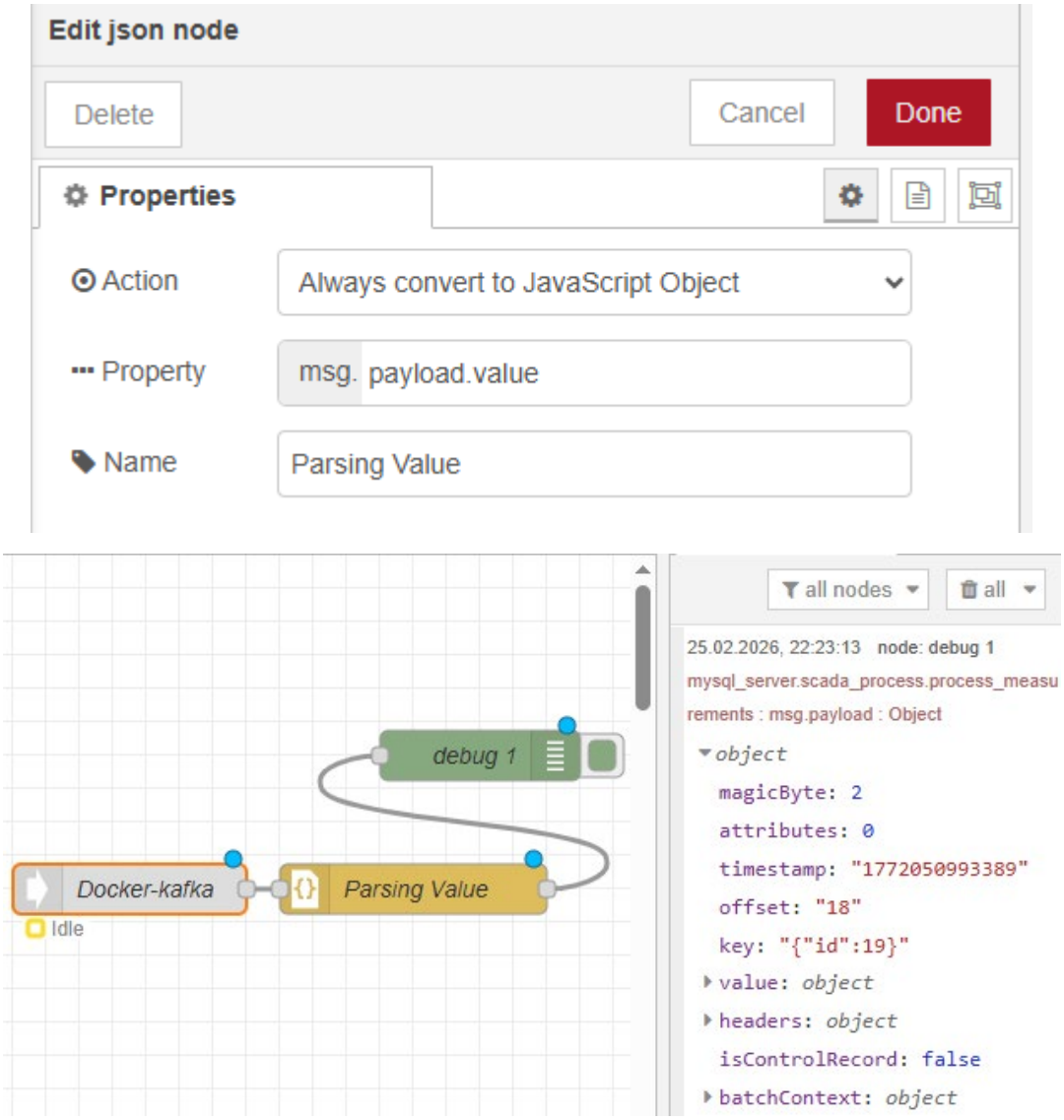


Рис. 9.36. Пайплайн *Node-RED* для обробки подій *Debezium*

Далі у вузлі *function* реалізують прикладну логіку перетворення повідомлення *Debezium* у формат, придатний для диспетчерського рівня. У межах прикладу обробляються операції вставки, оновлення та повідомлення початкового знімка, які визначаються значеннями *op c, u, r*. Актуальні дані беруться з *after*. У результаті формується *MQTT*-тема (*topic*)

SCADA/sensors/data та компактний *JSON* з параметрами процесу. Часова мітка *t_stamp* перетворюється у формат *ISO* 8601, оскільки це спрощує відображення у *SCADA* та подальше використання в історизації та трендах. Приклад логіки перетворення наведено нижче.

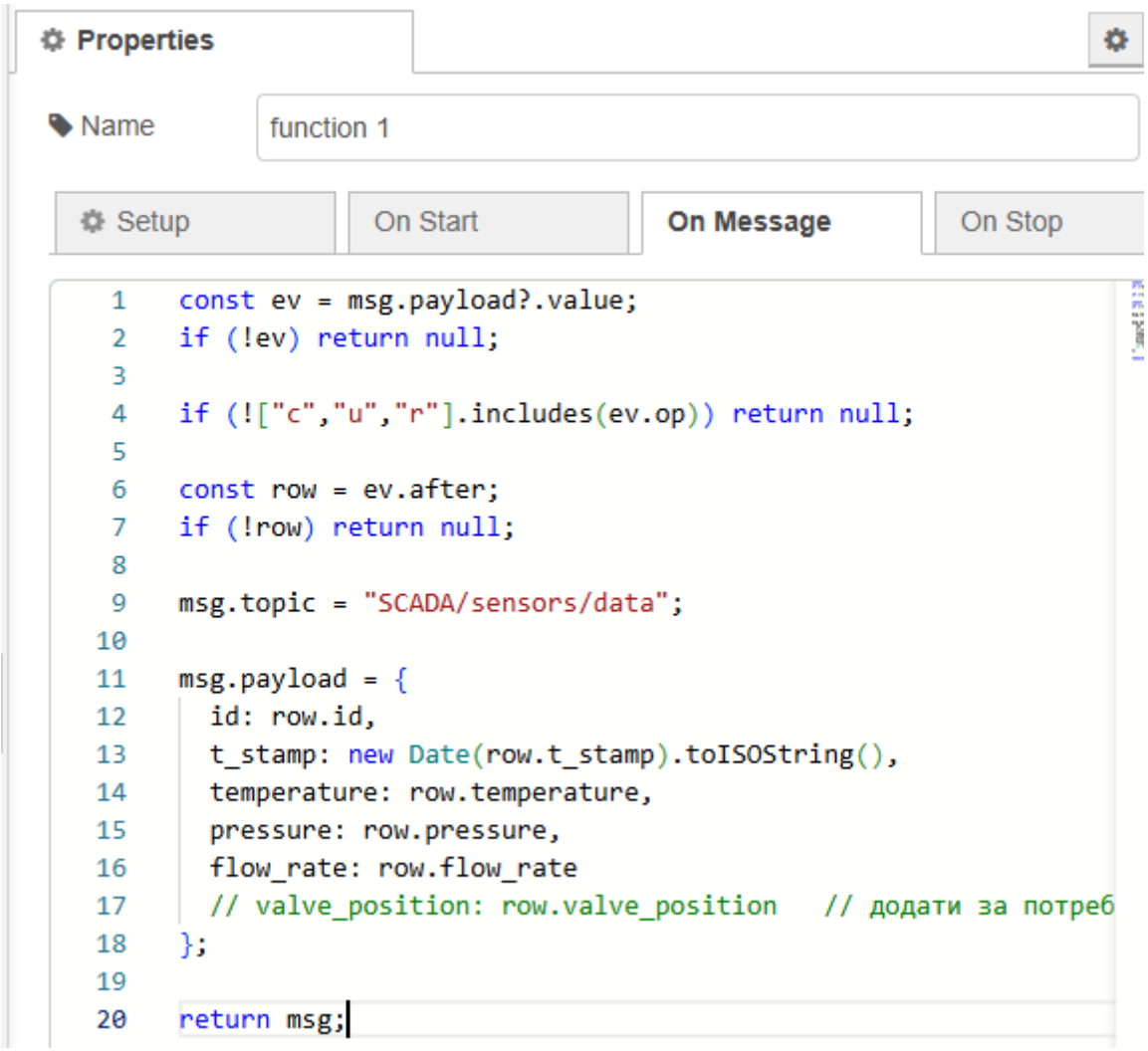
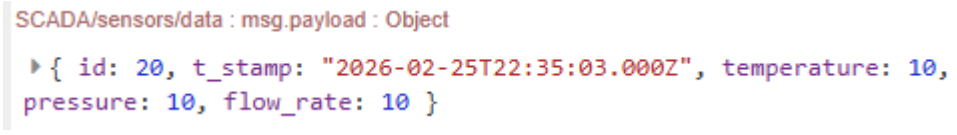


Рис. 9.37. Налаштування вузла *function*

Після вузла *function* доцільно залишити вузол *debug*, який відображає вже оброблене повідомлення. Приклад повідомлення після перетворення наведено нижче.



Публікацію сформованого повідомлення в *MQTT* виконують через вузол *mqtt out*. У налаштуваннях *mqtt out* створюють конфігурацію брокера *Mosquitto*, задаючи адресу *localhost* і порт 1883. Налаштування *TLS* не застосовують, якщо воно не увімкнене на стороні брокера. Тему в *mqtt out* можна не задавати явно, оскільки вона передається через *msg.topic*, сформований у вузлі *function*. Після *Deploy* вузол *mqtt out* має перейти в стан *connected*, що підтверджує встановлене з'єднання з *Mosquitto*. Приклад створення конфігурації *Mosquitto-local* наведено на рис. 9.38.

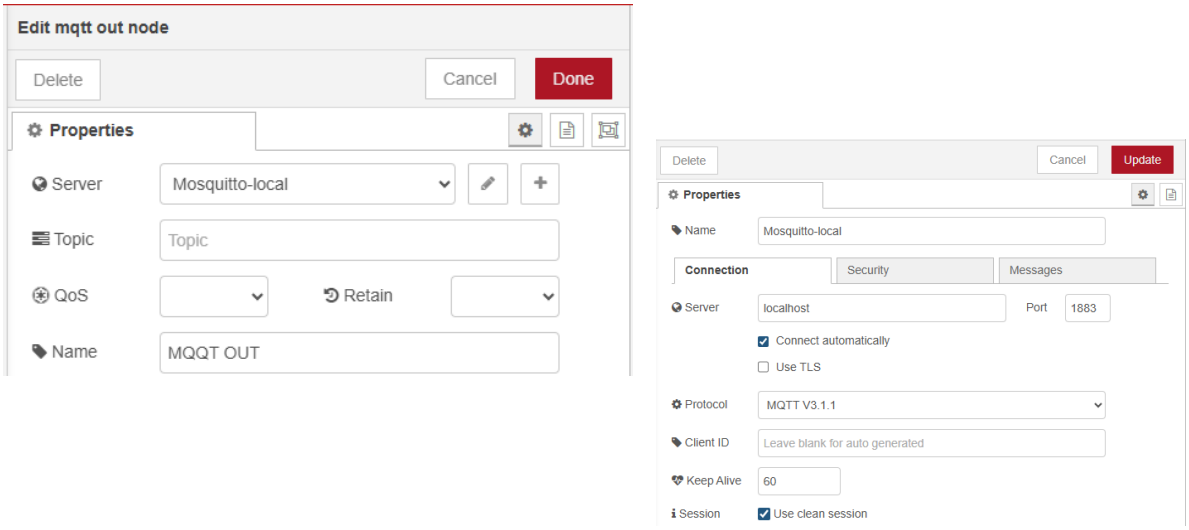


Рис. 9.38. Налаштування *MQTT*-брокера *Mosquitto-local* у *Node-RED* для вузла *mqtt out*

Для перевірки доставки повідомлень через *MQTT* без залучення *Ignition* додають вузол *mqtt in*, який підписується на ту саму тему *SCADA/sensors/data* на тому самому брокері *Mosquitto-local*. Вузол *mqtt in* є джерелом повідомлень у потоці, тому він не підключається лінією після *mqtt out*, а працює паралельно, отримуючи повідомлення з брокера після їхньої публікації. Вихід *mqtt in* під'єднують до окремого *debug*. У полі *Output* доцільно використати автоматичне визначення типу даних, щоб *JSON* відображався як об'єкт. Приклад налаштування *mqtt in* наведено на рис. 9.39.

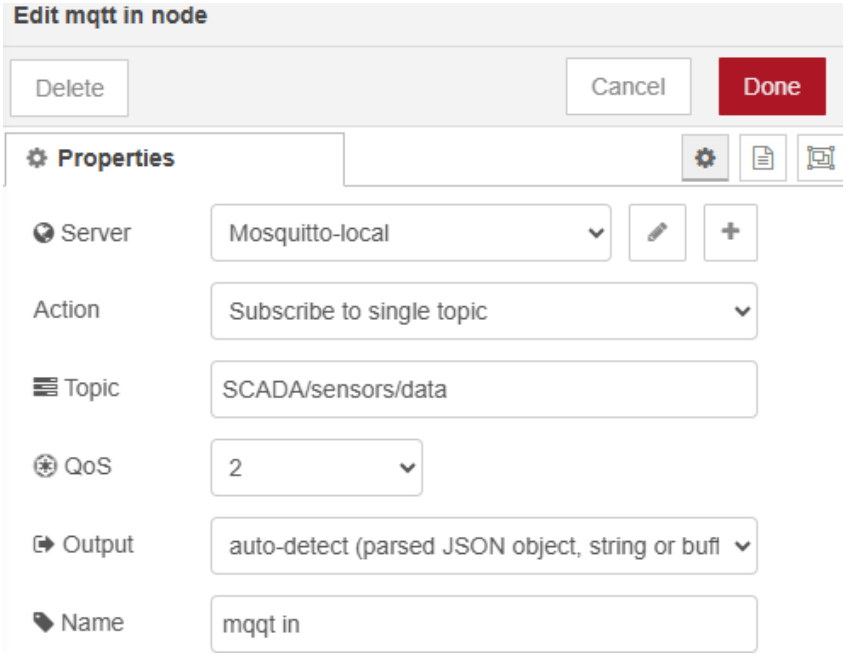


Рис. 9.39. Налаштування вузла *mqtt in* для підписки на тему *SCADA/sensors/data* у *Mosquitto*

Після виконання контрольної вставки нового запису в таблицю *MySQL* має спостерігатися надходження повідомлення в *Debug* не лише після вузла *function*, а й після *mqtt in*. Це підтверджує, що подія пройшла шлях *MySQL, Debezium, Kafka, Node-RED* і була доставлена через *Mosquitto* у вигляді *MQTT*-публікації.

На цьому етапі підтверджено працездатність проміжного рівня перетворення і транспортування даних між *Apache Kafka* та диспетчерським рівнем через *MQTT*. Наступним кроком є підключення *SCADA Ignition* до *MQTT*-брокера та відображення отриманих значень у вигляді змінних диспетчерського рівня.

9.10.4. Підключення SCADA Ignition до MQTT та приймання подій

Після підтвердження доставки повідомлень у *MQTT* на рівні *Node-RED* виконують підключення *SCADA Ignition* до *MQTT*-брокера *Mosquitto* та налаштовують відображення отриманих значень у вигляді тегів

диспетчерського рівня. Налаштування виконують у вебінтерфейсі *Ignition Gateway* у розділі *MQTT Engine Settings*.

На вкладці *Servers* перевіряють наявність підключення до брокера *Mosquitto*. Для локального стенда вказують адресу *tcp://localhost*:1883. Після збереження конфігурації статус з'єднання має бути *Connected*.

Далі переходять на вкладку *Namespaces* та переглядають доступні простори імен (*namespace*). У стандартній конфігурації відображаються вбудовані типи просторів імен, зокрема *Elecsys, Sparkplug A, Sparkplug B, Xirgo*. Вони призначені для певних форматів представлення даних і не є обов'язковими для сценарію, де *Node-RED* публікує звичайні *MQTT*-повідомлення з корисним навантаженням у форматі *JSON*. Приклад переліку просторів імен наведено на рис. 9.40.

Name	Namespace Type	Enabled
Elecsys	Elecsys	true
Sparkplug A	SparkplugA	true
Sparkplug B	SparkplugB	false
Xirgo	Xirgo	true

Рис. 9.40. Перелік просторів імен (*Namespaces*) у налаштуваннях *MQTT Engine* на рівні *Ignition Gateway*

Далі створюють *Custom*-простір імен для прийому повідомлень із теми *SCADA/sensors/data*. У вкладці *Custom* заповнюють параметри конфігурації. У полі Name задають назву, наприклад *SCADA_JSON*. У полі *Subscriptions* вказують тему, яку публікує *Node-R*ED. У межах стенда використовується *SCADA/sensors/data*. Якщо передбачається підписка на групу тем у межах одного каталогу, допускається використання шаблону *SCADA/sensors*/#.

Для впорядкування розміщення тегів задають *Root Tag Folder*, наприклад *SCADA_NODE_RED*. У цьому разі сформовані теги розміщуються всередині заданої папки. Поле *Tag Name* залишають порожнім. У такому режимі останній токен теми використовується як вузол нижнього рівня, а попередні токени теми утворюють папки. Для теми *SCADA/sensors/data* формуються папки *SCADA* і *sensors*, вузол *data*, а поля *JSON*-повідомлення відображаються як дочірні теги всередині *data*.

Опцію *JSON Payload* вмикають, тобто активують *Parse the payload as a JSON string*. Кодування *Encoding Charset* залишають *UTF_8*. Приклад налаштування *Custom Namespace* наведено на рис. 9.41.

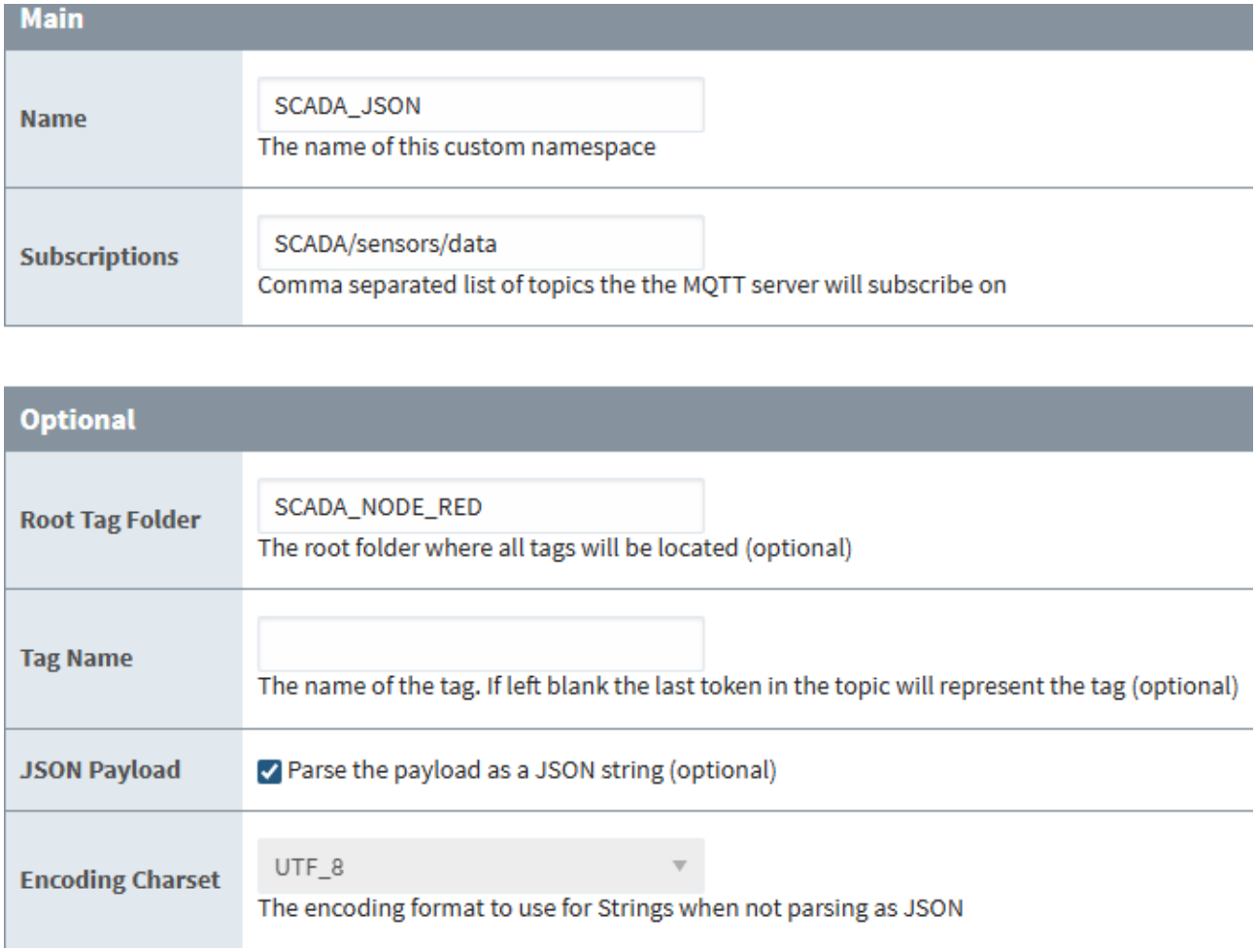


Рис. 9.41. Налаштування *Custom Namespace* у *MQTT Engine* для підписки на тему *SCADA/sensors/data* та розбору *JSON*-повідомлень

Після збереження конфігурації переходять в *Ignition Designer* і відкривають *Tag Browser*. У списку провайдерів тегів вибирають *MQTT Engine* та перевіряють появу структури тегів. Для наведеного прикладу очікується розміщення тегів у дереві *SCADA_NODE_RED*, далі *SCADA, sensors, data*, а всередині вузла *data* мають відображатися теги *id, t_stamp, temperature, pressure, flow_rate*. Значення тегів оновлюються під час надходження нових повідомлень від *Node-RED*. Приклад відображення тегів у *Tag Browser* наведено на рис. 9.42.

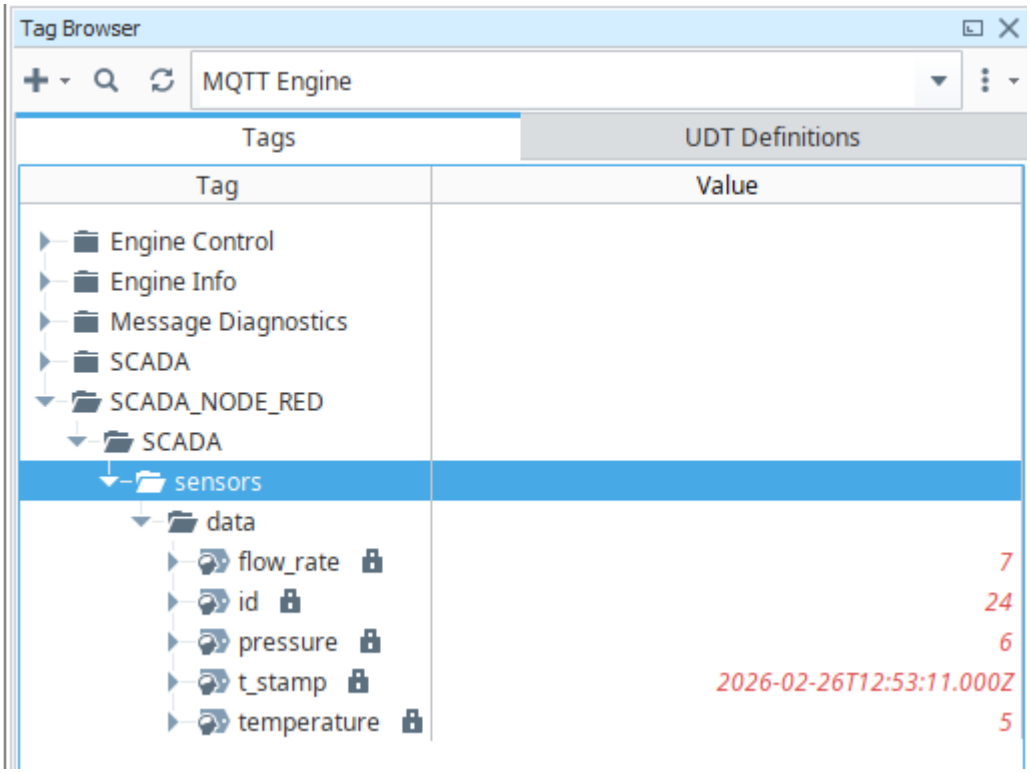


Рис. 9.42. Відображення параметрів з *MQTT*-теми *SCADA/sensors/data* у *Tag Browser SCADA Ignition* після розбору *JSON* у *MQTT Engine* (*Custom Namespace*)

СПИСОК ДЖЕРЕЛ ІНФОРМАЦІЇ

1. Болтон В. Програмовані логічні контролери та промислова автоматизація / В. Болтон. – 6-те вид. – Oxford : Newnes, 2015. – 432 с.
2. Бойєстад Р., Нашелскі Л. Електронні пристрої та схеми / Р. Бойєстад, Л. Нашелскі. – 11-те вид. – Hoboken : Pearson Education, 2018. – 912 с.
3, Дорожовець М. Основи метрології та вимірювальної техніки : підручник : у 2 т. / М. Дорожовець, В. Мотало, Б. Стадник ; за ред. Б. Стадника. – Львів : Вид-во Нац. ун-ту «Львівська політехніка», 2005. – Т. 1. – 560 с.
4. Каляєв І. А. Автоматизовані системи керування технологічними процесами : навч. посіб. / І. А. Каляєв. – Київ : Кондор, 2019. – 368 с.
5. Поліщук Є. С. Інформаційно-вимірювальні системи : навч. посіб. / Є. С. Поліщук. – Львів : Вид-во Львів. політехніки, 2018. – 312 с.
6. Harrington J. L. Relational Database Design and Implementation / J. L. Harrington. – 4th ed. – Amsterdam : Morgan Kaufmann, 2016. – 520 p.
7. Date C. J. An Introduction to Database Systems / C. J. Date. – 8th ed. – Boston : Addison-Wesley, 2004. – 1024 p.
8. Dubois P. MySQL Cookbook / P. Dubois. – 4th ed. – Sebastopol : O'Reilly Media, 2014. – 866 p.
9. Widenius M. MySQL Reference Manual / M. Widenius, D. Axmark, K. Arno. – Upper Saddle River : MySQL AB, 2022. – 1280 p.
10. Kleppmann M. Designing Data-Intensive Applications / M. Kleppmann. – Sebastopol : O'Reilly Media, 2017. – 616 p.
11. Hohpe G. Enterprise Integration Patterns : Designing, Building, and Deploying Messaging Solutions / G. Hohpe, B. Woolf. – Boston : Addison-Wesley, 2004. – 736 p.
12. Richards M. Software Architecture Patterns / M. Richards. – Sebastopol : O'Reilly Media, 2015. – 162 p.

13. Debezium Documentation : Change Data Capture for Databases [Electronic resource]. – Red Hat, 2024. – Available at: https://debezium.io (accessed: 09.03.2026).
14. Apache Kafka Documentation [Electronic resource]. – Apache Software Foundation, 2024. – Available at: https://kafka.apache.org/documentation (accessed: 09.03.2026).
15. Ignition User Manual [Electronic resource]. – Inductive Automation, 2024. – Available at: https://docs.inductiveautomation.com (accessed: 09.03.2026).
16. Banks J. Discrete-Event System Simulation / J. Banks, J. S. Carson, B. L. Nelson. – 5th ed. – Upper Saddle River : Pearson Education, 2010. – 608 p.
17. IEC 61131-3:2013. Programmable controllers – Part 3: Programming languages. – Geneva : International Electrotechnical Commission, 2013. – 256 p.
18. ISO/IEC 27001:2022. Information security, cybersecurity and privacy protection – Information security management systems – Requirements. – Geneva : International Organization for Standardization, 2022. – 36 p.

Навчальне видання

КРАСНІКОВ Ігор Леонідович
ЛИСАЧЕНКО Ігор Григорович
БАБІЧЕНКО Анатолій Костянтинович

ІНТЕГРАЦІЯ MySQL У SCADA-СИСТЕМИ:

ТЕОРІЯ І ПРАКТИКА

Навчальний посібник
для бакалаврів та магістрів
напряму підготовки
G7 «Автоматизація, комп'ютерно-інтегровані
технології та робототехніка»
усіх форм навчання

Відповідальна за випуск доц. Кравченко Я. О.

Роботу до видання рекомендувала доц. Крилова В. А.

В авторській редакції

План 2026 р., поз. 6

Гарнітура Times New Roman. Ум. друк. арк.14,63

Видавничий центр НТУ «ХПІ».
Свідоцтво про державну реєстрацію ДК № 5478 від 21.08.2017 р.
61002, Харків, вул. Кирпичова, 2.

Електронне видання