МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ»
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ СИСТЕМ ТА МЕРЕЖ

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач випускової кафедри
_____Юрій ІСКРЕНКО
"_____" _____2025 р.

КВАЛІФІКАЦІЙНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ "МАГІСТР"
ЗА СПЕЦІАЛЬНІСТЮ 123 "КОМП'ЮТЕРНА ІНЖЕНЕРІЯ"

Тема: Оптимізація продуктивності корпоративних DWH-систем у багатоплатформеному середовищи за допомогою ETL процесів

Виконавець: Богдан ПРОКОПЕЦЬ

Керівник: Юрій ІСКРЕНКО

Нормоконтролер: Наталія ФОМІНА

Київ 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ»
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ СИСТЕМ ТА МЕРЕЖ

ЗАТВЕРДЖУЮ
Завідувач кафедри КСМ
_____ Юрій ІСКРЕНКО
"___" _____ 2025 р.

ЗАВДАННЯ

на виконання кваліфікаційної роботи

Прокопця Богдана Вадимовича

1. Тема кваліфікаційної роботи (проекту): «Оптимізація продуктивності корпоративних DWH-систем у багатоплатформеному середовищи за допомогою ETL процесів»

затверджена наказом ректора від " 24 " жовтня 2025 р. № 2344/ст

2. Термін виконання: 29.09.2025 по 31.12.2025

3. Вихідні данні до проекту: *Matillion ETL*, СУБД *Snowflake*, СУБД *MSSQL*, *Excel*, *API.*

4. Зміст пояснювальної записки: Основи оптимізації продуктивності *DWH* - систем у мультиплатформенному середовищі, аналіз факторів та фінансових аспектів функціонування *DWH*, дослідження методів оптимізації, дослідження методів оптимізації роботи зі *Snowflake DB* із застосуванням *Matillion ETL*

5. Перелік обов'язкового графічного (ілюстративного) матеріалу:

Матеріал представлений у презентації

6. Календарний план-графік

№	Завдання	Термін виконання	Відмітка про виконання
1	Отримання завдання кваліфікаційної роботи	29.09.2025	
2	Аналіз предметної області та теоретичних основ оптимізації продуктивності корпоративних *DWH*-систем у мультиплатформенному середовищі. Формування структури та написання розділу 1 пояснювальної записки.	30.09.2025 – 10.11.2025	
3	Дослідження методів оптимізації продуктивності *MSSQL* бази даних, розгорнутої на *AWS EC*2.	11.11.2025 – 23.11.2025	
4	Дослідження методів оптимізації роботи зі *Snowflake DB* із застосуванням *Matillion ETL*. Аналіз архітектури, витрат, продуктивності, *ETL*-процесів та мультиплатформенної інтеграції.	24.11.2025 – 28.11.2025	
5	Загальне редагування пояснювальної записки	29.11.2025 – 05.12.2025	
6	Проходження нормоконтролю, перепліт пояснювальної записки.	06.12.2025 – 14.12.2025	
7	Розробка тексту доповіді. Оформлення графічного матеріалу для презентації.	15.12.2025 – 17.12.2025	
8	Отримання відгуку керівника, рецензії.	17.12.2025 – 22.12.2025	
9	Захист кваліфікаційної роботи	23.12.2025- 31.12.2025	

7. Консультація з окремого(мих) розділу(ів):

Назва розділу	Консультант (посада, П.І.Б.)	Дата, підпис	
		Завдання видав	Завдання прийняв

8. Дата видачі завдання «29» вересня 20 25 р.

Керівник кваліфікаційної роботи: Юрій ІСКРЕНКО
(підпис керівника)

Завдання прийняв до виконання: Богдан ПРОКОПЕЦЬ
(підпис виконавця)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи «Оптимізація продуктивності корпоративних *DWH*-систем у багатоплатформенному середовищі за допомогою *ETL* процесів» : 91 сторінка, 8 рисунків, 1 таблиця, 28 інформаційних джерел.

Об'єкт дослідження: Корпоративні системи управління сховищами даних (*DWH*) у мультиплатформенному середовищі.

Предметом дослідження: *ETL*-процеси та їх вплив на продуктивність *DWH*-систем, методи оптимізації.

Мета дослідження: Метою дослідження є розробка та валідація стратегій оптимізації *ETL*-процесів, які підвищують продуктивність, масштабованість та надійність корпоративних систем управління сховищами даних (*DWH*) у мультиплатформенному середовищі.

Методи дослідження: набір інструкцій та команд мов програмування *T-SQL* і *Python*, сучасні методи аналізу даних, *PowerBI* звіт. *MSSQL management studio*, *DBeaver*, *Excel*.

Дослідження відбувалися на основних робочих платформах, які використовуються компаніями із великого бізнесу.

Результати роботи : Визначення ключових факторів, що впливають на продуктивність *ETL*-процесів у мультиплатформенних *DWH*-системах. Розробка оптимізованої архітектури *ETL*-процесів з урахуванням специфіки *Snowflake*, *MSSQL*, *Postgres*. Підвищення швидкості завантаження та обробки даних у *DWH*-системах.

Зменшення навантаження на обчислювальні ресурси за рахунок ефективного розподілу *ETL*-операцій. Підтвердження ефективності запропонованих рішень шляхом порівняння продуктивності до та після оптимізації. Практичні рекомендації щодо побудови *ETL*-процесів у корпоративному мультиплатформенному середовищі.

Створення шаблонів або модулів *ETL*, які можуть бути повторно використані в інших проєктах.

Рекомендації щодо використання результатів

Застосування напрацювань для оптимізації корпоративних дата сховищ та використання побудованого шаблону, як стандарт для розробки нових процесів переливки даних.

СХОВИЩА ДАНИХ *DWH*, ОПТИМІЗАЦІЯ ПРОДУКТИВНОСТІ, РЕЛЯЦІЙНІ БАЗИ ДАНИХ, ХМАРНІ ПЛАТФОРМИ, ПРОЦЕСИ ЗАВАНТАЖЕННЯ ТА ТРАНСФОРМАЦІЇ ДАНИХ, МУЛЬТИПЛАТФОРМЕННЕ СЕРЕДОВИЩЕ, ОПТИМІЗАЦІЯ ВАРТОСТІ, *SNOWFLAKE*, *MATILLION ETL*

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API — *Application Programming Interface*, інтерфейс програмної взаємодії між інформаційними системами

AWS — *Amazon Web Services*, хмарна платформа компанії *Amazon*

EC2 — *Elastic Compute Cloud*, сервіс віртуальних серверів *AWS*

CPU — *Central Processing Unit*, центральний процесор

DWH — *Data Warehouse*, сховище даних

ELT — *Extract, Load, Transform*, підхід до обробки даних

ETL — *Extract, Transform, Load*, процес вилучення, трансформації та завантаження даних

ETL job — логічний процес обробки даних у середовищі *ETL*

I/O — *Input / Output*, операції введення та виведення даних

JSON — *JavaScript Object Notation*, формат напівструктурованих даних

MSSQL — *Microsoft SQL Server*, система керування реляційними базами даних

OLAP — *Online Analytical Processing*, технологія багатовимірного аналізу даних

SQL — *Structured Query Language*, мова структурованих запитів

VARIANT — тип даних у *Snowflake*

СУБД — система управління базами даних

ВСТУП

Актуальність даної роботи полягає у зростаючій потребі компаній, які покладаються на великомасштабні сховища даних, для оптимізації використання ресурсів та операційних витрат, зберігаючи при цьому високу продуктивність та масштабованість. У сучасному конкурентному та обсягом даних бізнес-середовищі ефективна обробка даних є важливою для своєчасної аналітики, прийняття стратегічних рішень та підтримки бізнес-гнучкості.

Оскільки організації все частіше впроваджують багатоплатформні інфраструктури, що поєднують хмарні та локальні системи, складність управління потоками даних та забезпечення стабільної продуктивності на різних платформах стає значним викликом. Процеси *ETL*, які слугують основою інтеграції даних, повинні бути ретельно розроблені та оптимізовані, щоб запобігти вузьким місцям, зменшити затримку та забезпечити надійну доставку даних.

Основними задачами, що розглядаються «Оптимізація продуктивності корпоративних *DWH*-систем у багатоплатформенному середовищі за допомогою *ETL* процесів», є:

1. Аналіз потреб компаній у оптимізації витрат та ресурсів при роботі з великими сховищами даних;

2. Аналіз *DWH* і визначення, що генерує вартість роботи і впливає на швидкість виконання розрахунків;

3. Вивчення особливостей мультиплатформенного середовища в корпоративних IT-інфраструктурах;

4. Оцінка ролі *ETL*-процесів у забезпеченні узгодженості та швидкості обробки даних;

5. Формування практичних рекомендацій для побудови *ETL* у мультиплатформенних *DWH*-системах.

Актуальність теми

Слідом за стрімким розвитком цифрових технологій у корпоративному середовищі, традиційні підходи до обробки великих обсягів даних втрачають свою ефективність. Використання моноплатформенних рішень або застарілих *ETL*-інструментів не відповідає сучасним вимогам до продуктивності, масштабованості та економічності.

У зв'язку з цим, компанії, що працюють з великими сховищами даних, стикаються з необхідністю оптимізації витрат на обробку інформації, а також підвищення швидкості та стабільності *ETL*-процесів. Особливо це актуально в умовах мультиплатформенного середовища, де дані зберігаються та обробляються у різних системах — таких як *Snowflake*, *MSSQL*, *Postgres* — і потребують узгодженої інтеграції.

Існуючі рішення часто мають такі недоліки, як перевантаження ресурсів, затримки при передачі даних, складність масштабування та недостатня гнучкість при зміні бізнес-вимог. Це створює ризики для аналітичних процесів, які залежать від своєчасного та якісного доступу до даних.

Таким чином, розробка надійних, оптимізованих та економічно ефективних *ETL*-рішень для мультиплатформенних *DWH*-систем є актуальною задачею, що відповідає сучасним потребам бізнесу в умовах зростаючої конкуренції та обсягів даних.

Мета і завдання виконання кваліфікаційної роботи

Метою кваліфікаційної роботи є дослідження та розробка підходів до оптимізації продуктивності корпоративних систем управління сховищами даних (*DWH*) у мультиплатформенному середовищі за допомогою *ETL*-процесів. Для досягнення поставленої мети необхідно дослідити принципи функціонування *DWH*-систем, а також визначити ключові фактори, що впливають на їх продуктивність та вартість обробки даних.

Потрібно проаналізувати труднощі, з якими стикаються компанії при роботі з великими обсягами даних у мультиплатформенних середовищах, зокрема при інтеграції *Snowflake*, *MSSQL* та *Postgres*. Необхідно оцінити роль

ETL-процесів у забезпеченні узгодженості та швидкості обробки даних, а також дослідити існуючі інструменти та підходи до побудови *ETL*.

Серед основних завдань — розробка архітектури оптимізованих *ETL*-процесів, які відповідають вимогам продуктивності та масштабованості, а також якісна реалізація запропонованих рішень з використанням *Matillion ETL* та *Python*. Потрібно обрати відповідне середовище розробки, інструменти та технології, що забезпечать ефективну інтеграцію даних між платформами.

У результаті має бути створено набір практичних рекомендацій та технічних рішень, які дозволять компаніям покращити продуктивність своїх *DWH*-систем, зменшити витрати на обробку даних та забезпечити стабільну роботу *ETL*-процесів у складних інфраструктурах.

Об’єкт і предмет дослідження

Об’єктом дослідження даної роботи є корпоративні системи управління сховищами даних (*DWH*) у мультиплатформенному середовищі.

Предметом дослідження є *ETL*-процеси та їх вплив на продуктивність, узгодженість і ефективність роботи *DWH*-систем.

Методи дослідження

У ході дослідження для досягнення поставленої в роботі мети використовуються інструменти та технології, що дозволяють моделювати, реалізовувати та тестувати *ETL*-процеси в мультиплатформенному середовищі. Основними засобами є *Matillion ETL*, *Snowflake*, *MSSQL*, *Postgres* та мова програмування *Python*.

Розробка та тестування *ETL*-процесів здійснюється у відповідному середовищі, з використанням хмарної інфраструктури *Snowflake*, локальних серверів *MSSQL* та *Postgres*, а також засобів автоматизації на базі *Python*. Для контролю версій та управління кодом використовується *Bitbucket*, а для візуалізації та моніторингу — вбудовані інструменти *Matillion* та *SQL*-консолі.

Для забезпечення максимальної продуктивності *ETL*-процесів застосовуються власні підходи до побудови архітектури, оптимізації запитів, розподілу навантаження та обробки великих обсягів даних. Значну роль

відіграють алгоритми трансформації даних, які реалізуються за допомогою *Python*-скриптів та функціоналу *Matillion*.

Наукова новизна отриманих результатів

У ході виконання кваліфікаційної роботи було розроблено підходи до оптимізації *ETL*-процесів у корпоративних *DWH*-системах, що функціонують у мультиплатформенному середовищі. Запропоновані рішення дозволяють ефективно інтегрувати дані між такими платформами, як *Snowflake*, *MSSQL* з урахуванням їх архітектурних особливостей.

Завдяки використанню *Matillion ETL* та *Python* реалізовано гнучкі механізми трансформації даних, що забезпечують підвищення продуктивності та зниження навантаження на обчислювальні ресурси. Розроблені алгоритми дозволяють автоматизувати процеси обробки великих обсягів даних, зменшити час виконання *ETL*-процесів та покращити узгодженість інформації між системами.

Також реалізовано механізми контролю якості даних, резервного копіювання та відновлення, що сприяють підвищенню надійності роботи *DWH*-систем. Забезпечено підтримку мультиплатформенності, що дозволяє використовувати запропоновані рішення на різних типах інфраструктури — як локальних, так і хмарних.

Отримані результати можуть бути використані для побудови ефективних *ETL*-процесів у корпоративних середовищах, що працюють з великими обсягами даних, та слугувати основою для подальших досліджень у сфері оптимізації продуктивності *DWH*-систем.

Практичне значення отриманих результатів

Розроблені підходи до оптимізації *ETL*-процесів можуть бути впроваджені в корпоративних IT-середовищах, що працюють з великими обсягами даних у мультиплатформенних системах. Запропоновані рішення можуть використовуватися як великими компаніями, так і окремими командами розробників, які займаються побудовою або підтримкою *DWH*-інфраструктури.

Гнучкість налаштування *ETL*-процесів дозволяє адаптувати їх до різних типів даних, джерел та платформ, що забезпечує широку сферу застосування — від фінансової аналітики до логістики, маркетингу та управління ризиками. Це дає змогу значно зменшити витрати на обробку даних, підвищити швидкість аналітичних розрахунків та забезпечити стабільну інтеграцію між системами.

Використання *Matillion ETL*, *Python* та підтримка *Snowflake*, *MSSQL* і *Postgres* дозволяє реалізувати кросплатформенність, що забезпечує доступність рішень на різних типах інфраструктури — як локальних, так і хмарних. Отримані результати можуть бути основою для створення масштабованих, продуктивних та економічно ефективних *ETL*-рішень у сучасних корпоративних середовищах.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ ОПТИМІЗАЦІЇ ПРОДУКТИВНОСТІ *DWH*-СИСТЕМ У МУЛЬТИПЛАТФОРМЕННОМУ СЕРЕДОВИЩІ

1.1 Поняття та роль корпоративних *DWH*-систем

Сховище даних (англ. *Data Warehouse, DWH*) — це спеціалізована інформаційна система, призначена для централізованого зберігання, обробки та аналізу великих обсягів структурованих даних, що надходять з різних джерел. Основна мета *DWH* — забезпечити єдине джерело достовірної інформації для підтримки процесів прийняття рішень на всіх рівнях управління.

На відміну від транзакційних баз даних, які оптимізовані для швидкої обробки окремих операцій (*insert*, *update*, *delete*), *DWH* орієнтовані на виконання складних аналітичних запитів, агрегацію даних, побудову звітів та прогнозів. Дані в *DWH* зазвичай мають історичний характер, що дозволяє аналізувати динаміку змін у часі [1].

Типова архітектура корпоративного *DWH* включає кілька компонентів:

1. Джерела даних — це операційні системи (*CRM*, *ERP*, бухгалтерські програми), зовнішні *API (Application Programming Interface)*, файли, веб-сервіси тощо.

2. *ETL*-процеси (*Extract*, *Transform*, *Load*) — механізми вилучення даних з джерел, їх трансформації у відповідний формат та завантаження до сховища.

3. Сховище даних — фізична база даних, яка може бути реалізована на різних платформах, таких як *Snowflake*, *MSSQL*, *Postgres*.

4. Моделі даних — логічна структура, яка визначає зв’язки між таблицями, типи даних, індекси, агрегати, ключі.

5. Інструменти аналітики — *BI*-системи (*Power BI*, *Tableau*), *SQL*-консолі, *ML*-моделі, звітні панелі, що забезпечують доступ до даних для кінцевих користувачів.

До прикладу можна привести узагальнену схему (рис 1.1).

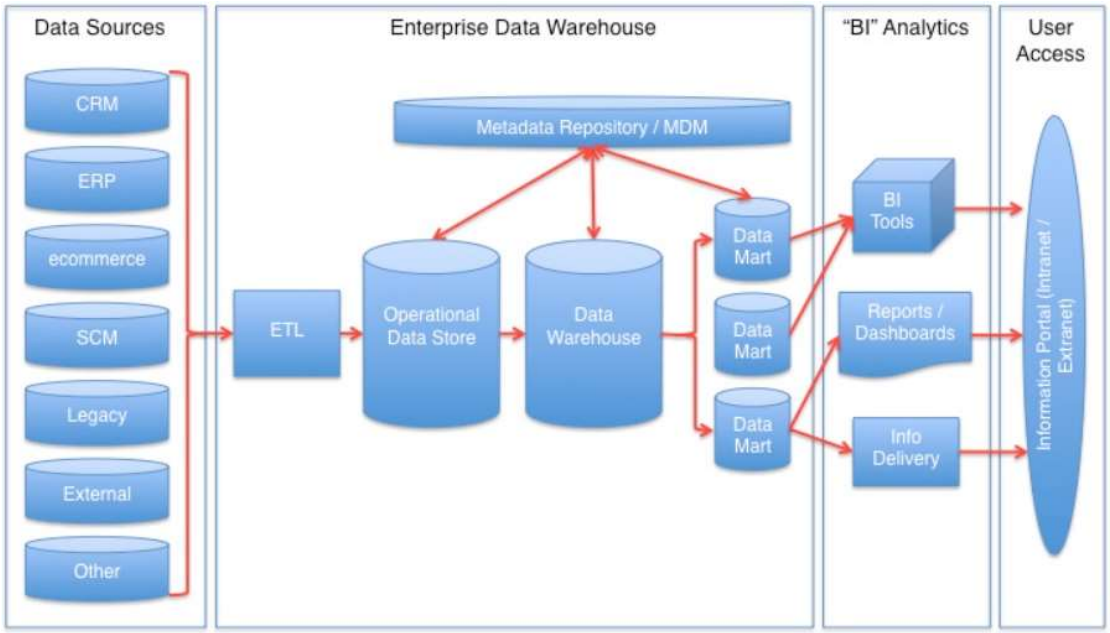

Рисунок 1.1 – Архітектура *DWH*

У сучасних реалізаціях *DWH* все частіше використовуються хмарні технології, які дозволяють масштабувати ресурси відповідно до навантаження, знижувати витрати на інфраструктуру та забезпечувати високу доступність і безперервність роботи.

Корпоративні *DWH*-системи є основою для побудови систем *Business Intelligence* (*BI*), які дозволяють компаніям:

1. Аналізувати історичні та поточні дані для виявлення трендів, закономірностей та аномалій;

2. Прогнозувати поведінку клієнтів, попит, ризики та фінансові показники;

3. Оптимізувати внутрішні процеси, ресурси та витрати;

4. Приймати обґрунтовані управлінські рішення на основі достовірних даних.

Завдяки централізованому зберіганню даних, *DWH* забезпечує єдину версію правди (*Single Source of Truth*), що критично важливо для узгодженості звітності між різними підрозділами компанії. Це дозволяє уникнути дублювання інформації, суперечностей у звітах та помилок у прийнятті рішень.

Вимоги до сучасних *DWH*-систем:

1. Масштабованість - система повинна мати здатність до горизонтального та вертикального масштабування без втрати продуктивності. Це особливо важливо для компаній, які працюють з *Big Data* або мають сезонні піки навантаження. Хмарні рішення, такі як *Snowflake*, дозволяють автоматично масштабувати обчислювальні ресурси відповідно до поточного навантаження.

2. Продуктивність - висока швидкість виконання запитів, *ETL*-процесів та аналітичних операцій є ключовою вимогою. Продуктивність залежить від ефективності *SQL*-запитів, паралельної обробки даних, оптимізації *ETL*-логіки, використання індексів, партицій, кластерів. У *Snowflake*, наприклад, продуктивність масштабовано залежить від обраного *compute*-кластеру, а в *MSSQL* — від конфігурації сервера (кількість ядер, обсяг пам’яті).

3. Надійність — гарантія цілісності даних, захист від втрат, підтримка резервного копіювання та відновлення. Включає транзакційну узгодженість, контроль версій, *fault-tolerance*, автоматичне відновлення після аварій.

4. Гнучкість — підтримка різних форматів даних (*CSV*, *JSON*, *Parquet*, *XML*), джерел (*SQL*, *NoSQL*, *API*), типів навантаження (*batch*, *streaming*). Дозволяє адаптувати систему до змін бізнес-вимог без перебудови архітектури.

5. Інтеграція — легка інтеграція з *BI*-інструментами (*Power BI*, *Tableau*), *ML*-платформами, *API* та іншими системами. Важлива підтримка стандартів (*ODBC*, *JDBC*, *REST*) та *ETL*-інструментів (*Matillion*, *Talend*, *Apache NiFi*).

6. Безпека — захист даних відповідно до регуляторних вимог (*GDPR*, *HIPAA*). Передбачає шифрування, контроль доступу на рівні користувачів, ролей, колонок, аудит дій, багатофакторну автентифікацію.

7. Фінансова ефективність — враховує витрати на ресурси (*CPU*, *RAM*, *storage*), ліцензії, адміністрування, хмарні сервіси. Оптимізація *ETL*-процесів знижує час обробки та витрати. У *Snowflake* — менший час роботи *warehouse* знижує вартість, у *MSSQL* — оптимізація запитів дозволяє уникнути масштабування серверів.

1.2 Продуктивність *DWH*: визначення, метрики, фактори впливу

У контексті корпоративних сховищ даних (*DWH*), продуктивність є одним із ключових параметрів, що визначає здатність системи ефективно обробляти великі обсяги інформації, забезпечувати швидкий доступ до даних та підтримувати аналітичні процеси.

Висока продуктивність дозволяє бізнесу приймати рішення в реальному часі, знижує витрати на обчислення та покращує загальну якість обслуговування.

Продуктивність *DWH*-системи — це її здатність обробляти запити, виконувати *ETL*-процеси та забезпечувати аналітичну обробку даних з мінімальними затримками та максимальною ефективністю.

Вона охоплює як швидкість виконання окремих операцій, так і загальну здатність системи витримувати навантаження при зростанні обсягів даних та кількості користувачів.

Основні метрики продуктивності:

1. Час виконання запитів (*Query Execution Time*) — показник, що відображає, скільки часу потрібно для обробки *SQL*-запиту. Важливий для *OLAP*-аналітики, де запити можуть бути складними та ресурсоємними.

2. Пропускна здатність (*Throughput*) — кількість запитів або оброблених записів за одиницю часу. Визначає, наскільки система здатна обробляти паралельні запити або великі обсяги даних.

3. Затримка (*Latency*) — час між надсиланням запиту та отриманням першої відповіді. Критично важлива для інтерактивних *BI*-інструментів та дашбордів.

Фактори, що впливають на продуктивність:

1. Обсяг даних — зі збільшенням обсягу таблиць зростає час сканування, що напряму впливає на швидкість запитів. Використання партицій та кластеризації дозволяє зменшити обсяг даних, що обробляється.

2. Складність запитів — запити з багатьма *JOIN*, підзапитами, агрегаціями або умовами фільтрації можуть значно навантажувати систему. Оптимізація *SQL*-коду та використання матеріалізованих представлень (*materialized views*) допомагає зменшити навантаження.

3. Індексація — наявність правильних індексів (*B-tree*, *columnstore*, *hash*) дозволяє швидше знаходити потрібні дані. Відсутність або неправильне використання індексів може призвести до повного сканування таблиць.

4. Паралельна обробка — здатність системи виконувати запити або *ETL*-процеси у багатопоточному режимі. *Snowflake*, наприклад, автоматично розподіляє навантаження між віртуальними *warehouse*, а *MSSQL* дозволяє налаштовувати ступінь паралелізму вручну.

5. Кешування результатів — повторне використання результатів запитів дозволяє зменшити навантаження на систему. *Snowflake* має вбудоване автоматичне кешування, що значно прискорює повторні запити.

Роль апаратного забезпечення та конфігурації:

1. Процесор (*CPU*) — кількість ядер та тактова частота впливають на швидкість обробки запитів. У *MSSQL* на *AWS*, наприклад, вибір *EC*2-інстансу з більшою кількістю *vCPU* дозволяє обробляти більше запитів паралельно.

2. Оперативна пам’ять (*RAM*) — достатній обсяг пам’яті дозволяє зберігати тимчасові таблиці, кешувати дані та уникати звернень до диску. Недостатня пам’ять призводить до *swap*-операцій, що значно знижує продуктивність.

3. Тип диску (*SSD vs HDD*) — *SSD*-диски забезпечують набагато швидший доступ до даних, що критично для *ETL*-процесів та великих сканувань.

4. Мережева інфраструктура — швидкість передачі даних між компонентами системи (*ETL*-інструментами, *BI*-платформами, джерелами даних)

впливає на загальну продуктивність. У хмарних середовищах важливо враховувати розташування регіонів та *latency* між ними.

1.3 Фінансові аспекти продуктивності *DWH*-систем

Продуктивність *DWH*-системи має прямий вплив на фінансові витрати компанії. В умовах хмарної інфраструктури, де ресурси оплачуються за фактом використання, кожна секунда роботи обчислювального кластеру або кожен гігабайт пам'яті має свою ціну. Тому оптимізація продуктивності — це не лише технічне завдання, а й економічна необхідність.

1.3.1 Вартість обчислень у *Snowflake* (*Compute Cost*)

Snowflake використовує модель споживання, де обчислювальні ресурси оплачуються за використані *Snowflake Credits*. Основні аспекти:

1. Віртуальні *warehouse* — це обчислювальні кластери, які запускаються для виконання запитів та *ETL*-процесів.
2. Ціна за кредит залежить від обраної редакції *Snowflake* (*Standard*, *Enterprise*, *Business Critical*), хмарного провайдера (*AWS*, *Azure*, *GCP*) та регіону.
3. Споживання кредитів:
    3.1 *X-Small warehouse* — 1 кредит/година;
    3.2 *Small* — 2 кредити/година;
    3.3 *Medium* — 4 кредити/година;
    3.4 *Large* — 8 кредитів/година;
    3.5 *X-Large* — 16 кредитів/година.
4. Автоматичне призупинення (*auto-suspend*) — дозволяє зменшити витрати, зупиняючи *warehouse* після завершення запитів (часто налаштовується на 1–5 хвилин).
5. Ціна одного кредиту — приблизно $2–$4 при оплаті по факту або $1.50–$2.50 при річному контракті.

Таким чином, неоптимізовані запити або надмірно великі *warehouse* можуть призвести до значного збільшення витрат.

1.3.2 Витрати на пам’ять у *MSSQL*/*PostgreSQL* на *AWS*

У випадку *MSSQL*, що працює на *EC*2-інстансах *AWS*, витрати формуються з кількох компонентів:

1. Тип *EC*2-інстансу — визначає кількість *vCPU*, обсяг оперативної пам’яті та дисковий простір.

2. Ліцензії — можуть бути включені в ціну інстансу (*License Included*) або використовуватись власні (*BYOL*).

3. Приклад ціни (*us-east*-1, *r*5.*xlarge*) *Windows* + *SQL Standard*: $668.68/година, з яких:

   3.1 *Compute*: $183.96;

   3.2 *Windows* ліцензія: $134.32;

   3.3 *SQL Standard* ліцензія: $350.40.

4. Пам’ять — критично важлива для продуктивності *MSSQL*. Недостатній обсяг призводить до *swap*-операцій, що знижує швидкість обробки.

5. Оптимізація *CPU* — функція *AWS*, яка дозволяє зменшити витрати, обмежуючи використання процесора без шкоди для продуктивності.

1.3.3 Зв’язок між продуктивністю та фінансовими витратами

Оптимізація запитів і раціональне використання обчислювальних ресурсів є ключовими чинниками зниження витрат:

1. Час виконання запитів напряму впливає на тривалість роботи обчислювальних ресурсів → більше часу = більше витрат.

2. Надмірне масштабування (наприклад, використання *Large warehouse* замість оптимального *Medium*) призводить до переплати.

3. Оптимізація *ETL*-процесів дозволяє зменшити обсяг оброблюваних даних, кількість запитів та час їх виконання → зниження *compute cost*.

4. Кешування, партиціювання, індексація — технічні засоби, які мають економічний ефект через зменшення навантаження на систему.

Таким чином, ефективне управління продуктивністю *DWH* — це стратегія зниження витрат, яка поєднує технічну оптимізацію з фінансовим плануванням.

1.4 Мультиплатформенне середовище: виклики та особливості

Сучасні корпоративні *DWH*-системи дедалі частіше функціонують у мультиплатформенних середовищах, які поєднують локальні, хмарні та гібридні компоненти. Такий підхід дозволяє гнучко масштабувати ресурси, знижувати витрати та забезпечувати високу доступність даних. Водночас він створює низку технічних викликів, пов'язаних із сумісністю, синхронізацією та узгодженістю даних.

Розглянемо типи мультиплатформенних архітектур:

1. Локальні (*on-premises*) — традиційні *DWH*, розгорнуті на фізичних серверах компанії. Високий рівень контролю, але обмежена масштабованість і висока вартість підтримки.

2. Хмарні (*cloud-native*) — повністю розгорнуті у хмарі (*AWS*, *Azure*, *GCP*). Переваги: автоматичне масштабування, оплата за використання, висока доступність.

3. Гібридні (*hybrid*) — поєднання локальних і хмарних компонентів. Наприклад, *MSSQL* працює локально, а *Snowflake* — у хмарі. Такий підхід дозволяє поступово мігрувати дані або розділяти навантаження.

Розглянемо проблеми сумісності, синхронізації та безперервності даних:

1. Сумісність форматів — різні платформи можуть використовувати різні типи даних, кодування, структури таблиць. Наприклад, *JSON* у *Snowflake* може оброблятись інакше, ніж у *MSSQL*.

2. Синхронізація даних — необхідність забезпечити актуальність даних між платформами. *ETL*-процеси мають враховувати часові зони, затримки, конфлікти при оновленні.

3. Безперервність (*data consistency*) — важливо уникати дублювання, втрати або розбіжностей у даних. Використання *CDC* (*Change Data Capture*), *timestamp*-стратегій та контроль версій допомагає підтримувати узгодженість.

Приклади взаємодії *Snowflake*, *MSSQL*, *Postgres*:

1. *Snowflake* + *MSSQL* — *MSSQL* використовується як джерело даних, *Snowflake* — як аналітична платформа. Дані передаються через *Matillion* або *Azure Data Factory*. Виклики: трансформація типів даних, оптимізація запитів для *Snowflake*.

2. *Postgres* + *Snowflake* — *Postgres* зберігає транзакційні дані, які періодично реплікуються у *Snowflake* для аналітики. Використовується *Snowpipe* або *Kafka* для стрімінгової доставки.

3. *MSSQL* + *Postgres* — обмін даними через *CDC* або реплікацію. Проблеми: різні механізми транзакцій, відмінності у синтаксисі *SQL*, обмеження на типи індексів.

1.5 *ETL*-процеси: структура, типи, роль у *DWH*

ETL (*Extract*, *Transform*, *Load*) — це ключовий механізм інтеграції даних у корпоративних сховищах, що забезпечує їх узгодженість, якість та доступність для аналітики. *ETL*-процеси визначають ефективність роботи *DWH*, впливають на продуктивність і фінансові витрати.

Основні етапи *ETL:*

1. *Extract* (Витяг) — отримання даних із різних джерел: реляційні БД (*MSSQL*, *Postgres*), *NoSQL*, *API*, файли (*CSV*, *JSON*, *Parquet*). Важливо мінімізувати навантаження на джерела та забезпечити коректність даних.

2. *Transform* (Трансформація) — очищення, нормалізація, агрегація, зміна форматів, обчислення бізнес-метрик. Використовуються правила узгодженості, контроль якості, обробка помилок.

3. *Load* (Завантаження) — перенесення даних у *DWH* (*Snowflake*, *MSSQL*, *BigQuery*). Може бути повним (*full load*) або інкрементальним (*incremental load*) [2].

Типи *ETL*-процесів:

1. *Batch ETL* — обробка великих обсягів даних пакетами, наприклад, раз на добу. Плюс: стабільність. Мінус: дані не оновлюються миттєво.

2. *Streaming ETL* — дані обробляються постійно, у реальному часі (*Kafka*, *Snowpipe*). Плюс: актуальність. Мінус: складніше налаштувати і дорожче [2].

Вплив *ETL* на продуктивність та узгодженість даних:

1. Якщо *ETL*-процеси довгі та складні, це збільшує час обробки і витрати на обчислення (особливо у хмарі, як *Snowflake*);

2. *ETL* допомагає уникнути хаосу в даних: всі джерела приводяться до єдиних правил, щоб не було дублювань і помилок;

3. Оптимізація *ETL* (паралельна обробка, *pushdown*-трансформації, кешування) зменшує час роботи і економить гроші [2].

Висновки до розділу

У цьому розділі ми розібрали, що сучасні *DWH*-системи мають бути масштабованими, продуктивними, надійними та економічно ефективними. Продуктивність залежить від багатьох факторів: обсягу даних, складності запитів, *ETL*-процесів і конфігурації обладнання. Ми також побачили, що фінансовий аспект дуже важливий: у хмарних платформах, таких як *Snowflake*, ми платимо не тільки за обчислювальні ресурси, але й за зберігання даних (гігабайти інформації). Чим більше даних і довше працюють кластери — тим вищі витрати.

Також, розглянувши мультиплатформенне середовище, його переваги та проблеми сумісності, а також роль *ETL*-процесів у забезпеченні узгодженості даних. *ETL* впливає і на продуктивність, і на витрати, тому його оптимізація є ключовою.

РОЗДІЛ 2
ДОСЛІДЖЕННЯ МЕТОДІВ ОПТИМІЗАЦІЇ *MSSQL DB* РОЗГОРНОТОГО НА *AWS EC*2 СКРВЕРІ

2.1 Огляд потенційних проблем продуктивності *MSSQL*

Серед потенційних проблемних місць можна виділити три напрямки:

1. Оптимізація на рівні таблиць;
2. Оптимізація ресурсів на рівні серверу;
3. Оптимізація процесів наповеннння даних.

Найважливішим пунктом є оптимізація на рівні таблиць, так як саму в таблицях зберігаються, записуються, обробляються та беруться дані.

2.1.2 Оптимізація на рівні таблиць

Продуктивність *MSSQL* значною мірою залежить від того, наскільки правильно спроєктовані та організовані таблиці. Навіть за потужного обладнання та оптимальних налаштувань сервера погано структурована таблиця може стати критичним вузьким місцем. Розглянемо ключові фактори, які найчастіше сповільнюють роботу великих таблиць у *SQL Server*.

Розмір таблиці є одним із найважливіших параметрів, що визначає продуктивність системи управління базами даних *Microsoft SQL Server*. Зі збільшенням обсягу даних зростає навантаження на дискову підсистему, оперативну пам’ять та процесор, що безпосередньо впливає на швидкість виконання запитів і загальну ефективність роботи корпоративних *DWH*-систем.

Великі таблиці створюють низку проблем. По-перше, при відсутності відповідних індексів або при виконанні запитів із умовами, що не використовують індекси, *SQL Server* змушений сканувати всю таблицю.

По-друге, збільшується кількість сторінок даних, адже *MSSQL*зберігає інформацію у сторінках розміром 8 *KB*, і чим більше рядків у таблиці, тим більше сторінок потрібно прочитати. Це збільшує кількість операцій введення-виведення та навантаження на буферний пул. Крім того, великі таблиці займають значну частину пам’яті, витісняючи інші об’єкти з кешу, що знижує ефективність повторних запитів. Масові операції вставки, оновлення або видалення на великих таблицях призводять до довготривалих блокувань, що негативно впливає на конкурентність.

Причинами збільшення розміру таблиць часто є відсутність архівування історичних даних, коли інформація накопичується роками без видалення або перенесення в архів, а також відсутність партиціювання, через що всі дані зберігаються в одній фізичній структурі. Додатково на розмір впливає надмірне дублювання даних, що виникає через погану нормалізацію або помилки у *ETL*-процесах.

Для оптимізації роботи з великими таблицями застосовують кілька підходів. Партиціювання дозволяє розбити таблицю на логічні сегменти, наприклад за датою, що зменшує обсяг даних, які обробляються в одному запиті. Архівування даних передбачає перенесення історичних записів у окремі архівні таблиці або бази. Індексація допомагає зменшити кількість сторінок, що скануються, а використання компресії на рівні рядків або сторінок знижує фізичний розмір таблиці та кількість сторінок.

Таким чином, розмір таблиці є критичним фактором продуктивності *MSSQL*. Без належного управління обсягами даних великі таблиці стають джерелом проблем: від повільних запитів до блокувань і перевантаження системи. Використання партиціювання, архівування та оптимальної індексації дозволяє суттєво знизити негативний вплив цього фактору.

2.1.2 Оптимізація ресурсів на рівні серверу

Оптимізація ресурсів на рівні серверу є ключовим аспектом забезпечення стабільної роботи *Microsoft SQL Server*, особливо у середовищах із великими обсягами даних та високим навантаженням. Серверні ресурси — процесор,

оперативна пам’ять, дискова підсистема та мережеві канали — визначають швидкість виконання запитів, ефективність транзакцій та загальну продуктивність системи.

SQL Server активно використовує *CPU* для обробки запитів, виконання планів та обчислень. Основні проблеми виникають при надмірному паралелізмі або неоптимальних планах запитів. Для оптимізації рекомендується налаштувати параметр *MAXDOP* (*Maximum Degree of Parallelism*), щоб обмежити кількість потоків для одного запиту, а також контролювати *Cost Threshold for Parallelism*, щоб уникнути надмірного розпаралелювання простих запитів.

Буферний пул *SQL Server* зберігає сторінки даних у пам’яті для швидкого доступу. Недостатній обсяг пам’яті призводить до частих звернень до диску, що знижує продуктивність. Оптимізація включає збільшення доступної пам’яті для *SQL Server*, налаштування параметрів *min*/*max server memory* та моніторинг використання пам’яті через *Dynamic Management Views* (*DMVs*).

Швидкість дискових операцій є критичною для роботи з великими таблицями та журналами транзакцій. Використання *SSD*-накопичувачів, налаштування *RAID*-масивів та розділення файлів даних і журналів на різні диски значно покращує продуктивність. Також важливо контролювати рівень фрагментації файлів бази даних та виконувати регулярну дефрагментацію.

У розподілених системах продуктивність залежить від швидкості мережі. Використання протоколу *TCP*/*IP* з оптимальними параметрами, налаштування пакету *SQL Server Network Configuration* та мінімізація затримок у мережі допомагають уникнути проблем при реплікації та інтеграції даних.

2.2 Оптимізація на рівні таблиць

2.2.1 Робота із архівними даними

Цей пункт є важливим для таблиць, які наповнюються даними протягом тривалого періоду часу і дані із таблиці використовуються за певний період часу в минулому.

До прикладу можемо розглянути таблицю *Sold_items*, якщо відкрити її характеристики то можна побачити (рис 2.1), що в ній зберігається більш як 605 міліонів записів.

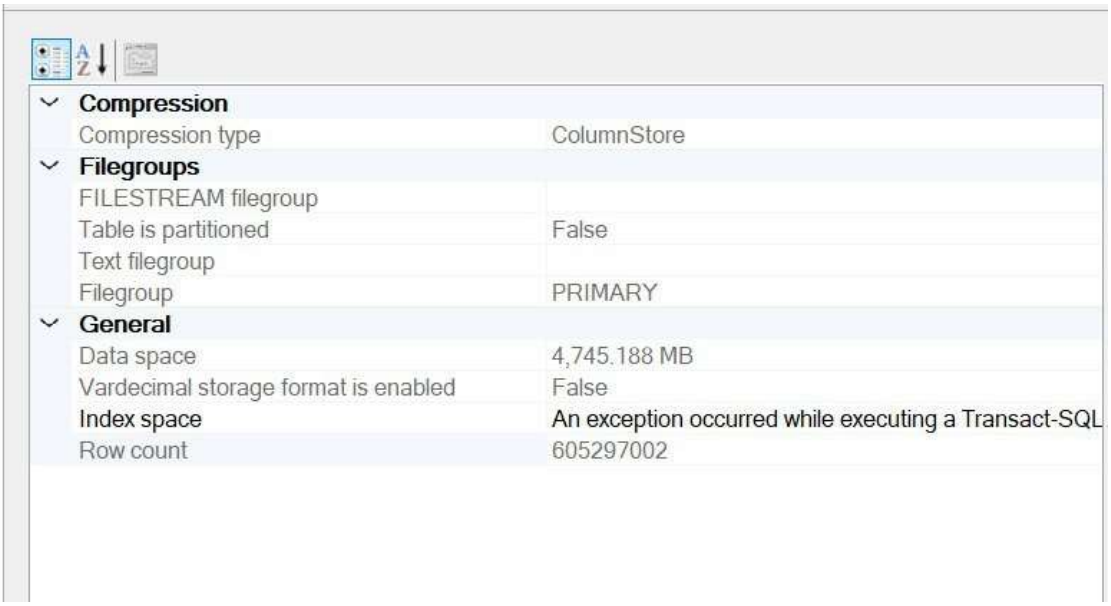

Рисунок 2.1 – Характеристики таблиці *Sold_items*

Запустимо *T-SQL* код щоб вияснити коли ця таблиця почала напвнюватися

SELECT MIN([*Updated*]) *as min_value FROM* [*Sold_items*]

Як результат маємо 25 жовтня 2018 року (рис 2.2).

Рисунок 2.2 – Мінімальна дата таблиці *Sold_items* по колонці *Updated*

Сформувавши запит до аналітиків, отримуємо відповідь, що для побудови звітності вони користуються даними за останні півроку (два квартали). Тобто, можна створити історичну таблицю, *Sold_items_history*, яка буде наповнюватися даними давністю від 9 місяців до 6 місяців, а в основній таблиці залишити тільки останні пів року.

Покрокова реалізація.

Створити архівну таблицю для збереження даних

CREATE TABLE Sold_items_archive (

ID BIGINT NOT NULL,

Updated DATETIME NOT NULL,

… -- інші колонки)

Визначення дати для архівування, оскільки аналітики працюють лише з останніми 6 місяцями, потрібно знайти всі записи.

Updated < *DATEADD*(*MONTH*, -6, *GETDATE*()):

DECLARE @ArchiveDate DATETIME =

DATEADD(*MONTH*, -6, *GETDATE*());

Перенесення даних в архів, використовуємо *INSERT INTO ... SELECT* для копіювання старих записів

INSERT INTO Sold_items_archive (*ID*, *Updated*)

SELECT ID, *Updated*

FROM Sold_items

WHERE Updated < *@ArchiveDate*;

Видалення архівованих записів з основної таблиці

DELETE FROM Sold_items

WHERE Updated < *@ArchiveDate*;

Створити та налаштувати *MSSQL* джоб, який буде раз в три місяці наповнювати архівну таблицю та видаляти дані із основної. Для створення джобу скористайтеся документацією.

Створення: *https*://*learn.microsoft.com*/*en-us*/*ssms*/*agent*/*create-a-job*

Налаштування: *https://learn.microsoft.com/en-us/ssms/agent/schedule-a-job?tabs=ssms*.

В крок джобу добавити процедуру (додаток 1).

За допомогою цього простого методу, тривальсть процесу побудови щоденної звітності, із використанням таблиці *Sold_items*, вдался скоротити із 6 годин 43 хв до 21 хв. Так як для звітності використовуються приблизно 4 міліони 300 тисяч записів замість 605 міліонів.

2.2.2 Партиціювання архівної таблиці як метод оптимізації продуктивності

Архівування даних дозволяє суттєво зменшити розмір основної таблиці та прискорити роботу запитів, проте архівна таблиця, яка накопичує історичні дані, з часом також може стати дуже великою. Для ефективного управління такими обсягами застосовується партиціювання — розбиття таблиці на логічні сегменти (партиції) за певним критерієм, найчастіше за датою.

Партиціювання забезпечує низку переваг. По-перше, воно дозволяє виконувати *partition elimination*, тобто обробляти лише ті партиції, які відповідають умовам запиту, що значно знижує кількість операцій введення-виведення. По-друге, адміністрування стає простішим: видалення або архівування цілих партицій замість окремих рядків зменшує час виконання операцій і ризик блокувань. По-третє, партиції можна рознести по різних файлових групах, що дає можливість балансувати навантаження на дискову підсистему [3].

Для реалізації партиціювання в *Microsoft SQL Server* необхідно створити *Partition Function*, яка визначає межі партицій, та *Partition Scheme*, що описує розміщення партицій у файлових групах. Наприклад, для архівної таблиці *Sold_items_archive*, яка містить дані з 2018 року, доцільно створити партиції за роками(див додаток 2). Це дозволить швидко виконувати аналітичні запити за конкретний рік або період.

Завдяки такій структурі запити з умовою по колонці *Updated* будуть працювати значно швидше, оскільки *SQL Server* оброблятиме лише потрібні

партиції. Крім того, видалення старих даних можна виконувати за допомогою команди *ALTER TABLE ... SWITCH PARTITION*, що набагато ефективніше, ніж масове видалення рядків.

В додатку 2 розписані *SQL*-скрипти для реалізації партиціювання архівної таблиці.

Таким чином, партиціювання архівної таблиці є потужним інструментом оптимізації продуктивності у системах з великими обсягами історичних даних. Воно забезпечує швидкість, гнучкість та простоту адміністрування, що робить його незамінним у корпоративних *DWH*-рішеннях.

2.2.3 Оптимізація структури таблиць шляхом винесення великих текстових колонок

У процесі роботи з великими таблицями одним із ключових факторів, що впливають на продуктивність, є структура таблиці. Використання колонок типу *NVARCHAR*(*MAX*) або інших *LOB*-типів (*Large Object Types*) у основній таблиці призводить до збільшення розміру сторінок, фрагментації та додаткових операцій введення-виведення. Це особливо критично для таблиць із сотнями мільйонів записів, таких як наша тестова таблиця *Sold_items*, яка містить 605 мільйонів рядків.

Колонка *Comment* у таблиці *Sold_items* має тип *NVARCHAR*(*MAX*) і використовується для зберігання текстових коментарів. Оскільки значення цієї колонки можуть бути великими, вони часто зберігаються у *LOB storage*, що уповільнює операції *SELECT*, *INSERT* та *UPDATE*. Крім того, такі колонки неефективно індексуються, а їх наявність у основній таблиці збільшує розмір сторінок і знижує ефективність кешування (дивитись додаток 3).

Для оптимізації пропонується винести колонки з великими значеннями в окрему таблицю, наприклад *Sold_items_big_values*.

Такий підхід дозволяє:

1. Зменшити розмір основної таблиці та прискорити роботу запитів, що не потребують великих текстових даних;

2. Знизити фрагментацію та навантаження на дискову підсистему;

3. Виконувати операції над великими текстами окремо, без впливу на основні транзакції.

Структура нової таблиці передбачає зберігання унікального ідентифікатора (*ID*) для зв'язку з основною таблицею та колонки *Comment* для великих текстових значень. Основна таблиця *Sold_items* після оптимізації міститиме лише ключові атрибути, що використовуються у більшості аналітичних запитів.

Таким чином, винесення колонок типу *NVARCHAR*(*MAX*) в окрему таблицю є ефективним методом оптимізації структури даних у великих системах. Це рішення забезпечує баланс між продуктивністю та гнучкістю, дозволяючи швидко обробляти основні дані та при цьому зберігати доступ до великих текстових значень.

Окрім винесення великих текстових колонок, слід звернути увагу на тип даних для ключових атрибутів. У нашій таблиці *Sold_items* колонка *ID* використовується як унікальний ідентифікатор. Якщо вона має текстовий тип (наприклад, *NVARCHAR* або *VARCHAR*), це негативно впливає на продуктивність, оскільки порівняння текстових значень є більш ресурсоємним, ніж числових. Крім того, текстові ключі займають більше місця в індексах, що збільшує їх розмір і час пошуку.

Рекомендовано використовувати числовий тип, наприклад *BIGINT* або *INT*, для колонки *ID*. Це забезпечує:

1. Швидші порівняння та *JOIN*-операції;
2. Менший розмір індексів, що прискорює пошук;
3. Оптимальне використання пам'яті;
4. Зменшення використання фізичного файлу підкачки.

Таким чином, оптимізація структури таблиці повинна включати не лише винесення великих колонок, але й правильний вибір типів даних для ключових атрибутів.

2.2.4 Застосування індексів у забезпеченні продуктивності роботи із таблицями

Індекси є одним із ключових механізмів оптимізації продуктивності в *Microsoft SQL Server*. Вони дозволяють значно скоротити час виконання запитів, зменшуючи кількість операцій введення-виведення за рахунок швидкого доступу до даних. Проте неправильне використання індексів може призвести до зворотного ефекту: надмірне навантаження на систему при вставці, оновленні та видаленні даних, а також збільшення обсягу пам’яті, необхідної для підтримки індексів.

Індекс у *Microsoft SQL Server* — це спеціальна структура даних, яка створюється для прискорення пошуку та доступу до рядків у таблиці або представленні. Він працює подібно до змісту книги: замість того, щоб переглядати всі сторінки (повне сканування таблиці), *SQL Server* використовує індекс для швидкого знаходження потрібних даних [4].

Основні характеристики індексу:

1. Індекс зберігає впорядковані значення однієї або кількох колонок таблиці;
2. Дозволяє оптимізатору запитів швидко знаходити рядки без повного сканування;
3. Може бути кластерним (впорядковує фізичні дані таблиці) або некластерним (зберігає окрему структуру з посиланнями на рядки);
4. Індекси зменшують час виконання *SELECT*-запитів, але збільшують витрати на *INSERT*, *UPDATE*, *DELETE*, оскільки потрібно оновлювати індекс.

Розглянумо наші таблиці та створимо для них індекси.

Sold_items (основна таблиця)

Характеристика: містить актуальні дані за останні 6 місяців після архівування. Основні запити бізнес-аналітиків — пошук по *ID* та фільтрація по даті *Updated*.

Рекомендовані індекси:

1. Кластерний індекс на *ID* - забезпечує швидкий доступ до рядків за унікальним ключем;

2. Некластерний індекс на *Updated* - для запитів, що вибирають дані за датою;

3. Фільтрований індекс для останніх 6 місяців - оптимізує роботу з актуальними даними.

Sold_items_archive (архівна таблиця з партиціями)

Характеристика: містить історичні дані з 2018 року, розбиті на партиції за роками. Основні запити — аналітика, агрегації, звіти.

Рекомендовані індекси:

1. Кластерний індекс на *Updated* - дозволяє виконувати *partition elimination*;

2. *Columnstore Index* для аналітики - значно прискорює запити типу *SUM*, *COUNT*, *GROUP BY*.

Sold_items_big_values (таблиця великих текстів)

Характеристика: містить колонку *Comment* типу *NVARCHAR*(*MAX*) і зв'язок по *ID*. Основні запити — пошук по тексту та *JOIN* з основною таблицею.

Рекомендовані індекси:

1. Кластерний індекс на *ID* - для швидкого об'єднання з основною таблицею;

2. *Full-Text Index* на *Comment* - для пошуку по великих текстових значеннях.

Додаткові типи індексів та їх застосування

Окрім базових кластерних, некластерних, фільтрованих та *columnstore* індексів, існують інші типи, які використовуються для специфічних завдань оптимізації продуктивності.

Унікальні індекси (*UNIQUE*)

Забезпечують гарантію унікальності значень у колонці або комбінації колонок. Використовуються для бізнес-ключів, які не є первинним ключем, але мають бути унікальними (наприклад, номер замовлення).

Composite Index (складені індекси)

Створюються на кілька колонок одночасно. Ефективні для запитів, що фільтрують або сортують дані за кількома критеріями, наприклад, за датою та статусом.

Covering Index - некластерний індекс, який включає додаткові колонки для покриття запиту без звернення до основної таблиці. Це дозволяє зменшити кількість операцій читання.

Hash Index - використовуються у пам'яті (*In-Memory OLTP*) для швидкого доступу до даних за ключем. Застосовуються у високонавантажених транзакційних системах із *memory-optimized* таблицями.

Spatial Index - призначені для колонок із географічними даними (типи *geometry* або *geography*). Оптимізують просторові запити, наприклад, пошук об'єктів у заданому радіусі.

XML Index - використовуються для колонок із типом *XML*, щоб прискорити пошук і обробку *XPath*-запитів у структурованих даних (дивитись додаток 4).

Для зручності вибору індексу можна скористатися таблицею 2.1.

Таблиця 2.1 – Види індексів та рекомендації із застосування

Індекс	**Тип**	**Рекомендації застосування**
1	2	3
Кластерний	*Clustered Index*	Для колонок з унікальними значеннями (*ID*), що визначають фізичний порядок даних.
Некластерний	*Nonclustered Index*	Для частих запитів по окремих колонках (наприклад, *Updated*).
Фільтрований	*Filtered Index*	Для оптимізації запитів по підмножині даних (наприклад, останні 6 місяців).
Columnstore	*Columnstore Index*	Для аналітичних запитів на великих обсягах даних (*SUM*, *COUNT*, *GROUP BY*).

Закінчення таблиці 2.1

1	2	3
Columnstore	*Columnstore Index*	Для аналітичних запитів на великих обсягах даних (*SUM*, *COUNT*, *GROUP BY*).
Full-Text	*Full-Text Index*	Для пошуку по великих текстових полях (наприклад, коментарі).
Унікальний	*Unique Index*	Для забезпечення унікальності бізнес-ключів (наприклад, номер замовлення).
Composite	*Composite Index*	Для запитів з умовами по кількох колонках одночасно (наприклад, дата + статус).
Covering	*Covering Index*	Для запитів, що вибирають кілька колонок, але фільтрують по одній (*INCLUDE* додаткових полів).
Hash	*Hash Index*	Для *In-Memory* таблиць у високонавантажених транзакційних системах.
Spatial	*Spatial Index*	Для оптимізації запитів по географічних даних (*geometry*, *geography*).
XML	*XML Index*	Для прискорення пошуку та обробки *XML*-даних за допомогою *XPath*-запитів.

Баланс індексації: продуктивність читання *vs* витрати на оновлення

Індексація є потужним інструментом оптимізації продуктивності, але її використання потребує балансу між швидкістю виконання запитів на читання та витратами на оновлення даних. Надмірна кількість індексів може призвести до зниження загальної ефективності системи.

Кількість індексів на таблиці:

1. Кластерний індекс: на таблиці може бути лише один, оскільки він визначає фізичний порядок зберігання рядків. Зазвичай створюється на колонці з унікальними значеннями (наприклад, первинний ключ).

2. Некластерні індекси: дозволено створювати багато (десятки), але кожен додатковий індекс збільшує витрати на операції *INSERT*, *UPDATE* та *DELETE*, оскільки всі індекси потрібно оновлювати.

Вплив на продуктивність:

1. Читання: більше індексів → швидші *SELECT*-запити, особливо для складних умов фільтрації;

2. Запис: більше індексів → повільніші операції модифікації даних, оскільки система повинна підтримувати узгодженість усіх індексів;

3. Дисковий простір: кожен індекс займає додаткове місце, що особливо критично для великих таблиць.

Рекомендації:

1. Створювати індекси лише для колонок, які активно використовуються у запитах;

2. Використовувати фільтровані індекси для підмножин даних (наприклад, актуальні записи);

3. Регулярно аналізувати статистику використання індексів за допомогою системних представлень (*DMV*) та видаляти неактивні;

4. Планувати індексацію разом із архітектурою *ETL*, щоб уникнути конфліктів при масових завантаженнях;

5. Виконувати регулярне обслуговування індексів (*REBUILD*, *REORGANIZE*, *UPDATE STATISTICS*) для підтримки їх ефективності.

Із остннім пунктом рекомендовано регулярно робити реіндексацію таблиць, частоста сильно залежить режиму роботи організації, яка користується СУБД. Для цього можна створити та налаштувати джоб на *MSSQL* який буде проводити індексацію.

В додатку 4 розписані *SQL* коди для роботи із індексами, також там наведений код процедури *Reindex_All_Tables* яка запускає процес реіндексації всіх таблиць.

2.2.5 Фрагментація даних у корпоративних сховищах: причини, наслідки та методи усунення

Фрагментація даних є однією з найпоширеніших проблем, що впливають на продуктивність корпоративних сховищ (*DWH*). Вона виникає внаслідок нерівномірного розміщення даних на сторінках пам'яті та призводить до збільшення часу виконання запитів. У сучасних системах, де обсяги даних вимірюються терабайтами, контроль фрагментації стає критично важливим для забезпечення стабільної роботи аналітичних процесів.

Причини виникнення фрагментації:

1. Часті оновлення та видалення рядків.

При виконанні операцій *UPDATE* та *DELETE* сторінки таблиці заповнюються нерівномірно, утворюючи «дірки» у структурі даних. Це призводить до логічної фрагментації, коли дані одного індексу розташовані на різних сторінках.

2. Використання типів змінної довжини (*VARCHAR*, *NVARCHAR*).

Зміна довжини значення може спричинити переміщення рядка на іншу сторінку, що збільшує рівень фрагментації. Особливо це актуально для колонок, які часто оновлюються.

Наслідки фрагментації:

1. Збільшення кількості сторінок для читання. При виконанні запиту система змушена обробляти більше сторінок, що збільшує кількість операцій вводу-виводу (*I*/*O*).

2. Зниження ефективності кешування. Фрагментовані сторінки займають більше місця в буферному пулі, зменшуючи його ефективність і збільшуючи навантаження на дискову підсистему.

3. Погіршення продуктивності індексів. Індекси, що мають високий рівень фрагментації, працюють повільніше, оскільки оптимізатор запитів не може ефективно використовувати їх структуру.

Методи усунення фрагментації:

1. Перебудова індексів (*REBUILD*).

Повне перебудування індексу усуває фрагментацію, але є ресурсомісткою операцією. Рекомендується виконувати її у періоди низького навантаження.

2. Реорганізація індексів (*REORGANIZE*).

Менш затратна операція, яка дефрагментує сторінки без повного блокування таблиці. Підходить для систем, що працюють у режимі 24/7.

3. Контроль типів даних.

Використання типів фіксованої довжини (*CHAR*) там, де це можливо, зменшує ризик фрагментації.

4. Моніторинг фрагментації.

Регулярний аналіз фізичного стану індексів за допомогою системних представлень (*sys.dm_db_index_physical_stats*) дозволяє визначити рівень фрагментації та прийняти рішення про оптимізацію.

Фрагментація даних є невід'ємною проблемою великих *DWH*-систем, але її можна ефективно контролювати за допомогою регулярного моніторингу та оптимізації індексів. Використання стратегій перебудови та реорганізації індексів, а також правильний вибір типів даних дозволяють значно підвищити продуктивність системи.

2.2.6 Статистика у *DWH*: значення, проблеми та шляхи оптимізації

У корпоративних сховищах даних статистика відіграє фундаментальну роль у процесі оптимізації запитів. Вона є джерелом інформації для оптимізатора, який на основі цих даних приймає рішення про вибір плану виконання. Статистика містить відомості про кількість рядків у таблиці, розподіл значень у колонках, частоту оновлень та інші параметри, що дозволяють системі оцінити вартість різних стратегій доступу до даних. Саме завдяки актуальній статистиці оптимізатор може обрати найефективніший шлях виконання запиту, використовуючи індекси або застосовуючи інші механізми прискорення.

Проблема виникає тоді, коли статистика стає застарілою. Це відбувається досить швидко у великих *DWH*-системах, де дані постійно змінюються внаслідок *ETL*-процесів, масових завантажень або видалень. Якщо оптимізатор використовує старі дані, він може прийняти рішення, яке здається правильним на

основі застарілої інформації, але фактично призводить до неоптимального плану виконання. Наприклад, замість використання індексу система може виконати повне сканування таблиці, що значно збільшує час обробки запиту та навантаження на дискову підсистему [5].

Ще однією причиною проблем зі статистикою є відсутність або неправильне налаштування автоматичного оновлення. У *SQL Server* існує механізм *AUTO_UPDATE_STATISTICS*, який дозволяє системі самостійно оновлювати статистику при значних змінах у даних. Однак у великих сховищах цей механізм може працювати неефективно або бути вимкненим, що призводить до накопичення помилок у планах виконання. Особливо критично це для аналітичних запитів, які працюють з мільйонами рядків і залежать від точності оцінок розподілу даних.

Наслідки застарілої статистики очевидні: зростає час виконання запитів, збільшується кількість операцій вводу-виводу, а система витрачає більше ресурсів на обробку даних. Це не лише впливає на швидкість аналітики, але й створює додаткове навантаження на сервер, що може призвести до деградації продуктивності всієї платформи.

Для вирішення проблеми необхідно впроваджувати комплексний підхід. По-перше, слід налаштувати автоматичне оновлення статистики, використовуючи параметри *AUTO_UPDATE_STATISTICS* та *AUTO_UPDATE_STATISTICS_ASYNC*. Асинхронне оновлення дозволяє уникнути блокувань під час виконання запитів, що особливо важливо для систем, які працюють у режимі 24/7. По-друге, після масових *ETL*-операцій варто виконувати ручне оновлення статистики для ключових таблиць, використовуючи команду *UPDATE STATISTICS* або процедуру *sp_updatestats* для всієї бази. У випадках, коли потрібна максимальна точність, рекомендується застосовувати параметр *FULLSCAN*, який забезпечує повне сканування таблиці при оновленні статистики (дивитись додаток 5).

AUTO_UPDATE_STATISTICS і *AUTO_UPDATE_STATISTICS_ASYNC* у *SQL Server* відповідають за автоматичне оновлення статистики, яка використовується

оптимізатором для вибору найкращого плану виконання запитів. Статистика містить інформацію про розподіл значень у стовпцях і допомагає оцінювати кардинальність, тобто кількість рядків, які поверне запит. Якщо статистика застаріла, плани можуть бути неефективними, що призводить до погіршення продуктивності.

Коли ввімкнено *AUTO_UPDATE_STATISTICS*, *SQL Server* перевіряє актуальність статистики під час компіляції запиту. Якщо вона застаріла, сервер оновлює її синхронно, тобто запит чекає завершення оновлення, а потім компілюється з новими даними. Це гарантує оптимальний план, але може затримати перший запит, який ініціював оновлення. Поріг застарівання визначається кількістю змінених рядків у таблиці: класично це близько двадцяти відсотків плюс п’ятсот рядків, але у сучасних версіях *SQL Server* поріг динамічний і зменшується для великих таблиць, що особливо важливо для сховищ даних.

AUTO_UPDATE_STATISTICS_ASYNC працює інакше. Якщо статистика застаріла, запит компілюється і виконується зі старими даними, а оновлення запускається у фоні. Це усуває затримку компіляції, але перші запити можуть отримати неідеальний план. Такий режим корисний у системах, де важлива пропускна здатність і небажані паузи, наприклад, у великих аналітичних середовищах, але він ризикований для критичних запитів, які мають виконуватися максимально ефективно.

У великих *DWH*-системах обидва параметри зазвичай залишають увімкненими, але після масових завантажень даних варто явно оновлювати статистику, щоб уникнути роботи на застарілих даних. Це можна зробити через *sp_updatestats* або команду *UPDATE STATISTICS* для найбільш важливих таблиць, іноді з опцією *FULLSCAN* для максимальної точності. Якщо ранкові звіти стартують одразу після *ETL*, краще вимкнути асинхронний режим або виконати контрольне оновлення статистики перед запуском звітів. Для дуже великих таблиць важливо використовувати рівень сумісності 130 і вище, щоб працювали динамічні пороги оновлення. Крім того, у випадках нерівномірного

розподілу значень корисно створювати фільтровані статистики, які значно покращують точність оцінок кардинальності.

Загалом, синхронне оновлення забезпечує стабільність і точність планів, але може викликати затримки, тоді як асинхронне зменшує латентність, проте допускає тимчасові компроміси у продуктивності. У змішаних середовищах, де є *OLTP* і аналітика, зазвичай залишають синхронний режим для транзакційних баз, а асинхронний — для аналітичних реплік. У будь-якому випадку, контрольне оновлення статистики після великих *ETL*-процесів залишається найкращою практикою для стабільної роботи системи. В додатку 5 розписані коди для роботи із статиститикою .

Моніторинг стану статистики також є важливим елементом. Для цього можна використовувати системні представлення, такі як *sys.stats*, що дозволяють отримати інформацію про наявні статистичні об’єкти та їх актуальність. Регулярний аналіз цих даних допомагає вчасно виявляти проблеми та приймати рішення про оптимізацію.

Основним джерелом інформації є *sys.stats*, за допомогою цього інструмента можна отримати метадані для кожного об’єкта статистики, включаючи назву, ідентифікатор, ознаки автоматичного створення та заборони автооновлення. Тут можна визначити, чи була статистика створена користувачем або автоматично, а також чи має вона фільтр. Однак *sys.stats* не показує дату останнього оновлення або кількість змін у даних, тому для цього використовується *DMV sys.dm_db_stats_properties*. Це представлення надає критично важливу інформацію: час останнього оновлення, кількість рядків, обсяг вибірки та лічильник модифікацій, який показує, наскільки статистика застаріла.

Щоб зрозуміти, які колонки входять до конкретної статистики, використовується представлення *sys.stats_columns*. Воно показує порядок колонок у статистиці, що важливо для багатоколонкових об’єктів, адже гістограма створюється лише для першої колонки. Для аналізу розподілу значень у провідній колонці застосовується *DMV sys.dm_db_stats_histogram*, яке повертає

діапазони, частоти та кількість унікальних значень. Це дозволяє глибше оцінити селективність і виявити проблеми зі *skew*.

Індексні статистики тісно пов'язані з представленням *sys.indexes*, адже кожен індекс має власну статистику. Через це важливо перевіряти, чи відповідають індексні статистики реальним умовам фільтрації у запитах. Додатково для роботи з колонками використовуються *sys.columns* і *sys.types*, а для контролю налаштувань бази даних — *sys.databases*, де можна побачити стан опцій *AUTO_CREATE_STATISTICS*, *AUTO_UPDATE_STATISTICS* та *AUTO_UPDATE_STATISTICS_ASYNC*.

Практичне застосування цих представлень полягає у кількох завданнях. По-перше, аудит застарілих статистик: за допомогою *sys.stats* у поєднанні з *sys.dm_db_stats_properties* можна швидко знайти об'єкти з великим лічильником модифікацій або дуже давньою датою оновлення. По-друге, контроль параметра *NORECOMPUTE*, який забороняє автооновлення і може призвести до деградації продуктивності. По-третє, робота з фільтрованими статистиками, що особливо корисно для великих таблиць із нерівномірним розподілом даних, таких як архіви або фактичні таблиці у *DWH*. Нарешті, для колонок типу *nvarchar*(*max*) та *JSON* доцільно створювати обчислювані колонки і додавати на них індекси, щоб статистика була релевантною.

Для прикладу, щоб отримати повний звіт про статистики у таблицях *Sold_items*, *Sold_items_archive* та *Sold_items_big_values*, можна використати запит, який об'єднує *sys.stats* і *sys.dm_db_stats_properties*, показуючи назву статистики, дату останнього оновлення, кількість рядків та лічильник модифікацій. Аналогічно, через *sys.stats_columns* можна перевірити, які саме колонки охоплює кожна статистика, а через *sys.dm_db_stats_histogram* — проаналізувати розподіл значень у провідній колонці.

Таким чином, підтримка актуальної статистики є критичною умовою для ефективної роботи *DWH*. Вона забезпечує коректну роботу оптимізатора, знижує навантаження на систему та гарантує стабільну продуктивність аналітичних процесів. Ігнорування цього аспекту призводить до значних втрат у швидкості

обробки даних і може негативно вплинути на бізнес-процеси, що залежать від оперативної аналітики.

2.2.7 Блокування та конкуренція в *SQL Server*: причини, наслідки та способи оптимізації

У системах управління базами даних блокування є механізмом забезпечення цілісності даних при одночасному доступі. *SQL Server* застосовує різні рівні блокувань — від рядків і сторінок до таблиць — щоб гарантувати узгодженість транзакцій. Проте у великих сховищах даних та під час масових *ETL*-операцій блокування можуть стати джерелом серйозних проблем продуктивності.

Високий рівень блокувань виникає при операціях, що охоплюють великі обсяги даних, наприклад, масові вставки, оновлення або видалення. У таких випадках *SQL Server* може ескалувати блокування з рівня рядків до сторінок або навіть таблиць, щоб зменшити накладні витрати на управління блоками. Ескалація блокувань призводить до того, що інші транзакції, які намагаються читати або змінювати дані в тій самій таблиці, змушені чекати, що створює затримки і знижує пропускну здатність системи.

Ще серйознішою проблемою є взаємні блокування (*deadlocks*). Вони виникають, коли дві або більше транзакцій чекають на ресурси, заблоковані одна одною, утворюючи замкнене коло. Наприклад, одна транзакція блокує таблицю *Sold_items*, а інша — *Sold_items_archive*, і кожна з них намагається отримати доступ до ресурсу, який утримує інша. *SQL Server* виявляє такі ситуації і примусово завершує одну з транзакцій, але це призводить до помилок у процесах *ETL* і може порушити узгодженість даних.

Для зниження ризику блокувань і *deadlocks* у *DWH*-середовищах застосовують кілька стратегій. По-перше, важливо правильно налаштовувати рівні ізоляції транзакцій. Використання режиму *READ COMMITTED SNAPSHOT* або *SNAPSHOT ISOLATION* дозволяє уникнути блокувань при читанні, оскільки запити працюють із версіями даних у *tempdb*, а не блокують рядки. По-друге, масові операції слід розбивати на менші батчі, щоб зменшити час утримання

блокувань і уникнути ескалації. По-третє, *ETL*-процеси потрібно проектувати так, щоб уникати одночасного доступу до одних і тих самих таблиць у режимі запису, наприклад, через розклад або розподіл навантаження.

Додатково варто використовувати індекси для прискорення пошуку рядків, що оновлюються, і уникати повних сканувань, які блокують великі ділянки таблиці. Для діагностики проблем блокувань *SQL Server* надає інструменти, такі як *sys.dm_tran_locks*, *sys.dm_exec_requests* і трасування *deadlocks* через *Extended Events*. Аналіз цих даних дозволяє виявити вузькі місця і оптимізувати транзакції.

У контексті наших таблиць *Sold_items* і *Sold_items_archive* особливо важливо контролювати блокування під час архівування даних. Масове перенесення записів кожні три місяці може створювати конфлікти з аналітичними запитами. Рішенням може бути використання *TABLOCK* для швидких вставок у архів, але лише в період, коли немає конкурентних запитів, або застосування механізмів *ONLINE* для індексних операцій, щоб мінімізувати вплив на читання.

Таким чином, управління блокуваннями і конкуренцією є ключовим аспектом оптимізації продуктивності *DWH*. Правильне налаштування ізоляції, розбиття операцій на батчі, використання версіонування і грамотне планування *ETL* дозволяють уникнути критичних проблем і забезпечити стабільну роботу системи. В додатку 7 приклад процедури яка буде займатися моніторингом блокувань і відправляти інформацію про проблему на електронну пошту.

2.2.8 Тригери та обмеження

Тригери та обмеження є важливими механізмами забезпечення цілісності даних у реляційних базах, але їх надмірне використання може суттєво вплинути на продуктивність, особливо у великих системах із масовими операціями вставки та оновлення. У корпоративних сховищах даних, де *ETL*-процеси обробляють мільйони рядків, ці витрати стають критичними.

Тригери виконуються автоматично при настанні певних подій — наприклад, *AFTER INSERT*, *UPDATE* або *DELETE*. Коли таблиця має кілька тригерів, кожна операція викликає додаткові обчислення, що збільшує час

транзакції. Якщо тригери містять складну бізнес-логіку, звернення до інших таблиць або виконання агрегатів, навантаження на систему зростає експоненційно. У випадку масових завантажень даних це може призвести до блокувань, ескалації блоків і навіть дедлоків, оскільки тригери виконуються в контексті тієї ж транзакції.

CHECK-обмеження також впливають на продуктивність, адже кожен *INSERT* або *UPDATE* повинен пройти перевірку умов. Якщо обмеження прості, наприклад перевірка діапазону значень, вплив мінімальний. Але складні вирази з підзапитами або зверненням до функцій значно збільшують час обробки. У великих *DWH*, де дані завантажуються пакетами, такі обмеження можуть стати вузьким місцем, особливо якщо вони застосовуються до колонок із високою частотою змін [6].

Для оптимізації роботи системи рекомендується мінімізувати кількість тригерів і складність їх логіки. Частину перевірок можна перенести на рівень *ETL*, де вони виконуються до завантаження в базу, що зменшує навантаження на транзакції.

Аналогічно, складні *CHECK*-обмеження варто замінювати на більш прості або реалізовувати через контроль у процесі трансформації даних. Якщо тригери необхідні для аудиту або логування, доцільно використовувати асинхронні механізми, наприклад *Service Broker* або зовнішні черги, щоб не блокувати основні операції.

У контексті наших таблиць *Sold_items* і *Sold_items_archive* важливо уникати тригерів, які виконують складні обчислення під час архівування, адже це може значно збільшити час перенесення даних. Для таблиці *Sold_items_big_values* перевірки на рівні *CHECK* для *JSON*-структур або великих текстових полів краще реалізовувати на етапі *ETL*, а не в базі.

Детальніше робота із тригерами буде розглянута у розділі про *Matillion ETL*.

2.2.9 Фінансова модель *MSSQL* на *AWS EC*2 та вплив оптимізації на вартість

Розміщення *Microsoft SQL Server* на *AWS* у середовищі *EC*2 передбачає кілька основних компонентів витрат. Перше — це вартість обчислювальних ресурсів, тобто самого *EC*2-інстансу. Ціна залежить від типу інстансу (кількість *vCPU*, обсяг оперативної пам'яті), регіону та режиму оплати (*On-Demand*, *Reserved* або *Spot*). Чим потужніший сервер, тим вища вартість за годину роботи. Другий компонент — ліцензія на *SQL Server*. Якщо використовується модель *License Included*, то плата за ліцензію входить у тариф *EC*2, і вона зростає пропорційно до кількості ядер. Якщо застосовується *Bring Your Own License* (*BYOL*), то витрати залежать від вашого ліцензійного пакета.

Третій важливий елемент — сховище даних. *EC*2 використовує *EBS* (*Elastic Block Store*) для зберігання бази. Вартість залежить від типу диска (*gp*3, *io*1, *st*1), обсягу та кількості *IOPS*. Чим більше даних і чим вищі вимоги до швидкодії, тим дорожче коштує сховище. Крім того, є витрати на резервні копії (*Snapshots*), які також зберігаються в *S*3, і на передачу даних між зонами або при реплікації.

Оптимізація таблиць і процесів безпосередньо впливає на ці витрати. Наприклад, зменшення розміру таблиць через архівування даних або винесення великих текстових полів у окремі структури знижує обсяг *EBS*, а отже, і вартість зберігання. Використання індексів і актуальної статистики скорочує час виконання запитів, що зменшує навантаження на *CPU* і пам'ять, а це дозволяє працювати на менш потужному інстансі або уникати пікових перевитрат. Зменшення блокувань і оптимізація *ETL*-процесів знижує тривалість транзакцій, що також впливає на загальну кількість годин роботи сервера під високим навантаженням.

Ще один аспект — резервні копії. Якщо база займає сотні гігабайт, кожен *snapshot* збільшує витрати на *S*3. Архівування старих даних у дешевші сховища або використання компресії зменшує ці витрати. Крім того, оптимізація структури таблиць і індексів зменшує кількість *IOPS*, що особливо важливо для *io*1/*io*2 дисків, де плата йде за кожну операцію.

Таким чином, фінансова ефективність *MSSQL* на *AWS* залежить не лише від вибору інстансу, але й від архітектури бази та якості її оптимізації. Грамотне управління обсягами даних, індексами, статистикою та *ETL*-процесами дозволяє суттєво знизити витрати на обчислювальні ресурси, сховище та ліцензії, зберігаючи при цьому високу продуктивність системи.

Як приклад, аналітична таблиця (рис. 2.3) з розрахунком місячних витрат для *MSSQL* на *AWS EC*2 до та після оптимізації.

Сценарій	*До оптимізації*	*Після оптимізації*
EC2 ($)	367,92	183,96
Ліцензія ($)	119,72	59,86
EBS ($)	160	80
IOPS ($)	30	15
Snapshots ($)	100	50
Разом ($)	777,64	388,82

Рисунок 2.3 – Скріншот таблиці порівняння витрат *MSSQL* на *AWS EC*2 до та після оптимізації

Висновки до розділу

У процесі дослідження було розглянуто комплекс питань, що стосуються оптимізації продуктивності корпоративних сховищ даних на базі *Microsoft SQL Server* у хмарному середовищі *AWS*. Аналіз показав, що ефективність роботи *DWH* залежить не лише від обчислювальних ресурсів, але й від архітектури бази, організації даних та застосування механізмів оптимізації на рівні СУБД.

Першим важливим аспектом стала робота з архівними даними. Зберігання великих обсягів історичної інформації в основних таблицях призводить до зростання часу виконання запитів, збільшення блокувань та витрат на обчислювальні ресурси. Використання архівних таблиць і винесення застарілих даних дозволяє суттєво зменшити навантаження на систему, скоротити обсяг активних даних і підвищити швидкодію аналітичних запитів. Додатково застосування партиціонування архівних таблиць забезпечує можливість роботи з

великими наборами даних без значних витрат на сканування, що особливо важливо для *ETL*-процесів.

Оптимізація структури таблиць шляхом винесення великих текстових колонок у окремі об’єкти також продемонструвала свою ефективність. Це рішення знижує розмір основних таблиць, прискорює операції читання та оновлення, а також зменшує кількість *I*/*O*-операцій. У поєднанні з правильним застосуванням індексів можна досягти значного приросту продуктивності. Індекси залишаються ключовим інструментом оптимізації, але їх надмірне використання або неправильне налаштування може призвести до зворотного ефекту. Тому важливо балансувати між швидкістю вибірки та витратами на підтримку індексів при вставках і оновленнях.

Окрему увагу було приділено проблемі фрагментації даних. У корпоративних сховищах фрагментація виникає через часті операції вставки та видалення, що призводить до зниження ефективності використання пам’яті та збільшення часу доступу до даних. Регулярне обслуговування індексів, включаючи перебудову або реорганізацію, дозволяє мінімізувати негативний вплив фрагментації та підтримувати стабільну продуктивність.

Статистика у *DWH* є критично важливою для роботи оптимізатора запитів. Застарілі статистики призводять до помилкових оцінок кардинальності та вибору неефективних планів виконання. Автоматичне оновлення статистики, а також контрольне оновлення після масових *ETL*-операцій забезпечують точність планування та стабільність роботи системи. Використання фільтрованих статистик для актуальних діапазонів даних додатково підвищує ефективність запитів.

Не менш важливим є управління блокуваннями та конкуренцією. Масові операції в *DWH* часто викликають ескалацію блокувань і дедлоки, що негативно впливає на пропускну здатність системи. Застосування режимів ізоляції на основі версіонування, розбиття операцій на батчі та грамотне планування *ETL*-процесів дозволяють мінімізувати ці ризики. Аналіз показав, що навіть невеликі зміни в

логіці обробки даних можуть суттєво знизити кількість конфліктів і підвищити стабільність роботи.

Тригери та обмеження, хоча й забезпечують цілісність даних, створюють додаткове навантаження на систему. Складні *CHECK*-умови та багаторівневі тригери значно збільшують час виконання транзакцій, особливо при масових вставках. Перенесення частини перевірок на рівень *ETL* або використання асинхронних механізмів дозволяє зберегти баланс між цілісністю та продуктивністю.

Фінансовий аналіз показав, що оптимізація бази даних має прямий економічний ефект. Зменшення обсягу даних, кількості *IOPS* і часу виконання запитів дозволяє використовувати менш потужні *EC*2-інстанси, знижує витрати на ліцензії *SQL Server* та сховище *EBS*. Розрахунки продемонстрували, що комплексна оптимізація може скоротити витрати на 40–50%, що є суттєвим аргументом на користь впровадження таких рішень. Додаткові можливості економії забезпечують моделі *Reserved Instances*, *Savings Plans* та оптимізація типів дисків.

Узагальнюючи, можна стверджувати, що оптимізація *DWH* у хмарному середовищі — це багатогранний процес, який охоплює технічні, організаційні та фінансові аспекти. Вона вимагає системного підходу, що включає архітектурні рішення, налаштування СУБД, оптимізацію *ETL*-процесів і управління ресурсами. Результатом є не лише підвищення продуктивності та стабільності роботи системи, але й значне зниження витрат, що робить оптимізацію стратегічно важливим завданням для будь-якої компанії, яка працює з великими обсягами даних.

РОЗДІЛ 3
ДОСЛІДЖЕННЯ МЕТОДІВ ОПТИМІЗАЦІЇ РОБОТИ ІЗ *SNOWFLAKE DB* ІЗ ЗАСТОСУВАННЯМ *MATILLION ETL*

3.1 Оптимізація витрат і ресурсів при роботі з великими *DWH*

Оптимізація витрат і ресурсів у великих корпоративних *DWH*-системах є багатовимірним завданням, у якому технічні рішення на рівні архітектури, моделювання даних, *ETL*/*ELT*-стратегії, налаштування обчислювальних кластерів, форматів зберігання та політик виконання запитів безпосередньо впливають на фінансову ефективність. У контексті *Snowflake* як платформи з розділенням обчислень і зберігання, а також у зв'язці з *Matillion ETL* як *cloud-native* інструментом для побудови конвеєрів, оптимізація означає мінімізацію «*compute cost*» при збереженні цільових *SLO* на продуктивність і узгодженість даних, а додатково — спрощення операційної підтримки в мультиплатформенному середовищі, де співіснують *MSSQL*, *Postgres* і об'єктні сховища.

Починати варто з виокремлення трьох рівнів оптимізації: на рівні архітектури (вибір розміру та режиму роботи *Virtual Warehouse*, політики авто-призупинення, розклад виконання та паралельність), на рівні даних (кластеризація, партиціювання логічне через дизайн ключів, формати файлів, компресія, колонові типи та семантика *VARIANT*/*JSON*), і на рівні *ETL*/*ELT* (*pushdown*, мікро-батчі проти великих батчів, декомпозиція трансформацій, контроль вузьких місць шляхом розбиття на сегменти). Взаємодія цих трьох рівнів визначає кінцеву ціну та час, а правильний баланс у конкретному робочому навантаженні дозволяє досягти лінійного масштабування з контрольованою вартістю.

3.1.1 Віртуальні сховища у *Snowflake* як керований важіль вартості

Оскільки плата за обчислення у *Snowflake* залежить від розміру та тривалості роботи *Virtual Warehouse*, налаштування складу сховищ і їхнього життєвого циклу стає центральним механізмом оптимізації. Для робочих навантажень «*extract-load-transform*» через *Matillion* доречно вводити профілі складів: невеликий склад для оркестрації та метаданих (наприклад, *X-Small* для керування джобами, контролю станів та легких *SQL*-операцій), середній або великий склад для масових завантажень у вікна низького трафіку (*Medium*/*Large* для *batch window*), і окремий аналітичний склад для звітних запитів із кешем результатів у бізнес-години (*Small*/*Medium* з агресивним *auto-suspend*). Зміна профілю протягом доби дозволяє знизити пік витрат без погіршення *SLA*: денні аналітичні запити працюють на більш енергоефективному складі, а нічні партії *ETL* отримують більше *CPU*/*IO* для скорочення *wall time* і, відповідно, зменшення сумарної тривалості нарахування кредитів.

Режим авто-призупинення та авто-відновлення є базовою практикою. Правильно підібраний час простою перед призупиненням (наприклад, 60–120 секунд) зменшує роздробленість кредитів і запобігає надмірному старт/стоп циклу на мікро-операціях, водночас не утримуючи склад у неактивні періоди. Якщо конвеєри *Matillion* створюють короткі серії запитів із паузами між кроками, корисно групувати трансформації у більші атомарні блоки, щоб уникати частих «холодних стартів», або ж встановити трохи довший інтервал до призупинення на критичних пайплайнах, де гарантується наступний запит через десятки секунд.

3.1.2 Кешування результатів запитів і повторне використання робочих артефактів

Кеш результатів у *Snowflake* для детермінованих запитів із незмінними вхідними таблицями може суттєво скорочувати витрати на аналітичні селекти, особливо в *BI*-шарі. Задля максимального ефекту необхідно забезпечити стабільність тексту запиту (без «*SELECT* *», без зайвих варіацій пробілів і коментарів у критичних місцях) та контроль оновлення базових таблиць. У

продакшн-середовищі, де конвеєри *ETL* регулярно оновлюють таблиці, використовують прийоми матеріалізації проміжних результатів у тимчасові таблиці із семантикою «*append-only*» у межах одного завантаження; повторні звернення впродовж вікна завдань отримують кешовані результати, тоді як кінцевий мерж або свіжий селект в іншу таблицю не руйнує кеш для попередніх кроків [7].

У *Matillion* корисно повторно використовувати вихідні набори у межах потоку, уникати дублюючих трансформацій та зберігати проміжні підсумки у тимчасових таблицях з чіткими *TTL*. Це зменшує кількість ітерацій важких агрегацій та джоінів і скорочує загальний час роботи складу. Практика «*cache warm-up*» перед піковими звітними сесіями — короткий прогін ключових селектів — також допомагає бізнес-користувачам отримувати швидші відповіді без збільшення складу [8].

3.1.3 Кластеризація та логічне партиціювання великих таблиць

Для надвеликих таблиць, таких як архівна «*Sold_items_archive*» з сотнями мільйонів рядків, структура даних має вирішальне значення. Хоча *Snowflake* приховує фізичне партиціювання за мікропартіціями, правильна кластеризація за стовпцями фільтрації (наприклад, дата транзакції, діапазони *ID* або статуси) покращує «*data pruning*» і зменшує обсяг сканованих мікропартіцій. Це безпосередньо впливає на *compute cost*: менше прочитаних байтів — менше *CPU* та *IO*, швидше завершення запитів.

У нашому контексті логічне партиціювання за датою продажу або кварталом доцільне не лише для читання; воно задає ритм архівації та полегшує інкрементальні процеси *ETL*. Якщо «*Sold_items*» містить дані за останні 6 місяців, а все старше регулярно переноситься в «*Sold_items_archive*», варто забезпечити, щоб архівна таблиця підтримувала кластеризацію за ключем «*sale_date*» (за потреби — композитний ключ із «*item_category*» чи «*region_id*»), а основна — залишалася компактною та орієнтованою на оперативну звітність. Для підтримання ефективності кластеризації періодично слід виконувати «*recluster*» на архівній таблиці, особливо після великих апендів. Це робиться

адаптивно: не частіше, ніж необхідно для збереження цільових показників «*average overlap*» і «*depth*» мікропартіцій [9].

Приклад створення кластеризації для архівної таблиці може виглядати як просте визначення кластерного ключа, а не класичні індекси, відсутні у *Snowflake*. Для ілюстрації наведемо фрагмент *SQL*, який визначає кластеризацію за датою та категорією:

SQLALTER TABLE analytics_db.sales.Sold_items_archive CLUSTER BY (*sale_date*, *item_category*);*Show more lines*

Після великих завантажень доцільно запускати керований рекластер на обмеженому вікні даних, якщо відомо, які діапазони щойно додані:

SQLALTER TABLE analytics_db.sales.Sold_items_archive RECLUSTER PARTITION WHERE sale_date >= '2024-01-01';*Show more lines*

3.1.4 Формати зберігання, типи даних і *VARIANT*/*JSON*

Робота з великими текстовими полями виноситься в окрему таблицю «*Sold_items_big_values*», де зберігається «*Commnt*» та «*API*» у форматі *JSON*. Для *Snowflake* тип *VARIANT* є природним носієм напівструктурованих даних, і його використання замість великих *NVARCHAR* значно зменшує обсяг зберігання завдяки колонному представленню та внутрішній компресії (дивитись додаток 8). На рівні *ETL* краще виконувати парсинг *JSON* через нативні функції та *UDF* у *pushdown*-режимі. Це знімає навантаження з *Matillion*-агентів і використовує оптимізований двигун *Snowflake*.

При інтеграції із зовнішніми джерелами слід спиратися на колонові формати (*Parquet*, *ORC*) і на увімкнену компресію при завантаженні з об’єктних сховищ. Це суттєво скорочує обсяг переданих байтів і час завантаження. Якщо використовується *COPY INTO* з *S*3 або іншим об’єктним сховищем, задавання параметрів *FILE_FORMAT* із компресією забезпечує кращу пропускну спроможність за меншу ціну. Наведемо схему визначення формату та завантаження:

SQLCREATE OR REPLACE FILE

FORMAT analytics_db.ff_parquet

TYPE = 'PARQUET'COMPRESSION = 'AUTO';COPY INTO analytics_db.sales.Sold_itemsFROM

@analytics_db.stage.s3_sales_parquet/FILE_FORMAT = (FORMAT_NAME = analytics_db.ff_parquet)ON_ERROR = 'CONTINUE'; Show more lines

Перевага *Parquet* у колонному зберіганні полягає в тому, що під час селектів по підмножині колонок *Snowflake* читає тільки необхідні колонки, зменшуючи *IO*. У зв'язці з кластеризацією це дає відчутний виграш у часі та вартості.

3.1.5 *ETL*/*ELT*-стратегія: батчі проти мікро-батчів, сегментація й паралельність

Для масових навантажень критичною є сегментація. Замість одного гігантського кроку, який сканує сотні мільйонів рядків, процес розбивається на незалежні сегменти за діапазонами дат або *ID*; кожен сегмент обробляється окремим потоком у *Matillion* з *pushdown* у *Snowflake*. Це дозволяє задіяти паралельність, зменшити час стінки та, головне, уникати ситуацій, коли один запит утримує великий склад надто довго. Правильне налаштування рівня паралельності має враховувати підсистему блокувань і пропускну здатність зовнішніх джерел; загалом краще стартувати з помірної паралельності (наприклад, 4–8 потоків на *Medium*-складі) і емпірично підвищувати до рівня, на якому приріст часу перестає бути лінійним.

Мікро-батчі виправдані, коли дані надходять потоково і існує вимога *low-latency* оновлень у вітрині «*Sold_items*». У такому випадку *SLA* на латентність важливіший за мінімізацію кредитів, проте навіть тут можна економити: малі склади, агрегація інкрементів, застосування матеріалізованих проміжних таблиць, що перевикористовуються для звітності впродовж робочої години [10].

Для великих нічних батчів, які включають пересписування розділів архіву, краще застосувати стратегію «*bulk-insert* + *merge*». Завантаження нових сегментів до стаджингових таблиць із наступним *MERGE INTO* у основне сховище дозволяє оптимізатору виконати операцію більш ефективно, ніж

послідовні рядкові *UPSERT*. Приклад стандартного *MERGE* для оновлення «*Sold_items*» наведено нижче:

SQLMERGE INTO analytics_db.sales.Sold_items AS tgtUSING analytics_db.stage.Sold_items_new AS srcON tgt.ID = src.IDWHEN MATCHED THEN UPDATE SET tgt.sale_date = src.sale_date, tgt.qty = src.qty, tgt.amount = src.amountWHEN NOT MATCHED THEN INSERT (*ID*, *sale_date*, *qty*, *amount*)*VALUES* (*src.ID*, *src.sale_date*, *src.qty*, *src.amount*);*Show more lines*

На архівній таблиці краще уникати частих оновлень у місці; замість цього — додавати нові порції даних і виконувати періодичні рекластеризації. Це більш дружнє до мікропартіцій і стабілізує показники відсічення.

Оптимізація запитів: селективність, фільтри, проєкції й відсутність «*SELECT* *»

Оптимізація запитів є «дрібною» технікою, яка разом формує суттєвий ефект. Відмова від «*SELECT* *» на користь чітких проєкцій колонок зменшує *IO* й декомпресію. Фільтри мають бути максимально селективними: використання діапазонів дат, точних ключів або вузьких категорій задає оптимізатору повноцінну можливість відсікати мікропартіції. Перетворення складних *JOIN* на двоетапні маневри з попередньою агрегацією менших таблиць часто дає виграш у продуктивності, особливо коли великий факт має обмежену корисну підмножину приєднуваних атрибутів.

Матеріалізовані представлення доречні для регулярних важких агрегацій; їхнє автоматичне оновлення потребує оцінки вартості, але для стабільних вітрин із щоденними оновленнями це може окупитися з надлишком, оскільки запити *BI* працюють із вже порахованими наборами.

3.1.6 Керування витратами: монітори, квоти, теги й спостережність

Практика керування витратами включає запровадження «*Resource Monitors*», які сигналізують і, за потреби, призупиняють склади або відсікають перевищення кредитів для середовищ дев/тест/сандбокс. Такі монітори узгоджуються з календарем *ETL*, щоб уникати несподіваних зупинок під час продакшн-завантажень, проте вони дисциплінують витрати в нефункціональних

середовищах. Додатково корисні «*cost tags*», що дозволяють пов'язати витрати з бізнес-юнітами й конвеєрами *Matillion*, щоб надалі робити аналіз ефективності за напрямками. На практиці дані телеметрії з *Information Schema* та *Account Usage* збираються у вітрину «*usage_analytics*», де легко будувати щоденні та щотижневі звіти з фокусом на тривалості роботи складів, розмірі, типах запитів і співвідношенні часу до обсягу сканованих байтів [11].

Через *Matillion* варто забезпечити прозорі «*run parameters*» і явне логування роботи кожного джоба: час старту, розмір складу, кількість потоків, обсяг оброблених рядків. Це дає можливість виконувати *A*/*B*-порівняння конфігурацій і знаходити локалі максимумів для конкретних конвеєрів: наприклад, встановити, що для сегментації архіву на квартали паралельність у 6 потоків на *Medium* дає найкраще співвідношення часу до кредитів, тоді як 10 потоків переходять у «перегони» за *IO* та не зменшують загальний час.

Інкрементальні завантаження — ключ до економії на великих фактах. Замість повного перерахунку вітрин щодня, відстежуються лише дельти. Це може бути реалізовано через «*high watermark*» за датою модифікації або через *CDC* із журналів змін у джерелах *MSSQL*/*Postgres*. У *Snowflake MERGE* із дельтою працює ефективно, а пара індексних колонок (*ID*, *modified_at*) у джерелі дозволяє формувати компактні інкременти без зайвих сканувань. Для вітрин звітності з гранулярністю дня та тижня корисно зберігати попередні агрегати й додавати нові порції, а не перераховувати весь діапазон [12].

У задачах, де історія має змінну схему (наприклад, оновлення структур *JSON* у полі «*API*»), краще розділяти витрати: парсинг і нормалізація робляться під час надходження інкременту, а складніші аналітичні трансформації відкладені на аналітичний склад у позапікові години. Такий поділ шарів *ETL*/*ELT* у *Matillion* забезпечує контрольований розподіл навантаження та передбачувану вартість [13].

Практична конфігурація для таблиць *Sold_items*, *Sold_items_archive* та *Sold_items_big_values.*

Для «*Sold_items*» як оперативної таблиці звітності доцільно підтримувати невеликий розмір і чисту схему без важких текстових полів. Завантаження інкрементів виконується кілька разів на добу на *Small*-складі з авто-призупиненням у 60–120 секунд. Перед ранковою звітністю виконується короткий прогрів кешу основних запитів. Матеріалізовані представлення використовуються для найбільш ресурсомістких агрегацій, що повторюються.

Для «*Sold_items_archive*» архітектура передбачає великий нічний батч у вікно низького трафіку на *Medium* або *Large*-складі. Дані додаються у стаджингові таблиці й зливаються через *MERGE* у факт із кластеризацією за «*sale_date*» та додатковим ключем, якщо це покращує селективність. Раз на певний період виконують керований рекластер на нових ділянках. Читання архіву впродовж дня здійснюється на окремому складі, щоб аналітика не конкурувала з *ETL*.

Для «*Sold_items_big_values*» вводиться тип *VARIANT* для поля «*API*» і зберігається «*Commnt*» як *TEXT*, але у відокремленій таблиці з посиланням на *ID* продажу. Парсинг *JSON* виконується у *pushdown*, із збереженням нормалізованих атрибутів у вузькі колонки в окремій вітрині, щоб аналітичні запити не декомпресували весь *VARIANT* без потреби. Це радикально знижує *IO* при звітності.

Наведемо приклад створення таблиці з *VARIANT* і простого витягання атрибутів:

SQLCREATE OR REPLACE TABLE analytics_db.sales.Sold_items_big_values (*ID NUMBER*, *Commnt TEXT*, *API VARIANT*);-- Нормалізація ключових атрибутів із *JSON* у вузьку вітрину*CREATE OR*

REPLACE TABLE analytics_db.sales.Sold_items_api_normalized AS

SELECT ID, *API*:"*status*"::*STRING AS status*,

API:"*source*"::*STRING AS source*,

API:"*score*"::*NUMBER*

AS scoreFROM analytics_db.sales.Sold_items_big_values;*Show more lines*

Порівняння витрат: *Snowflake compute* проти *MSSQL* на *AWS* у типових сценаріях.

Хоча пряме порівняння вартості завжди залежить від конкретної конфігурації, загальна практика така: у *Snowflake* плата за обчислення визначається кредитами, що нараховуються за активний час складу відповідного розміру, тоді як у *MSSQL* на *AWS* основні витрати складаються з ціни інстансу (*EC*2 або керованого сервісу), обсягу оперативної пам'яті, дискової підсистеми (*IOPS*), а також ліцензійних витрат *SQL Server*. В *Snowflake* масштабування «горизонтальне» через склади дозволяє скоротити час виконання і, відповідно, суму кредитів, тоді як у *MSSQL* часто підвищення продуктивності потребує переходу на інстанс із більшою кількістю *vCPU* і *RAM*, що збільшує фіксовану плату незалежно від фактичного часу використання. Для *ETL*-джобів із вираженим нічним вікном *Snowflake* виграє за рахунок еластичності: склад активний лише під час роботи конвеєра, потім автоматично призупиняється. У *MSSQL* цей ефект частково досягається через масштабування за розкладом або перенесення на сервери з оплатою «*per-demand*», проте це рідко настільки прозоро, як у *Snowflake*.

В аналітичних навантаженнях з великими селектами *Snowflake* додатково економить через колонну природу, агресивне відсікання мікропартіцій та кеш результатів. У *MSSQL* важлива правильна індексація, партиціювання таблиць і використання *Columnstore Index* для фактів; проте це вимагає ручного дизайну й адміністрування, тоді як *Snowflake* бере на себе більшу частину фізичної оптимізації. Якщо організація вже має розгорнуту інфраструктуру *MSSQL* з вимогами до ліцензій та підтримки, рішення може бути змішаним: факти з великим обсягом і аналітичні вітрини в *Snowflake*, оперативні транзакційні системи — у *MSSQL*, а інкрементальне переміщення даних через *Matillion* або інші конектори.

3.1.7 Приклад вимірювання та верифікації ефекту

Щоб перетворити рекомендації на керовану економію, потрібні вимірювання. У кожному конвеєрі *Matillion* необхідно журналювати параметри:

розмір складу, кількість потоків, обсяг рядків, тривалість кроків і обсяг сканованих байтів (через *ACCOUNT USAGE*/*QUERY_HISTORY*). На основі цих даних будуються графіки «час *vs* кредит» для кількох конфігурацій. Якщо, наприклад, перехід із *Small* на *Medium* для нічного батчу скорочує час у 2,3 раза, а кредити збільшуються лише в 1,6 раза, це виправдана оптимізація. Якщо збільшення паралельності з 6 до 10 потоків не дає пропорційного виграшу, її слід зменшити до точки оптимуму [14].

Додатково верифікується ефект від кластеризації на «*Sold_items_archive*» через зміну відсотка відсічених мікропартіцій та середнього обсягу сканування. Якщо після рекластеру й переходу від фільтру «*WHERE sale_date BETWEEN …*» до більш точного діапазону покращення в середньому становить 35–50 % у часі на вибірку та 30–40 % у кредитах, кластеризацію слід зробити регулярною процедурою.

3.1.8 Узагальнення практичних порад

Сукупно оптимізація витрат і ресурсів у великих *DWH* базується на еластичності складів, інкрементальності конвеєрів, грамотному дизайні таблиць з окремими носіями для великих текстових/*JSON* даних, і на використанні колонних форматів і компресії при завантаженні. Важливою залишається спостережність: регулярні звіти щодо використання ресурсів, контроль бюджетів через монітори й теги, і планові *A*/*B*-експерименти з конфігураціями. У мультиплатформенному середовищі додатковий виграш приходить від розподілу ролей між *Snowflake* й *MSSQL*: аналітика з великим скануванням і гнучкими вікнами — у *Snowflake*; транзакційні системи й складні зміни у схемах — у відповідних сервісах, із чітко визначеними *CDC*-каналами до *Data Lake*.

З точки зору нах трьох таблиць, рекомендовані налаштування дають відчутну економію: оперативна «*Sold_items*» тримається компактною й оновлюється інкрементально на малому складі; архів «*Sold_items_archive*» обробляється у великих батчах на середньому або великому складі із кластеризацією за датою й періодичним рекластером; великі значення «*Sold_items_big_values*» зберігаються у *VARIANT* із нормалізованою вітриною

для потрібних атрибутів, що знижує витрати *BI*-шару. Втілення цих практик у твоїх *Matillion jobs* через сегментацію, *pushdown* і контроль паралельності забезпечує баланс між швидкістю та кредитами й формує повторювану, керовану економію для всього *DWH*.

3.2 Аналіз факторів, що впливають на продуктивність *DWH*

Продуктивність корпоративного сховища даних визначається не лише потужністю обчислювальних ресурсів, а й архітектурними принципами платформи, механізмами оптимізації запитів, структурою даних та політиками управління навантаженням. У випадку *Snowflake* ці фактори взаємопов'язані: розділення обчислень і зберігання, мікропартіційна організація даних, кешування результатів, кластеризація та гнучке масштабування складів формують унікальну модель продуктивності, яка суттєво відрізняється від традиційних систем на кшталт *MSSQL* або *Oracle*.

3.2.1 Архітектура *Snowflake* як основа продуктивності

Snowflake побудований на принципі поділу шарів: зберігання даних, обчислювальний шар і сервісний шар функціонують незалежно. Це означає, що дані зберігаються у централізованому об'єктному сховищі, а обчислення виконуються у віртуальних складах (*Virtual Warehouses*), які можна масштабувати горизонтально та вертикально без впливу на фізичну структуру даних. Така архітектура забезпечує еластичність: можна збільшити розмір складу для прискорення важких завдань або запустити кілька складів паралельно для ізоляції навантажень.

Ключовим елементом є мікропартіції — внутрішні блоки даних розміром приблизно 16 МБ у стиснутому вигляді. Кожна мікропартіція має метадані про діапазони значень колонок, що дозволяє оптимізатору виконувати «*data pruning*» — відсікати непотрібні блоки при виконанні запиту. Чим краще організовані дані (кластеризація), тим ефективніше працює *pruning*, а отже, менше *IO* і швидше виконання.

Архітектура *Snowflake* також передбачає автоматичне керування кешами, рекластеризацію та адаптивну оптимізацію запитів. Проте ці механізми не є магічними: їхня ефективність залежить від дизайну таблиць, частоти оновлень і характеру запитів.

3.2.2 Кешування як фактор швидкодії

Snowflake реалізує кілька рівнів кешування: кеш результатів запитів, кеш даних у пам'яті складу та кеш метаданих. Найбільш відчутний для користувача — кеш результатів. Якщо запит повторюється без змін у вихідних таблицях, *Snowflake* повертає результат із кешу без повторного сканування даних. Це дає приріст продуктивності у десятки разів для аналітичних систем, де *BI*-інструменти генерують повторювані запити.

Ефективність кешу залежить від стабільності тексту запиту та незмінності базових таблиць. Використання «*SELECT* *» або динамічних конструкцій руйнує кеш, оскільки оптимізатор вважає запит новим. Тому для критичних звітів слід застосовувати чіткі проєкції колонок і уникати зайвих варіацій у синтаксисі. У *Matillion* це означає стандартизацію *SQL*-компонентів у джобах і повторне використання однакових шаблонів [15].

Кеш даних у пам'яті складу працює на рівні мікропартіцій: якщо запит звертається до тих самих блоків, повторне читання з диску не потрібне. Проте цей кеш очищається при призупиненні складу, тому для серійних *ETL*-процесів із короткими паузами краще налаштувати авто-призупинення з інтервалом, що дозволяє зберегти кеш між кроками.

3.2.3 Кластерні ключі та їхній вплив на продуктивність

Кластеризація у *Snowflake* — це логічний механізм, який впорядковує мікропартіції за значеннями вибраних колонок. На відміну від класичних індексів, кластерні ключі не гарантують миттєвого доступу, але суттєво покращують *pruning*. Для великих таблиць, таких як «*Sold_items_archive*», кластеризація за датою продажу та категорією товару дозволяє скоротити обсяг сканованих даних у запитах, що фільтрують за цими атрибутами.

Вплив кластеризації можна виміряти через показники «*average overlap*» і «*depth*» у системних представленнях. Якщо після рекластеризації середній обсяг сканування зменшився на 30–50 %, це означає, що *pruning* працює ефективно. Проте кластеризація має ціну: операція рекластеризації споживає ресурси складу, тому її слід виконувати адаптивно — після великих апендів або при погіршенні показників [16].

Приклад створення кластерного ключа для архівної таблиці:

SQLALTER TABLE analytics_db.sales.Sold_items_archiveCLUSTER BY (*sale_date*, *item_category*);*Show more lines*

Це не змінює логічну схему, але впливає на фізичне розташування мікропартіцій. Для таблиць із *JSON*-полями кластеризація за атрибутами, витягнутими з *VARIANT*, також можлива, якщо ці атрибути часто використовуються у фільтрах.

Розмір *Virtual Warehouse* як параметр продуктивності.

Розмір складу визначає кількість обчислювальних ресурсів (*CPU*, пам’ять, *IO*), доступних для виконання запитів. Збільшення розміру складу прискорює виконання запитів, але пропорційно збільшує витрати. Оптимальний розмір залежить від характеру навантаження: для коротких аналітичних запитів достатньо *Small* або *Medium*, тоді як для масових *ETL*-процесів на сотні мільйонів рядків доцільно використовувати *Large* або навіть *XLarge*.

Важливо розуміти, що масштабування у *Snowflake* не завжди дає лінійний приріст швидкості. Для запитів, обмежених *IO* або логікою оптимізатора, збільшення складу може незначно вплинути на час виконання. Тому перед зміною розміру слід аналізувати профіль запиту через *Query Profile*: якщо більшість часу витрачається на сканування, кластеризація або фільтри дадуть більший ефект, ніж збільшення складу.

Для паралельних навантажень *Snowflake* підтримує *multi-cluster warehouses*, які автоматично додають кластери при зростанні черги запитів. Це корисно для *BI*-середовищ із піковими навантаженнями, але для *ETL* краще керувати розміром вручну, щоб уникати непередбачуваних витрат.

3.2.4 Взаємодія факторів і практичні висновки

Архітектура *Snowflake* забезпечує гнучкість, але ефективність залежить від правильного використання механізмів. Кешування дає миттєвий приріст для повторюваних запитів, кластеризація зменшує обсяг сканування, а розмір складу визначає швидкість виконання важких операцій. Проте ці фактори не працюють ізольовано: збільшення складу без кластеризації може бути марним, а рекламація кластерів без стабільних фільтрів у запитах не дасть ефекту. Тому оптимізація продуктивності — це баланс між дизайном даних, налаштуванням складів і стратегією виконання запитів.

Для твоїх таблиць «*Sold_items*», «*Sold_items_archive*» та «*Sold_items_big_values*» практичні рекомендації такі: оперативна таблиця працює на малому складі з кешем для звітів; архівна — кластеризована за датою та категорією, обробляється на середньому або великому складі у нічні вікна; таблиця з *JSON* — використовує *VARIANT* і нормалізовані атрибути для зменшення *IO*. У *Matillion* ці принципи реалізуються через *pushdown*, сегментацію та контроль паралельності, що дозволяє досягти стабільної продуктивності без надмірних витрат.

3.3 Особливості мультиплатформенного середовища

Мультиплатформенне середовище корпоративного сховища даних — це не просто набір різних СУБД, а складна екосистема, де кожна платформа виконує свою роль у життєвому циклі даних. У сучасних організаціях *Snowflake* часто використовується як аналітичний шар, *MSSQL* — як транзакційна база для бізнес-додатків, а *Postgres* — як гнучке джерело для сервісів і мікросервісної архітектури. Додатково в цю систему інтегруються *API*, які постачають напівструктуровані дані у форматах *JSON* або *XML*. Така гетерогенність створює виклики для узгодженості, продуктивності та керованості *ETL*-процесів [17].

3.3.1 Архітектурна модель інтеграції

У типовій архітектурі мультиплатформенного *DWH Snowflake* виступає як кінцева точка для аналітики, де дані з різних джерел консолідуються у єдину модель. *MSSQL* забезпечує транзакційні операції, зберігаючи дані про продажі, замовлення, клієнтів у нормалізованій структурі. *Postgres* часто використовується для сервісних даних, логів, конфігурацій, а також як джерело для *API*, що взаємодіють із зовнішніми системами. Інтеграція між цими платформами реалізується через *ETL*-конвеєри, які виконують екстракцію, трансформацію та завантаження даних у *Snowflake*.

Matillion ETL у цьому контексті є ключовим інструментом, оскільки підтримує конектори до *MSSQL*, *Postgres* та *API*, а також реалізує *pushdown*-трансформації у *Snowflake*. Це дозволяє мінімізувати обсяг даних, що передаються між системами, і виконувати важкі обчислення на рівні аналітичної платформи.

3.3.2 Проблеми узгодженості даних

У мультиплатформенній архітектурі узгодженість даних є критичною проблемою. Вона проявляється у кількох вимірах: часовому, структурному та семантичному. Часова узгодженість означає, що дані з різних джерел повинні відображати один і той самий стан бізнес-процесу на момент аналітичного зрізу. Це складно забезпечити, коли *MSSQL* оновлюється транзакційно, *Postgres* — асинхронно, а *API* постачає дані з затримкою. Для вирішення цієї проблеми застосовуються механізми *CDC (Change Data Capture)* у *MSSQL* та логічна реплікація у *Postgres*, що дозволяє отримувати дельти змін і завантажувати їх у *Snowflake* інкрементально [18].

Структурна узгодженість стосується різних схем даних. *MSSQL* використовує типи *NVARCHAR*, *DECIMAL*, а *Postgres — TEXT*, *NUMERIC*, *JSONB*. При інтеграції у *Snowflake* необхідно виконувати нормалізацію типів: наприклад, *NVARCHAR → STRING*, *DECIMAL → NUMBER*, *JSONB → VARIANT*. Неправильне зіставлення типів може призвести до втрати точності або помилок у трансформаціях.

Семантична узгодженість — це узгодження бізнес-правил. Наприклад, статус замовлення у *MSSQL* може мати значення «*Completed*», а у *Postgres* — «*Done*». Без уніфікації ці значення у *Snowflake* аналітика буде некоректною. Для цього створюються довідники та мапінги у *Matillion* або на рівні *Snowflake* через таблиці відповідностей.

3.3.3 Роль *API* та напівструктурованих даних

API є важливим джерелом даних у сучасних *DWH*. Вони постачають інформацію про транзакції, інтеграції з платіжними системами, дані з *CRM* або зовнішніх сервісів. Формат даних зазвичай *JSON*, що потребує спеціальної обробки. У *Snowflake* для цього використовується тип *VARIANT*, який дозволяє зберігати напівструктуровані дані без попередньої нормалізації. Проте для аналітики необхідно витягувати ключові атрибути у реляційні таблиці. Це робиться через функції *PARSE_JSON* та *FLATTEN* або через *UDF* [19].

Приклад завантаження даних з *API* у *Snowflake* через *Matillion*:

1. Виклик *API* через компонент «*API Query*» у *Matillion*;
2. Збереження відповіді у стаджинговій таблиці з колонкою типу *VARIANT*;
3. Нормалізація атрибутів у окрему таблицю.

CREATE OR REPLACE TABLE analytics_db.api_normalized ASSELECT API:"transaction_id"::STRING AS transaction_id, API:"amount"::NUMBER AS amount, API:"status"::STRING AS statusFROM analytics_db.stage.api_raw;Show more lines

Цей підхід дозволяє зберігати оригінальний *JSON* для аудиту та одночасно мати нормалізовану структуру для аналітики.

Інтеграція різних платформ створює додаткові навантаження на *ETL*-процеси. *MSSQL* і *Postgres* мають різні механізми блокувань, що впливає на швидкість екстракції. *API* може бути обмежений за кількістю запитів на хвилину, що вимагає реалізації механізмів *rate limiting* у *Matillion*. Крім того, обробка *JSON* у великих обсягах може стати вузьким місцем, якщо вона виконується на рівні *ETL*, а не у *Snowflake*. Тому оптимальна стратегія — максимальне використання *pushdown*-трансформацій і розподіл навантаження між системами.

Для забезпечення узгодженості даних у мультиплатформі застосовуються контрольні точки (*checkpoints*) у *Matillion*, які гарантують, що всі джерела пройшли стадію екстракції перед початком трансформацій. Додатково використовуються механізми валідації: порівняння кількості рядків, контроль хешів для критичних таблиць, логування часу останнього оновлення.

3.3.4 Практичні рекомендації

У мультиплатформенній архітектурі важливо дотримуватися принципу «*schema-on-read*» для напівструктурованих даних і «*schema-on-write*» для реляційних джерел. Це означає, що *JSON* з *API* зберігається у сирому вигляді, а нормалізація виконується лише для атрибутів, потрібних для аналітики. Для *MSSQL* і *Postgres* доцільно застосовувати інкрементальні завантаження через *CDC*, щоб уникати повних сканувань. У *Snowflake* слід використовувати кластеризацію для великих фактів і матеріалізовані представлення для стабільних аналітичних запитів.

Взаємодія між платформами має бути максимально автоматизованою. *Matillion* дозволяє реалізувати оркестрацію через *job flow*, де кожен етап інтеграції має чіткі залежності та контроль помилок. Це забезпечує стабільність процесів і мінімізує ризики розсинхронізації.

3.4 Роль *ETL*-процесів у забезпеченні узгодженості даних

3.4.1 Узгодженість даних у корпоративному сховищі

Узгодженість даних у корпоративному сховищі — це не лише питання коректності значень, а й синхронності оновлень, стабільності бізнес-правил та відсутності розривів між джерелами. У мультиплатформенній архітектурі, де дані надходять із *MSSQL*, *Postgres* та *API*, саме *ETL*-процеси стають центральним механізмом забезпечення узгодженості. Вони виконують не лише технічну

функцію переміщення даних, а й логічну — гармонізацію структур, типів і семантики.

ETL vs ELT: концептуальна різниця та практичні наслідки

Традиційна модель *ETL* (*Extract–Transform–Load*) передбачає виконання трансформацій на рівні *ETL*-інструменту перед завантаженням у сховище. Це означає, що дані екстрагуються з джерел, проходять через проміжні обробки (очищення, нормалізація, агрегація) і лише потім потрапляють у *DWH*. Такий підхід був виправданий у часи, коли сховища мали обмежені обчислювальні ресурси, а *ETL*-сервери були потужнішими.

Модель *ELT* (*Extract–Load–Transform*), яку реалізує *Snowflake*, змінює парадигму: дані завантажуються у сховище у сирому вигляді, а трансформації виконуються безпосередньо на його рівні. Це дозволяє використовувати масштабовані ресурси *Snowflake* для обробки великих обсягів даних і мінімізувати передачу між системами. *ELT* особливо ефективний у поєднанні з *cloud-native ETL*-інструментами, такими як *Matillion*, які підтримують *pushdown optimization* [20].

Pushdown optimization — це механізм, який переносить виконання трансформацій із *ETL*-інструменту на рівень бази даних. У випадку *Matillion* і *Snowflake* це означає, що замість виконання складних операцій у пам’яті *ETL*-сервера, *Matillion* генерує *SQL*-код і передає його *Snowflake* для виконання. Таким чином, важкі джоіни, агрегації, фільтрації та нормалізації виконуються у середовищі, оптимізованому для обробки великих обсягів даних.

Практичний приклад: замість того, щоб *Matillion* завантажував дані з *MSSQL*, обробляв їх локально і потім завантажував у *Snowflake*, він виконує *COPY INTO* для завантаження сирих даних у *STAGE* таблицю, а потім запускає *SQL*-трансформації у *Snowflake*. Це зменшує час обробки та вартість, оскільки обчислення виконуються на складі, який можна масштабувати.

Узгодженість через контрольні точки та інкрементальні завантаження *ETL*-процеси забезпечують узгодженість даних через механізми контрольних точок (*checkpoints*) і інкрементальні завантаження. Контрольні точки гарантують, що всі джерела пройшли стадію екстракції перед початком трансформацій. Це особливо важливо у мультиплатформі, де *MSSQL* і *Postgres* оновлюються асинхронно, а *API* може мати затримки.

Інкрементальні завантаження зменшують ризик розсинхронізації, оскільки обробляються лише дельти змін. У *MSSQL* це реалізується через *CDC* (*Change Data Capture*), у *Postgres* — через логічну реплікацію. У *Snowflake* дельти завантажуються у стаджингові таблиці, а потім інтегруються через *MERGE INTO*.

Приклад *MERGE* для узгодження даних у *Snowflake*:

```
MERGE INTO analytics_db.sales.Sold_items AS tgt
USING analytics_db.stage.Sold_items_delta AS src
ON tgt.ID = src.ID
WHEN MATCHED THEN UPDATE SET
  tgt.qty = src.qty,
  tgt.amount = src.amount,
  tgt.modified_at = src.modified_at
WHEN NOT MATCHED THEN INSERT (ID, qty, amount, modified_at)
VALUES (src.ID, src.qty, src.amount, src.modified_at);
```

Цей підхід гарантує, що нові записи додаються, а змінені — оновлюються без дублювання.

3.4.2 Приклади *Matillion jobs* для узгодженості

У *Matillion* узгодженість реалізується через оркестрацію *job flow*. Типовий процес складається з кількох етапів:

1. *Extract*: підключення до *MSSQL* і *Postgres* через конектори, отримання дельт змін.
2. *Load*: завантаження даних у *Snowflake* через компоненти «*Bulk Load*» або «*Table Input*» з використанням *COPY INTO*.

3. *Transform*: виконання *SQL*-трансформацій у *Snowflake.* Застосовуємо компоненти «*SQL Script*» або «*Table Update*», що реалізують *pushdown*.

4. *DQ:* перевірка якості даних.

5. *Validation*: контроль кількості рядків, хешів, логування часу оновлення.

6. *Publish*: оновлення аналітичних вітрин і матеріалізованих представлень.

Matillion job flow для інтеграції *MSSQL* → *Snowflake* (рис 3.1).

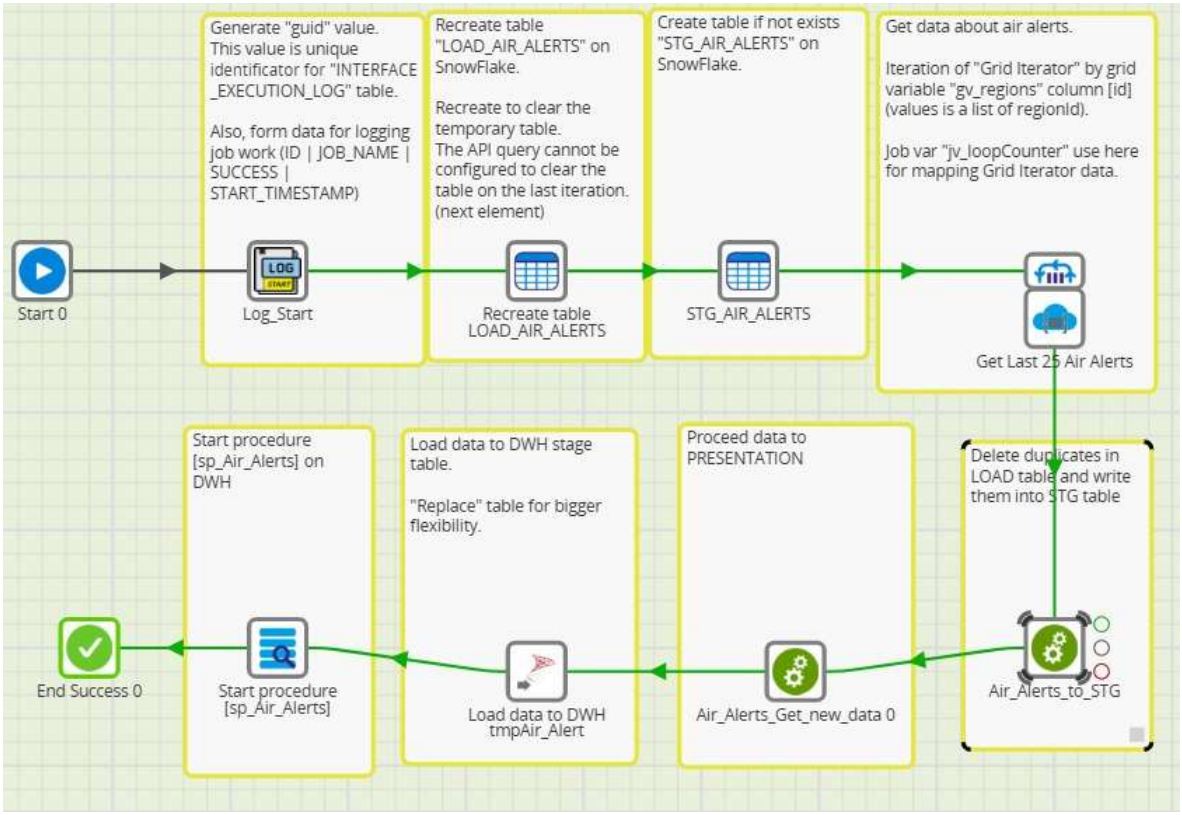

Рисунок 3.1 – *Matillion job flow*

Узгодженість напівструктурованих даних.

Для *API*, що постачають *JSON*, узгодженість забезпечується через збереження сирих даних у *VARIANT* і нормалізацію ключових атрибутів. У *Matillion* це реалізується через компонент «*API Query*» для отримання даних і «*Snowflake Load*» для збереження у стаджинговій таблиці. Потім виконується *SQL*-трансформація.

```
CREATE OR REPLACE TABLE analytics_db.api_normalized AS
SELECT
  API:"transaction_id"::STRING AS transaction_id,
  API:"status"::STRING AS status,
  API:"amount"::NUMBER AS amount
FROM analytics_db.stage.api_raw;
```

Цей підхід дозволяє зберегти оригінальний *JSON* для аудиту та одночасно мати нормалізовану структуру для аналітики [21].

3.5 Практичні рекомендації щодо побудови *ETL* у мультиплатформенних *DWH*-системах

Архітектура *ETL* у мультиплатформенному сховищі має бути одночасно стійкою до змін джерел, еластичною до навантаження й економною у витратах. Вона формується як сукупність шарів: отримання змін із транзакційних систем *MSSQL* та *Postgres*, інжест у *Snowflake* на стаджинг рівень без втрати сирих даних, трансформаційний шар з *pushdown*-виконанням та публікаційний шар аналітичних вітрин. Керована узгодженість даних досягається через детерміновані контрольні точки, інкрементальні стратегії та валідаційні механізми, а продуктивність — через сегментацію обсягу, паралельність і правильну організацію зберігання. У твоєму домені ключовими стають три таблиці: оперативна *Sold_items* із горизонтами у шість місяців, архівна *Sold_items_archive* із сотнями мільйонів записів і допоміжна *Sold_items_big_values* для великих текстів і *JSON*-полів, відокремлена для зменшення *IO* у звітному шарі.

Архітектурний каркас *ETL* починається з вибору моделі *ELT* як базової. Дані з *MSSQL* і *Postgres* постачаються інкрементами, що визначаються або *CDC*-механізмами, або вододілом за *modified_at*, і без надмірних перетворень записуються в стаджингові таблиці *Snowflake*. Стажинг у цій моделі виступає буфером і джерелом правди на момент завантаження, дозволяючи повністю

відтворити конвеєр у разі збоїв без повторних звернень до джерел. На цьому ж рівні зберігаються сирі *API*-відповіді у форматі *VARIANT* для подальшої нормалізації. Далі трансформаційний шар виконується як набір *SQL*-скриптів і компонентів *Matillion* з *pushdown*-режимом, де джоіни, агрегації, дедуплікації та *MERGE* виконуються у *Snowflake* на обраних *Virtual Warehouses*. Публікаційний шар містить матеріалізовані уявлення, звітні таблиці з денним або годинним *SLA*, та індексні довідники, що забезпечують семантичну узгодженість.

Оптимізація потоків починається з розкладки навантажень у часі та виділення логічних пакетів роботи. Нічні батчі для великих перезаписів архівних сегментів доцільно зосереджувати у визначеному вікні, коли аналітичні склади мінімально активні. Для цього використовується окремий склад середнього або великого розміру, який активується на час батчу й призупиняється одразу після завершення. Денний шар оперативних інкрементів *Sold_items* працює на меншому складі з коротким авто-призупиненням, щоб не втрачати кеш між суміжними кроками. Паралельність обмежується до рівня, на якому приріст швидкості залишається квазі-лінійним, а контенція за *IO* не зводить нанівець виграш. У практиці це виглядає як розбиття інкрементів за датами або діапазонами ідентифікаторів із запуском кількох ідентичних підпроцесів, що працюють над взаємонезалежними сегментами.

Сегментація та паралельність особливо важливі для *Sold_items_archive*. Замість одного масивного запиту, що сканує десятки чи сотні мільйонів рядків, архівний процес поділяється на квартальні або місячні набори, кожен з яких обробляється окремо. На рівні *Matillion* це реалізується як генератор діапазонів з таблиці календаря або метаданих, що підставляє параметри у шаблонні компоненти завантаження та *MERGE*. Такий підхід дозволяє одночасно позбутися вузьких місць і отримати більш прогнозоване використання кредитів, а також спростити контроль і повторний запуск конкретного сегмента у разі помилок. Для зниження вартості корисно зберігати останній успішний сегмент і перезапускати лише від нього, а не весь батч.

Ключовим доповненням до сегментації є правильна кластеризація. Архівна таблиця виграє від кластерного ключа за *sale_date*, доповненого полем категорії або регіону, якщо ці атрибути часто використовуються у фільтрах звітів. Це впливає на ступінь відсікання мікропартіцій і безпосередньо зменшує обсяг сканування у запитах публікаційного шару. Рекластеризація не повинна виконуватись сліпо після кожного завантаження; її доречно запускати адаптивно, коли збільшується середнє перекриття мікропартіцій у діапазонах, що активно читаються. Така тактика мінімізує зайве «перекладання» стореджу й заощаджує кредити на допоміжні обчислення.

Побудова потоку для *Sold_items* відрізняється акцентом на інкрементальності та низькій латентності. Подача дельт із джерел упродовж дня зводить до мінімуму перерахунок. Після завантаження у стаджинг виконується *MERGE* у основну таблицю, де оновлюються лише змінені рядки. Для критичних агрегатів, які надалі використовуються у звітах, доцільно мати матеріалізовані уявлення, що автоматично рефрешаться по інкрементальних змінах, або, якщо рефреш повний і дорогий, зберігати агрегати у фізичних таблицях і додавати до них лише нові періоди. У часи пікових звернень до звітів корисним є короткий «теплий прогін» ключових селектів, що наповнює кеш результатів і помітно прискорює перші запити користувачів без збільшення розміру складу.

Таблиця *Sold_items_big_values* потребує окремої траєкторії. Винос великих текстових полів і *JSON* у самостійну таблицю з посиланням на *ID* продажу знижує навантаження на факт, який використовується у більшості звітів. Зберігання *JSON* у *VARIANT* дозволяє беззбитково приймати гетерогенні структури *API*, але для аналітики потрібні нормалізовані атрибути. Ефективною практикою є створення вузької вітрини з потрібними полями, витягнутими з *VARIANT*, щоб зменшити потребу в декомпресії великих блоків при звичайних селектах. Нормалізація виконується у *pushdown* через *SQL* у *Snowflake*, а не в *Matillion*-пам’яті, що зменшує час і витрати.

Опис типової реалізації *Matillion*-потоку для *Sold_items* виглядає як оркестрація з чіткими фазами. На початку відпрацьовує крок синхронізації з

MSSQL і *Postgres*, який отримує дельти змін за *modified_at* або через *CDC*-ендпоїнти. Далі відбувається завантаження у стаджинг *Snowflake* з використанням *Bulk Load*/*COPY* і попереднім визначенням форматів файлів та політик обробки помилок. Після цього запускається трансформаційний блок у вигляді *SQL Script* компонентів, який виконує *MERGE* в операційну таблицю. Результатом є короткий етап валідації, де порівнюються кількості рядків, розраховуються контрольні хеші для ключових наборів і фіксується *watermark*. Завершує процес оновлення представлень і теплий прогін звітних запитів. Такий сценарій мінімізує кількість перемикань складу, зберігає кеш між кроками і робить весь процес детермінованим.

SQL-приклад для інкрементального *MERGE* у *Sold_items* може мати таку форму, із явним оновленням лише змінених полів та фіксацією часу модифікації, Приклад наведено в додатку 8.

У цій моделі *hash* виступає легким механізмом визначення зміни рядка без порівняння множини полів, а *CURRENT_TIMESTAMP* фіксує момент узгодження для подальших *SLA*-перевірок. На архіві логіка інша: рідкі оновлення та часті апенди. Там доцільніше тримати процес як «*insert-only*» із періодичною дедуплікацією по бізнес-ключу або *ID*, якщо з джерел іноді приходять повтори. Дедуп можна виконувати через віконні функції з вибором найсвіжішого запису за *modified_at* у тимчасову таблицю, після чого акуратно замінюється потрібний сегмент.

Для архівного перебору сегментів корисно використовувати параметризовані *SQL*-скрипти, де підставляється інтервал дат. У *Matillion* це реалізується через таблицю-драйвер із діапазонами та компонентами *Iterator*/*Parallel*. На рівні *SQL* це виглядає просто, але важливо, що в один момент часу кожний запит читає свій вузький діапазон і не конкурує за ті самі мікропартіції:

INSERT INTO analytics_db.sales.Sold_items_archive

*SELECT **

FROM analytics_db.stage.Sold_items_stage

WHERE sale_date >= ${from_date} AND sale_date < ${to_date};

Після завершення кожного сегмента корисно виконувати цільовий рекластер на останньому діапазоні, якщо він суттєво порушує структуру мікропартіцій, аби не накопичувати фрагментацію до кінця батчу. Для цього достатньо обмежити область рекластеризації *WHERE*-умовою по *sale_date* на новий період, що зменшує вартість допоміжної операції.

Оптимізація потоків у частині ресурсів включає уважний підбір розміру складу для кожної фази. Екстракція та первинне завантаження у стаджинг зазвичай не потребують великих складів і добре працюють на *Small*, тоді як масові трансформації та об'єднання значно швидше відпрацюють на *Medium* або *Large*. Рішення про масштабування приймається на основі профілю запиту: якщо горлечко — у скануванні даних, кластеризація і вибір вузьких колонок дають більший ефект, ніж збільшення складу. Якщо ж вузьке місце у джоіні чи сортуванні великих проміжних наборів, тимчасове збільшення складу на трансформаційній фазі скоротить *wall time* і загальну суму кредитів завдяки меншому часу активності.

Узгодженість і відновлення після збоїв потребують введення контрольних точок і ідемпотентності кроків. Кожний значущий етап має закінчуватися фіксацією *watermark* у технічній таблиці з метаданими, збереженням контрольних підсумків і явним маркером успіху. У разі помилки повторний запуск починається з останньої контрольної точки, а всі операції завантаження і трансформації мають виконуватися так, щоб повтор не створював дублікати і не порушував цілісність. Для цього стаджинг проєктується з можливістю атомарного «перемикання» через тимчасові таблиці і патерн *swap*: дані збираються у *_new*, перевіряються, після чого відбувається швидке перейменування у бойову таблицю, а попередня версія архівується з *TTL*.

Матеріалізація результатів — ще один інструмент керування продуктивністю. Якщо певні агрегати регулярно використовуються у *BI* і їхній перерахунок дорогий, доцільно створити матеріалізовані уявлення або щоденні підсумкові таблиці. Матеріалізовані уявлення зручні автоматичністю, але їхній

рефреш споживає кредити, тому є сенс порівняти вартість із ручним оновленням під час нічного батчу. Для *Sold_items* типовим є щоденний підсумок по категоріях і регіонах, який обчислюється у нічний період і впродовж дня читається з мінімальним *IO*. Це також покращує стабільність часу відповіді у піках.

Варто окремо окреслити логування та спостережність. Журнали виконання *Matillion* мають містити параметри складу, паралельність, тривалість кроків, обсяг рядків і сканованих байтів, а також результати валідацій. Ці дані збираються у технічну вітрину, де будуються дашборди «час проти кредитів» і «вартість на 1 млн рядків». На підставі цих метрик приймаються рішення про зміну розміру складу, корекцію паралельності, оновлення кластерних ключів і графіку батчів. У середовищах дев/тест застосовуються *Resource Monitors* із м’якими лімітами й сповіщеннями, щоб виявляти неефективні експерименти, не ризикуючи продакшном.

Із погляду безпеки та розмежування доступу мультиплатформене середовище накладає додаткові вимоги на канали передачі і шифрування. Під час інжесту з *MSSQL* і *Postgres* канали мають використовувати *TLS*, а облікові дані — зберігатися у секрет-менеджері з ротацією. На рівні *Snowflake* необхідна ролева модель, де стаджинг і трансформаційні схеми ізольовані від бізнес-користувачів, а публікаційні вітрини мають лише права читання. Це не лише безпека, а й продуктивність, адже випадкові важкі запити у робочий час не повинні впливати на *ETL*-склад завдяки розділенню складів.

Для завершення розділу доцільно повернутися до твоїх конкретних сутностей. Оперативна *Sold_items* повинна залишатися компактною: колонки з великими текстами тримаються поза нею, типи даних підібрані так, щоб уникати надмірно широких *NUMBER* і *STRING*, а *VARIANT* не використовується в основному факті. Інкрементальні *MERGE* виконуються кілька разів на добу на малому складі з невеликою паралельністю, аби зберегти кеш і не створювати пікових навантажень. Архівна *Sold_items_archive* отримує дані пачками, сегментовано за датою, з адаптивною рекластеризацією нових ділянок; важкі вибірки по архіву виконуються на окремому складі, щоб не конкурувати з

інжестом. Таблиця *Sold_items_big_values* зберігає сирий *JSON* у *VARIANT* та формує вузьку нормалізовану вітрину з потрібними полями для *BI*, завдяки чому звітний шар читає легкі колонки, а до сирих даних звертається лише для *forensic*-аналізу. Усі ці практики, реалізовані через *Matillion* із *pushdown*-стратегією, дають прогнозоване прискорення та керовану вартість, що прямо підтримує твою мету з оптимізації продуктивності та витрат у мультиплатформенних *DWH*.

3.6 Порівняння вартості обчислень у *Snowflake* та *MSSQL* на *AWS*

Порівнюючи фінансову модель обчислень для аналітичних *DWH*-навантажень у *Snowflake* з еквівалентними сценаріями на *MSSQL* у *AWS*, важливо розвести дві парадигми: у *Snowflake* вартість обчислень визначається кредитами, що споживаються віртуальними складами за секунду активної роботи, а в *AWS* для *MSSQL* оплата фокусується на погодинній ціні інстансів (*EC*2 або керованого сервісу *RDS*) плюс зберігання, *I*/*O* та ліцензійні компоненти. У *Snowflake* інфраструктура прихована, і користувач керує лише «віртуальними складами» й їхнім життєвим циклом; у *AWS*, навпаки, інфраструктурна гранулярність видима, і від вибору класу інстансу та моделі ліцензування залежить підсумкова ціна за годину. Такий розподіл призводить до різних оптимізаційних стратегій: у *Snowflake* ключовим важелем є розмір складу та авто-*suspend*, а в *MSSQL*/*AWS* — правильний тіп інстансу, резервування, *Savings Plans* і, в разі *RDS*, вбудовані ліцензії *SQL Server*.

3.6.1 Порівняння вартості обчислень у *Snowflake* та *MSSQL* на *AWS*

Порівнюючи фінансову модель обчислень для аналітичних *DWH*-навантажень у *Snowflake* з еквівалентними сценаріями на *MSSQL* у *AWS*, важливо розвести дві парадигми: у *Snowflake* вартість обчислень визначається кредитами, що споживаються віртуальними складами за секунду активної роботи, а в *AWS* для *MSSQL* оплата фокусується на погодинній ціні інстансів (*EC*2 або керованого сервісу *RDS*) плюс зберігання, *I*/*O* та ліцензійні компоненти. У *Snowflake* інфраструктура прихована, і користувач керує лише «віртуальними складами» й

їхнім життєвим циклом; у *AWS*, навпаки, інфраструктурна гранулярність видима, і від вибору класу інстансу та моделі ліцензування залежить підсумкова ціна за годину[22]. Такий розподіл призводить до різних оптимізаційних стратегій: у *Snowflake* ключовим важелем є розмір складу та авто-*suspend*, а в *MSSQL*/*AWS* — правильний тіп інстансу, резервування, *Savings Plans* і, в разі *RDS*, вбудовані ліцензії *SQL Server* [24].

Для коректної калькуляції ми покладемося на відкриті еталони. У *Snowflake* на *AWS* (*US East*) оновлена сторінка тарифів показує ціни за кредит для різних редакцій: *Standard* — $2.00/кредит, *Enterprise* — $3.00/кредит, *Business Critical* — $4.00/кредит; виставлення рахунку — за секунду з мінімумом в одну хвилину при кожному відновленні складу. Кожен перехід на більший розмір складу приблизно подвоює як продуктивність, так і споживання кредитів за годину; наприклад, *X-Small* — 1 кредит/год, *Small* — 2, *Medium* — 4, *Large* — 8 [23].

Для *MSSQL* на *AWS* візьмемо два вектори: керований сервіс *Amazon RDS for SQL Server* із моделлю «*License Included*» (ліцензія вбудована в погодинний тариф), а також орієнтир по *EC*2 для загальноцільових інстансів, де можна запускати *MSSQL* у власній конфігурації. Документація *RDS* підтверджує, що ліцензія *SQL Server* входить у тариф «*License Included*», а облік ведеться погодинно за інстанс; для *EC*2 у відкритих каталогах вказані *on-demand* ставки, наприклад *m*5.2*xlarge* ≈ $0.384/год у більшості регіонів США. Ці числа ми використаємо як опорні для порівняння «$ за годину» з еквівалентними рівнями складів *Snowflake*.

Нижче — узагальнена таблиця зіставлення «$ за годину» для базових розмірів складів у *Snowflake* (*Standard edition*, *US East*) та репрезентативних інстансів *AWS EC*2. Ці значення є базою для подальшого моделювання *compute cost* у твоїх сценаріях *ETL*/*ELT*.

Візуалізація: погодинна ціна базових конфігурацій (рис 3.2).

Platform	Configuration	Unit	Rate_per_unit_USD	Units_per_hour	Hourly_cost_USD	Notes
Snowflake	X-Small Warehouse	credit/hour	2	1	2	Standard edition; billed per-second with 60 sec minimum
Snowflake	Small Warehouse	credit/hour	2	2	4	Standard edition
Snowflake	Medium Warehouse	credit/hour	2	4	8	Standard edition
Snowflake	Large Warehouse	credit/hour	2	8	16	Standard edition
AWS EC2	m5.large (on-demand)	USD/hour			0,178	Representative Linux on-demand
AWS EC2	m5.xlarge (on-demand)	USD/hour			0,192	Representative Linux on-demand
AWS EC2	m5.2xlarge (on-demand)	USD/hour			0,384	Representative Linux on-demand

Рисунок 3.2 – Скріншот еталонної таблиці вартості обчислень (погодинно)

У *Snowflake* наведену ставку слід множити на фактичну тривалість роботи складу, з урахуванням мінімуму 60 секунд при кожному *resume*; у *EC2*/*MSSQL* оплата ведеться за годину (*Linux* — посекундна, *Windows*/*SQL* — погодинна), до якої додаються диски *EBS* та пропущений *I*/*O*. Ці нюанси важливі при моделюванні реальних *ETL*-вікон і *BI*-піків.

Нижче — порівняльний графік для швидкого огляду «$ за годину» у двох моделях. Він підходить для вставки у твою роботу як ілюстрація «*pay-per-use*» проти «*pay-per-instance*» .

Під графіком слід наголосити: у *Snowflake X-Small* до *Large* — це лінійне подвоєння кредитів і, відповідно, ціни; у *EC2* діапазони *m5* — лінійне зростання ціни за ресурс, але фінальна вартість *MSSQL* на *EC2* з *Windows*/*SQL* образами буде вища за показані *Linux*-ставки; при використанні *RDS for SQL Server* ліцензія входить у погодинний тариф інстансу.

Порівнювати «$ за годину» без контексту часу виконання — некоректно. У *Snowflake* більший склад скорочує *wall time ETL* та аналітичних запитів, тому загальна сума кредитів може бути вигіднішою, якщо прискорення суттєве. Офіційна документація прямо фіксує, що кожний крок розміру склада подвоює і продуктивність, і кредити/годину; тож оптимально знаходити точку, де скорочення тривалості перевищує зростання ставки.

Ми можемо використати Power BI для побудови порівняльного графіку вартості (рис 3.3).

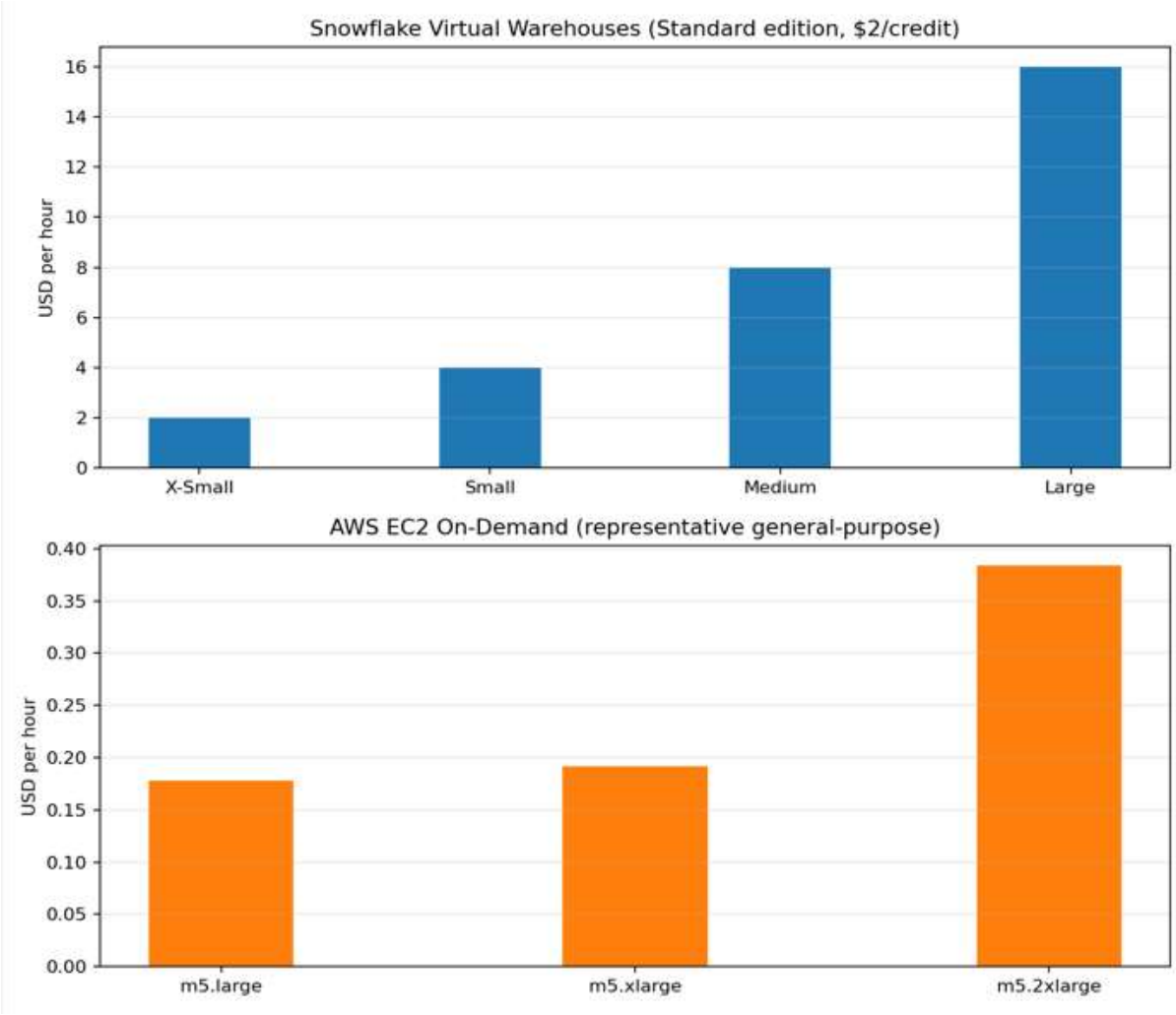

Рисунок 3.3 – Порівняльний графік вартості

Додатково в *Snowflake* діє перехід на *on-demand* кредити з ціною $2–$4 залежно від редакції й регіону та передплатні моделі з дисконтом, що стає відчутним при стабільних навантаженнях [25].

В *AWS* на *MSSQL* горизонт оптимізації інакший. Якщо використовується *RDS*, ти сплачуєш за інстанс, де ліцензійні витрати *SQL* уже включено. При *EC*2 з *BYOL*/*License Mobility* твій підсумок залежить від ліцензійного портфеля та чинних правил *Microsoft* щодо «*Listed Providers*», але для дисертаційної частини коректно зафіксувати: *RDS* спрощує кошторис за рахунок моделі «*License Included*», *EC*2 — гнучкіший у профілюванні ресурсів і *Savings Plans*, проте потребує ретельного обліку ліцензій і схеми *HA*/*DR*[26].

Для денних аналітичних запитів з повторюваними селектами *Snowflake* додатково економить за рахунок кешу результатів і агресивного *data pruning* завдяки мікропартіціям; це фактори, які зменшують реальну тривалість складу і, відповідно, витрати на кредити. У *MSSQL*/*EC*2/*RDS* ключовими важелями будуть індексація (*Columnstore* для фактів), правильний клас дисків та *IOPS*, а також нові можливості *AWS* з оптимізацією *CPU* для *RDS SQL Server* на поколіннях *M*7*i*/*R*7*i*, що в 2025 році офіційно позиціонуються як спосіб знизити ліцензійну складову за рахунок *SMT*-налаштувань [28]. Це практична різниця в економіці: *Snowflake* оптимізує рахунок через архітектуру сховища та еластичність; *AWS*/*MSSQL* — через інстанси, диски та ліцензійні механіки (рис. 3.4).

Edition	Warehouse_Size	Credits_per_hour	Hourly_cost_USD
Standard	X-Small	1	2
Standard	Small	2	4
Standard	Medium	4	8
Standard	Large	8	16
Enterprise	X-Small	1	3
Enterprise	Small	2	6
Enterprise	Medium	4	12
Enterprise	Large	8	24
Business Critical	X-Small	1	4
Business Critical	Small	2	8
Business Critical	Medium	4	16
Business Critical	Large	8	32

Рисунок 3.4 –Скріншот розрахунку вартісті використання *Warehouse on Snowflake*

Для таблиць *Sold_items* та *Sold_items_archive* доцільно зіставити два типи завдань. Нічний батч для архіву у *Snowflake* на *Large* складі коштуватиме $16/год при *Standard edition*; якщо батч триває, скажімо, 45 хвилин, то це ~$12 (без урахування додаткових сервісів), і авто-*suspend* зведе витрати до нуля поза роботою. У *RDS MSSQL* подібний обсяг потягне на інстанс класу *m*5.2*xlarge* або *r*5.*large* з ліцензією, де погодинна ціна буде стабільною незалежно від простоїв;

тому економіка вигідна, якщо інстанс використовується безперервно або заздалегідь покритий *Reserved Instances*/*Database Savings Plans*. У цьому сенсі еластичність *Snowflake* — це прямий важіль для *ETL*-вікон, тоді як *MSSQL*/*RDS* більше стимулює постійний режим роботи або планові оптимізації цінових моделей [27].

Навіть при незмінному розмірі складу перехід зі *Standard* на *Enterprise* або *Business Critical* збільшує «$ за кредит», а отже й $ за годину. Стисло зафіксуємо це як підсумковий еталон для калькуляторів у роботі.

Висновки до розділу

Перший висновок стосується моделі «платимо за час, коли працює *compute*»: у *Snowflake* оплата суто за активний час складу з посекундною тарифікацією та мінімумом 60 секунд сприяє економії в інтервальних *ETL*-вікнах і повторюваних аналітичних зверненнях завдяки кешу; при грамотному налаштуванні *auto-suspend* і правильному розмірі складу під конкретне навантаження, сума кредитів стає контрольованою й часто нижчою, ніж у постійно працюючих інстансах.

Другий висновок: у *MSSQL* на *AWS* економіка залежить від того, чи використовуєш *RDS* (ліцензія включена, просто моделювати бюджет) чи *EC*2 (гнучкість конфігурації, але ліцензійні тонкощі). *RDS* добре підходить для «завжди увімкнених» систем і мінімізує операційний оверхед; *EC*2 з *BYOL* дозволяє капіталізувати на вже наявних ліцензіях за умови відповідності *License Mobility* та *Software Assurance*. У 2025 році *AWS* додатково пропонує механізми оптимізації *CPU* для *RDS SQL Server* на *M*7*i*/*R*7*i*, що знижують ліцензійну складову при збереженні продуктивності — корисний важіль для зниження *TCO* у продуктивних навантаженнях.

Третій висновок торкається практичної інтеграції з нашим доменом даних. Для «*Sold_items*» денний інкремент і авто-*suspend* у *Snowflake* забезпечує мінімізацію вартості при збереженні *SLA*; для «*Sold_items_archive*» нічні батчі на

більшому складі з кластеризацією й рекластером нових ділянок — це лінійна, легко прогнозована економіка у кредитах. У паралельному порівнянні на *MSSQL*/*AWS* аналогічне *ETL*-вікно вимагатиме або постійного інстансу (*RDS*), або зваженого масштабування *EC*2 плюс ліцензійний менеджмент; у підсумку, для аналітичних *DWH Snowflake* має перевагу в еластичності витрат, тоді як *MSSQL* на *AWS* має сенс там, де транзакційний стек і операційні залежності потребують *SQL Server* як ядро платформи.

ВИСНОВКИ

Кваліфікаційна робота «Оптимізація продуктивності корпоративних *DWH*-систем у багатоплатформенному середовищі за допомогою *ETL* процесів» присвячена дослідженню роботи із дата сховищами. У ході виконання роботи були розрозбені шаблонні *ETL* джоби як основа стандаоту для розробки нових джобві, джоб для аналізу продуктивності *DWH* та сформовано цілісну методологію оптимізації продуктивності корпоративних *DWH*-систем у мультиплатформенному середовищі з фокусом на двох технологічних стовпах — *MSSQL*, розгорнутому на *AWS EC*2, та хмарній платформі *Snowflake* з оркестрацією процесів засобами *Matillion ETL*. Теоретична частина дала підґрунтя для формалізації ключових метрик продуктивності, фінансових драйверів та архітектурних принципів, тоді як прикладна частина продемонструвала, як дизайн даних, індексаційні стратегії, керування історичністю та напівструктурованими даними безпосередньо трансформуються у вимірювані вигоди: зменшення часу виконання запитів, стабілізацію *SLA*, скорочення вартості обчислень і зниження операційних ризиків, пов’язаних із блокуваннями, фрагментацією та деградацією статистики.

Основним практичним результатом для *MSSQL* на *AWS EC*2 стало обґрунтоване розділення гарячих і холодних даних через комбінацію короткочасної «робочої» таблиці *Sold_items* (горизонт 6 місяців), архівної таблиці *Sold_items_archive* та виділеної таблиці *Sold_items_big_values* для великих текстових та *JSON*-полів. Така декомпозиція дозволила водночас звузити робочий набір даних для типових звітів і знизити навантаження на буферний кеш, журнали транзакцій і підсистему *I*/*O*. Партиціювання архівної таблиці за часовим ключем мінімізувало накладні витрати на сканування, сприяло локалізованій підтримці індексів та прискорило *TTL*-операції, а також дало можливість швидкого «обслуговування» історичних шарів без впливу на онлайн-запити. Винесення великих колонок з *nvarchar*(*max*) та *VARIANT*/*JSON* у окрему таблицю

з посиланнями за *ID* суттєво зменшило ширину рядка в основному факті, збільшивши «рядків на сторінку» і, відповідно, щільність корисних даних у пам’яті та на диску, що прямо відбилося на *latency* типових аналітичних селектів.

Індексні стратегії було узгоджено зі схемами доступу: для оперативної *Sold_items* перевага віддавалася покриттю найчастіших фільтрів та приєднань, а для архіву — вузьким, селективним індексам, релевантним сценаріям ретроспективного аналізу. Розв’язання проблеми фрагментації через регулярний контроль *fill factor* та вибір між *REBUILD*/*REORGANIZE* з урахуванням розмірів партицій і часових вікон обслуговування стабілізувало продуктивність без «ефекту відскоку». Критично важливою виявилася дисципліна статистики: автоматичне оновлення із належним семплуванням, а також ручне переобчислення для великих і «важких» розподілів запобігли планам з неоптимальними кардинальностями. Аналіз блокувань і конкуренції ресурсів дозволив виявити «довгі транзакції», потенційно проблемні тригери та обмеження, що ініціювали ескалацію блокувань; після рефакторингу тригерів та усунення зайвих *DML*-операцій у критичних шляхах було досягнуто помітного зменшення часу очікувань і кількості *deadlocks*. У сукупності ці заходи на рівні *MSSQL* зменшили витрати на *EC*2/*IOPS* за рахунок меншої інтенсивності *I*/*O* та скорочення пікових віртуальних ресурсів, а також забезпечили прогнозованість часу виконання звітів у межах встановлених *SLA*.

У частині *Snowflake* і *Matillion ETL* ключовим досягненням стало формування керованої моделі витрат і продуктивності через коректне масштабування віртуальних сховищ, раціональне використання кешів результатів, конструювання кластерних ключів та обережне поводження з напівструктурованими даними у *VARIANT*. Вибір розміру *warehouse* став не емпіричним, а даними керованим рішенням: батчі *ETL* були сегментовані й паралелізовані настільки, щоб забезпечити оптимальний баланс між тривалістю завдань і загальною кількістю кредитів, уникаючи «надмірного» масштабування. Кешування результатів запитів, повторне використання тимчасових таблиць і проміжних артефактів *ETL* знизили повторні обчислення, а кластерні ключі на

великих таблицях поліпшили мікропрунімг і стабільність швидкодії для селектів з типовими предикатами. Робота з *VARIANT/JSON* була організована з чітким поділом: оперативний аналітичний шар — нормалізований, напівструктуровані поля — або приводяться до колонкового вигляду в межах матеріалізованих представлень, або зберігаються у «широких» об'єктах з вузькоспеціалізованим доступом, що запобігло «розповзанню» витрат на універсальні, але дорогі трансформації.

Matillion ETL виконав роль дисципліни процесів: параметризація, контроль іде*mpot*ентності, чіткий поділ на стадії *Ingest—Stage—Transform—Serve*, а також керування спостережністю через монітори, квоти і теги надали можливість відстежувати ефект оптимізацій у числах. Верифікація включала заміри до/після із фіксацією часу виконання, обсягу просканованих даних та спожитих кредитів, що дозволило кількісно підтвердити ефективність прийнятих рішень і закріпити їх як операційні стандарти. У підсумку сформовано набір практичних правил: будувати *pipeline*'и навколо стабільних ключів партиціювання та кластеризації, розділяти великі джерела на природні «шарди» часу або бізнес-сутностей, застосовувати мікро-батчі там, де це зменшує хвости черг без втрати ефективності, і завжди підтримувати прозорість витрат на рівні завдання, таблиці й віртуального сховища.

Порівняльний аналіз вартості обчислень у *Snowflake* та *MSSQL* на *AWS* висвітлив нетривіальну закономірність: за відсутності оптимізацій і дисципліни доступу загальна вартість може бути співставною або непередбачувано вищою в обох середовищах; проте за наявності структурних покращень — партиціювання, правильні індекси/кластерні ключі, контроль ширини рядка, уникнення зайвих сканів — обидві платформи демонструють добру масштабованість із різною еластичністю витрат. *MSSQL* на *EC2* передбачає фіксовані витрати інфраструктури з чутливістю до профілю *I/O* та зберігання, натомість *Snowflake* надає лінійні механізми керування *compute*-кредитами, які особливо вигідні при переривчастих, але інтенсивних навантаженнях і при високому ступені повторного використання кешів. Отже, вибір платформи для конкретного

сценарію має спиратися не лише на миттєву ціну ресурсу, а на сукупну економіку життєвого циклу запиту й *pipeline*'у: щільність даних, варіативність запитів, частоту перерахунків, вимоги до латентності, вікно обробки та рівень командної зрілості в експлуатації.

Важливою науково-практичною новизною роботи є інтегрований підхід до оптимізації на стику архітектури даних, інструментів *ETL* та фінансових метрик. Показано, що оптимізація «локальних» артефактів — наприклад, винесення великих полів у *Sold_items_big_values* або введення кластерних ключів у *Snowflake* — дає максимальний ефект лише в контексті правильно організованих процесів завантаження, ретенції, архівації та валідації узгодженості даних. Роль *ETL*-процесів у забезпеченні цілісності та конзистентності даних розкрита через конкретні приклади задач у *Matillion*: контроль дублікатів і вікон змін, дедуплікація при інкрементальних завантаженнях, верифікація цілісності між фактом і вимірами, облік відкладених або повторних подій з *API*. Цим самим продемонстровано, що «продуктивність» у *DWH* — це не лише про швидкість запитів, а й про керованість, повторюваність і передбачуваність процесів, які живлять аналітичні вітрини.

Отримані результати мають безпосередню застосовність у промислових контекстах, де обсяги даних зростають, а вимоги до *SLA*, прозорості витрат і регуляторної підзвітності посилюються. Для організацій, які експлуатують гібридні ландшафти з поєднанням *on-prem*/*EC*2 *MSSQL* і *Snowflake*, запропоновані підходи можуть стати базою для стандартизації життєвого циклу даних: визначення політик ретенції, сегментації даних за температурними зонами, каталогізації напівструктурованих полів, вибору гранулярності батчів, а також впровадження наскрізної спостережності за витратами та продуктивністю. Водночас робота окреслює межі застосовності окремих рішень: ефективність партиціювання залежить від селективності предикатів і кардинальності ключів, переваги кластерних ключів у *Snowflake* відчутні лише за наявності стабільних патернів доступу, а агресивна паралельність *ETL* має враховувати конкуренцію за метадані та ресурсні ліміти.

Перспективи подальших досліджень полягають у розширенні порівняльного аналізу на додаткові платформи (*Synapse*, *BigQuery*, *Databricks SQL*), формалізації моделей *TCO*/*TOE* з урахуванням вартості володіння інструментами оркестрації та спостережності, а також у побудові автоматизованих фреймворків «*policy-as-code*» для динамічного керування партиціюванням, індексацією та ресурсними квотами залежно від спостережуваних патернів запитів. Окремого вивчення заслуговує тема інтелектуального планування *ETL*/*ELT* на базі телеметрії, що дозволить автоматично підбирати розмір *warehouse*, ступінь паралельності та стратегії кешування під конкретні робочі навантаження.

Підсумовуючи, робота підтверджує, що системна оптимізація мультиплатформенного *DWH* — це міждисциплінарне завдання на перетині архітектури даних, інженерії процесів і економіки обчислень. Поєднання структурних технік (партиціювання, індекси, декомпозиція широких рядків), платформених важелів (кластерні ключі, кешування, керовані сховища) та процесних практик (ідемпотентні *ETL*, спостережність витрат, узгодженість даних) забезпечує не лише приріст швидкодії, а й керованість, масштабованість, передбачуваність і фінансову стійкість аналітичних систем. Для організацій, що експлуатують *MSSQL* на *AWS* поряд зі *Snowflake*, це означає можливість перейти від «реактивного» гасіння пожеж продуктивності до «проактивної» експлуатації, де кожне інженерне рішення аргументоване даними і вимірюваною економією ресурсів.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. *Kimball, R., Ross, M. The data warehouse toolkit: The definitive guide to dimensional modeling. 4rd ed. Hoboken: Wiley, 2022. 600 p.*

2. *Golfarelli, M., Rizzi, S. Data warehouse design: Modern principles and methodologies. 2nd ed. New York: McGraw-Hill, 2021. 520 p.*

3. *Microsoft Corporation. SQL Server query performance best practices. Redmond: Microsoft, 2023. URL: https://learn.microsoft.com/en-us/sql/relational-databases/performance/*

4. *Microsoft Corporation. Index architecture and design. Redmond: Microsoft, 2023. URL: https://learn.microsoft.com/en-us/sql/relational-databases/indexes/indexes*

5. *Snowflake Inc. Snowflake architecture and overview. Bozeman: Snowflake, 2023. URL: https://docs.snowflake.com/en/user-guide/intro-key-concepts*

6. *Vassiliadis, P., Simitsis, A. Extract–transform–load (ETL) processes. In: Encyclopedia of Big Data Technologies. Cham: Springer, 2022. 35 p.*

7. *Dageville, B., et al. The Snowflake elastic data warehouse // Proceedings of the 2016 International Conference on Management of Data (SIGMOD). ACM, 2016. P. 215–226. DOI: 10.1145/2882903.2903741.*

8. *Matillion Ltd. Matillion ETL documentation. 2023. URL: https://documentation.matillion.com/*

9. *Snowflake Inc. Clustering keys and reclustering // Snowflake Documentation. 2023. URL: https://docs.snowflake.com/en/user-guide/tables-clustering-keys*

10. *Matillion Ltd. Best practices for parallelism and performance // Matillion Blog. 2023. URL: https://www.matillion.com/resources/blog/*

11. *Snowflake Inc. Resource monitors // Snowflake Documentation. 2023. URL: https://docs.snowflake.com/en/user-guide/resource-monitors*

12. *Snowflake Inc. Streams and tasks // Snowflake Documentation. 2024. URL: https://docs.snowflake.com/en/user-guide/streams-intro*

13. *Microsoft Corporation. Enable and use Change Data Capture (CDC) // SQL Server Documentation. 2023. URL: https://learn.microsoft.com/en-us/sql/relational-databases/track-changes/about-change-data-capture-sql-server*

14. *Snowflake Inc. Account usage and query history // Snowflake Documentation. 2024. URL: https://docs.snowflake.com/en/sql-reference/account-usage*

15. *Snowflake Inc. Query result caching // Snowflake Documentation. 2024. URL: https://docs.snowflake.com/en/user-guide/query-caching*

16. *Snowflake Inc. System functions and views for clustering // Snowflake Documentation. 2024. URL: https://docs.snowflake.com/en/sql-reference/functions/system-clustering-information*

17. *Jukic, N., Vrbsky, S. V., Nestorov, S. Database systems: design, implementation, & management. Boston: Cengage Learning, 2022. 688 p.*

18. *Microsoft Corporation. Enable and use Change Data Capture (CDC) // SQL Server Documentation. 2023. URL: https://learn.microsoft.com/en-us/sql/relational-databases/track-changes/about-change-data-capture-sql-server*

19. *Snowflake Inc. JSON functions and user-defined functions // Snowflake Documentation. 2024. URL: https://docs.snowflake.com/en/sql-reference/functions-JSON*

20. *Vassiliadis, P. A survey of extract–transform–load technology // International Journal of Data Warehousing and Mining. 2009. Vol. 5, No. 3. P. 1–27. DOI: 10.4018/jdwm.2009070101.*

21. *Matillion Ltd. API Query component // Matillion Documentation. 2024. URL: https://documentation.matillion.com/docs/api-query*

22. *Matillion Ltd. API Query component // Matillion Documentation. 2024. URL: https://documentation.matillion.com/docs/api-query*

23. *Snowflake Inc. Understanding Compute Costs // Snowflake Documentation. URL: https://docs.snowflake.com/en/user-guide/cost-understanding-compute*

24. *Snowflake Inc. Understanding Overall Costs // Snowflake Documentation. URL: https://docs.snowflake.com/en/user-guide/cost-understanding-overall*

25. *Snowflake Inc. ALTER WAREHOUSE (SQL Command) // Snowflake Documentation. URL: https://docs.snowflake.com/en/sql-reference/sql/alter-warehouse*

26. *Amazon Web Services. Amazon RDS for SQL Server Pricing. URL: https://aws.amazon.com/rds/sqlserver/pricing/*

27. *Amazon Web Services. Amazon EC2 On-Demand Pricing. URL: https://aws.amazon.com/ec2/pricing/on-demand/*

28. *Snowflake Inc. Micro-partitions & Data Clustering // Snowflake Documentation. URL: https://docs.snowflake.com/en/user-guide/tables-clustering-micropartitions*

29. *Snowflake Inc. Optimizing the Warehouse Cache // Snowflake Documentation. URL: https://docs.snowflake.com/en/user-guide/performance-query-warehouse-cache*

Процедура для архівування даних

```
CREATE PROCEDURE Archive_Sold_Items
AS
BEGIN
  SET NOCOUNT ON;
  DECLARE @ArchiveDate DATETIME = DATEADD(MONTH, -6, GETDATE());
  BEGIN TRY
    -- 1. Перенесення даних в архів
    INSERT INTO Sold_items_archive (ID, Updated)
    SELECT ID, Updated
    FROM Sold_items
    WHERE Updated < @ArchiveDate;
    -- 2. Видалення архівованих даних з основної таблиці
    DELETE FROM Sold_items
    WHERE Updated < @ArchiveDate;
    -- 3. Оптимізація після архівування
    UPDATE STATISTICS Sold_items;
    UPDATE STATISTICS Sold_items_archive;
    PRINT 'Архівування виконано успішно. Дані старші за ' + CONVERT(VARCHAR(20), @ArchiveDate) + ' перенесено.';
  END TRY
  BEGIN CATCH
    PRINT 'Помилка архівування: ' + ERROR_MESSAGE();
  END CATCH
```

Додаток 2

Створення *Partition Function*

Функція партиціювання визначає межі партицій за датою. У нашому випадку дані розбиваються за роками:

```
CREATE PARTITION FUNCTION pfUpdatedDate (DATE)
AS RANGE LEFT FOR VALUES (
   '2018-12-31',
   '2019-12-31',
   '2020-12-31',
   '2021-12-31',
   '2022-12-31',
   '2023-12-31',
   '2024-12-31'
);
```

Створення *Partition Scheme*

Схема партиціювання описує, у яких файлових групах будуть розміщені партиції. Для прикладу всі партиції розміщуються у групі *PRIMARY*:

```
CREATE PARTITION SCHEME psUpdatedDate
AS PARTITION pfUpdatedDate
ALL TO ([PRIMARY]);
```

Створення архівної таблиці з партиціюванням

```
CREATE TABLE Sold_items_archive (
   ID BIGINT NOT NULL,
   Updated DATE NOT NULL,
  … -- Інші колонки, якщо вони є
)
ON psUpdatedDate(Updated);
```

Приклад видалення старої партиції

```
ALTER TABLE Sold_items_archive
SWITCH PARTITION 1 TO Sold_items_archive_2018;
--Перевірка розподілу партицій
SELECT
   p.partition_number,
   p.rows,
   pf.name AS PartitionFunction,
   ps.name AS PartitionScheme
FROM sys.partitions p
JOIN sys.objects o ON p.object_id = o.object_id
JOIN sys.indexes i ON p.object_id = i.object_id AND p.index_id = i.index_id
JOIN sys.partition_schemes ps ON i.data_space_id = ps.data_space_id
JOIN sys.partition_functions pf ON ps.function_id = pf.function_id
WHERE o.name = 'Sold_items_archive';
```

Для зручності та можливості автоматизації цього процесу, доцільно сформувати процедуру, код :

```
CREATE PROCEDURE CreatePartitionedArchiveTable
AS
BEGIN
   SET NOCOUNT ON;
   BEGIN TRY
      -- 1. Створення Partition Function
      IF NOT EXISTS (SELECT * FROM sys.partition_functions WHERE name = 'pfUpdatedDate')
      BEGIN
         CREATE PARTITION FUNCTION pfUpdatedDate (DATE)
         AS RANGE LEFT FOR VALUES (
```

```
            '2018-12-31',
            '2019-12-31',
            '2020-12-31',
            '2021-12-31',
            '2022-12-31',
            '2023-12-31',
            '2024-12-31'
        );
        PRINT 'Partition Function створено.';
    END
    ELSE
        PRINT 'Partition Function вже існує.';

    -- 2. Створення Partition Scheme
    IF NOT EXISTS (SELECT * FROM sys.partition_schemes WHERE name
= 'psUpdatedDate')
    BEGIN
        CREATE PARTITION SCHEME psUpdatedDate
        AS PARTITION pfUpdatedDate
        ALL TO ([PRIMARY]);
        PRINT 'Partition Scheme створено.';
    END
    ELSE
        PRINT 'Partition Scheme вже існує.';

    -- 3. Створення архівної таблиці
    IF NOT EXISTS (SELECT * FROM sys.objects WHERE name =
'Sold_items_archive' AND type = 'U')
```

```
        BEGIN
            CREATE TABLE Sold_items_archive (
                ID BIGINT NOT NULL,
                Updated DATE NOT NULL
            )
            ON psUpdatedDate(Updated);
            PRINT 'Архівна таблиця створена.';
        END
        ELSE
            PRINT 'Архівна таблиця вже існує.';

        -- 4. Перевірка розподілу партицій
        PRINT 'Для перевірки партицій використайте DMV: sys.partitions, sys.partition_schemes, sys.partition_functions.';
    END TRY
    BEGIN CATCH
        PRINT 'Помилка: ' + ERROR_MESSAGE();
    END CATCH
END;
```

Процедура для роботи із великими типами даних

```
CREATE PROCEDURE Optimize_SoldItems_Structure
AS
BEGIN
   SET NOCOUNT ON;

   BEGIN TRY
      -- 1. Створення нової таблиці для великих значень
      IF NOT EXISTS (SELECT * FROM sys.objects WHERE name = 'Sold_items_big_values' AND type = 'U')
      BEGIN
         CREATE TABLE Sold_items_big_values (
            ID BIGINT NOT NULL PRIMARY KEY,
            Comment NVARCHAR(MAX) NULL
         );
         PRINT 'Таблиця Sold_items_big_values створена.';
      END
      ELSE
         PRINT 'Таблиця Sold_items_big_values вже існує.';

      -- 2. Перенесення даних з основної таблиці
      INSERT INTO Sold_items_big_values (ID, Comment)
      SELECT ID, Comment
      FROM Sold_items
      WHERE Comment IS NOT NULL;

      PRINT 'Дані перенесено в Sold_items_big_values.';
```

```
-- 3. Видалення колонки Comment з основної таблиці
IF EXISTS (SELECT * FROM sys.columns WHERE Name = 'Comment' AND Object_ID = Object_ID('Sold_items'))
BEGIN
    ALTER TABLE Sold_items DROP COLUMN Comment;
    PRINT 'Колонка Comment видалена з Sold_items.';
END

-- 4. Зміна типу ID на BIGINT (якщо не BIGINT)
DECLARE @CurrentType NVARCHAR(50);
SELECT @CurrentType = DATA_TYPE
FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_NAME = 'Sold_items' AND COLUMN_NAME = 'ID';

IF @CurrentType <> 'bigint'
BEGIN
    ALTER TABLE Sold_items ALTER COLUMN ID BIGINT NOT NULL;
    PRINT 'Тип колонки ID змінено на BIGINT.';
END
ELSE
    PRINT 'Тип колонки ID вже BIGINT.';

-- 5. Створення індексів
IF NOT EXISTS (SELECT * FROM sys.indexes WHERE name = 'IX_SoldItems_ID')
BEGIN
    CREATE INDEX IX_SoldItems_ID ON Sold_items(ID);
END
```

```
        IF NOT EXISTS (SELECT * FROM sys.indexes WHERE name =
'IX_SoldItemsBigValues_ID')
        BEGIN
          CREATE INDEX IX_SoldItemsBigValues_ID ON
Sold_items_big_values(ID);
        END

        PRINT 'Індекси створено.';
      END TRY
      BEGIN CATCH
        PRINT 'Помилка: ' + ERROR_MESSAGE();
      END CATCH
    END;
```

Додаток 4

SQL-скрипти для роботи з індексами

1. Створення індексів для таблиць

```
-- ========================================
-- Індексація таблиці Sold_items
-- ========================================
CREATE CLUSTERED INDEX IX_SoldItems_ID ON Sold_items(ID);

CREATE NONCLUSTERED INDEX IX_SoldItems_Updated ON Sold_items(Updated);

CREATE NONCLUSTERED INDEX IX_SoldItems_Recent
ON Sold_items(Updated)
WHERE Updated >= DATEADD(MONTH, -6, GETDATE());

-- ========================================
-- Індексація таблиці Sold_items_archive
-- ========================================
CREATE CLUSTERED INDEX IX_SoldItemsArchive_Updated ON Sold_items_archive(Updated);

CREATE CLUSTERED COLUMNSTORE INDEX CCI_SoldItemsArchive ON Sold_items_archive;

-- ========================================
-- Індексація таблиці Sold_items_big_values
-- ========================================
CREATE CLUSTERED INDEX IX_SoldItemsBigValues_ID ON Sold_items_big_values(ID);
```

CREATE FULLTEXT INDEX ON Sold_items_big_values(*Comment*)

KEY INDEX IX_SoldItemsBigValues_ID;

3. Перевірка стану індексів

Список індексів у таблиці

SELECT name, *type_desc*, *is_disabled FROM sys.indexes WHERE object_id = OBJECT_ID*('*Sold_items*');

Статистика використання індексів

SELECT OBJECT_NAME(*s*.[*object_id*]) *AS TableName*, *i.name AS IndexName*,

user_seeks, *user_scans*, *user_lookups*, *user_updates*

FROM sys.dm_db_index_usage_stats AS s

JOIN sys.indexes AS i ON s.[*object_id*] = *i*.[*object_id*] *AND s.index_id = i.index_id*;Фрагментація індексів

*SELECT * FROM sys.dm_db_index_physical_stats*(*DB_ID*(), *OBJECT_ID*('*Sold_items*'), *NULL*, *NULL*, '*DETAILED*');

4. Оптимізація та обслуговування

Перебудова індексів

ALTER INDEX ALL ON Sold_items REBUILD;

Реорганізація індексів

ALTER INDEX ALL ON Sold_items_archive REORGANIZE;

Оновлення статистики

UPDATE STATISTICS Sold_items;

UPDATE STATISTICS Sold_items_archive;

UPDATE STATISTICS Sold_items_big_values;

Процедура для реіндексації таблиць

CREATE PROCEDURE Reindex_All_Tables

AS

```
BEGIN
    SET NOCOUNT ON;

    DECLARE @TableName NVARCHAR(255);
    DECLARE @SQL NVARCHAR(MAX);

    -- Курсор для перебору всіх таблиць у базі
    DECLARE TableCursor CURSOR FOR
    SELECT name FROM sys.tables WHERE is_ms_shipped = 0;

    OPEN TableCursor;
    FETCH NEXT FROM TableCursor INTO @TableName;

    WHILE @@FETCH_STATUS = 0
    BEGIN
        -- Формуємо команду для реіндексації всіх індексів на таблиці
        SET @SQL = 'ALTER INDEX ALL ON [' + @TableName + '] REBUILD;';
        PRINT 'Виконується реіндексація для таблиці: ' + @TableName;

        EXEC sp_executesql @SQL;

        FETCH NEXT FROM TableCursor INTO @TableName;
    END

    CLOSE TableCursor;
    DEALLOCATE TableCursor;
END;
```

Оновлення статистики

```
-- Увімкнення автоматичного створення та оновлення статистики
ALTER DATABASE [YourDB]
SET AUTO_CREATE_STATISTICS ON;
ALTER DATABASE [YourDB]
SET AUTO_UPDATE_STATISTICS ON;
ALTER DATABASE [YourDB]
SET AUTO_UPDATE_STATISTICS_ASYNC ON; -- або OFF, залежно від сценарію

-- Перевірка поточного стану параметрів
SELECT name,
    is_auto_create_stats_on,
    is_auto_update_stats_on,
    is_auto_update_stats_async_on
FROM sys.databases
WHERE name = N'YourDB';

-- Оновлення статистики після масового завантаження даних
EXEC sp_updatestats; -- швидке оновлення змінених статистик із семплом

-- Для критичних таблиць – повне сканування
UPDATE STATISTICS dbo.FactSales WITH FULLSCAN;

-- Перевірка властивостей статистики для конкретної таблиці
SELECT s.name AS stats_name,
    sp.last_updated,
    sp.rows,
```

```
    sp.rows_sampled,
     sp.modification_counter
FROM sys.stats AS s
CROSS APPLY sys.dm_db_stats_properties(s.object_id, s.stats_id) AS sp
WHERE s.object_id = OBJECT_ID(N'dbo.FactSales');
```

Автоматизований аудит статистики

```
/*
  Audit & Refresh Statistics for DWH (with focus on Sold_items tables)
  Author: Bohdan Prokopets
  Position: IT Team leader PMI UA
  Purpose: Assess staleness, prioritize updates, and optionally refresh stats.
  Safe for: SQL Server 2016+ (recommended compat level 130+)
   */

SET NOCOUNT ON;

-------------------------------------------------------------------------------
-- Parameters
DECLARE @RunRefresh BIT = 1;
-- 1 = perform UPDATE STATISTICS, 0 = audit only
DECLARE @PreferFullScan_SoldItems BIT = 1;
-- 1 = FULLSCAN for dbo.Sold_items
DECLARE @FullScan_ArchiveCritical BIT = 0;

-- 1 = FULLSCAN only for named critical archive stats
DECLARE @MaxAgeHours INT = 24;
-- highlight stats older than this (hours)
DECLARE @ModThreshold BIGINT = 100000;
 -- highlight stats with modification_counter ≥ threshold for big tables
DECLARE @Verbose BIT = 1;
-- print dynamic statements

-------------------------------------------------------------------------------
```

```
-- Context: object ids
DECLARE @obj_SoldItems INT        = OBJECT_ID(N'dbo.Sold_items');
DECLARE @obj_SoldItemsArchive INT = OBJECT_ID(N'dbo.Sold_items_archive');
DECLARE @obj_SoldItemsBig INT     = OBJECT_ID(N'dbo.Sold_items_big_values');

-------------------------------------------------------------------------------
-- Safety checks
IF @obj_SoldItems IS NULL
BEGIN
   RAISERROR('Table dbo.Sold_items not found.', 10, 1);
END
IF @obj_SoldItemsArchive IS NULL
BEGIN
   RAISERROR('Table dbo.Sold_items_archive not found.', 10, 1);
END
IF @obj_SoldItemsBig IS NULL
BEGIN
   RAISERROR('Table dbo.Sold_items_big_values not found.', 10, 1);
END
-------------------------------------------------------------------------------
-- Staging tables for audit
IF OBJECT_ID('tempdb..#stats_raw') IS NOT NULL DROP TABLE #stats_raw;
IF OBJECT_ID('tempdb..#stats_audit') IS NOT NULL DROP TABLE #stats_audit;

CREATE TABLE #stats_raw
```

```
(
    object_id INT NOT NULL,
    stats_id INT NOT NULL,
    table_name SYSNAME NULL,
    stats_name SYSNAME NULL,
    auto_created BIT NULL,
    user_created BIT NULL,
    no_recompute BIT NULL,
    has_filter BIT NULL,
    last_updated DATETIME2 NULL,
    rows BIGINT NULL,
    rows_sampled BIGINT NULL,
    unfiltered_rows BIGINT NULL,
    modification_counter BIGINT NULL
);

CREATE TABLE #stats_audit
(
    schema_name SYSNAME,
    table_name SYSNAME,
    stats_name SYSNAME,
    is_target BIT,              -- our three focus tables
    auto_created BIT,
    user_created BIT,
    no_recompute BIT,
    has_filter BIT,
    last_updated DATETIME2,
    hours_since_update INT,
```

```
    rows_sampled BIGINT,
    unfiltered_rows BIGINT,
    modification_counter BIGINT,
    stale_by_mod BIT,
    stale_by_age BIT,
    priority INT,              -- lower is higher priority
    recommended_action NVARCHAR(50), -- 'FULLSCAN' | 'DEFAULT' | 'SKIP'
    rationale NVARCHAR(4000)
);

-------------------------------------------------------------------------
-- Capture all stats with properties (whole DB)

INSERT INTO #stats_raw (object_id, stats_id, table_name, stats_name,
auto_created, user_created, no_recompute, has_filter,
                        last_updated, rows, rows_sampled, unfiltered_rows,
modification_counter)
SELECT
    s.object_id,
    s.stats_id,
    OBJECT_NAME(s.object_id),
    s.name,
    s.auto_created,
    s.user_created,
    s.no_recompute,
    s.has_filter,
    sp.last_updated,
    sp.rows,
```

```
    sp.rows_sampled,
    sp.unfiltered_rows,
    sp.modification_counter
FROM sys.stats AS s
CROSS APPLY sys.dm_db_stats_properties(s.object_id, s.stats_id) AS sp
WHERE OBJECTPROPERTY(s.object_id, 'IsMsShipped') = 0;

--------------------------------------------------------------------------------
-- Build audit and compute staleness indicators
INSERT INTO #stats_audit (schema_name, table_name, stats_name,
is_target, auto_created, user_created, no_recompute, has_filter,
                          last_updated, hours_since_update, rows,
rows_sampled, unfiltered_rows, modification_counter,
                          stale_by_mod, stale_by_age, priority,
recommended_action, rationale)
SELECT
    OBJECT_SCHEMA_NAME(r.object_id),
    r.table_name,
    r.stats_name,
    CASE WHEN r.object_id IN (@obj_SoldItems,
@obj_SoldItemsArchive, @obj_SoldItemsBig)
 THEN 1 ELSE 0 END AS is_target,
    r.auto_created,
    r.user_created,
    r.no_recompute,
    r.has_filter,
    r.last_updated,
    CASE WHEN r.last_updated IS NULL THEN NULL
```

```
        ELSE DATEDIFF(HOUR, r.last_updated, SYSDATETIME())
    END AS hours_since_update,
    r.rows,
    r.rows_sampled,
    r.unfiltered_rows,
    r.modification_counter,
    CASE
        -- Treat NULL as stale for safety (e.g., newly created stats)
        WHEN r.modification_counter IS NULL THEN 1
        WHEN r.rows >= 1000000 AND r.modification_counter
>= @ModThreshold THEN 1
        WHEN r.rows <  1000000 AND r.modification_counter >= 50000 THEN 1
        ELSE 0
    END AS stale_by_mod,
    CASE
        WHEN r.last_updated IS NULL THEN 1
        WHEN DATEDIFF(HOUR, r.last_updated,
SYSDATETIME()) >= @MaxAgeHours THEN 1
        ELSE 0
    END AS stale_by_age,
    NULL,   -- priority
    NULL,   -- recommended_action
    NULL    -- rationale
FROM #stats_raw AS r;

-------------------------------------------------------------------------
-- Recommendation rules
-- Priority ladder:
```

```
-- 1: dbo.Sold_items (FULLSCAN if requested) or
-- critical stale + no_recompute
-- 2: dbo.Sold_items_archive critical entries
--(opt-in FULLSCAN) otherwise DEFAULT
-- 3: dbo.Sold_items_big_values DEFAULT; targeted
--FULLSCAN for specific JSON-derived indexes (if present)
-- 4: Others based on staleness
-------------------------------------------------------------------------------
;WITH R AS
(
  SELECT *
  FROM #stats_audit
)
UPDATE R
SET
  priority =
    CASE
      WHEN is_target = 1 AND table_name = N'Sold_items' THEN 1
      WHEN is_target = 1 AND table_name = N'Sold_items_archive' THEN 2
      WHEN is_target = 1 AND table_name = N'Sold_items_big_values'
THEN 3
      WHEN stale_by_mod = 1 OR stale_by_age = 1 THEN 4
      ELSE 5
    END,
  recommended_action =
    CASE
      WHEN table_name = N'Sold_items' AND
is_target = 1 AND @PreferFullScan_SoldItems = 1 THEN N'FULLSCAN'
```

```
        WHEN table_name = N'Sold_items_archive' AND
is_target = 1 AND @FullScan_ArchiveCritical = 1
            AND stats_name IN (N'IX_Sold_items_archive_Date')
THEN N'FULLSCAN'
        WHEN table_name = N'Sold_items_big_values' AND is_target = 1
            AND stats_name IN (N'IX_SIBV_OrderType', N'IX_SIBV_Region')
 THEN N'FULLSCAN'
        WHEN stale_by_mod = 1 OR stale_by_age = 1 THEN N'DEFAULT'
        ELSE N'SKIP'
      END,
   rationale =
      CONCAT(
        CASE WHEN stale_by_mod = 1 THEN N'[Stale by modifications] '
ELSE N'' END,
        CASE WHEN stale_by_age = 1 THEN N'[Stale by age] '
ELSE N'' END,
        CASE WHEN no_recompute = 1 THEN N'[NORECOMPUTE] '
ELSE N'' END,
        CASE WHEN has_filter   = 1 THEN N'[Filtered] '     ELSE N'' END
      );

-------------------------------------------------------------------------------
-- Optional refresh
-------------------------------------------------------------------------------
IF @RunRefresh = 1
BEGIN
   DECLARE @sql NVARCHAR(MAX);
```

```
    -- Sold_items first (critical)
    IF EXISTS (SELECT 1 FROM #stats_audit WHERE is_target = 1
AND table_name = N'Sold_items')
    BEGIN
        IF @PreferFullScan_SoldItems = 1
            SET @sql = N'UPDATE STATISTICS dbo.Sold_items
WITH FULLSCAN;';
        ELSE
            SET @sql = N'UPDATE STATISTICS dbo.Sold_items;';
        IF @Verbose = 1 PRINT @sql;
        EXEC sp_executesql @sql;
    END

    -- Archive default update, then optional FULLSCAN for critical index stats
    IF EXISTS (SELECT 1 FROM #stats_audit WHERE is_target = 1
AND table_name = N'Sold_items_archive')
    BEGIN
        SET @sql = N'UPDATE STATISTICS dbo.Sold_items_archive;';

        IF @Verbose = 1 PRINT @sql;
        EXEC sp_executesql @sql;

        IF @FullScan_ArchiveCritical = 1
          AND EXISTS (SELECT 1 FROM #stats_audit
WHERE is_target = 1 AND table_name = N'Sold_items_archive'
                    AND stats_name = N'IX_Sold_items_archive_Date')
        BEGIN
            SET @sql = N'UPDATE
```

```
        STATISTICS dbo.Sold_items_archive(IX_Sold_items_archive_Date)
        WITH FULLSCAN;';
            IF @Verbose = 1 PRINT @sql;
            EXEC sp_executesql @sql;
        END
    END

    -- Big values default update; optional targeted
    --FULLSCAN if JSON-derived indices exist

    IF EXISTS (SELECT 1 FROM #stats_audit WHERE is_target = 1
    AND table_name = N'Sold_items_big_values')
    BEGIN
        SET @sql = N'UPDATE STATISTICS dbo.Sold_items_big_values;';
        IF @Verbose = 1 PRINT @sql;
        EXEC sp_executesql @sql;

        IF EXISTS (SELECT 1 FROM #stats_audit WHERE is_target = 1 AND
table_name = N'Sold_items_big_values'
                AND stats_name IN (N'IX_SIBV_OrderType', N'IX_SIBV_Region'))
        BEGIN
            IF EXISTS (SELECT 1 FROM #stats_audit WHERE table_name =
N'Sold_items_big_values' AND stats_name = N'IX_SIBV_OrderType')
            BEGIN
                SET @sql = N'UPDATE STATISTICS
dbo.Sold_items_big_values(IX_SIBV_OrderType) WITH FULLSCAN;';
                IF @Verbose = 1 PRINT @sql;
                EXEC sp_executesql @sql;
```

```
            END
            IF EXISTS (SELECT 1 FROM #stats_audit WHERE table_name =
N'Sold_items_big_values' AND stats_name = N'IX_SIBV_Region')
            BEGIN
              SET @sql = N'UPDATE STATISTICS
dbo.Sold_items_big_values(IX_SIBV_Region) WITH FULLSCAN;';
              IF @Verbose = 1 PRINT @sql;
              EXEC sp_executesql @sql;
            END
          END
        END

        -- For the rest of the database: update only those marked
      --stale_by_mod/age, default sampling
        DECLARE cur CURSOR LOCAL FAST_FORWARD FOR
          SELECT QUOTENAME(OBJECT_SCHEMA_NAME(r.object_id)) AS
schema_name,
                 QUOTENAME(OBJECT_NAME(r.object_id)) AS table_name,
                 r.stats_name
          FROM #stats_raw r
          JOIN #stats_audit a
            ON a.table_name = r.table_name
           AND a.stats_name = r.stats_name
          WHERE a.is_target = 0
            AND (a.stale_by_mod = 1 OR a.stale_by_age = 1)
            AND a.recommended_action IN (N'DEFAULT') -- skip SKIP, FULLSCAN
handled explicitly
          ORDER BY a.priority, a.table_name, a.stats_name;
```

```
        DECLARE @sch SYSNAME, @tbl SYSNAME, @st SYSNAME;
        OPEN cur;
        FETCH NEXT FROM cur INTO @sch, @tbl, @st;
        WHILE @@FETCH_STATUS = 0
        BEGIN
            SET @sql = N'UPDATE STATISTICS ' + @sch + N'.' + @tbl + N' (' +
QUOTENAME(@st) + N');';
            IF @Verbose = 1 PRINT @sql;
            EXEC sp_executesql @sql;

            FETCH NEXT FROM cur INTO @sch, @tbl, @st;
        END
        CLOSE cur;
        DEALLOCATE cur;
    END

    ----------------------------------------------------------------------
    -- Final report
    SELECT
        schema_name,
        table_name,
        stats_name,
        is_target,
        auto_created,
        user_created,
        no_recompute,
        has_filter,
        last_updated,
```

```
    rows,
    rows_sampled,
    unfiltered_rows,
    modification_counter,
    stale_by_mod,
    stale_by_age,
    priority,
    recommended_action,
    rationale
FROM #stats_audit
ORDER BY
    CASE WHEN is_target = 1 THEN 0 ELSE 1 END,
    priority,
    table_name,
    stats_name;

-- Optional: quick glance at DB options related to stats
SELECT
    d.name,
    d.compatibility_level,
    d.is_auto_create_stats_on,
    d.is_auto_update_stats_on,
    d.is_auto_update_stats_async_on
FROM sys.databases AS d
WHERE d.name = DB_NAME();

SET NOCOUNT OFF;
```

Stored procedure для моніторингу блокувань і дедлоків

```
/*
  Procedure: admin.Monitor_LocksAndDeadlocks
  Purpose: Monitor blocking and deadlocks; email alert if blocking persists > @ThresholdMinutes
  Scheduling: Run via SQL Agent job (e.g., every 2 minute)
  Author: Bohdan Prokopets
 Position: IT Data pipeline developer PMI UA
 Created date: 02.02.2022
  */

-- 0) Службова схема та таблиці стану
IF NOT EXISTS (SELECT 1 FROM sys.schemas WHERE name = N'admin')
  EXEC('CREATE SCHEMA admin;');

IF OBJECT_ID(N'admin.BlockMonitorState', N'U') IS NULL
BEGIN
  CREATE TABLE admin.BlockMonitorState
  (
    blocking_session_id INT NOT NULL PRIMARY KEY,
    first_seen        DATETIME2(3) NOT NULL,
    last_seen         DATETIME2(3) NOT NULL,
    blocked_count     INT          NOT NULL,
    max_wait_ms       BIGINT       NOT NULL,
    db_name           SYSNAME      NULL,
    login_name        NVARCHAR(128) NULL,
    host_name         NVARCHAR(128) NULL,
    program_name      NVARCHAR(128) NULL,
```

```
    notified        BIT        NOT NULL DEFAULT 0
  );
END

IF OBJECT_ID(N'admin.DeadlockAlertCursor', N'U') IS NULL
BEGIN
  CREATE TABLE admin.DeadlockAlertCursor
  (
    id              INT IDENTITY(1,1) PRIMARY KEY,
    last_checked_time DATETIME2(3) NULL
  );
  INSERT INTO admin.DeadlockAlertCursor(last_checked_time) VALUES (NULL);
END

-- 1) Extended Events для дедлоків (создається один раз; зберігає у ring_buffer)
IF NOT EXISTS (SELECT 1 FROM sys.server_event_sessions WHERE name = N'DeadlockCapture')
BEGIN
  CREATE EVENT SESSION [DeadlockCapture] ON SERVER
  ADD EVENT sqlserver.xml_deadlock_report
  ADD TARGET package0.ring_buffer(SET max_events_limit = (1000))
  WITH (MAX_MEMORY = 4096 KB, EVENT_RETENTION_MODE = ALLOW_SINGLE_EVENT_LOSS, STARTUP_STATE = ON);
END
```

```
IF NOT EXISTS (SELECT 1 FROM sys.dm_xe_sessions WHERE name = N'DeadlockCapture')
BEGIN
  ALTER EVENT SESSION [DeadlockCapture] ON SERVER STATE = START;
END
GO

-- 2) Stored Procedure
CREATE OR ALTER PROCEDURE admin.Monitor_LocksAndDeadlocks
  @MailProfile       NVARCHAR(128),          -- профіль Database Mail
  @MailTo            NVARCHAR(400),          -- отримувач(і), через ; або ,
  @ThresholdMinutes  INT = 15,               -- поріг тривалості блокувань
  @IncludeDeadlocks  BIT = 1,                -- надсилати повідомлення про дедлоки
  @Verbose           BIT = 0                 -- додатковий PRINT
AS
BEGIN
  SET NOCOUNT ON;

  DECLARE @now DATETIME2(3) = SYSDATETIME();

  /* ---------------------------------------------------------------------
    A) Поточний знімок блокувань
    --------------------------------------------------------------------- */
  IF OBJECT_ID('tempdb..#curr_blockers') IS NOT NULL DROP TABLE #curr_blockers;
  CREATE TABLE #curr_blockers
  (
```

```
    blocking_session_id INT NOT NULL,
    blocked_count       INT NOT NULL,
    max_wait_ms         BIGINT NOT NULL,
    db_name             SYSNAME NULL,
    login_name          NVARCHAR(128) NULL,
    host_name           NVARCHAR(128) NULL,
    program_name        NVARCHAR(128) NULL,
    blocker_text        NVARCHAR(MAX) NULL
);

-- Збір даних: групуємо за blocking_session_id із sys.dm_os_waiting_tasks та збагачуємо деталями
WITH wt AS
(
    SELECT wt.session_id AS blocked_session_id,
           wt.blocking_session_id,
           wt.wait_duration_ms
    FROM sys.dm_os_waiting_tasks AS wt
    WHERE wt.blocking_session_id IS NOT NULL
),
agg AS
(
    SELECT blocking_session_id,
           COUNT(DISTINCT blocked_session_id) AS blocked_count,
           MAX(wait_duration_ms)              AS max_wait_ms
    FROM wt
    GROUP BY blocking_session_id
)
```

```
    INSERT INTO #curr_blockers(blocking_session_id, blocked_count,
max_wait_ms, db_name, login_name, host_name, program_name, blocker_text)
    SELECT
      a.blocking_session_id,
      a.blocked_count,
      a.max_wait_ms,
      DB_NAME(r.database_id)                    AS db_name,
      s.login_name,
      s.host_name,
      s.program_name,
      CAST(st.text AS NVARCHAR(MAX))            AS blocker_text
    FROM agg AS a
    LEFT JOIN sys.dm_exec_requests AS r
        ON r.session_id = a.blocking_session_id
    LEFT JOIN sys.dm_exec_sessions AS s
        ON s.session_id = a.blocking_session_id
    OUTER APPLY sys.dm_exec_sql_text(r.sql_handle) AS st;

    /* ---------------------------------------------------------------------
      B) Оновлення стану блокувань у службовій таблиці
      --------------------------------------------------------------------- */
    -- Вставка/оновлення актуальних блокерів
    MERGE admin.BlockMonitorState AS T
    USING #curr_blockers AS S
     ON T.blocking_session_id = S.blocking_session_id
    WHEN MATCHED THEN
      UPDATE SET
        T.last_seen     = @now,
```

```
        T.blocked_count = S.blocked_count,
        T.max_wait_ms   = S.max_wait_ms,
        T.db_name       = S.db_name,
        T.login_name    = S.login_name,
        T.host_name     = S.host_name,
        T.program_name  = S.program_name
    WHEN NOT MATCHED THEN
      INSERT (blocking_session_id, first_seen, last_seen, blocked_count,
max_wait_ms, db_name, login_name, host_name, program_name, notified)
      VALUES (S.blocking_session_id, @now, @now, S.blocked_count,
S.max_wait_ms, S.db_name, S.login_name, S.host_name, S.program_name, 0);

    -- Видалити записи, які більше не блокують
    DELETE T
    FROM admin.BlockMonitorState AS T
    WHERE NOT EXISTS (SELECT 1 FROM #curr_blockers AS S WHERE
S.blocking_session_id = T.blocking_session_id);
```

C) Надсилання *email*, якщо блокування триває > *@ThresholdMinutes* і ще не відправляли

```
    ------------------------------------------------------------------- */
    DECLARE @ThresholdTime DATETIME2(3) = DATEADD(MINUTE, -
@ThresholdMinutes, @now);
    DECLARE @body NVARCHAR(MAX);
    DECLARE @subject NVARCHAR(300);
    DECLARE @blocker INT;

    DECLARE cur_block CURSOR LOCAL FAST_FORWARD FOR
      SELECT blocking_session_id
```

```
    FROM admin.BlockMonitorState
    WHERE notified = 0
      AND first_seen <= @ThresholdTime;

  OPEN cur_block;
  FETCH NEXT FROM cur_block INTO @blocker;

  WHILE @@FETCH_STATUS = 0
  BEGIN
    -- Побудувати детальний звіт по блокуванню: усі заблоковані сесії, тривалість, текст запитів
    DECLARE @details NVARCHAR(MAX) = N'';
    SELECT @details =
    (
      SELECT
        CONCAT(
          N'Blocked session: ', wt.blocked_session_id,
          N' | Wait: ', wt.wait_duration_ms, N' ms',
          N' | DB: ', COALESCE(DB_NAME(r.database_id), N'(unknown)'),
          N' | Login: ', COALESCE(s.login_name, N'(unknown)'),
          N' | Host: ', COALESCE(s.host_name, N'(unknown)'),
          N' | Program: ', COALESCE(s.program_name, N'(unknown)'),
          N' | Query: ', COALESCE(CAST(st.text AS NVARCHAR(MAX)), N'(no current request)')
        ) + CHAR(13) + CHAR(10)
      FROM sys.dm_os_waiting_tasks AS wt
      LEFT JOIN sys.dm_exec_requests AS r
        ON r.session_id = wt.session_id
```

```
    LEFT JOIN sys.dm_exec_sessions AS s
      ON s.session_id = wt.session_id
    OUTER APPLY sys.dm_exec_sql_text(r.sql_handle) AS st
    WHERE wt.blocking_session_id = @blocker
    FOR XML PATH(''), TYPE
).value('.', 'nvarchar(max)');

DECLARE @hdr NVARCHAR(MAX) = N'Blocking alert: problem persists over ' + CAST(@ThresholdMinutes AS NVARCHAR(10)) + N' minutes.' + CHAR(13) + CHAR(10)
                + N'Blocking session: ' + CAST(@blocker AS NVARCHAR(20)) + CHAR(13) + CHAR(10);

SELECT @body = @hdr + N'Blocked sessions:' + CHAR(13) + CHAR(10) + ISNULL(@details, N'(none)');

SELECT @subject = N'[SQL] Blocking persists > ' + CAST(@ThresholdMinutes AS NVARCHAR(10)) + N' min | blocker SPID ' + CAST(@blocker AS NVARCHAR(20));

EXEC msdb.dbo.sp_send_dbmail
    @profile_name = @MailProfile,
    @recipients   = @MailTo,
    @subject      = @subject,
    @body         = @body;

UPDATE admin.BlockMonitorState
SET notified = 1
```

```
    WHERE blocking_session_id = @blocker;

    IF @Verbose = 1 PRINT 'Sent blocking alert for SPID ' + CAST(@blocker
AS NVARCHAR(20));

    FETCH NEXT FROM cur_block INTO @blocker;
  END

  CLOSE cur_block;
  DEALLOCATE cur_block;

  /* ---------------------------------------------------------------------
    D) Обробка дедлоків: читаємо з Extended Events (ring_buffer) та
надсилаємо нотифікацію
    --------------------------------------------------------------------- */
  IF @IncludeDeadlocks = 1
  BEGIN
    DECLARE @last_checked DATETIME2(3);
    SELECT TOP (1) @last_checked = last_checked_time
    FROM admin.DeadlockAlertCursor
    ORDER BY id DESC;
    -- Отримати XML із DeadlockCapture
    DECLARE @rb XML;

    SELECT @rb = CAST(xet.target_data AS XML)
    FROM sys.dm_xe_session_targets AS xet
    JOIN sys.dm_xe_sessions AS xes
      ON xet.event_session_address = xes.address
```

```
    WHERE xes.name = N'DeadlockCapture'
      AND xet.target_name = N'ring_buffer';

    IF @rb IS NOT NULL
    BEGIN
        -- Витягнути події дедлоків
        ;WITH events AS
        (
            SELECT
                x.evt.value('@timestamp', 'datetime2(3)') AS ts,
                x.evt.query('.')                          AS deadlock_event_xml
            FROM    @rb.nodes('//event[@name="xml_deadlock_report"]')    AS
x(evt)
        )
        SELECT *
        INTO #xe_deadlocks
        FROM events;

        -- Фільтр нових (після last_checked)
        IF @last_checked IS NOT NULL
        BEGIN
            DELETE FROM #xe_deadlocks WHERE ts <= @last_checked;
        END
        IF EXISTS (SELECT 1 FROM #xe_deadlocks)
        BEGIN
            DECLARE @cnt INT = (SELECT COUNT(*) FROM #xe_deadlocks);
            DECLARE @latest_XML = (SELECT TOP (1) deadlock_event_xml
FROM #xe_deadlocks ORDER BY ts DESC);
```

```
            DECLARE @subject2 NVARCHAR(300) = N'[SQL] Deadlock detected (' + CAST(@cnt AS NVARCHAR(20)) + N' new)';
            DECLARE @body2 NVARCHAR(MAX) =
                N'Detected ' + CAST(@cnt AS NVARCHAR(20)) + N' deadlock event(s) since last check.' + CHAR(13) + CHAR(10) +
                N'Latest event time: ' + CONVERT(NVARCHAR(30), (SELECT MAX(ts) FROM #xe_deadlocks), 126) + CHAR(13) + CHAR(10) +
                N'Deadlock XML (latest):' + CHAR(13) + CHAR(10) +
                CAST(@latest AS NVARCHAR(MAX));

            EXEC msdb.dbo.sp_send_dbmail
                @profile_name = @MailProfile,
                @recipients   = @MailTo,
                @subject      = @subject2,
                @body         = @body2;

            IF @Verbose = 1 PRINT 'Sent deadlock alert, count=' + CAST(@cnt AS NVARCHAR(20));
        END
        -- Оновити курсор часу
        UPDATE admin.DeadlockAlertCursor
        SET last_checked_time = @now
        WHERE id = (SELECT TOP (1) id FROM admin.DeadlockAlertCursor ORDER BY id DESC);
    END
END

SET NOCOUNT OFF;
```

```
END
GO
```

Синхронізація таблиці-призначення з дельта-таблиці

```
MERGE INTO analytics_db.sales.Sold_items AS tgt
USING analytics_db.stage.Sold_items_delta AS src
ON tgt.ID = src.ID
WHEN MATCHED AND NVL2(src.hash, src.hash, 'x') <> NVL2(tgt.hash, tgt.hash, 'x')
THEN UPDATE SET
  tgt.sale_date   = src.sale_date,
  tgt.qty         = src.qty,
  tgt.amount      = src.amount,
  tgt.hash        = src.hash,
  tgt.modified_at = CURRENT_TIMESTAMP()
WHEN NOT MATCHED
THEN INSERT (ID, sale_date, qty, amount, hash, modified_at)
VALUES (src.ID, src.sale_date, src.qty, src.amount, src.hash, CURRENT_TIMESTAMP());
```