

Koen Decorte



# **Modern SQL Mastery**

## Advanced Techniques for DB2 for i Performance and Productivity

[www.cdinvest.eu](http://www.cdinvest.eu)

International IBM i ISV and  
IBM business partner .

located in Antwerp, Belgium

Expertise in RPG, SQL,  
Python, HTML, Ai,  
NodeJS, Linux...

Working with IBM i  
and its predecessors  
for more than 40  
year

IBM Champion  
since 2018 and  
CEAC member

Applications : CDErp,  
CDReport CD-Account  
accountancy, CDVts  
and MES.

What others talk about,  
we do.

Who are we ?



# Agenda

- Introduction to Modern SQL on IBM i
- Common Table Expressions (CTEs)
- Advanced Joins and Subqueries
- Window Functions
- JSON and XML Support
- Advanced Indexing Strategies
- Performance Optimization Techniques
- Replacing RPG/Record-Level Access with SQL
- Omnifind
- Data analysis techniques

# Session goals

- Master advanced SQL techniques for DB2 for i
- Learn to write efficient, maintainable SQL code
- Understand performance optimization strategies
- Apply modern SQL patterns to business problems
- Leverage the latest DB2 for i features



# Introduction

Modern SQL on IBM i

# Modern SQL on IBM i

- SQL's evolution on the IBM i platform
- SQL vs. traditional record-level access
- The modern SQL mindset: set-based processing
- SQL as a first-class language on IBM i

# IBM i SQL Capabilities

- Full SQL standard compliance
- Integration with IBM i security model
- Seamless access to traditional physical/logical files
- SQL stored procedures and functions
- Database-centric application design

# Why Modern SQL Matters

- Improved developer productivity
- Better performance with optimizer improvements
- Standardized access method across platforms
- Simplification of complex business logic
- Easier to maintain and understand





# Common Table Expressions

# Common Table Expressions (CTEs) - Overview

- Definition and basic syntax
- Temporary result sets within a SELECT, INSERT, UPDATE, or DELETE
- Benefits over derived tables and temporary tables
- Readability and maintainability advantages

# Basic CTE Example

```
-- Find departments with more than 3 employees
WITH employee_counts AS (
    SELECT
        DEPTNO,
        COUNT(*) AS EMP_COUNT
    FROM EMPLOYEE
    GROUP BY DEPTNO
)
SELECT
    DEPTNO
    EMP_COUNT
FROM employee_counts
WHERE EMP_COUNT > 3
ORDER BY EMP_COUNT DESC;
```

# Non-Recursive CTEs

- Multiple CTEs in a single query
- Joining CTEs with other tables
- Using CTEs in views
- Performance considerations

# Multiple CTEs Example

```
WITH
dept_count AS (
  SELECT DEPTNO, COUNT(*) AS EMP_COUNT
  FROM EMPLOYEE
  GROUP BY DEPTNO
),
avg_salary AS (
  SELECT DEPTNO, AVG(SALARY) AS AVG_SAL
  FROM EMPLOYEE
  GROUP BY DEPTNO
),
salary_variance AS (
  SELECT
    DEPTNO,
    STDDEV(SALARY) AS SALARY_STDDEV
  FROM EMPLOYEE
  GROUP BY DEPTNO
)
SELECT
  d.DEPTNAME,
  dc.EMP_COUNT,
  s.AVG_SAL,
  sv.SALARY_STDDEV,
  (s.AVG_SAL / dc.EMP_COUNT) AS AVG_SALARY_PER_EMPLOYEE
FROM DEPARTMENT d
JOIN dept_count dc ON d.DEPTNO = dc.DEPTNO
JOIN avg_salary s ON d.DEPTNO = s.DEPTNO
JOIN salary_variance sv ON d.DEPTNO = sv.DEPTNO
ORDER BY dc.EMP_COUNT DESC;
```

# Recursive CTEs

- Recursive query definition
- UNION ALL with base and recursive parts
- Solving hierarchical data problems
- Avoiding infinite loops

# Recursive CTE Example - Org Chart

See demo cte.sql

- Let's look at a classic use case: an organizational chart
- Break down the query: The base case starts with the CEO (employees with no manager)
- The recursive part finds all employees who report to managers already in our result set
- We add a LEVEL column to track the hierarchy depth
- This query would be nearly impossible without recursive CTEs

# CTE Use Cases

- Hierarchical data traversal
- Report generation with complex aggregations
- Simplifying complex joins
- Breaking down complex logic into manageable pieces
- Multi-step data transformations



# CTEs vs. Temporary Tables

- Performance considerations
- Scope and persistence
- When to use each approach
- Materialized query tables as an alternative



# Advanced joins and subqueries

# Advanced Joins - Overview

- Review of basic join types
- Multiple-table joins
- Self-joins for hierarchical data
- Non-equi joins
- LATERAL joins

# Complex Join Example

- This example joins four tables to create a comprehensive order report
- Notice the join order: we start with customer and work our way through related tables
- The WHERE clause adds a time filter for recent orders
- This type of query replaces what might have been multiple record-level access operations in traditional RPG

# Self-Join Example

- A self-join is when a table joins to itself—useful for hierarchical relationships within a single table
- In this example, we're joining the employee table to itself to connect employees with their managers
- The LEFT JOIN ensures we don't lose employees who don't have a manager
- This is a simple but powerful technique for organizational data

# LATERAL Join

- LATERAL joins are a powerful feature that lets you reference columns from tables earlier in the FROM clause
- In this example, we're finding the top 3 highest-paid employees in each department
- Without LATERAL, this would require much more complex code
- Think of LATERAL as enabling a 'for each' type of processing within your SQL

# Advanced Subqueries

- Correlated subqueries
- Scalar subqueries
- EXISTS and NOT EXISTS
- ANY, SOME, and ALL operators
- Performance considerations

# Correlated Subquery Example

- A correlated subquery references columns from the outer query
- In this example, we're calculating each employee's salary difference from their department average
- The subquery executes once for each row in the outer query
- This can be powerful but may have performance implications for large datasets



# EXISTS vs. IN vs. JOIN

- Correlated subqueries
- Scalar subqueries
- EXISTS and NOT EXISTS
- ANY, SOME, and ALL operators
- Performance considerations

# Exists approach

## **How it works:**

- The database checks each customer and sees if at least one matching order exists
- It stops checking orders as soon as it finds the first match (short-circuit evaluation)
- Only needs to know IF a match exists, not HOW MANY or WHICH ONES

## **Best when:**

- You only need to verify existence
- The subquery table (ORDERS) is large
- Few rows in the outer table (CUSTOMER) will match the condition
- You have indexes on the join columns

# IN approach

## **How it works:**

- Creates a list of all matching customer numbers from the orders table
- Then checks if each customer is in that list
- Usually materializes the complete list of matching values

## **Best when:**

- The subquery returns a small, distinct set of values
- You have indexes on the columns being compared
- The subquery table (ORDERS) is much smaller than the outer table

# JOIN approach

## **How it works:**

- Directly combines the two tables based on the matching condition
- Must use DISTINCT to avoid duplicates if multiple orders match a customer
- Retrieves all matching rows from both tables

## **Best when:**

- You need additional columns from both tables
- You have good indexes on join columns
- The database optimizer is sophisticated
- You need to perform additional operations on the joined data

# EXISTS vs. IN vs. JOIN

## Which is best?

The answer depends on your specific scenario:

- **EXISTS** is often most efficient when checking for existence in a large table, especially when you expect many matches and can benefit from short-circuit evaluation.
- **IN** works well when the inner query returns a small set of distinct values and those values are indexed.
- **JOIN** is preferable when you need data from both tables or when further filtering/operations will be performed on the joined result.

# Window functions

# Window functions overview

- Definition and basic syntax
- Differences from GROUP BY
- OVER clause and window frame specification
- Window function types

# Types of Window Functions

- Aggregate functions: SUM, AVG, MIN, MAX, COUNT
- Ranking functions: RANK, DENSE\_RANK, ROW\_NUMBER
- Distribution functions: PERCENT\_RANK, CUME\_DIST
- Navigation functions: LEAD, LAG, FIRST\_VALUE, LAST\_VALUE



# Basic Window Function Example

- The query starts like a standard SELECT, retrieving employee information. But notice what happens when we get to the AVG function. Instead of a GROUP BY, we use OVER (PARTITION BY DEPTNO).
- This OVER clause is the defining feature of window functions. It creates a 'window' of rows—in this case, all employees in the same department—and calculates the average salary across that window.
- The magic is that unlike GROUP BY, each employee row remains in the result set. We're not collapsing rows; we're adding calculated values to each row based on its relationship to other rows.
- This allows us to immediately calculate the difference between each employee's salary and their department average in a single expression: SALARY - AVG(SALARY) OVER (PARTITION BY DEPTNO).

# Ranking Functions Example

- Ranking functions are perfect for top-N reporting and analysis
- In this example, we're calculating both department ranks and overall company ranks
- The PARTITION BY clause resets the ranking for each department
- This would be extremely complex without window functions

# ROW\_NUMBER vs. RANK vs. DENSE\_RANK

- ROW\_NUMBER gives a unique number to each row, even if values are tied
- RANK gives the same rank to tied values, then skips the next rank
- DENSE\_RANK also gives the same rank to tied values but doesn't skip ranks

# Navigation Functions

- Navigation functions let you access data from other rows without joins
- LAG retrieves data from a previous row, while LEAD looks at the next row
- In our example, we're calculating the change in order amount from a customer's previous order

# Window Frame Specification

```
SELECT  
  EMPNO,  
  LASTNAME,  
  HIREDATE,  
  SALARY,  
  AVG(SALARY) OVER (  
    ORDER BY HIREDATE  
    ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING  
  ) AS ROLLING_AVG_SALARY  
FROM EMPLOYEE  
ORDER BY HIREDATE;
```

# Window Function Use Cases

- Running totals and moving averages
- Percentile calculations
- Gap analysis (finding missing values)
- Year-over-year comparisons
- Top-N per group queries
- Handling time-series data

# Window Functions Performance Tips

- Avoid unnecessary PARTITION BY clauses
- Be cautious with large window frames
- Consider materialized query tables for complex windows
- Monitor sort operations in the query plan
- Window functions vs. self-joins for performance

# JSON and XML support



# JSON Support - Overview

- DB2 for i JSON capabilities
- JSON data types
- JSON\_TABLE for result set generation
- Built-in JSON functions

# JSON Basic Functions

- `JSON_OBJECT` builds a JSON object from your relational data
- You can nest objects and arrays to create complex structures
- This is particularly useful for API responses or document storage

# Extracting JSON Data with JSON\_TABLE

- JSON\_TABLE is like a virtual table function that extracts data from JSON
- It converts JSON into relational rows and columns that you can join with other tables
- The path expressions use a syntax similar to JavaScript dot notation

# JSON\_VALUE and JSON\_QUERY

- JSON\_VALUE extracts a scalar value from JSON and converts it to a SQL type
- JSON\_QUERY extracts a JSON object or array without converting it
- Use JSON\_VALUE when you need a single value for comparisons or output
- Use JSON\_QUERY when you need to preserve the JSON structure

# JSON Performance Considerations

- Indexes on JSON\_VALUE expressions
- Materializing JSON into relational tables
- VALIDATE vs. LAX parsing modes
- Memory usage with large JSON objects
- Storing JSON vs. generating JSON on-the-fly

# JSON Use Cases

- API integration
- Configuration storage
- Document storage within the database
- Flexible schema requirements
- Modern web application development

# XML Support in DB2 for i

- XMLTABLE function
- XMLPARSE and XMLSERIALIZE
- XPath expressions
- XML vs. JSON considerations
- Legacy system integration



# Advanced indexing strategies



# Advanced Indexing - Overview

- Traditional index types review
- Expression-based indexes
- Covering indexes
- Clustered indexes
- Index optimization strategies

# Expression-Based Indexes

-- Create an index on an expression

```
CREATE INDEX ix_employee_upper_lastname  
ON EMPLOYEE (UPPER(LASTNAME));
```

-- Query that can use the expression index

```
SELECT * FROM EMPLOYEE  
WHERE UPPER(LASTNAME) = 'SMITH';
```

# When to Use Expression-Based Indexes

- Case-insensitive searches
- Date/time component extraction
- Function-based WHERE clauses
- JSON property extraction (JSON\_VALUE)
- Computed columns used in predicates

# Covering Indexes

-- Creating a covering index for a common query

```
CREATE INDEX ix_orders_cust_date  
ON ORDERS (CUSTNO, ORDERDATE)  
INCLUDE (ORDERNO, TOTAL_AMOUNT);
```

-- Query that can use the covering index

```
SELECT ORDERNO, ORDERDATE, TOTAL_AMOUNT  
FROM ORDERS  
WHERE CUSTNO = 1001  
AND ORDERDATE >= '2023-01-01';
```

# Index Advisor

- Using DB2 for i Index Advisor
- Interpreting Index Advisor recommendations
- Visual Explain for index analysis
- When to accept or reject recommendations
- Testing index effectiveness

# Indexing Best Practices

- Balance index count with maintenance overhead
- Consider column cardinality and selectivity
- Index column order matters
- Avoid over-indexing small tables
- Monitor for unused indexes



# Performance Optimization

# Advanced Indexing - Overview

- SQE vs. CQE optimization
- Query rewrites for better performance
- Statistics and histograms
- Join optimization
- Batch vs. interactive performance



# Basic Performance Tuning Checklist

- Review access plans with Visual Explain
- Verify proper indexing
- Monitor temporary storage usage
- Evaluate file sizes and fragmentation

# Advanced Query Optimization

- Join order optimization
- Temporary result handling
- Partitioning and parallelism
- Materialized query tables (MQTs)

# Query Rewrites for Performance

- Join order optimization
- Temporary result handling
- Partitioning and parallelism
- Materialized query tables (MQTs)

# Query Rewrites for Performance

- Original query:

```
SELECT
  c.CUSTNAME,
  SUM(o.TOTAL_AMOUNT) AS TOTAL_ORDERS
FROM CUSTOMER c
LEFT JOIN ORDERS o ON c.CUSTNO = o.CUSTNO
WHERE (o.ORDERDATE >= '2023-01-01' OR o.ORDERDATE IS NULL)
GROUP BY c.CUSTNAME
ORDER BY TOTAL_ORDERS DESC;
```

# Query Rewrites for Performance

```
SELECT
  c.CUSTNAME,
  COALESCE(o.TOTAL_ORDERS, 0) AS TOTAL_ORDERS
FROM CUSTOMER c
LEFT JOIN (
  SELECT
    CUSTNO,
    SUM(TOTAL_AMOUNT) AS TOTAL_ORDERS
  FROM ORDERS
  WHERE ORDERDATE >= '2023-01-01'
  GROUP BY CUSTNO
) o ON c.CUSTNO = o.CUSTNO
ORDER BY TOTAL_ORDERS DESC;
```

# Using OLAP Features for Performance

- GROUPING SETS
- ROLLUP and CUBE
- Performance comparison against multiple queries
- Memory requirements

# GROUPING SETS Example

- GROUPING SETS
- ROLLUP and CUBE
- Performance comparison against multiple queries
- Memory requirements

# Materialized Query Tables (MQTs)

- Definition and syntax
- Performance benefits
- Maintenance strategies
- When to use MQTs
- Real-world performance improvements



# Materialized Query Tables (MQTs)

- DATA INITIALLY IMMEDIATE means it's populated when created
- REFRESH DEFERRED means it won't automatically update when underlying tables change
- MAINTAINED BY USER means you're responsible for refreshing it
- These settings give you control over the performance impact of MQT maintenance

# MQT Maintenance

- MQTs need to be refreshed to remain current with the underlying data
- The REFRESH TABLE statement updates the MQT data
- For regular refreshes, schedule a job that runs during off-peak hours



# Replacing RPG with SQL

# Replacing RPG with SQL - Overview

- Benefits of SQL over record-level access
- Migration strategies
- Performance considerations
- Hybrid approaches

# Record-Level Access vs. SQL Comparison

Traditional RPG with CHAIN:

```
CHAIN (CUSTNO) CUSTFILE;  
IF %FOUND;  
    CHAIN (CUSTNO) ORDERFILE;  
    IF %FOUND;  
        // Process order  
    ENDIF;  
ENDIF;
```

# Record-Level Access vs. SQL Comparison

SQL equivalent:

```
SELECT c.*, o.*  
FROM CUSTOMER c  
LEFT JOIN ORDERS o ON c.CUSTNO = o.CUSTNO  
WHERE c.CUSTNO = :custno;
```

# Set-Based Operations with SQL

- RPG processing:

```
DOW NOT %EOF(CUSTFILE);  
  READ CUSTFILE;  
  IF NOT %EOF(CUSTFILE);  
    // Process each customer  
    // Update status  
    UPDATE CUSTSTATUS;  
  ENDIF;  
ENDDO;
```

# Set-Based Operations with SQL

- SQL equivalent:

UPDATE CUSTOMER

SET STATUS = 'ACTIVE'

WHERE LASTORDERDATE >= CURRENT DATE - 90  
DAYS;



# Complex Business Logic in SQL

```
UPDATE CUSTOMER c
SET c.DISCOUNT_PCT =
CASE
  WHEN (SELECT SUM(TOTAL_AMOUNT)
        FROM ORDERS
        WHERE CUSTNO = c.CUSTNO
        AND ORDERDATE >= CURRENT DATE - 1 YEAR) >= 10000 THEN 10
  WHEN (SELECT SUM(TOTAL_AMOUNT)
        FROM ORDERS
        WHERE CUSTNO = c.CUSTNO
        AND ORDERDATE >= CURRENT DATE - 1 YEAR) >= 5000 THEN 5
  ELSE 0
END
WHERE c.STATUS = 'ACTIVE';
```

# Moving Business Logic to the Database

- Stored procedures vs. application logic
- Database functions for complex calculations
- Triggers for data integrity
- Views for data abstraction
- Security implications

# Creating a Stored Procedure

- Stored procedures encapsulate business logic on the server
- This example calculates customer metrics like average order amount and total orders
- The LANGUAGE SQL directive indicates it's written in SQL rather than RPG or another language
- The caller receives calculated results without having to know the underlying tables or logic

# Creating a User-Defined Function

- User-defined functions extend SQL's capabilities with custom logic
- This example determines a customer status based on complex business rules
- The DETERMINISTIC keyword indicates that the function always returns the same result for the same input
- Functions can be used in SELECT lists, WHERE clauses, and anywhere an expression is allowed

# Creating a User-Defined Function

- User-defined functions extend SQL's capabilities with custom logic
- This example determines a customer status based on complex business rules
- The DETERMINISTIC keyword indicates that the function always returns the same result for the same input
- Functions can be used in SELECT lists, WHERE clauses, and anywhere an expression is allowed



# Anti-Patterns and Best practices

# SQL Code Patterns Library

- Reusable SQL patterns for common tasks
- Templates for reporting queries
- Data transformation patterns
- Performance optimization patterns
- Modernization patterns

# Anti-Patterns to Avoid

- `SELECT *` in production code
- Unnecessary subqueries
- Cursor-based processing when set-based is possible
- Excessive use of temporary tables
- String concatenation in SQL
- Lack of parameterization



# Omnifind

## IBM OmniFind Text Search Server for DB2 for i

- New IBM i product offering: 5733-OMF
  - **No-charge offering**
- Delivers common DB2 Family text search technology
  - **Advanced, linguistic high-speed searches**
  - **Support enabled for any character-based column**
  - **Search technology also supports Rich Text document formats**
    - Example: LOB columns containing PDF or Microsoft® Word documents
    - IFS documents can be indexed with extra programming
  - **Includes support for 26 different languages**

# OmniFind Search Capabilities

- Example Searches:

```
SELECT author, story FROM books  
WHERE CONTAINS (story, 'mice chasing cats') = 1  
AND pubDate >= '1/1/2000'
```

```
SELECT feedSrc, feedDate,  
       SCORE (feedDoc, 'insurance settlement')  
FROM newsfeeds  
WHERE CONTAINS(feedDoc, 'insurance settlement') = 1  
ORDER BY 3 DESC
```

# Database Requirements

- Table must have primary key, unique key constraint or ROWID column
  - Physical files are supported
  - Unique-keyed physical file requires usage of ADDPFCST to register key as constraint
- Supported column data types:
  - BINARY, VARBINARY
  - BLOB
  - CHAR, VARCHAR
  - CLOB
  - DBCLOB
  - GRAPHIC, VARGRAPHIC

# Database

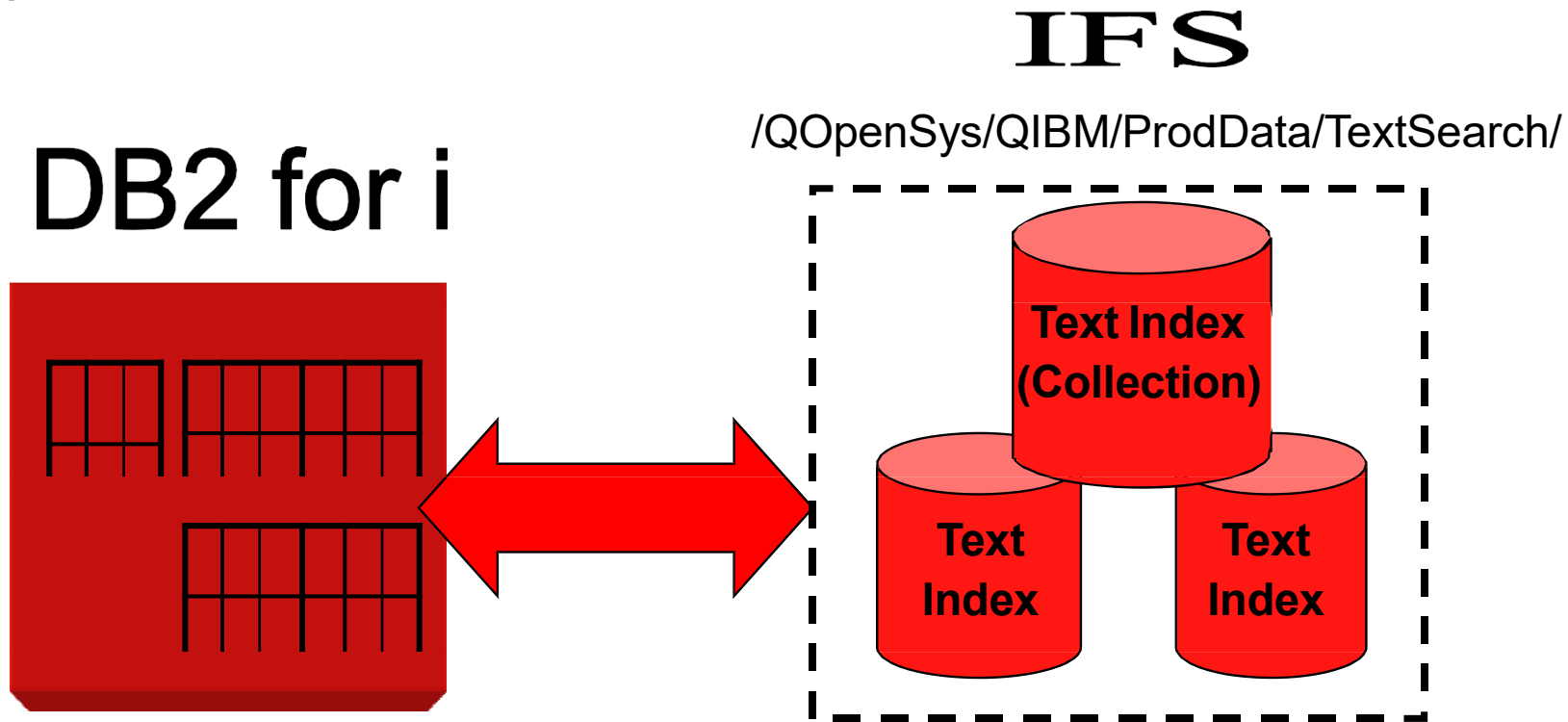
- Supported document format types:
  - Plain text
  - XML
  - HTML
  - Adobe PDF
  - Rich Text Format (RTF)
  - JustSystems Ichitaro
  - Microsoft Excel
  - Microsoft PowerPoint
  - Microsoft Word
  - Lotus® 123
  - Lotus Freelance®
  - Lotus WordPro
  - OpenOffice 1.1 & 2.0
  - OpenOffice Calc
  - Quattro Pro

# Text Index vs

- Text indexes stored outside of DB2
- Text indexes have delayed maintenance
  - Changes logged to staging table
  - Index maintenance is scheduled
- Text indexes not protected by IBM i SMAPP
- Text indexes utilize different indexing methods
  - Table with VARCHAR(32000) column and 175,000 rows
    - DB2 Index object size: 1.7 GB
    - Text Index object size: 0.1 GB

# Text Search Server Configuration

- OmniFind directory structure created during product install



# Configuration & Administration

- System stored procedures in SYSPROC
  - Starting and ending the text search server
    - SYSTS\_START
    - SYSTS\_STOP
  - Creating, maintaining, and dropping a text search index
    - SYSTS\_CREATE
    - SYSTS\_UPDATE
    - SYSTS\_DROP



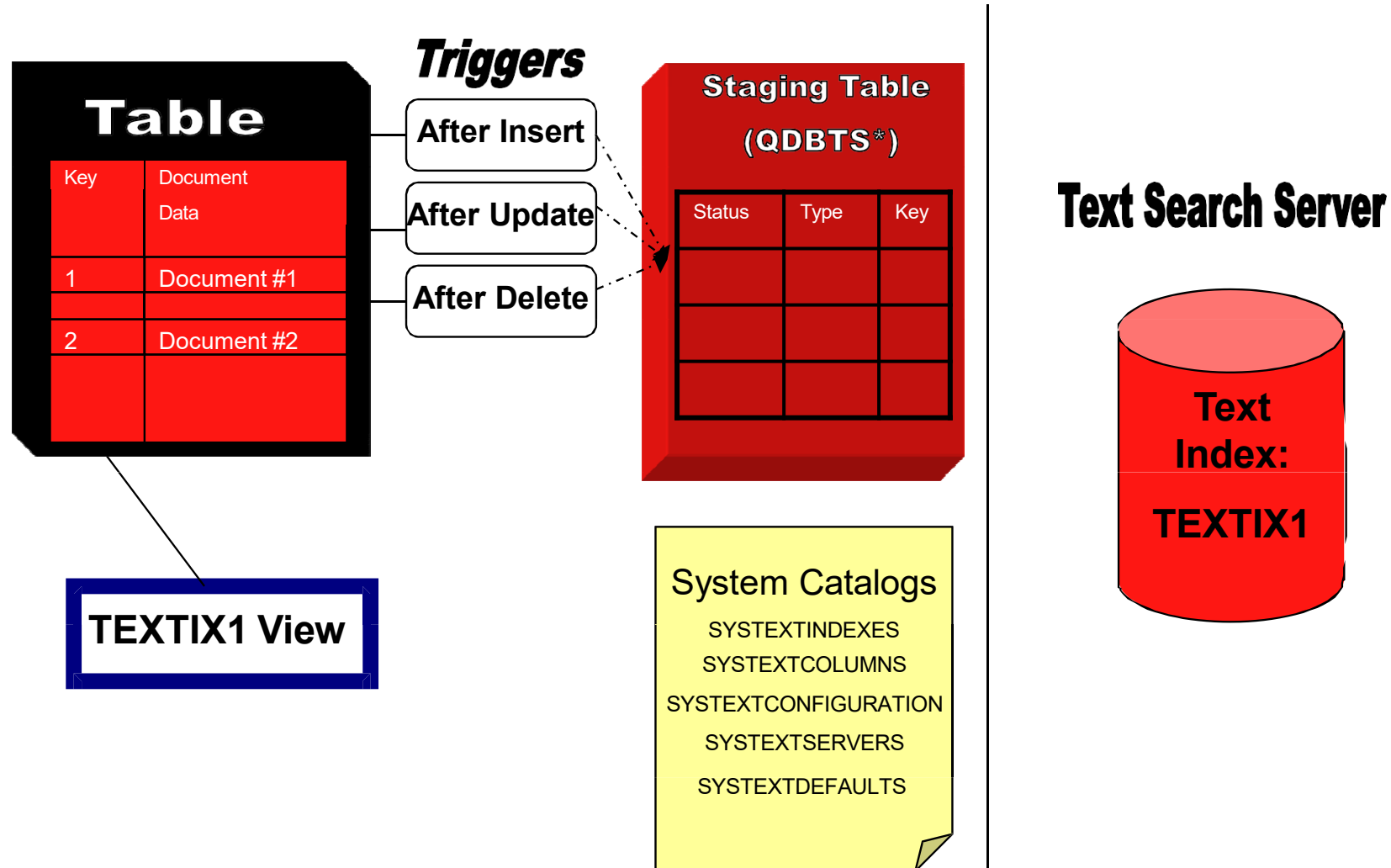
# Start and Stop Text Search Processing

- `systs_start(servernum)`
  - Starts the specified text search server
    - Default is all local servers
    - This function must be called before using other text search function.
- `systs_stop(servernum)`
  - Stops the specified text search server
    - Default is all local servers
    - Changes are still logged in the staging table, even when the server is stopped.

# Text Index Creation

- A text index is created by a call to SYSTS\_CREATE
  - Object created in IFS text server directory
  - Text index initially has no data
- When a Text Index is created, the following DB2 objects are created:
  - A view is created with the same name as the text search index
  - A staging table is created in the QSYS2 schema
  - After Insert, After Update, and After Delete triggers are added to the base table
  - The system catalogs are updated with information about the new index

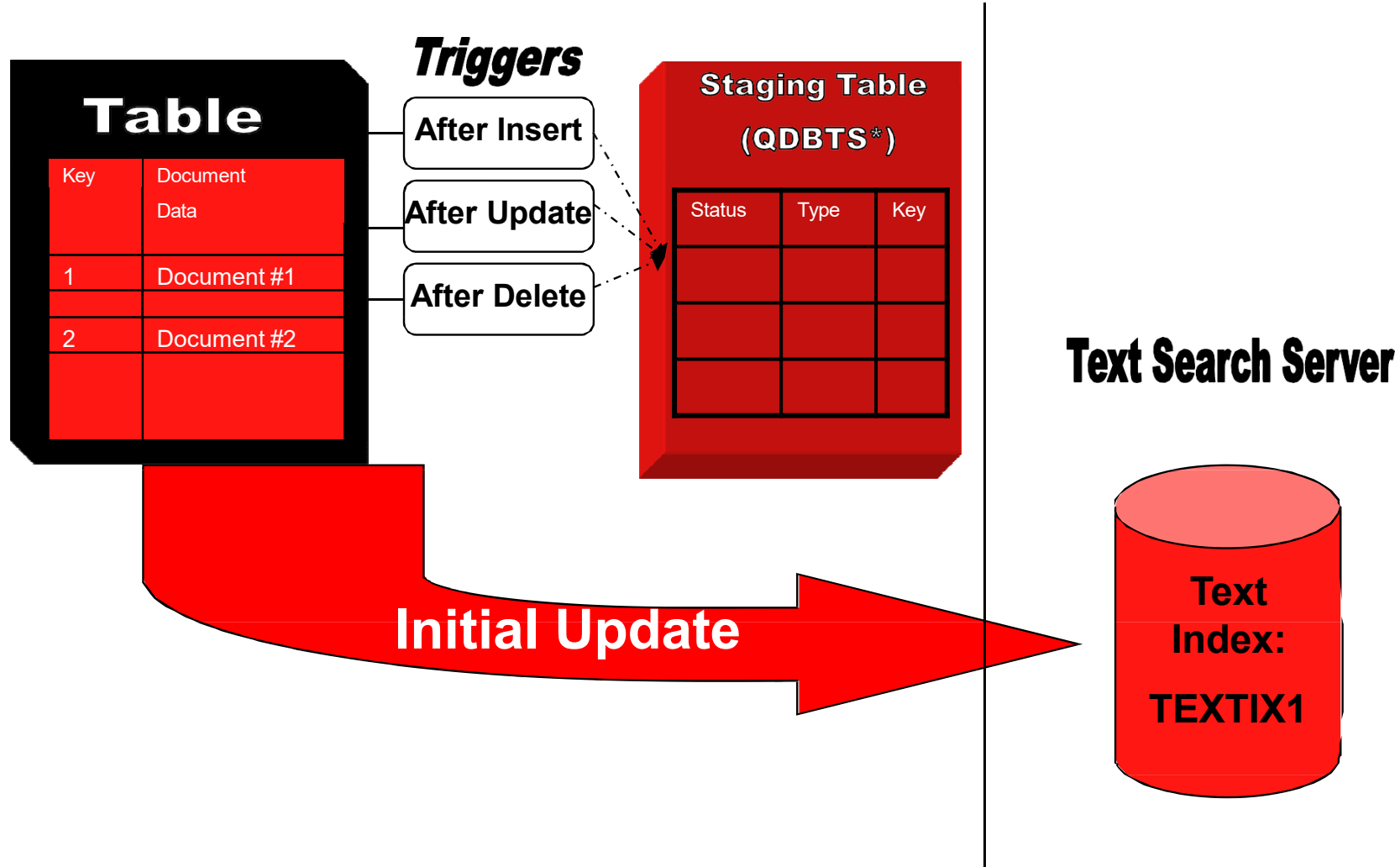
## Objects Created by SYSTS\_CREATE for a Text Index – TEXTIX1



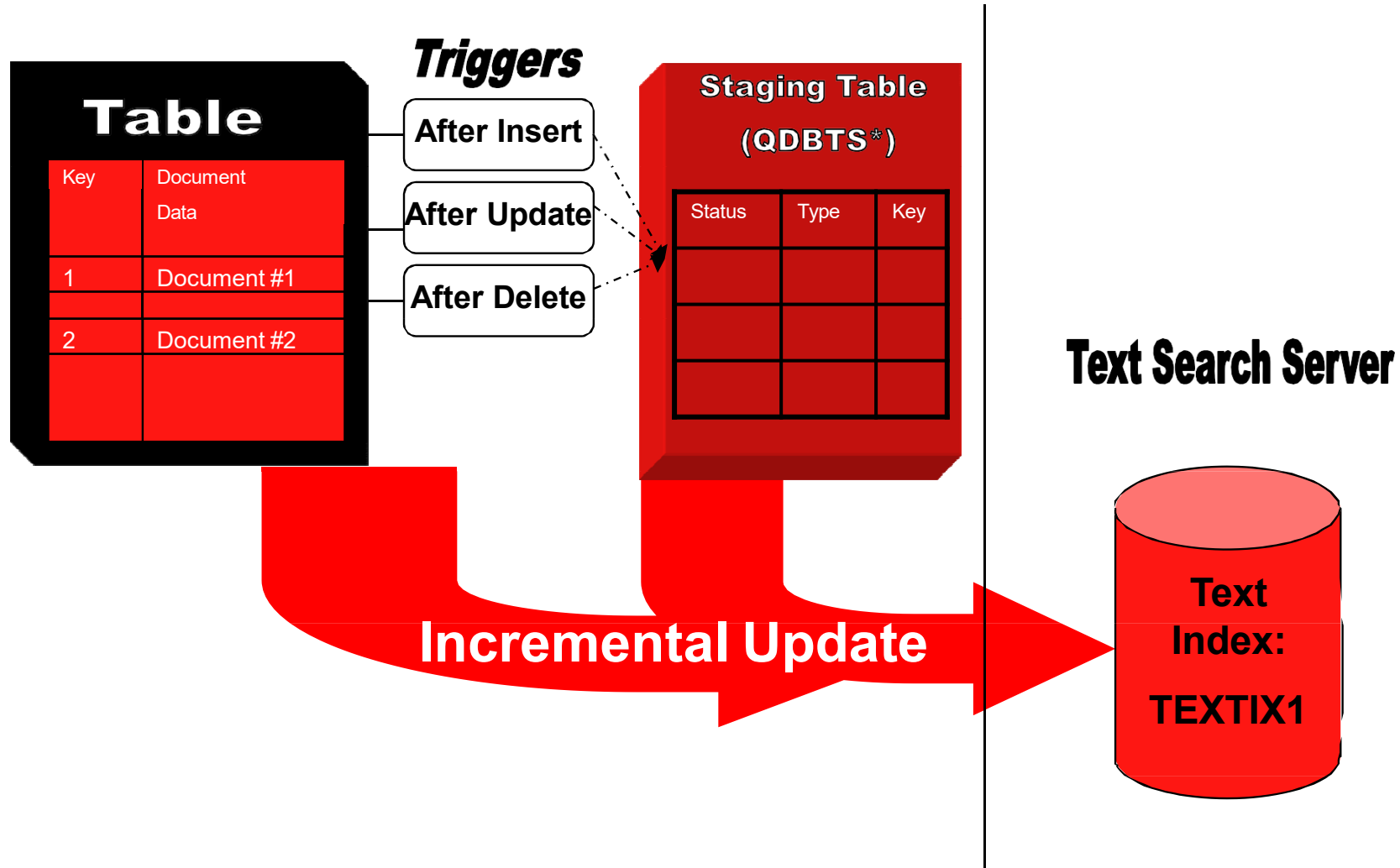
# Text Index Maintenance

- Index maintenance controlled with SYSTS\_UPDATE procedure
  - Initial update processes all text strings or text documents
  - Future index updates are incremental
    - Changes logged in the staging table are processed
    - IBM i Job Scheduler entries created to perform incremental updates
  - Minimum update threshold can be specified at creation time to only perform index updates when threshold exceeded

## SYSTS\_UPDATE Initial Update for a Text Index – TEXTIX1



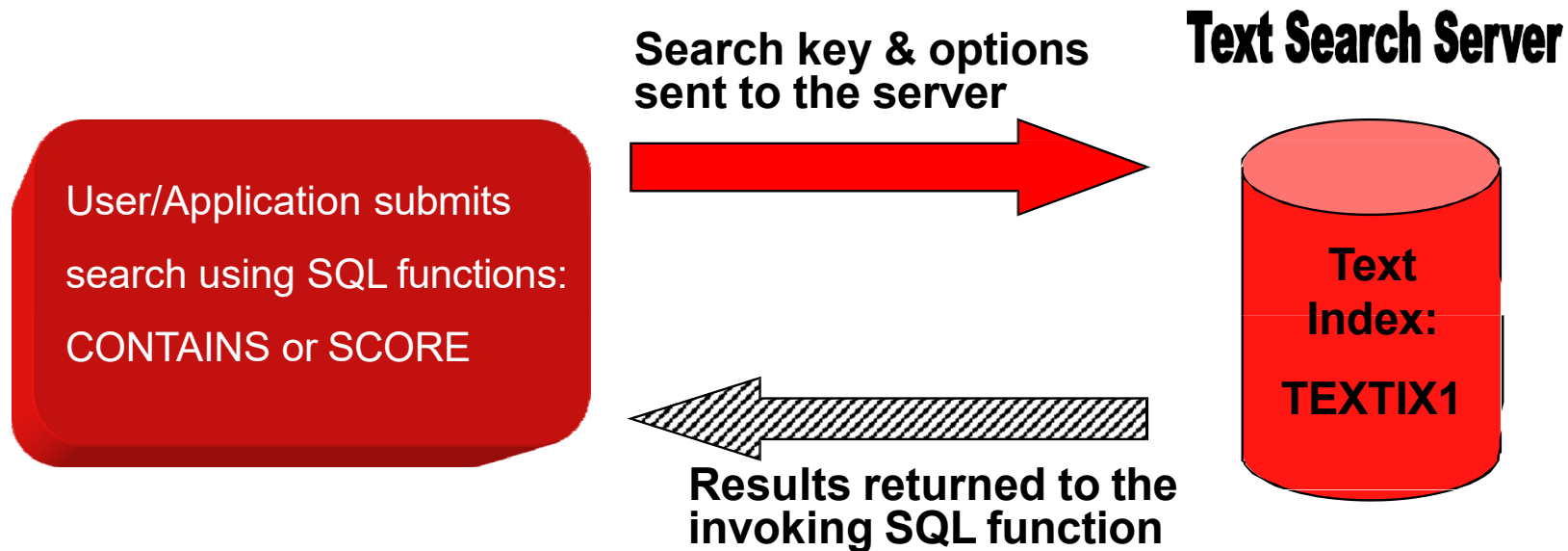
## SYSTS\_UPDATE Incremental Update for a Text Index – TEXTIX1



# Text Search Index Example

- CALL SYSPROC.SYSTS\_CREATE(  
    'myschema',  
    'resumes\_indx',  
    'myschema.resumes(applicant\_resume)',  
    'FORMAT INSO  
    UPDATE FREQUENCY D(\*) H(0) M(0)')
- FORMAT values: TEXT,HTML,XML,INSO
- UPDATE FREQUENCY controls index maintenance

# Search Processing



**\*\*\*NOTE: CONTAINS and SCORE functions only supported by the SQL Query Engine (SQE)**



# CONTAINS function

- **CONTAINS(*column-name*, *search-argument*, *options*)**
  - Column-name: column over which the text search index is built
  - Search-argument: text being searched for
  - Options: optional parameter, can be used to modify the query language or activate synonym matching
- **Function Output (integer value):**
  - 1 - match was found
  - 0 - no match found

# CONTAINS example

- Find matches on exact string 'New product interest' in the COMMENT column, use a host variable to pass search string.

```
char search_arg[100];
```

```
...
```

```
EXEC SQL DECLARE C1 CURSOR FOR  
    SELECT custkey FROM customers WHERE  
    CONTAINS(comment, :search_arg) = 1  
    ORDER BY CUSTKEY;
```

```
EXEC SQL SET :search_arg = "New product interest";
```

```
EXEC SQL OPEN C1;
```

# SCORE Function

- **SCORE(column-name, search-argument, options)**
  - Column-name: column over which the text search index is built
  - Search-argument: text being searched for
  - Options: optional parameter, can be used to modify the query language or activate synonym matching
- **Function Output:**
  - Value between 0 and 1, up to 3 decimal points
  - Higher value indicates a better match on the specified search

# SCORE Example

- Find those thesis reports that discuss programming from a performance or parallel perspective along with the normalized score rating

```
SELECT projID, projAuthor,  
       INTEGER(SCORE(thesis,  
                    'programming AND (parallel OR performance')*100)  
              ) AS relevance  
FROM projects  
WHERE  
CONTAINS(thesis,  
         'programming AND (parallel OR performance')=1  
ORDER BY relevance DESC
```

# Search argument options

- Simple Search
  - Enter one or more query terms, default operator is AND
  - Other logical operators: OR, NOT
- Use the minus sign (-) to exclude terms
  - To show all documents with terms “SQL performance”, but not “Oracle”, search argument is:  
SQL performance – Oracle
- Surround exact phrases in double quotes
  - To find hits for the product name DB2 Web Query, search argument is: “DB2 Web Query”

# Search argument options

- Wildcard character (\*) helps find documents when the exact spelling is not known, or many variations are desired.
  - 'Sh\* Gree\*'Would return: Shonn Greene, Shawn Green, etc
  - 'John \* Kennedy'
    - Would return: John F Kennedy or John Fitzgerald Kennedy, but NOT John Kennedy

# Advanced search options

- Score Customization with ^n
  - Weights specific terms more than others
  - DB2 AND “IBM i”^5
    - “IBM i” will be weighted more in the score results than ‘DB2’
- XML search
  - A subset of the XPATH query language is supported
  - Search for text contained in specified XML element/tag

# Search Example

- Find all of the applicants who have RPG experience in their resume.
  - **“RPG” search term delimited, so only matches returned for upper-case**

```
SELECT first_name || ' ' || last_name  
FROM resumes  
WHERE  
    CONTAINS(applicant_resume, "RPG") = 1
```



# Search Example

- Find resumes with a match on 'software engineer' in Spanish

```
SELECT first_name || ' ' || last_name, email_address,  
       SCORE(applicant_resume, 'ingeniero de software',  
             'QUERYLANGUAGE=es_ES')  
FROM myschema.resumes  
WHERE  
      CONTAINS( applicant_resume, 'ingeniero de software',  
              'QUERYLANGUAGE=es_ES') = 1  
ORDER BY 3 DESC
```

# Data Cleansing

# Finding Duplicate Rows in SQL

A great way to find duplicate rows is by using window functions – supported by DB2 on IBM i.

Consider a follow table dedup with duplicates:

name	email
mike	mike@example.com
mike k	mike@example.com
sean	sean@example.com
sean	sean@example.com
taylor	taylor@example.com

# Finding Duplicate Rows in SQL

```
select email, count(*)  
from dedup  
group by email  
having count(*) > 1;
```

EMAIL	00002
sean@example.com	2
mike@example.com	2

# Finding Duplicate Rows in SQL

The next step is to number the duplicate rows with the row\_number window function:

```
select row_number() over (partition by email),  
       name,  
       email  
from dedup;
```

00001	NAME	EMAIL
1	mike	mike@example.com
2	mike k	mike@example.com
1	sean	sean@example.com
2	sean	sean@example.com
1	taylor	taylor@example.com

# Finding Duplicate Rows in SQL

We can then wrap the above query filtering out the rows with row\_number column having a value greater than 1.

```
select * from (  
    select row_number() over (partition by email) as rij,  
           name, email from dedup ) t  
where t.rij < 2;
```

RIJ	NAME	EMAIL
1	mike	mike@example.com
1	sean	sean@example.com
1	taylor	taylor@example.com

# Finding Duplicate Rows in SQL

Notes about the ROW\_NUMBER window function

The row\_number is a standard window function and supports the regular parameters for a window function. We'd like to point out two cases that are of interest:

In a case where you want to pick a deduplicate row according a different criteria, you can make use of the ORDER clause inside the window function to order the partition.

In a case where you want to deduplicate on multiple columns, you can specify those columns as parameters to the partition clause.

# Concatenating Rows of String Values for Aggregation

What would an aggregation of rows of string columns look like?

For numeric columns, we can calculate averages and sums over many rows to aggregate them.

Rows with string columns require a different treatment.

This tip is useful in many-to-many relationships, when you want to aggregate one side of a relationship for reporting purposes:

security management – a table of user and their corresponding roles in an organization

account management – a table of sales people / account managers and the customers they manage

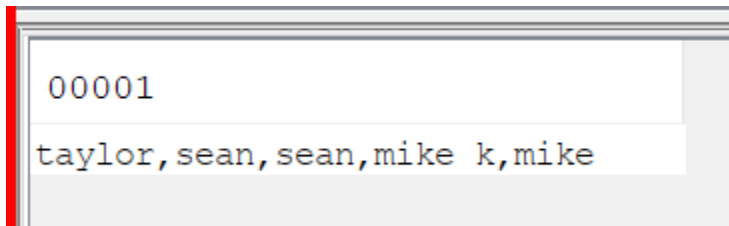
inventory management – a table of stores and a list of products they hold or have sold

tag management – a table of tags applied to resources and you want to show a summary of tags against a resource



# Concatenating Rows of String Values for Aggregation

```
select listagg(trim(name), ',')  
      within group(order by name desc)  
from dedup;
```



00001	taylor, sean, sean, mike k, mike
-------	----------------------------------

# Finding Patterns & Matching Substrings using Regular Expressions

A common analysis task is to count the number of emails in your database that are of commercial value, i.e., not using disposable email addresses or free providers like gmail or hotmail.

We'll make an assumption that the @ character splits the email address to a name and domain name. If your database table named "emails" contains email addresses like these:

first@gmail.com
second@gmail.com
third@business.com
fourth@provider.com

# Finding Patterns & Matching Substrings using Regular Expressions

```
select email,  
substr(email,1,locate('@',email)-1) as name,  
substr(email,locate('@',email)+1,  
length(substr(email,locate('@',email)+1))) as domain  
from emails;
```

email	name	domain
first@gmail.com	first	gmail.com
second@gmail.com	second	gmail.com
third@business.com	third	business.com
fourth@provider.com	fourth	provider.com

# Finding Patterns & Matching Substrings using Regular Expressions

We'll next need a database of services that hand out free and disposable email addresses.

The github project <https://github.com/willwhite/freemail> has an actively maintained list of such services.

If the data for these services is in a table named "free\_email\_domains", a simple join with this table will classify an email as a business user or a consumer.

# Finding Patterns & Matching Substrings using Regular Expressions

```
select emails.email as email,  
       case when  
         free_email_domains.domain is not null  
         then 'free'  
         else 'business'  
       end as email_type  
from emails  
join free_email_domains  
on free_email_domains.domain = substr(email,locate('@',email)+1,  
length(substr(email,locate('@',email)+1)));
```

# Finding Patterns & Matching Substrings using Regular Expressions

email	email_type
first@gmail.com	free
second@gmail.com	free
third@business.com	business
fourth@provider.com	business

# Finding Patterns & Matching Substrings using Regular Expressions

If you've ever worked with marketers, you would have come across UTM parameters.

These are small bits of information attached to every URL to track campaign and channel effectiveness amongst others.

A tracked URL will look like something like this:

```
http://www.example.com/?utm_source=facebook&utm_medium=social&utm_campaign=black-friday
```

# Finding Patterns & Matching Substrings using Regular Expressions

The parameters are attached as query parameters after the "?" and have standard definitions like source, medium, campaign, etc. You can use a handy tool like the Google Campaign URL Builder to build URLs like these.

Suppose we have a table named pageviews with the following schema:

dt	url
2022-01-01	http://www.example.com/?utm_source=facebook&utm_medium=social&utm_campaign=black-friday
2022-01-02	http://www.example.com/?utm_source=google&utm_medium=cpc&utm_campaign=black-friday

and we want a result set that looks like this:

dt	source	medium	campaign
2022-01-01	facebook	social	black-friday
2022-01-02	google	cpc	black-friday



# Finding Patterns & Matching Substrings using Regular Expressions

We are going to make use the substring and regexp\_replace functions to first extract a substring that matches the regular expression for campaign, source and medium and then replace the matched text to get what we want. Our regular expression is going to take the form:

```
(?!&)utm_campaign=[^&]*(?=&)
```

# Finding Patterns & Matching Substrings using Regular Expressions

```
SELECT dt,  
regexp_replace(regexp_substr(url, '(?!&)utm_campaign=  
[^&]*(?=&)') , 'utm_[^=]*=', '') as utm_campaign FROM  
pageviews;
```

dt	utm_campaign
2022-01-01	facebook
2022-01-02	google

# SQL's NULL values: comparing, sorting, converting and joining with real values

```
select nullable_column from data_table where  
nullable_column is null;
```

Or

```
select nullable_column from data_table where  
nullable_column is not null;
```

# SQL's NULL values: comparing, sorting, converting and joining with real values

```
SELECT DEPTNO, DEPTNAME, COALESCE(MGRNO, 'ABSENT'),  
ADMIRDEPT FROM DEPARTMENT
```

The COALESCE function takes a number of arguments and returns the first non-NULL argument.

```
VALUES COALESCE(DATE1, DATE2, CURRENT DATE)
```

# Filling Missing Data and Gaps by Generating a Continuous Series in SQL

We want to find out the total sum of products sold on a day-by-day basis. A simple query like this will give us that:

```
select dt, sum(sales) as sales from transactions  
group by dt  
order by dt;
```

dt	sales
2023-01-01	27
2023-01-06	4
2023-01-07	28
2023-01-08	25
...	...

# Filling Missing Data and Gaps by Generating a Continuous Series in SQL

Unfortunately DB2 on IBM i does not have like

PostgreSQL has an function called `generate_series` that returns a continuous series as multiple rows.

So we have to write a more complex query than

```
select series as dt,  
       sum(sales) as sales  
from generate_series('2023-01-01'::date, '2023-05-31'::date, '1 day'::interval) as series  
left join transactions on transactions.dt = series  
group by dt  
order by dt;
```

# Filling Missing Data and Gaps by Generating a Continuous Series in SQL

```
with series as (SELECT
    d.min + num.n DAYS as dag
FROM
    -- create inline table with min max date
    (VALUES (DATE('2023-01-01'), DATE('2023-04-01'))) AS d(min, max)
INNER JOIN
    -- create inline table with numbers from 0 to 999
    (
```

# Filling Missing Data and Gaps by Generating a Continuous Series in SQL

```
SELECT
    n1.n + n10.n + n100.n AS n
FROM
    (VALUES(0),(1),(2),(3),(4),(5),(6),(7),(8),(9)) AS n1(n)
CROSS JOIN
    (VALUES(0),(10),(20),(30),(40),(50),(60),(70),(80),(90)) AS n10(n)
CROSS JOIN
    (VALUES(0),(100),(200),(300),(400),(500),(600),(700),(800),(900)) AS n100(n)
) AS num
ON
    d.min + num.n DAYS<= d.max
ORDER BY
    num.n)
```



# Filling Missing Data and Gaps by Generating a Continuous Series in SQL

```
select dag,  
       coalesce(sum(ohtotprice), 0) as sales  
from series  
left join ordrhdr on dag = ordrhdr.ohconfdate  
group by dag  
;
```

# Filling Missing Data and Gaps by Generating a Continuous Series in SQL

DAG	SALES
2023-01-01	0.00
2023-01-02	0.00
2023-01-03	0.00
2023-01-04	35.66
2023-01-05	0.00
2023-01-06	0.00
2023-01-07	0.00
2023-01-08	0.00
2023-01-09	0.00
2023-01-10	0.00
2023-01-11	0.00
2023-01-12	0.00
2023-01-13	0.00
2023-01-14	0.00
2023-01-15	0.00

# Smoothing data

# Calculating Running Total in SQL

For example, if your dataset is a "users\_joined" table like so:

date	user_id
2023-01-01	1
2023-01-02	2
2023-01-02	3
2023-01-02	4
2023-01-02	5
2023-01-02	6
2023-01-02	7
2023-01-03	8
...	...

# Calculating Running Total in SQL

You can compute the total number of users joined in a day like this:

```
select date, count(user_id) from users_joined  
group by date order by date;
```

Which will give you a result like this:

date	count
2023-01-01	1
2023-01-02	6
2023-01-03	1
...	...

# Calculating Running Total in SQL

```
select date,  
       count(user_id) as count,  
       sum(count(user_id)) over (order by date) as running_total  
from users_joined  
group by date  
order by date;
```

date	count	running_total
2023-01-01	1	1
2023-01-02	6	7
2023-01-03	1	8
....		

# Calculating Running/Moving Average in SQL

Sometimes you'll want to compute a running average over a selection of rows for the past N number of time periods. The running average is also called "moving average" or "rolling average".

The reason to use a running average is to smooth out the highs and lows of the data set and get a feel for the trends in the data.

Let's consider a table with quarterly revenues :

quarter	revenue
2022-1	700.356
2022-2	667.625
2022-3	639.281
2022-4	1115.171
...	...

# Calculating Running/Moving Average in SQL

To smooth out these huge jumps in revenue, we can compute a moving average that averages the previous three periods:

```
select quarter,  
       revenue,  
       avg(revenue) over (order by quarter rows between 3 preceding and  
current row)  
from table_revenue;
```



# Calculating Weighted Moving Average in SQL

**A weighted moving average is a moving average where the previous values within the sliding window are given different weights. This is usually done to make recent points more important.**

We are going to calculate a 4-period moving average. A simple way to compute the weights is to sum up the number of periods, and create fractional weights by dividing the weights by the total.

quarter	fraction	weight
current	4/10	0.4
current-1	3/10	0.3
current-2	2/10	0.2
current-3	1/10	0.1

You can pick what ever weights you want according to you needs, just make sure the weights add up to one.

# Calculating Weighted Moving Average in SQL

## 1. SQL row\_number to number the rows

```
select quarter, revenue, row_number() over ()  
from table_revenue;
```

quarter	revenue	row_number
2022-1	700.356	1
2022-2	667.625	2
2022-3	639.281	3
2022-4	1115.171	4
...	...	...

# Calculating Weighted Moving Average in SQL

2. SQL self-join to create a sliding window

with t as

```
(select quarter, revenue, row_number() over ()
```

```
    from table_revenue)
```

```
select t.quarter, t.row_number as row_number, t2.quarter as quarter_2, t2.row_number as row_number_2
```

```
from t
```

```
join t t2 on t2.row_number between t.row_number - 3 and t.row_number
```

quarter	row_number	quarter_2	row_number_2
2022-1	1	2022-1	1
2022-2	2	2022-1	1
2022-2	2	2022-2	2
2022-3	3	2022-1	1
2022-3	3	2022-2	2
...	...	...	...

# Calculating Weighted Moving Average in SQL

## 3. SQL case to use the fractional weights

Now it's just a matter of finding the difference between the row numbers and applying the fractional weights using a SQL CASE statement.

case

when t.row\_number - t2.row\_number = 0 then 0.4 \* t2.revenue

when t.row\_number - t2.row\_number = 1 then 0.3 \* t2.revenue

when t.row\_number - t2.row\_number = 2 then 0.2 \* t2.revenue

when t.row\_number - t2.row\_number = 3 then 0.1 \* t2.revenue

end

# Calculating Weighted Moving Average in SQL

```
with t as
  (select quarter, revenue, row_number() over ()
   from table_revenue)
select t.quarter, avg(t.revenue) as revenue,
sum(case
  when t.row_number - t2.row_number = 0 then 0.4 * t2.revenue
  when t.row_number - t2.row_number = 1 then 0.3 * t2.revenue
  when t.row_number - t2.row_number = 2 then 0.2 * t2.revenue
  when t.row_number - t2.row_number = 3 then 0.1 * t2.revenue
end)
from t
join t t2 on t2.row_number between t.row_number - 3 and t.row_number
group by 1
order by 1
```

# Calculating Exponential Moving Average in SQL with Recursive CTEs

Similar to simple/weighted moving averages, exponential moving averages (EMA) smooth out the observed data values.

The difference is that these methods use the previously calculated EMA value as a basis rather than the original (non-smooth) data value.

Since EMA builds upon itself, all previous data values have some effect on the new EMA, though the effect diminishes quickly with time.

You would implement this in SQL when for example you have sales data that shows a strong seasonal component.

Moreover, when it's trending higher period-over-period.

# Calculating Exponential Moving Average in SQL with Recursive CTEs

Queries can have a WITH clause, that allows you use statements – know as common table expressions (CTE) – that behave like temporary tables that only exist during the execution of the query.

If the CTE contains a recursive component, that's a recursive query. Recursive queries are useful to query data that demonstrate a hierarchical nature.

To define a recursive query, we need two parts – an initial query that is non-recursive and the recursive part. The general form of a recursive query is as follows:

```
with cte_name (  
    cte_query_definition -- non-recursive portion  
    union [all]  
    cte_query_definion  -- recursive portion  
)
```

# Calculating Exponential Moving Average in SQL with Recursive CTEs

```
with recursive t as (  
    select dt,  
           0.5 as alpha,  
           row_number() over (),  
           sales  
    from sales_data  
),
```



# Calculating Exponential Moving Average in SQL with Recursive CTEs

```
ema as (  
  select *, sales as sales_ema from t  
  where row_number = 1  
  union all  
  select t2.dt,  
         t2.alpha,  
         t2.row_number,  
         t2.sales,  
         t2.alpha * t2.sales + (1.0 - t2.alpha) * ema.sales as sales_ema  
  from ema  
  join t t2 on ema.row_number = t2.row_number - 1  
)  
  
select dt, sales, sales_ema  
from ema;
```

# Calculations per group

# Calculating Percentage (%) of Total Sum in SQL

This question comes up frequently when you want to the relative contribution of a row against the backdrop of the total sum of all the rows. For example:

- how is the browser marketshare changing over time ?
- What's each sales person's contribution to your company's revenue ?

Consider a table with the number of page view (in billions) with each browser:

Browser	Pageviews
Chrome	7.1685
Safari	1.935
Firefox	1.3455
UC Browser	1.0965
IE	1.341
Opera	0.816
Android	0.7245
Rest	1.2

# Calculating Percentage (%) of Total Sum in SQL

```
with total as
  ( select sum(pageviews) as total
    from pageviews )
select browser,
       pageviews / total.total as share
from pageviews, total
```

Which gives a ratio of each browser to the total:

Browser	Share
Chrome	0.895
Safari	0.241875
...	

# Percentage to Total per Group

What we are attempting to do here is to group our data into months, compute the total for that group and for each row within that group compute a ratio. An overall total wouldn't make sense.

dt	Browser	Pageviews
2022-01-01	Chrome	7.1685
2022-01-01	Safari	1.935
2022-01-01	...	...
2022-01-02	Chrome	7.2485
2022-01-02	Safari	1.721
2022-01-02	...	...

We once again to resort to window functions with a partition over the month portion of the datetime.

# Percentage to Total per Group

```
select extract(MONTH FROM dt),  
       browser,  
       pageviews / sum(pageviews) over(partition by extract(MONTH FROM  
dt))  
from pageviews;
```

# Calculating Difference from Beginning/First Row in SQL

First value in the entire table

If you simple the first value in the entire table, the first\_value is what you're looking for. The window definition excludes any parameters (the OVER clause) and so your window is over the entire table:

```
select dt,  
       price,  
       first_value(price) over ()  
from trades;
```

# Calculating Difference from Beginning/First Row in SQL

```
select dt,  
       price,  
       first_value(price) over (partition by dt)  
from trades;
```

Also, last\_value

Just like the first\_value window function, you also have access to the last\_value function that picks the last value in the current partition.



# Calculating Top N items per Group in SQL

How to find the top / bottom "N" rows in each group?

```
select country, city, population, row_number() over (order by population desc) as country_rank from cities;
```

country	city	population	country_rank
United States	New York	8175133	1
United Kingdom	London	7825300	2
United States	Los Angeles	3792621	3
United States	Chicago	2695598	4
France	Paris	2181000	5
...			

# Calculating Top N items per Group in SQL

```
select country,  
       city,  
       population,  
       row_number() over (partition by country order by population desc) as country_rank  
from cities;
```

country	city	population	country_rank
France	Paris	2181000	1
France	Marseille	808000	2
France	Lyon	422000	3
United Kingdom	London	7825300	1
United Kingdom	Birmingham	1016800	2

# Calculating Top N items per Group in SQL

```
select * from (  
    select country, city, population,  
        row_number() over (partition by country order by population desc) as country_rank  
    from cities) ranks where country_rank <= 2;
```

country	city	population	country_rank
France	Paris	2181000	1
France	Marseille	808000	2
United Kingdom	London	7825300	1
United Kingdom	Birmingham	1016800	2
United States	New York	8175133	1
United States	Los Angeles	3792621	2

# Calculating Top N items and Aggregating (sum) the remainder into "All Other"

```
select name, sales
from deals
order by sales desc
Fetch first 10 rows only;
```

name	sales
Alice	8405837
Bob	3884307
Chris	2718782
Dan	2195914
Eve	1553165
Fae	1513367
George	1409019
...	

# Calculating Top N items and Aggregating (sum) the remainder into "All Other"

What we are trying to accomplish is a final row appended to the results above that is the aggregation of the rest of the data, something like this:

name	sales
Alice	8405837
Bob	3884307
Chris	2718782
...	...
Justine	998537
All Other	7833544

# Calculating Top N items and Aggregating (sum) the remainder into "All Other"

```
with top10 as
  (select name, sales from deals order by sales desc
   fetch first 10 rows only)
select * from top10
union all
select 'All other' as name, sum(sales) as sales
from sales
where name not in
  (select name
   from top10);
```

# Summarizing Data

# Calculating Summary Statistics in SQL

- When you first get your hands on a data set, what's it like:
- quickly get a feel for the data?
- are there outliers?
- is the data shaped abnormally?

These are questions you might have about your data. The disadvantage of a spreadsheet-like interface is that it's difficult to understand your data, speically when you're seeing one for the first time.

For this purpose, we have summary statistics. Fortunately, SQL has a robust set of functions to do exactly that.



# Calculating Summary Statistics in SQL

As an example, we'll use a list of the fastest growing companies in the United States according to Inc. magazine. The data table is formatted as:

Name	State	Industry	Employees	Revenue
Fuhu	California	Consumer Products & Services	227	195640000
Quest Nutrition	California	Food & Beverage	191	82640563
Reliant Asset Management	Virginia	Business Products & Services	145	85076502
Superfish	California	Software	62	35293000
Acacia Communications	Massachusetts	Telecommunications	92	77652360
Provider Power	Maine	Energy	50	137977203
Crescendo Bioscience	California	Health	129	27308000

# Calculating Summary Statistics in SQL

```
select 'total',  
       sum(employees) as employees,  
       sum(revenue) as revenue  
from inc5000  
union  
select 'average',  
       avg(employees),  
       avg(revenue)  
from inc5000
```

# Calculating Summary Statistics in SQL

```
union
select 'min',
       min(employees),
       min(revenue)
from inc5000
union
select 'max',
       max(employees),
       max(revenue)
from inc5000
```

# Calculating Summary Statistics in SQL

?column?	employees	revenue
Total	65301	15088861318
Avg	96.45642540620383	22320800.76627219
Max	5603	985737000
Min	0	1953000

# Calculating Summary Statistics in SQL

```
select * from ( select 5, company, employees,  
revenue from inc5000  
union select 1, 'total', sum(employees) as  
employees, sum(revenue) as revenue from inc5000  
union select 2, 'avg', avg(employees), avg(revenue)  
from inc5000  
union select 3, 'min', min(employees), min(revenue)  
from inc5000  
union select 4, 'max', max(employees), max(revenue)  
from inc5000) stats order by 1
```

# Calculating Summary Statistics in SQL

?column?	company	employees	revenue
1	Total	65301	15088861318
2	Avg	96.45642540620383	22320800.76627219
3	Min	0	1953000
4	Max	5603	985737000
5	Transactis	61	5485216
5	Bareburger	268	31150000
5	SmartZip Analytics	96	11396150

# Making Histogram Frequency Distributions in SQL

PostgreSQL has a function `width_bucket` that will return the bucket number for a numeric value based on a range and the number of buckets necessary. Unfortunately DB2 for i does not have it so we need to do it manually :

```
select
  case
    when salary between 75000 and 90000 then '75000-90000'
    when salary between 90000 and 120000 then '90000-120000'
    else '120000+'
  end as salary_band,
  count(*)
from employee_salary
group by salary order by salary;
```

# Making Histogram Frequency Distributions in SQL

SQL `ntile` can be used for histograms with equal height bucket widths

If you want to optimize for bucket widths so that each bucket has the same number of salary counts, you can use the `ntile` window function to find the bucket widths.

In the field of image processing, this is similar to histogram equalization.



# Making Histogram Frequency Distributions in SQL

```
select quartile , min(ohtotprice), max(ohtotprice)
from (
    select ohtotprice,
           ntile(4) over (order by ohtotprice) as quartile from ordhdr
) x
group by quartile
order by quartile ;
```

# Making Histogram Frequency Distributions in SQL

QUARTILE	00002	00003
1	0.00	94.07
2	98.31	250.00
3	250.00	302.50
4	302.50	118325.87

# Calculating percentiles, quartiles, deciles, and N-tiles in SQL

- A percentile is a measure used in statistics indicating the value below which a given percentage of observations in a group of observations fall. For example, the 60th percentile is the value below which 60% of the observations may be found.
- Given a set of observations that has been sorted, the median, first quartile and third quartile can be used split the data into four pieces. The first quartile is the point at which one fourth of the data lies below it. The median is located exactly in the middle of the data set, with half of all of the data below it. The third quartile is the place where three fourths of the data lies below it.
- Percentiles have a lot of applications:
  - Used to monitor SLA violations (eg. meet 300ms response time)
  - Burstable billing (which is the example we'll use)

# Calculating percentiles, quartiles, deciles, and N-tiles in SQL

Burstable billing is a common practice among service providers to not penalize occasional overages. Consider a table with 1 day apart samples of bandwidth consumed:

dt	customer_id	bandwidth
2023-01-01	1	4.0
2023-01-01	2	1.2
2023-01-02	1	8.0
2023-01-02	2	3.2
2023-01-03	1	1.4
...	...	...

# Calculating percentiles, quartiles, deciles, and N-tiles in SQL

You want to charge your customers based on the 95th percentile.

Fortunately, we have access to the NTILE window function that divides an ordered partition into buckets and assigned a bucket number to each row in the partition.

Step by step:

- Use `ntile(100)` to split the data into 100 roughly even sized buckets.
- Use the `partition` parameter in the window definition to specify a partition by `customer_id`.
- Pick the max value in the 95th bucket for that customer.

# Calculating percentiles, quartiles, deciles, and N-tiles in SQL

```
select dt,  
customer_id,  
sum(bandwidth) as bandwidth  
from usage  
group by dt, customer_id;
```

# Calculating percentiles, quartiles, deciles, and N-tiles in SQL

```
select customer_id,  
       bandwidth,  
       ntile(100) over (partition by customer_id order by  
                        bandwidth asc)  
       percentile from ( ... )  
customer_bandwidth_percentiles;
```

# Calculating percentiles, quartiles, deciles, and N-tiles in SQL

```
select customer_id, max(bandwidth)

from ( ... ) where
    customer_bandwidth_percentiles.percentile = 95
group by customer_id;
```



# Gap Analysis to find missing values in a sequence

There are times when you want to find all gaps in a sequence of numbers, dates or data with a logically consecutive nature. A related scenario is to find ranges of unbroken sequences. There are a variety of use-cases of gap analysis:

- If you require your employees to check in to a timesheet everyday, a gap analysis can show you days where the employee was absent.
- If you have a fleet of vehicles doing deliveries, a gap analysis can show periods of time when all vehicles are not being used. Useful for maintenance or downtime.
- If you have a service level agreement for 24/7 coverage, a gap analysis can show you contractual breaches.

# Gap Analysis to find missing values in a sequence

```
create table gap (counter integer);
```

```
insert into gap (counter) values (1);
```

```
insert into gap (counter) values (2);
```

```
insert into gap (counter) values (5);
```

```
insert into gap (counter) values (6);
```

```
insert into gap (counter) values (8);
```

```
insert into gap (counter) values (9);
```

```
insert into gap (counter) values (10);
```

# Gap Analysis to find missing values in a sequence

```
select counter from gap;
```

COUNTER
1
2
5
6
8
9
10

# Finding Gaps with an exclusion join

```
with dummy(id) as (  
    select 1 from SYSIBM.SYSDUMMY1  
    union all  
    select id + 1 from dummy where id < 10  
)  
select id from dummy;
```

ID
1
2
3
4
5
6
7
8
9
10

# Finding Gaps with an exclusion join

```
with dummy(id) as (  
    select 1 from SYSIBM.SYSDUMMY1  
    union all  
    select id + 1 from dummy where id < 10  
)  
select id,  
       gap.counter  
from dummy  
left join gap on id = gap.counter;
```

ID	COUNTER
1	1
2	2
3	-
4	-
5	5
6	6
7	-
8	8
9	9
10	10

# Finding Gaps with an exclusion join

```
with dummy(id) as (  
  select 1 from SYSIBM.SYSDUMMY1  
  union all  
  select id + 1 from dummy where id < 10  
)  
select id,  
       gap.counter  
from dummy  
left join gap on id = gap.counter  
where counter is null;
```

ID	COUNTER
3	-
4	-
7	-

# Finding ranges of missing gaps

```
select gap.counter + 1 as start
  from gap
 left join gap r on gap.counter = r.counter - 1
 where r.counter is null;
```

START
3
7
11

# Finding ranges of missing gaps

```
select min(fr.counter) - 1 as stop
  from gap
 left join gap fr on gap.counter < fr.counter
 where fr.counter is not null
 group by gap.counter;
```

STOP
1
4
5
7
8
9



# Finding ranges of missing gaps

```
select gap.counter + 1 as start,  
       min(fr.counter) - 1 as stop  
from gap  
left join gap r on gap.counter = r.counter - 1  
left join gap fr on gap.counter < fr.counter  
where r.counter is null  
       and fr.counter is not null  
group by gap.counter, r.counter;
```

START	STOP
3	4
7	7

# Finding ranges of continuous values

```
select *  
from gap  
left join gap s on s.counter = gap.counter - 1;
```

COUNTER	COUNTER
1	—
2	1
5	—
6	5
8	—
9	8
10	9

# Finding ranges of continuous values

```
select gap.counter as stop, e.counter  
from gap  
left join gap e on e.counter = gap.counter + 1  
where e.counter is null;
```

STOP	COUNTER
2	-
6	-
10	-

# Finding ranges of continuous values

```
select gap.counter as start,  
(select a.counter as counter from gap a  
left join gap b on b.counter = a.counter + 1  
where b.counter is null and a.counter >= gap.counter  
limit 1) as stop  
from gap left join gap s on s.counter = gap.counter  
- 1 where s.counter is null;
```

START	STOP
1	2
5	6
8	10

# Making Correlation Coefficient Matrices to understand relationships in SQL

A scatter X-Y plot is a straightforward way to visualize the dependency between two variables. However, at times you want to understand how more than two variables are related. The correlation coefficient  $r$  can be calculated between pairs of variables that scales between +1 and -1 demonstrating the degree of positive or negative correlation.

You can use the correlation matrix to figure out what activities are correlated, to plan future activities. For example, within your customer support department, how are net promoter scores (NPS) related to support wait times.

# Making Correlation Coefficient Matrices to understand relationships in SQL

Mailchimp campaign example table

```
CREATE TABLE stats ( campaign_id varchar(16),  
sent_time timestamp, subject varchar(256), email  
varchar(64), open_count integer, click_count integer  
);
```

CAMPAIGN_ID	SENT_TIME	SUBJECT	EMAIL	OPEN_COUNT	CLICK_COUNT
TVSPOT01	2023-03-23 14:47:00.000000	Une nouvelle campagne télévisée avec notre spot bien connu	4	4	1
TVSPOT02	2023-03-23 14:48:00.000000	Une nouvelle campagne télévisée avec notre spot bien connu	169	167	2
TVSPOT03	2023-03-23 14:11:32.000000	Een nieuwe TV-campagne met onze gekende spot	44	43	1
TVSPOT04	2023-03-23 11:32:00.000000	Een nieuwe TV-campagne met onze gekende spot	479	468	11
CUBEDE	2023-02-28 17:46:00.000000	Bestätigung Beteiligung Cube Medienkampagne	40	40	32

# Making Correlation Coefficient Matrices to understand relationships in SQL

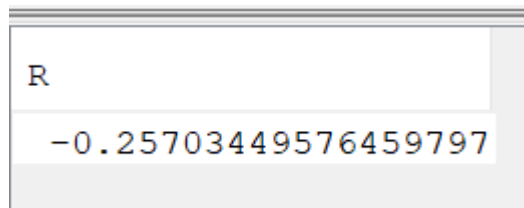
```
with table_mean as
( select avg(length(subject)) as mean_subject_length,
        avg(open_count) as mean_open_rate
  from stats ),
table_corrected as
( select length(subject) - mean_subject_length as mean_subject_length_corrected,
        open_count - mean_open_rate as mean_open_rate_corrected
  from table_mean, stats )
select sum(mean_subject_length_corrected * mean_open_rate_corrected) /
sqrt(sum(mean_subject_length_corrected * mean_subject_length_corrected) *
sum(mean_open_rate_corrected * mean_open_rate_corrected)) as r
from table_corrected;
```

R
-0.2564859923469527

# Making Correlation Coefficient Matrices to understand relationships in SQL

Fortunately, we don't have to repeat this each time, we can simply use the in-built corr function to calculate the correlation for us. Notice the small difference with the manual calculation.

```
select corr(length(subject), open_count) as r from stats;
```

A screenshot of a SQL query result. It shows a single row with a column header 'R' and a value '-0.25703449576459797'.

R
-0.25703449576459797



# Making Correlation Coefficient Matrices to understand relationships in SQL

```
select 'subject_length' as rij,  
       'subject_length' as col,  
       corr(length(subject), length(subject)) as coeff  
from stats  
union  
select 'subject_length' as rij,  
       'open_rate' as col,  
       corr(length(subject), open_count) as coeff  
from stats  
union  
select 'subject_length' as rij,  
       'click_rate' as col,  
       corr(length(subject), click_count) as coeff  
from stats
```

# Making Correlation Coefficient Matrices to understand relationships in SQL

```
union
select 'open_rate' as rij,
      'open_rate' as col,
      corr(open_count, open_count) as coeff
from stats
union
select 'open_rate' as rij,
      'click_rate' as col,
      corr(open_count, click_count) as coeff
from stats
union
select 'click_rate' as rij,
      'click_rate' as col,
      corr(click_count, click_count) as coeff
from stats
```

# Making Correlation Coefficient Matrices to understand relationships in SQL

ROW	COL	COEFF
open rate	open rate	1.0
subject length	subject length	1.0
open rate	click rate	0.002180780704690579
subject length	click rate	-0.5822709249114572
subject length	open rate	-0.25703449576459797
click rate	click rate	1.0

# Making Correlation Coefficient Matrices to understand relationships in SQL

```
select rij,  
sum(case when col='subject_length' then coeff else 0 end) as  
subject_length,  
sum(case when col='open_rate' then coeff else 0 end) as  
open_rate,  
sum(case when col='click_rate' then coeff else 0 end) as  
click_rate  
from (  
// ... query as before  
)  
group by rij  
order by rij DESC
```

# Making Correlation Coefficient Matrices to understand relationships in SQL

RIJ	SUBJECT_LENGTH	OPEN_RATE	CLICK_RATE
subject length	1.0	-0.25703449576459797	-0.5822709249114572
open rate	0.0	1.0	0.002180780704690579
click rate	0.0	0.0	1.0

# Calculating Z-Score in SQL

In statistics, the z-score (or standard score) of an observation is the number of standard deviations that it is above or below the population mean.

$$z = \frac{x - \mu}{\sigma}$$

where,

$x$  is the observation,  $\mu$  is the mean and  $\sigma$  is the standard deviation.

# Calculating Z-Score in SQL

You can use the z-score to answer questions like the following:

- What percentage of values fall below a specific value?
- What values can be expected to be outliers?
- What is the relative score of one distribution versus another?  
Literally comparing apples and oranges.

# Calculating Z-Score in SQL

It is generally accepted that z-scores lower than -1.96 or higher than 1.96 to be outliers, or at the least worth a second look. These values are statistically significant at the 95% confidence level. For a higher confidence level of 99%, you are interested in z-scores of higher than 2.576 or lower than -2.576. We are assuming that your data fits a normal distribution.



# Calculating Z-Score in SQL

Consider an ecommerce store that sells products online. The two metrics our analytics team is tracking everyday is sales and website visitors. The table will look something like this:

date	sales	visitors	conversion
2022-01-01	21	3373	0.62%
2022-01-02	50	3820	1.31%
2022-01-03	50	3175	1.57%
2022-01-04	33	4013	0.82%
2022-01-05	58	4022	1.44%
2022-01-06	5	4873	0.25%
2022-01-07	36	1924	1.87%
2022-01-08	44	3867	1.14%
...	...	...	....

# Calculating Z-Score in SQL

```
with sales_stats as
    (select avg(sales) as mean,
            stddev(sales) as sd
     from zscore),
visitor_stats as
    (select avg(visitors) as mean,
            stddev(visitors) as sd
     from zscore)
```

# Calculating Z-Score in SQL

```
select dt,  
       abs(sales - sales_stats.mean) / sales_stats.sd  
       as z_score_sales,  
       abs(visitors - visitor_stats.mean) / visitor_stats.sd as z_score_visitors  
from sales_stats, visitor_stats, zscore;
```

# Calculating Z-Score in SQL

date	z_score_sales	z_score_visitors
2022-01-01	1.02	0.07
2022-01-02	0.77	0.39
2022-01-03	0.77	0.28
2022-01-04	0.28	0.59
2022-01-05	1.26	0.60
2022-01-06	2.00	1.48
2022-01-07	0.09	1.57
2022-01-08	0.40	0.44



# Forecasting & predicting the future

# Calculating Linear Regression Coefficients in SQL

Regression is an incredibly powerful statistical tool, when used correctly, has the ability to help you predict the future. One could argue that hypothesis testing and prediction together are the pillars of data analysis.

The basic regression analysis uses fairly simple formulas to get a "best fit" line through your data. We are going to be specifically looking at simple regression analysis.

REMARK DB2 FOR I has regression functions which not all db's have.

# Calculating Linear Regression Coefficients in SQL

Businesses love regression analysis because it gives you powerful tools to:

- forecast next year's revenue
- predict site visitors given past historical and seasonal trends
- generate a risk score given demographics.

# Calculating Linear Regression Coefficients in SQL

```
select avg(x) as x_bar,  
       avg(y) as y_bar  
from ols;
```

x_bar	y_bar
255	395.671



# Calculating Linear Regression Coefficients in SQL

By using window functions, we can repeat the averages  $\bar{x}$  and  $\bar{y}$  row by row. This is to calculate the term :

```
select x,  
       avg(x) over () as x_bar,  
       y,  
       avg(y) over () as y_bar  
from ols;
```

# Calculating Linear Regression Coefficients in SQL

```
select
sum((x - x_bar) * (y - y_bar)) /
sum((x - x_bar) * (x - x_bar))
as slope
from ( select x, avg(x) over () as x_bar,
              y, avg(y) over () as y_bar
        from ols) s;
```

slope
1.5930700120048

# Calculating Linear Regression Coefficients in SQL

```
select slope,  
       y_bar_max - x_bar_max * slope as intercept  
from (  
    select sum((x - x_bar) * (y - y_bar)) / sum((x -  
x_bar) * (x - x_bar)) as slope, max(x_bar) as  
x_bar_max, max(y_bar) as y_bar_max  
from ( select x, avg(x) over () as x_bar, y,  
            avg(y) over () as y_bar from ols) s; )
```

# Calculating Linear Regression Coefficients in SQL

slope	intercept
1.5930700120048	-10.5618530612244

Hence, our best fit regression line has the equation:

$$y = 1.5930700120048 * x - 10.5618530612244$$

# Forecasting when you have Seasonal effects using the Ratio to Moving Average method in SQL

Many sales activities, product adoption and other business activities – customer support, website traffic, etc. show two kinds of patterns: a upward/downward trend and a seasonal variation. For example, not only has Amazon's revenue been trending up (growing) year after year for the last 20 years, but also shows seasonality with most of the sales made in the last four months of year (specially during Black Friday and Christmas.)

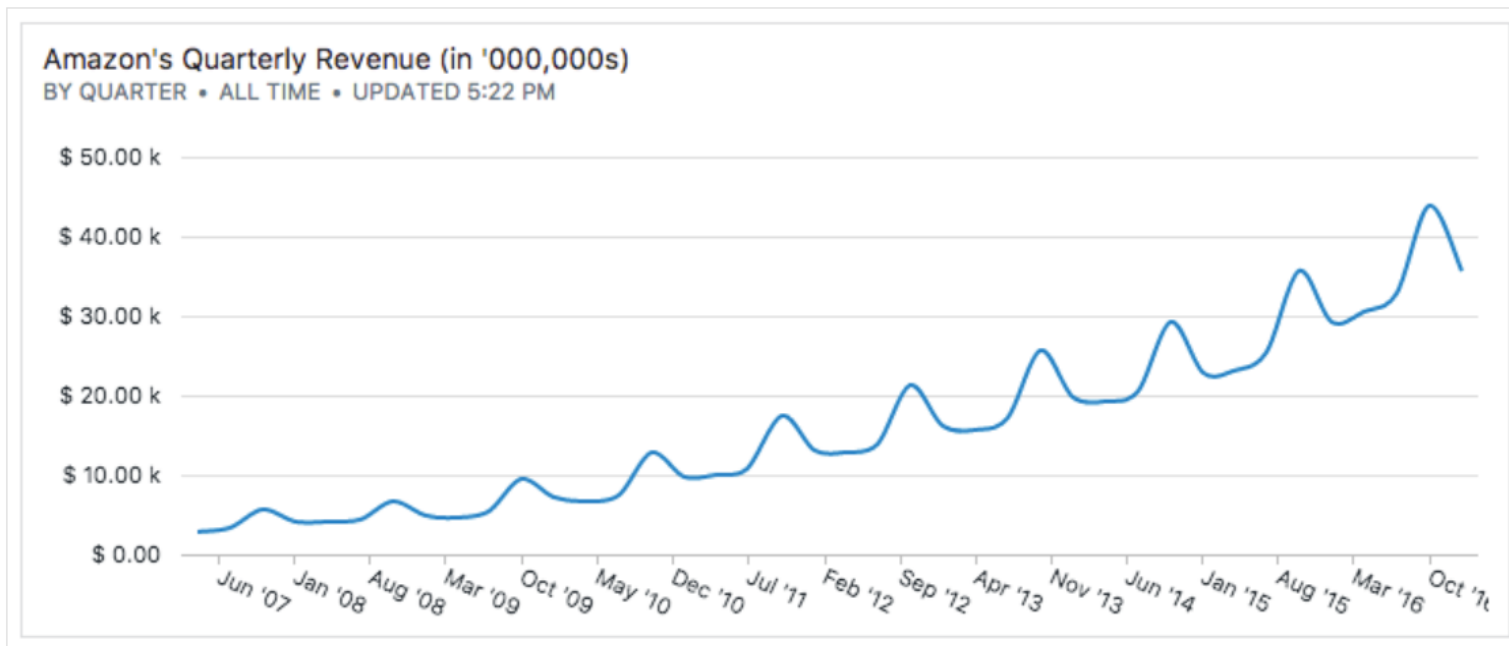
# Forecasting when you have Seasonal effects using the Ratio to Moving Average method in SQL

As a data analyst, if you were interested in knowing whether sales is trending up or down – the seasonality will throw you off. The sales for January and February is always less than December (because of Christmas) and without correcting for seasonality, it would appear that sales is trending down.

Modeling seasonal effects is important in order to accurately predict how you need to provision resources and maintain sufficient lead time. The ratio-to-moving average method is an easy to use method to pick out the seasonal effects and forecast future values.

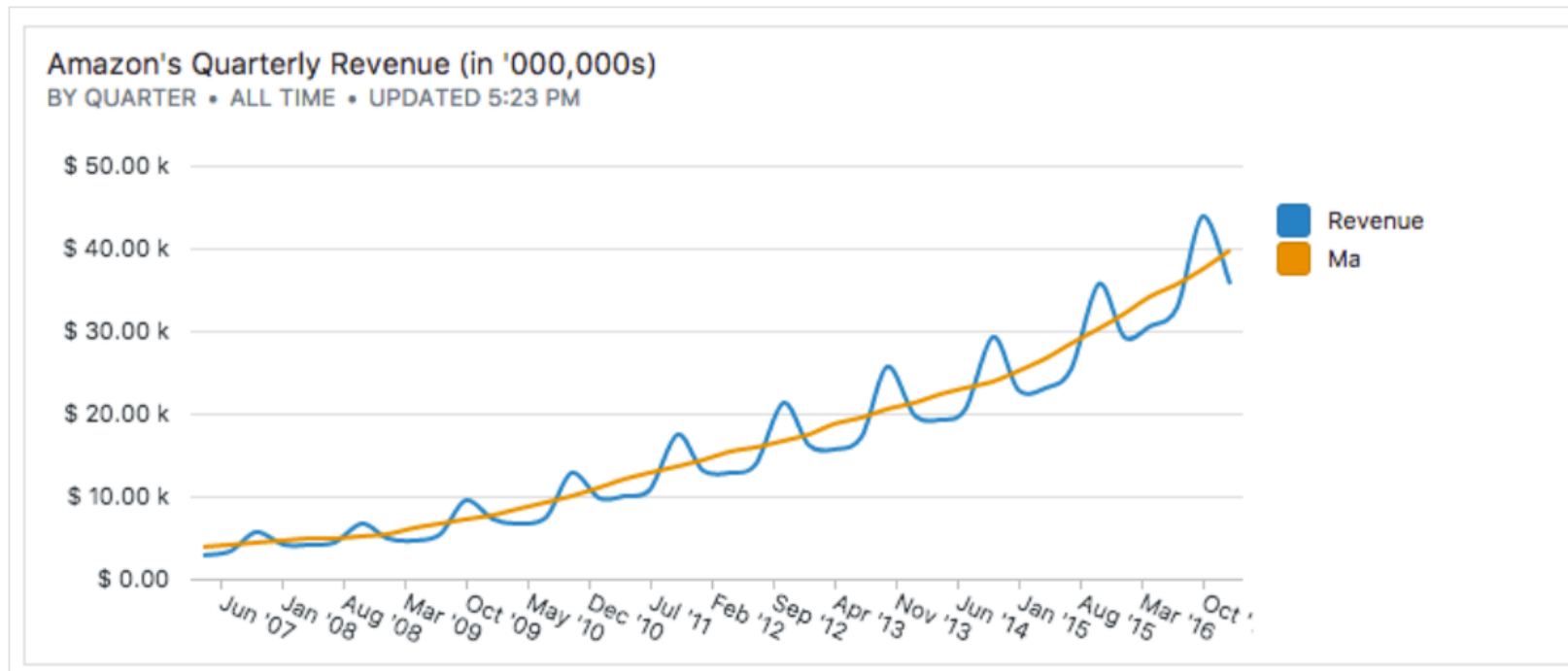
# Determine a Seasonal Index using Moving Averages

```
select quarter, revenue from amazon_revenue order by quarter asc
```



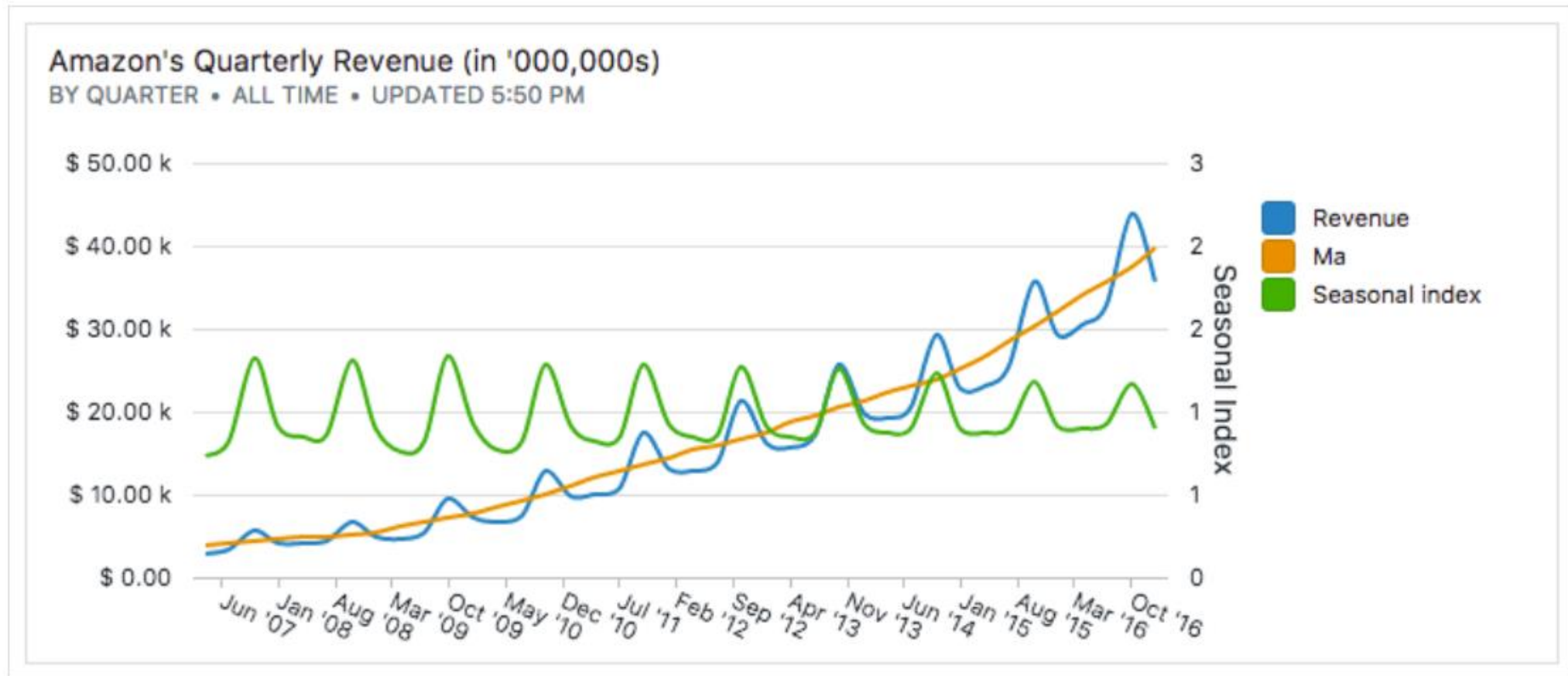
# Determine a Seasonal Index using Moving Averages

```
select quarter, revenue, avg(revenue) over (order by  
quarter rows between 1 preceding and 2 following) as  
ma from amazon_revenue order by quarter asc
```





# Determine a Seasonal Index using Moving Averages



# Determine a Seasonal Index using Moving Averages

```
select substr(yyymmdd, 1, 6) as d,  
       avg(revenue * 1.0 / ma) as si  
from ( select row_number() over (order by quarter)  
as n, [quarter:quarter] as q, revenue,  
avg(revenue) over(order by quarter rows between 1  
preceding and 2 following) as ma  
from amazon_revenue order by q asc) c  
group by substr(yyymmdd, 1, 6)
```

# Determine a Seasonal Index using Moving Averages

d	si
1	0.9121396120732432
4	0.8246402054415246
7	0.8599421103521137
10	1.2633667197200424

# Identifying the trend portion of the time series

Intuitively, Amazon's revenue is trending upwards year-over-year. We can run a regression using least squares to identify the trend portion.

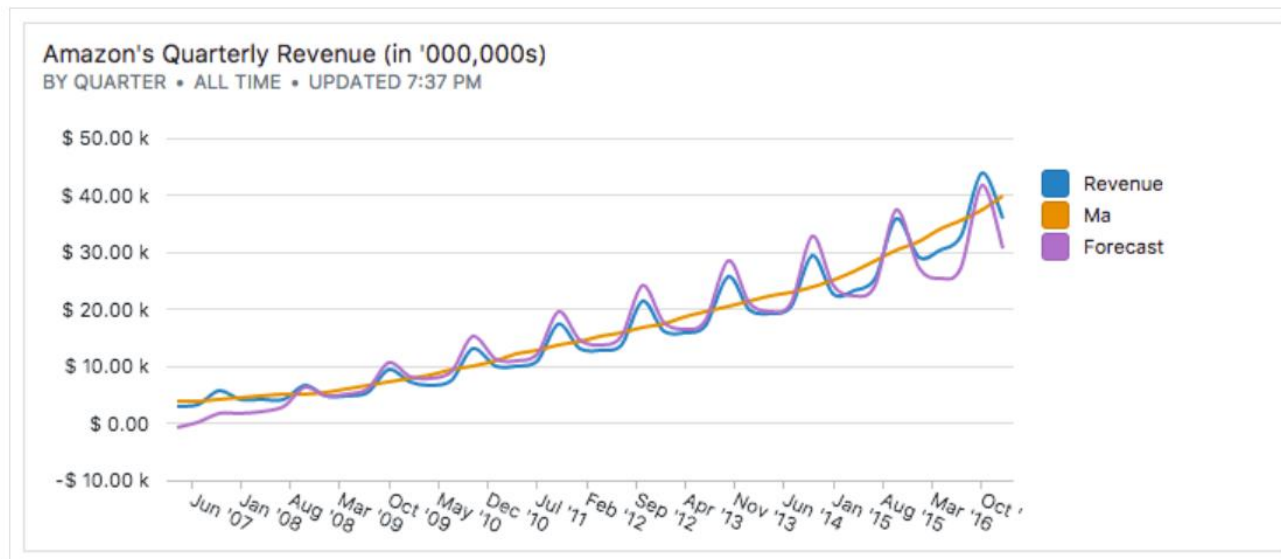
We'll take a shortcut and use `regr_slope` and `regr_intercept` to compute the regression line's slope and intercept.

# Identifying the trend portion of the time series

```
select
regr_slope(ma, extract(epoch from quarter)) as
slope,
regr_intercept(ma, extract(epoch from quarter)) as
intercept
from (
    select quarter, revenue, avg(revenue) over
    (order by quarter rows between 1 preceding and 2
following) as ma
    from amazon_revenue order by quarter asc ) a
```

# Forecasting using Deseasonalized data

By combining the regression as well as the seasonality index, we have now acquired the capability to forecast future sales! To actually perform the forecast, we'll project the trend line from the linear regression into future periods, and then adjust these trend values to account for the seasonal factors.



# Forecasting using Deseasonalized data

with base as

```
( select n, q, revenue, ma from ( select  
row_number() over (order by quarter) as n,  
[quarter:quarter] as q, revenue, avg(revenue)  
over(order by quarter rows between 1 preceding  
and 2 following) as ma  
from amazon_revenue order by q asc )  
b),
```

# Forecasting using Deseasonalized data

```
si as (  
    select substr(yyymmdd, 1, 6),  
    avg(revenue * 1.0 / ma) as si from ( select  
    row_number() over (order by quarter) as n,  
    [quarter:quarter] as q, revenue, avg(revenue)  
    over(order by quarter rows between 1 preceding  
    and 2 following) as ma from amazon_revenue order  
    by q asc) c group by 1 ),
```



# Forecasting using Deseasonalized data

trend as

```
( select regr_slope(ma, n) as slope,  
  regr_intercept(ma, n) as intercept from (select  
  row_number() over (order by quarter) as n,  
  [quarter:quarter] as q, revenue, avg(revenue)  
  over(order by quarter rows between 1 preceding  
  and 2 following) as ma from amazon_revenue order  
  by q asc) a )
```

# Forecasting using Deseasonalized data

```
select base.q,  
       base.revenue,  
       base.ma, trend.intercept + trend.slope * base.n  
       * case extract('MONTH', base.q)  
           when 1 then 0.9121396120732432  
           when 4 then 0.8246402054415246  
           when 7 then 0.8599421103521137  
           when 10 then 1.2633667197200424 end as  
forecast from trend, base
```

# Growth rates

# Calculating Month-Over-Month Growth Rate in SQL

```
select trunc_timestamp(ohconfdate, 'MONTH') as date,  
       count(*) as count  
from ordrhdr  
group by trunc_timestamp(ohconfdate, 'MONTH')  
order by 1;
```

date	count
2023-01-01	10
2023-02-01	12
2023-03-01	15
...	...

The above query should give us a neat table with the number of sales created every month.

# Calculating Month-Over-Month Growth Rate in SQL – LAG function

```
WITH MonthlySales AS (  
  select trunc_timestamp(ohconfdate, 'MONTH') as Maand,  
         count(*) as Aantal  
  from ordrhdr  
  group by trunc_timestamp(ohconfdate, 'MONTH')  
)  
SELECT q.*,  
       lag(aantal,1) over(order by Maand) as Vorige  
  FROM MonthlySales q  
 ORDER BY Maand
```

# Calculating Month-Over-Month Growth Rate in SQL

date	count	lag
2023-01-01	10	
2023-02-01	12	10
2023-03-01	15	12
...	...	...

The lag function returns a value evaluated at the row that is definable offset before the current row within the partition. In this particular we have simply picked the value from the previous row (offset of 1).

# Calculating Month-Over-Month Growth Rate in SQL

```
WITH MonthlySales AS (  
    select trunc_timestamp(ohconfdate, 'MONTH') as Maand,  
           count(*) as Aantal  
    from ordrhdr group by trunc_timestamp(ohconfdate, 'MONTH')  
)  
SELECT q.*,  
       100 * (aantal - lag(aantal,1) over(order by Maand) / lag(aantal,1)  
over(order by Maand)) || '%' as growth  
FROM MonthlySales q ORDER BY Maand;
```

# Visualize the 80/20 principle in SQL

A social scientist in Italy, Vilfredo Pareto discovered that roughly 20% of the population owned 80% of the wealth. From there, he noticed that these proportions described many other aspects of society. This led him to propose the 80-20 rule. The Pareto chart, named in his honor, shows the large contribution from a small proportion of the population.

You'll notice the 80-20 principle in action across your business:

- a small proportion of customers generate the most customer support load
- a small proportion of customers contribute the most to the over all revenue – financial statements usually list this as a risk factor
- a small proportion of web pages generate the most pageviews
- a small proportion of adwords keywords generate the most clicks and conversions, etc.



# Frequency distribution using GROUP BY

```
select title,  
       count(*) as pageviews  
from visits  
where visits.dt > (current date - 14 days)  
-- just calculate for the last two weeks  
group by title  
order by pageviews desc
```

# Frequency distribution using GROUP BY

title	pageviews
Page 0002	354
Page 0004	156
Page 0003	156
Home page	134
...	

# Cumulative sum using Window functions

```
select title,  
       pageviews,  
       sum(pageview) over (order by pageviews desc) as  
       cumsum  
from (  
  // query as before  
) tbl;
```

# Visualize the 80/20 principle in SQL

```
with total_pageviews as (  
    select count(pageviews) as cnt  
    from visits  
    where visits.dt > (current date - 14 days))  
    select title,  
    pageviews,  
    sum(pageview) over (order by pageviews desc) *  
1.0 / total_pageviews.cnt as cumsum
```

# Visualize the 80/20 principle in SQL

```
from (  
    select title,  
           count(*) as pageviews  
    from visits  
    where visits.dt > (current date - 14 days)  
    group by title  
    order by pageviews desc )  
tbl;
```



# Ranking your best and worst customers

# Analyzing Recency, Frequency and Monetary value to index your best customers

**Recency-Frequency-Monetary** (RFM) analysis is a indexing technique that uses past purchase behavior to segment customers.

Given a table with purchase transaction data, we calculate a score based on how recently the customer purchased, how often they make purchases and how much they spend in dollars on average on each purchase. Using these scores, we can segment our customer list to:

- identify our most loyal customers by segmenting one-time buyers from customers with repeat purchases
- increase customer retention and lifetime value
- increase average order size, etc.

# Calculating RFM indices

```
select    customer_id,  
          max(order_date) as last_order_date,  
          count(*) as count_order,  
          avg(amount) as avg_amount  
from customer_orders  
group by customer_id;
```



# Calculating RFM indices

```
Select customer_id,  
       ntile(4) over (order by last_order_date) as rfm_recency,  
       ntile(4) over (order by count_order) as rfm_frequency,  
       ntile(4) over (order by avg_amount) as rfm_monetary  
from ( /* .. previous query .. */ )
```

# Calculating RFM indices

```
select customer_id,  
        rfm_recency*100 + rfm_frequency*10 +  
        rfm_monetary  
as rfm_combined  
from ( /* .. previous query .. */ )
```

# Understanding the RFM indices

		Monetary Value Quartiles			
Recency Quartiles	Frequency Quartiles	1	2	3	4
1st	1	lost			
	2				
	3				
	4				
2nd	1	at risk			
	2				
	3				
	4				
3rd	1	slipping away			
	2				
	3				
	4				
4th	1	active			
	2				
	3				
	4				

# Conclusion and takeaways

# Conclusions

- AI and machine learning integration
- SQL-based automation
- Cloud and hybrid deployment models
- Integration with modern development frameworks

# Key Takeaways

- Modern SQL capabilities dramatically improve productivity
- Performance optimization requires both SQL skill and system knowledge
- Moving logic to the database often improves maintainability
- Leverage the latest features for competitive advantage
- Focus on set-based processing over record-level access



**Thank You**

For more information contact:

Koen Decorte

[kdecorte@cdinvest.be](mailto:kdecorte@cdinvest.be)