Migrating a data warehouse from a relational database to a document store and lessons learned

A Thesis

submitted to the designated
by the General Assembly of Special Composition
of the Department of Computer Science and Engineering
Examination Committee

by

Ioannis Lazos

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
IN COMPUTER SCIENCE

WITH SPECIALIZATION
IN COMPUTER SYSTEMS

University of Ioannina
February 2020

Examination Committee:

- **Stergios Anastasiadis,** Associate Professor, Computer Science & Engineering Department, University of Ioannina (Supervisor)

- **Nikolaos Mamoulis**, Professor, Computer Science & Engineering Department, University of Ioannina

- **Christos Nomikos**, Assistant Professor, Computer Science & Engineering Department, University of Ioannina

Dedication

To my wife, Vaia.

ACKNOWLEDGMENTS

I would like to express my sincerest thanks and gratitude to my advisor Associate Prof. Stergios Anastasiadis for the valuable guidance, advice he has offered during the elaboration of this thesis. Our collaboration has been a pleasant and memorable experience that has helped me develop strong research skills as well as develop my critical thinking.

I would also like to thank my manager Evgeny Borodkin who made it easy for me to fulfill the prerequisite of attending the classes of this master's degree.

Finally, I would like to thank my parents, Elias and Eleni for always believing and supporting me.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Ioannis Lazos, M.Sc. in Computer Science, Department of Computer Science and Engineering, University of Ioannina, Greece, February 2020.
Migrating a data warehouse from a relational database to a document store and lessons learned.
Thesis Supervisor: Stergios Anastasiadis, Associate Professor.

Traditional database systems have been based on the relational model for storage. These are widely known as SQL databases named after the language they were queried by. In the last few years, however, non-relational databases have dramatically risen in popularity. These databases are commonly known as NoSQL databases, clearly marking them different from the traditional SQL databases. A big number of these databases are based on document stores which provides flexibility on schema creation and are claimed to perform better than SQL databases.

Medium to big sized companies use huge amounts of structured, semi structured, and unstructured corporate data. With the availability of cheap storage, this data is stored for reporting, decision taking and other various applications. Processing such vast amount of data requires speed and flexible schemas. Document stores claim to satisfy these requirements.

In this thesis we are migrating a medium sized company's SQL database to a document store, taking notes on all procedures needed and finally evaluating the performance of both databases on query execution times, using alternative hardware configurations.

We used two market leading products from the same vendor, Microsoft SQL Server and Microsoft Azure Cosmos. The reason we chose Microsoft Azure Cosmos DB document store is because it is also a cloud database solution, that provides all the tools needed to deploy a datastore to the cloud.

In the experimental evaluation we conducted, it has been shown that document store, does not perform better on all cases, compared to SQL database.

ΕΚΤΕΤΑΜΕΝΗ ΠΕΡΙΛΗΨΗ

Ιωάννης Λάζος. Μ.Δ.Ε. στην Πληροφορική, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Φεβρουάριος 2020.

Διδάγματα που αποκτήθηκαν μεταφέροντας μια αποθήκη δεδομένων από μια σχεσιακή βάση δεδομένων σε μια βάση δεδομένων εγγράφων.

Επιβλέποντας: Στέργιος Αναστασιάδης, Αναπληρωτής Καθηγητής

Τα παραδοσιακά συστήματα βάσεων δεδομένων βασίζονται στο σχεσιακό μοντέλο. Είναι ευρέως γνωστά ως βάσεις δεδομένων SQL από τη γλώσσα που χρησιμοποιείται για την διαχείρισή τους. Τα τελευταία χρόνια, ωστόσο, οι μη σχεσιακές βάσεις δεδομένων έχουν αυξηθεί δραματικά σε δημοτικότητα. Αυτές οι βάσεις δεδομένων είναι κοινώς γνωστές ως βάσεις δεδομένων NoSQL, σηματοδοτώντας σαφώς ότι είναι διαφορετικές από τις παραδοσιακές βάσεις δεδομένων SQL. Ένας μεγάλος αριθμός αυτών, χρησιμοποιεί για την αποθήκευση των δεδομένων έγγραφα (documents) τα οποία παρέχουν ευελιξία στη δημιουργία των σχημάτων της βάσης και θεωρείται πως επιτυγχάνουν καλύτερες επιδόσεις από τις βάσεις δεδομένων SQL.

Εταιρείες μεσαίου έως μεγάλου μεγέθους χρησιμοποιούν τεράστιες ποσότητες, δομημένων, ημιδομημένων και αδόμητων εταιρικών δεδομένων. Με την πληθώρα χαμηλού κόστους μέσων αποθήκευσης, τα δεδομένα αυτά αποθηκεύονται και χρησιμοποιούνται για τη δημιουργία αναφορών, τη λήψη αποφάσεων και πολλές άλλες εφαρμογές. Η επεξεργασία τόσο μεγάλου όγκου δεδομένων απαιτεί ταχύτητα και ευέλικτα σχήματα. Οι βάσεις δεδομένων εγγράφων θεωρείται ότι μπορούν να καλύψουν αυτές τις απαιτήσεις.

Σε αυτή τη διατριβή μεταφέρουμε τα δεδομένα μιας βάσης SQL μιας εταιρείας μεσαίου μεγέθους σε μια βάση δεδομένων εγγράφων, καταγράφοντας όλα τα βήματα που απαιτούνται και αξιολογώντας την απόδοση των δύο βάσεων δεδομένων κατά την εκτέλεση ερωτημάτων αναζήτησης, χρησιμοποιώντας διαφορετικές διαμορφώσεις υλικού.

Χρησιμοποιήσαμε δύο κορυφαία προϊόντα της αγοράς του ίδιου προμηθευτή, τον Microsoft SQL Server και το Microsoft Azure CosmosDB. Επιλέξαμε το Microsoft Azure CosmosDB σαν βάση δεδομένων εγγράφων, καθώς υποστηρίζει την τεχνολογία νέφους (cloud) και παρέχει όλα τα εργαλεία που απαιτούνται για την μεταφορά της βάσης στο νέφος.

Στην πειραματική αξιολόγηση που διεξαγάγουμε, παρατηρείται ότι οι βάσεις δεδομένων αποθήκευσης εγγράφων, δεν αποδίδουν καλύτερα σε όλες τις περιπτώσεις, σε σύγκριση με τις βάσεις δεδομένων SQL.

CHAPTER 1.

INTRODUCTION

As technology nowadays is tireless and evolves continuously every day, the amount of data is increasing and an application to handle a huge volume of data efficiently is needed, it is important to choose the right model of the database. Relational database model has a quite rigid schema that means that a schema must be designed in advance before data had been loaded and all attributes of the schema are uniform for all elements [1].

Non-relational databases do not use the RDBMS principles (Relational Data Base Management System) and do not store data in tables and have schema-less approach to data management. Non-relational databases do not require schema definition before inserting data nor changing the schema when data collection and management need to evolve. Instead, they use identification keys and data can be found based on the assigned keys. A big number of these are based on document stores which provides flexibility on schema creation and are claimed to perform better than SQL databases.

Medium to big sized companies use huge amounts of structured, semi structured, and unstructured corporate data. With the availability of cheap storage, this data is stored for reporting, decision taking and other various applications. Processing such vast amount of data requires speed and flexible schemas. Document store databases claim to satisfy these requirements.

1.1. Contribution

Our contribution in this thesis is a practical approach on the comparison of a document store data warehouse implementation with the corresponding relational one. We focus on query execution times for selecting records because these operations are mandatory for report generation and in some cases take too much time to complete. We do not compare architectural details such as supported data models, consistency, fault-tolerance, etc. We keep notes on the issues we observed on the data migration process from a relational to a non-relational store. We use and compare two market leading products from the same vendor, Microsoft SQL Server and Microsoft Azure Cosmos DB, which is also a cloud database solution that provides all the tools needed to move our database to the cloud. For practical reasons we used the Microsoft Azure Cosmos DB Emulator version, that lacks some database distribution options, but for our experiments this is not crucial, because our databases run on local machine resources.

The dataset used is part of a database that stores sales data of a medium sized company. Now SQL Server is used for reporting but in some cases, the query execution time exceeds 16 minutes. We migrated all data to an equivalent Azure Document DB and compared the execution times of the same queries. We had to overcome the lack of aggregations and joins between containers by creating a denormalized partitioned table.

1.2. Roadmap

The rest of this thesis is organized as follows. Chapter 2 reviews related work. Chapter 3 presents a comparison between relational and NoSQL databases. Chapter 4 has a detailed presentation of Microsoft SQL Server and its tools. Chapter 5 presents Microsoft Azure Cosmos DB. Chapter 6 presents the experimental framework of our tests. Chapter 7 presents the results of our tests and Chapter 8 concludes with a summary.

CHAPTER 2.

RELATED WORK

There is a lot of discussion and papers about advantages and disadvantages of relational and non-relational database systems. It is claimed that NoSQL databases have the speed and scalability advantage, however on the other side they have several drawbacks compared to traditional relational databases [2].

Litwin introduces the relational database design theory, including a discussion on keys, relationships, integrity rules and the often-dreaded "Normal Forms" [3].

Hecht and Jablonski evaluate the underlying techniques of NoSQL databases considering their applicability for certain requirements, comparing their data models, query possibilities, concurrency controls, partitioning and replication opportunities [4].

Davoudian et al. explain the design decisions of NoSQL stores with regards to the four non-orthogonal design principles of distributed database systems: data model, consistency model, data partitioning, and the CAP theorem. For each principle, the authors explain its available strategies and corresponding features, strengths, and drawbacks. They also exemplify various implementations of each strategy through a collection of representative academic and industrial NoSQL technologies. Finally, they disclose some existing challenges in developing effective NoSQL stores, which need attention of the research community, application designers, and architects [5].

Leavitt notes that NoSQL databases, even though they are fast for simple tasks, they are also time-consuming for complex operations. Besides, queries for complex operations can be hard to form [6]. In the same paper, he also notes that NoSQL is a technology that many organizations are yet to learn and there is a lack of support and management tools to help.

Bartholomew gives a tutorial introduction to the history of and differences between SQL and NoSQL databases [7].

Similarly, Indrawan-Santiago qualitatively compares relational databases to NoSQL databases, and concludes that NoSQL are likely to complement relational databases and enhance an organization's enhance database management capabilities [8].

Li and Manoharan [9] compare key-value stores implementations on NoSQL and SQL databases and although NoSQL databases are generally designed for optimized key-value stores and SQL databases are not, their findings suggest that not all NoSQL databases perform better than SQL databases.

CHAPTER 3.

RELATIONAL VS NOSQL DATABASES

SQL database or relational database is a collection of data items organized in formally described tables from which data can be accessed or reassembled in many ways. Relational database is a set of tables referred to as relation with data category described in columns in a way similar to spreadsheets. Each row contains a unique instance of data for the corresponding data category. While creating a relational database, possible values along with constrains are applied to the data. It is the relation between the tables that makes it a 'relation' table. They require few assumptions about how data will be extracted from the database. As a result, the same database can be viewed in many ways.

Mostly all the relational databases use the Structured Query Language (SQL) to access and modify the data stored in the database. Originally it was based upon relational calculus and relational algebra and is subdivided into elements such as clauses, predicates, queries and statements.

Some of the benefits of the database designed according to the relational model are [10]:

- Most of the information is stored in the database and not in the application, so the database is self-documenting.

- It is easy to add, update or delete data.
- It gives benefits of data summarization, retrieval and reporting.
- The database is structured in a tabular form with highly related tables.
- Also, any changes required to make in the schema of the database is quite simple.

SQL database systems designed with traditional ACID (atomicity, consistency, isolation, durability) guarantees and choose consistency over availability.

A NoSQL (originally referred to "non-SQL" or "non-relational") database provides a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases. The data structures used by NoSQL databases (e.g. key-value, wide column, graph, or document) are different from those used by default in relational databases, making some operations faster in NoSQL. Most NoSQL stores lack true ACID transactions, although a few databases have made them central to their designs and are designed around the BASE philosophy (explained below) choosing availability over consistency.

3.1. ACID, BASE and CAP Theorem

ACID refers to the following four properties of transactions:

- Atomicity: stands for ‘everything or nothing’. If any part of the transaction is left incomplete, then the entire transaction is considered failed.
- Consistency: ensures that a database before and after any transaction is stable at a valid state.
- Isolation: ensures that multiple transactions executing at the same time do not affect each other's execution. Which requires from the concurrent transactions to be serialized.
- Durability: ensures that once a transaction has been committed it will remain in the same state i.e. stored permanently even if there are some errors, or even if the system crash or power loss occurs.

BASE consists of three principles:

- *Basically Available*: Guarantees the availability of the data. There will be a response to any request (can be failure too).
- *Soft state*: The state of the system could change over time.
- *Eventual consistency*: The system will eventually become consistent once it stops receiving input.

The CAP theorem states that it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees [11]:

- *Consistency*: Every read receives the most recent write or an error
- *Availability*: Every request receives a (non-error) response, without the guarantee that it contains the most recent write
- *Partition tolerance*: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

When a network partition failure occurs, we should decide to

- Cancel the operation and thus decrease the availability but ensure consistency, or
- Proceed with the operation and thus provide availability but risk inconsistency

The CAP theorem implies that in the presence of a network partition, one must choose between consistency and availability. Note that consistency as defined in the CAP theorem is quite different from the consistency guaranteed in ACID database transactions.

3.2. Comparing Relational and NoSQL databases

The comparison of both technologies results in the following [12]:

- *Transaction reliability*: Relational databases guarantee very high transaction reliability because they fully support ACID unlike the NoSQL databases that range from BASE to ACID.
- *Data Model*: Relational databases are based on the concepts of sets in mathematics. All the data is represented as mathematical n-ary relations, with an n-ary relation being a subset of the Cartesian product of N domains. The data inside the database is

represented as tuples and grouped into relations. The relation (represented by table) contains a set of Tuples (rows) which is a sequence of attributes, named columns in the relation table. The type of an attribute is identified by a set of values that have a common meaning. This data model is very specific and well organized. Columns are described by well-defined schema. The set of related data stored in rows has the same structure. NoSQL databases apply several modelling techniques like key value stores, graph, and document data model. The main distinguishing feature of the NoSQL data model is that it does not use the table as storage structure of the data; instead, it is schema-less and very efficient in handling the unstructured data like word or pdf files, images, and video files, etc.

- *Scalability*: Scalability is the greatest challenge faced by relational databases. The vertical scalability dependence on improving hardware is very costly and impractical for the reason of hardware limitation. Another type of scalability is the horizontal scalability according to which more commodity nodes or system unites are added. When the relational databases were created it was not a design goal to support the web applications that spread among many servers and serve millions of users in the way that is happening nowadays. As a result, the relational model does not support horizontal scalability very well. NoSQL databases depend on the horizontal scalability.

- *Cloud*: The relational databases are not well suited for cloud environments because they do not support full content data search and are hard to scale beyond a limit. However, NoSQL databases are the best solution for cloud databases because all the characteristics that define the NoSQL databases are very desirable for cloud databases. The cloud databases are not ACID compliant, and they provide improved availability, scalability, performance and flexibility but they deal with unstructured, semi-structured data or structured data.

- *Big data handling*: Big data handling is a major issue in relational databases. The solution is the scalability and data distribution which take two forms, vertical or horizontal. Accordingly, the data must be portioned into multiple servers which raise an issue of complexity in the joining for these data and the performance related to these operations. NoSQL databases are designed to handle big data, so they implemented methods to improve the performance of storing and retrieving data.

- *Data warehouse*: The relational databases are used for data warehousing resulting from gathering data from many sources and over time. As a result, the data warehouses end up with an increased size of stored data which raises problems like performance degradation when doing an OLAP, data mining or statistical process. In the other hand NoSQL databases are not designed to serve data warehouse applications because the designers focused on high performance, scalability, availability and storing big data in order to address the increasing size of stored data.
- *Complexity*: Complexity in relational databases rises because the user must convert data into tables. When the data does not fit into those tables, the structure of the database could be quite complex, difficult, and slow working with. Instead, the NoSQL databases have the capabilities to store unstructured, semi-structured or structured data.
- *Crash Recovery*: Relational databases manage crash recoveries via a recovery manager that is responsible for ensuring transaction atomicity and durability by using log files (e.g., ARIES algorithm [13]). On the other hand, crash recovery in NoSQL databases depends on replication as backup to recover from the crash, or alternative mechanisms such as the use of a journal file in MongoDB.
- *Security*: Relational databases have adopted very secure mechanisms to provide the security services although they face many security threats such as SQL injection, Cross-Site Scripting, Root Kits, Weak communication protocols etc. Many studies today investigate and try to solve these vulnerabilities. NoSQL databases came mainly to solve the problem of storing big data and to increase the performance of big data sets. This decreases the security level of the database, although many of the current NoSQL products provide improved database security.

3.3. Disadvantages of Relational and NoSQL databases

Relational Databases though conventionally accepted suffer from several drawbacks:

- Relational databases do not support high scalability. Better hardware can be employed up to a point, beyond which the database must be distributed.

- One major disadvantage is that the data is stored in relational databases in form of tables, this structure can give rise to high complexity when the data cannot be easily encapsulated in a table.
- Much of the features provided by relational databases are not used but they simply add to the cost as well as the complexity of the database.
- Relational Databases use SQL, which is featured to work on structured data, but SQL can be highly complex when working with unstructured data.
- When the amount of data turns huge, the database has to be partitioned across multiple servers, this partitioning poses several problems because joining tables in distributed servers is not an easy task.

Apart from various advantages such as high throughput, horizontal scalability, avoidance of expensive object relational mapping, non-relational databases have some shortcomings:

- Most of the non-relational databases are available as open source software. Although this is valuable, it compromises the reliability as nobody is responsible in times of failures.
- Many of the Non-Relational databases are disk based which implement buffer pool and multithreading hence require buffer management and locking features which increase the performance overhead.
- Many non-relational databases provide BASE properties and sacrifice conventional ACID properties as a step to increase performance. This could mean than non-relational databases compromise the consistency within the database.
- Because of lacking the ACID properties, the degree of reliability provided by non-relational databases is lower than what is provided by the relational databases. Developers have to code the ACID constraints easily provided in relational databases.

3.4. Document Store NoSQL databases

The Document store databases are NoSQL databases whose records consist of documents [14]. They store unstructured (text) or semi-structured (XML) documents which are usually hierarchical in nature. Each document consists of a set of keys and values which are almost

same as in the key-value databases. Each database residing in a document store points to its fields using pointers with the technique of hashing. Document store databases are schema free and are not fixed in nature.

3.4.1. Characteristics of Document Store Database

- A collection is a group of documents. The documents within a collection are usually related to the same subject, such as employees, products, and so on.
- A document is a set of ordered key-value pairs, where key is a string used to reference a particular value, and value can be either a string or a document.
- There are number of varieties to organize data that is collections, tags, non-visible metadata and directory hierarchies.
- JSON (JavaScript Object Notation), BSON (Binary JSON), and XML (eXtensible Markup Language) are formats commonly used to define documents.
- Embedded documents are documents within documents. An embedded document enables users to store related data in a single document to improve database performance.
- Document store databases do not require users to formally specify the structure of documents prior to adding documents to a collection. Therefore, document databases are called schema-less. Application programs should verify rules about the structure of a document.

3.5. Introduction to Data Warehouses

A data warehouse is a type of data management system that is designed to enable and support business intelligence (BI) activities, especially analytics. Data warehouses are solely intended to perform queries and analysis and often contain large amounts of historical data. The data within a data warehouse is usually derived from a wide range of sources such as application log files and transaction applications.

A data warehouse centralizes and consolidates large amounts of data from multiple sources. Its analytical capabilities allow organizations to derive valuable business insights from their data to improve decision-making. Over time, it builds a historical record that can

be invaluable to data scientists and business analysts. Because of these capabilities, a data warehouse can be considered an organization's "single source of truth."

A typical data warehouse often includes the following elements:

- A database to store and manages data
- An extraction, loading, and transformation (ELT) solution for preparing the data for analysis
- Statistical analysis, reporting, and data mining capabilities
- Client analysis tools for visualizing and presenting data to business users
- Other, more sophisticated analytical applications that generate actionable information by applying machine learning and artificial intelligence (AI) algorithms

3.5.1. Benefits of a Data Warehouse

Data warehouses offer the overarching and unique benefit of allowing organizations to analyze large amounts of variant data and extract significant value from it, as well as to keep a historical record.

Four unique characteristics (described by computer scientist William Inmon, who is considered the father of the data warehouse) allow data warehouses to deliver this overarching benefit. According to this definition, data warehouses are

- *Subject-oriented.* They can analyze data about a particular subject or functional area (such as sales).
- *Integrated.* Data warehouses create consistency among different data types from disparate sources.
- *Nonvolatile.* Once data is in a data warehouse, it's stable and doesn't change.
- *Time-variant.* Data warehouse analysis looks at change over time.

A well-designed data warehouse will perform queries very quickly, deliver high data throughput, and provide enough flexibility for end users to "slice and dice" or reduce the volume of data for closer examination to meet a variety of demands—whether at a high level or at a very fine, detailed level. The data warehouse serves as the functional foundation for middleware BI environments that provide end users with reports, dashboards, and other interfaces.

3.5.2. Data Warehouse Architecture

The architecture of a data warehouse is determined by the organization's specific needs. Common architectures include

- *Simple.* All data warehouses share a basic design in which metadata, summary data, and raw data are stored within the central repository of the warehouse. The repository is fed by data sources on one end and accessed by end users for analysis, reporting, and mining on the other end.
- *Simple with a staging area.* Operational data must be cleaned and processed before being put in the warehouse. Although this can be done programmatically, many data warehouses add a staging area for data before it enters the warehouse, to simplify data preparation.
- *Hub and spoke.* Adding data marts between the central repository and end users allows an organization to customize its data warehouse to serve various lines of business. When the data is ready for use, it is moved to the appropriate data mart.
- *Sandboxes.* Sandboxes are private, secure, safe areas that allow companies to quickly and informally explore new datasets or ways of analyzing data without having to conform to or comply with the formal rules and protocol of the data warehouse.

3.5.3. Designing a Data Warehouse

When an organization sets out to design a data warehouse, it must begin by defining its specific business requirements, agreeing on the scope, and drafting a conceptual design. The organization can then create both the logical and physical design for the data warehouse. The logical design involves the relationships between the objects, and the physical design involves the best way to store and retrieve the objects. The physical design also incorporates transportation, backup, and recovery processes.

Any data warehouse design must address the following:

- Specific data content
- Relationships within and between groups of data
- The systems environment that will support the data warehouse
- The types of data transformations required

- Data refresh frequency

A primary factor in the design is the needs of the end users. Most end users are interested in performing analysis and looking at data in aggregate, instead of as individual transactions. However, often end users don't really know what they want until a specific need arises. Thus, the planning process should include enough exploration to anticipate needs. Finally, the data warehouse design should allow room for expansion and evolution to keep pace with the evolving needs of end users.

CHAPTER 4.

MICROSOFT SQL SERVER

The SQL Server is a relational database management system, or RDBMS, developed and marketed by Microsoft. SQL Server is a client-server architecture. The SQL Server process starts with the client application sending a request. The SQL Server accepts, processes and replies to the request with processed data.

Similarly to other RDBMS software, the SQL Server is built on top of SQL, a standard programming language for interacting with the relational databases. The SQL Server is tied to Transact-SQL, or T-SQL, the Microsoft's implementation of SQL that adds a set of proprietary programming constructs.

There are three major components in SQL Server Architecture [15]:

- Protocol Layer – SQL Server Network Interface (SNI)
- Relational Engine
- Storage Engine
- SQLOS

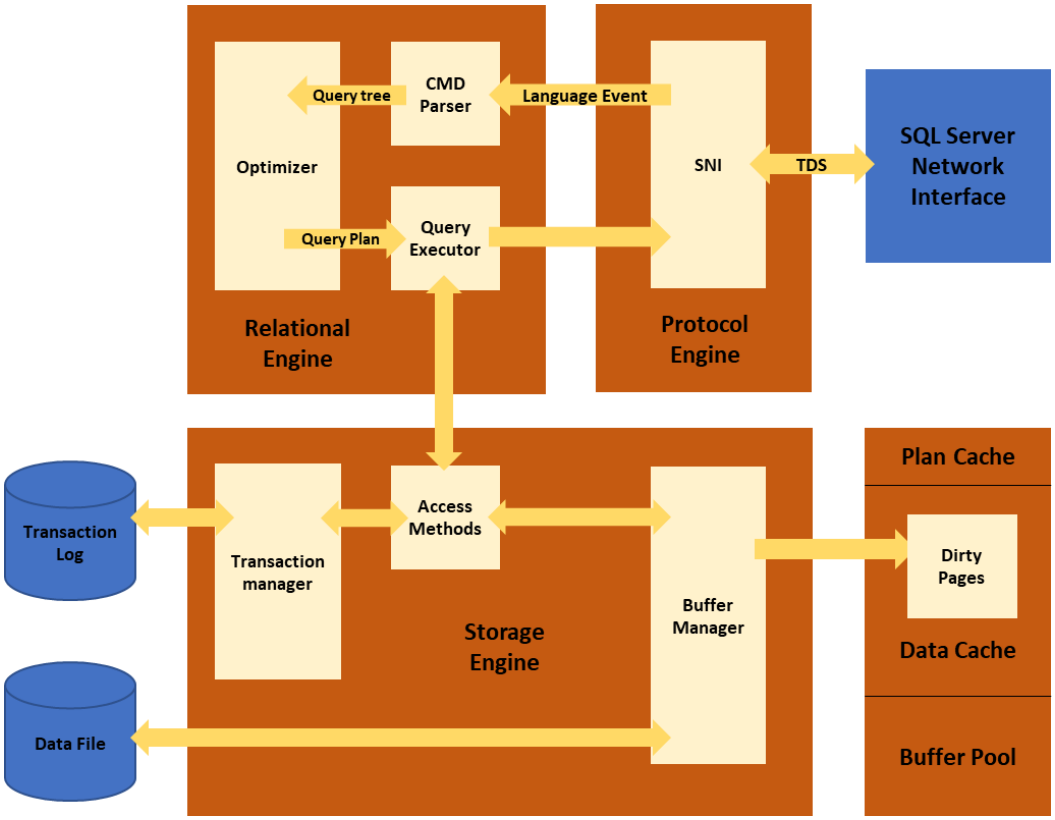

Figure 4.1 Diagram of the SQL Server architecture

4.1. Protocol Layer- SQL Server Network Interface

The SQL Server Network Interface (SNI) is a protocol layer that establishes the network connection between the client and the server. It consists of a set of APIs that are used by both the database engine and the SQL Server Native Client (i.e. SNAC).

The SQL Server supports the following protocols:

- Shared memory
- TCP/IP
- Named Pipes
- Virtual Interface Adapter (VIA)

Regardless of the network protocol used, once the connection is established, SNI creates a secure connection to a Tabular Data Stream (TDS) endpoint.

4.1.1. Shared Memory

Simple and fast, shared memory is the default protocol used to connect from a client running on the same computer as SQL Server. It can only be used locally, has no configurable properties, and is always tried first when connecting from the local machine. The limitation is that the client applications must reside on the same machine with the installed SQL Server.

4.1.2. TCP/IP

TCP/IP is the popular protocol widely used throughout the industry today. It communicates across interconnected networks and is a standard for routing network traffics and offers advanced security features. It enables you to connect to the SQL Server by specifying an IP address and a port number. Typically, this happens automatically when somebody specifies an instance to connect to. The internal name resolution system resolves the hostname part of the instance name to an IP address, and either you connect to the default TCP port number 1433 for default instances or the SQL Browser service will find the right port for a named instance using UDP port 1434.

4.1.3. Names Pipes

This protocol can be used when the application and the SQL Server reside on a local area network. A part of the memory is used by one process to pass information to another process, so that the output of one is the input of the other. The second process can be local (on the same computer as the first) or remote (on a networked computer). TCP/IP and Named Pipes are comparable protocols in the architectures in which they can be used. Named Pipes was developed for local area networks (LANs) but it can be inefficient across slower networks such as wide area networks (WANs). To use Named Pipes, you first need to enable it in SQL Server Configuration Manager (if you'll be connecting remotely) and then create a SQL Server alias, which connects to the server using Named Pipes as the protocol. Named Pipes uses TCP port 445, to ensure that the port is open on any firewalls between the two computers, including the Windows Firewall.

4.1.4. Virtual Interface Adapter

The Virtual Interface Adapter protocol enables high-performance communications between two systems. It requires specialized hardware at both ends and a dedicated connection. The VIA protocol should be enabled in the SQL Server Configuration Manager before it can be

used. This protocol has been deprecated and will no longer be available in the future versions of SQL Server.

4.1.5. Tabular Data Stream (TDS) Protocol

The Tabular Data Stream (TDS) Protocol is an application-level protocol used for the transfer of requests and responses between clients and database server systems. In such systems, the client will typically establish a long-lived connection with the server. Once the connection is established using a transport-level protocol, TDS messages are used to communicate between the client and the server. A database server can also act as the client if needed, in which case a separate TDS connection must be established. Note that the TDS session is directly tied to the transport-level session, meaning that a TDS session is established when the transport-level connection is established, and the server receives a request to establish a TDS connection. It persists until the transport-level connection is terminated (for example, when a TCP socket is closed). In addition, TDS does not make any assumption about the transport protocol used, but it does assume the transport protocol supports reliable, in-order delivery of the data.

TDS includes facilities for authentication and identification, channel encryption negotiation, issuing of SQL batches, stored procedure calls, returning data, and transaction manager requests. Returned data is self-describing and record oriented. The data streams describe the names, types and optional descriptions of the rows being returned.

4.2. Relational Engine

The Relational Engine is also known as the Query Processor. It has the SQL Server components that determine what exactly a query needs to do and how it can be done best. It is responsible for the execution of user queries by requesting data from the storage engine and processing the results that are returned.

As depicted in *Figure 4.1* there are 3 major components of the Relational Engine:

- CMD Parser
- Optimizer
- Query Executor

4.2.1. CMD Parser

The data once received from the Protocol Layer is passed to Relational Engine. The CMD Parser is the first component of Relational Engine to receive the query data. The main job of CMD Parser is to check the query for syntactic and semantic error. Finally, it generates a Query Tree. Note that, all the different trees have the same desired output. Each node in the tree represents a logical operation that the query must perform, such as reading a table, or performing an inner join. This logical tree is then used to run the query optimization process.

4.2.2. Optimizer

The work of the optimizer is to create an execution plan, using the logical tree, for the user's query. This is the plan that will determine how the user query will be executed.

Note that not all queries are optimized. Optimization is done for DML (Data Modification Language) commands like SELECT, INSERT, DELETE, and UPDATE. Such queries are first marked then send to the optimizer. DDL commands like CREATE and ALTER are not optimized, but they are instead compiled into an internal form. The query cost is calculated based on factors like CPU usage, memory usage, and input/ output needs. Optimizer's role is to find the cheapest, not the best, cost-effective execution plan.

4.2.3. Query Executor

The Query Executor calls the Access Methods to provide an execution plan for the data fetching logic. Once the data is received from the Storage Engine, the result gets published to the Protocol layer. Finally, the data is sent to the end user.

4.2.4. Processing a SELECT Statement

The basic steps that SQL Server uses to process a single SELECT statement include the following [16]:

- The parser scans the SELECT statement and breaks it into logical units such as keywords, expressions, operators, and identifiers.
- A Query Tree, sometimes referred to as a sequence tree, is built describing the logical steps needed to transform the source data into the format required by the result set.
- The Query Optimizer analyzes different ways the source tables can be accessed. It then selects the series of steps that returns the fastest results with fewer resources. The

Query Tree is updated to record this exact series of steps. The final, optimized version of the query tree is called the execution plan.

- The Relational Engine starts executing the execution plan. As the steps that require data from the base tables are processed, the Relational Engine requests from the Storage Engine to pass up data from the requested row sets.
- The Relational Engine processes the data returned from the Storage Engine into the format defined for the result set and returns the result set to the client.

4.3. Storage Engine

The work of the Storage Engine is to store the data in a storage device such as a Hard Disk Drive, a Solid State Drive or a Storage Area Network and retrieve the data when needed. Data is physically stored in Data Files [17], in the form of *data pages*, with each data page having a size of 8KB, forming the smallest storage unit in SQL Server. These data pages are logically grouped to form extends. No object is assigned a page in SQL Server.

The maintenance of the object is done via extents. The page has a section called the Page Header with a size of 96 bytes, carrying the metadata information about the page like the Page Type, Page Number, Size of Used Space, Size of Free Space, and Pointer to the next page and previous page, etc.

Every database contains one *Primary* file, which stores all the important data related to tables, views, triggers, etc. The extension of the file is usually .mdf. A database may or may not contain multiple *Secondary* files, which contain user-specific data. The extension is usually .ndf. Finally, a database contains *Log* files, also known as Write ahead logs, with an extension of .ldf, which are used for transaction management. They are used to recover from any unwanted instances and perform important task of rollback from uncommitted transactions.

The Storage Engine has the following main components:

- Access Methods
- Buffer Manager
- Transaction Manager
- Log Manager

- Lock Manager

4.3.1. Access Methods

It acts as an interface between the query executor and Buffer Manager/Transaction Logs. The Access Method itself does not do any execution. The first action is to determine whether the query is a Select Statement (DDL) or a Non-Select Statement (DDL & DML). Depending upon the result, the Access Method takes the following steps:

- If the query is DDL, SELECT statement, the query is passed to the Buffer Manager for further processing.
- And if query is DDL, NON-SELECT statement, the query is passed to the Transaction Manager. This mostly includes the UPDATE statement.

4.3.2. Buffer Manager

The Buffer Manager provides access to the data required and manages core functions for modules below:

- Plan Cache:
  Existing Query plan: The Buffer Manager checks if the execution plan is stored in the Plan Cache. If Yes, then it uses the query Plan Cache and its associated data cache.
  First time Cache plan: If the query execution plan is being run for first-time and it is complex, it makes sense to store it in the Plan Cache. This will ensure faster availability the next time that the SQL Server gets the same query.
- Data Parsing:
  Buffer cache: The Buffer Manager looks for data in the data cache. If present, then this data is used by the Query Executor. This improves the performance as the number of I/O operations is reduced when fetching data from the data cache as compared to fetching data from data storage.
  Data storage: If the data is not present in the Buffer Manager then the required data is searched in Data Storage and also stored in the data cache for future use.
- Dirty Page:
  Page which stores the data. The data is stored by the processing logic of the Transaction Manager.

4.3.3. Log Manager

The Log Manager keeps track of all the updates done in the system via logs in Transaction Logs. Logs have Logs Sequence Number with the Transaction ID and Data Modification Record. This is used for keeping track of Transaction Committed and Transaction Rollback.

4.3.4. Lock Manager

During a transaction, the associated data in Data Storage is in the Lock state. This process is handled by the Lock Manager. This process ensures data consistency and isolation (also known as ACID properties).

4.3.5. Execution Process

- The Log Manager start logging and the Lock Manager locks the associated data.
- A copy of the data is maintained in the Buffer cache.
- A copy of the data that should be updated is maintained in the Log buffer and all the events update the data in the Data buffer.
- The pages that store the data are also known as Dirty Pages.
- Checkpoint and Write-Ahead Logging: A data page can have more than one logical write made before it is physically written to disk. For each logical write, a transaction log record is inserted in the log cache that records the modification. The log records must be written to disk before the associated Dirty Page is removed from the Buffer cache and written to disk. The checkpoint process periodically scans the Buffer cache for buffers with pages from a specified database and writes all Dirty Pages to disk. Checkpoints save time during a later recovery by creating a point at which all dirty pages are guaranteed to have been written to disk.
- Lazy Writer: The Dirty Page can remain in memory. When the SQL Server observes a huge load and the buffer memory is needed for a new transaction, it frees up Dirty Pages from the cache. It operates with an LRU (Least Recently Used) Algorithm for cleaning pages from buffer pool to disk.

4.4. SQLOS

SQLOS is the SQL Server application layer where all operating system resources are managed [18]. It takes care of memory and buffer management, scheduling, resource governance, exception handling, extended events and IO. It calls the operating system on behalf of other SQL layers, and in cases such as scheduling functions as an operating system to efficiently schedule SQL engine resources. The SQLOS layer provides an API for other layers to call when they require operating resources. The API allows programmers in the other layers to write code without needing to optimize for specifics of the underlying machine architecture.

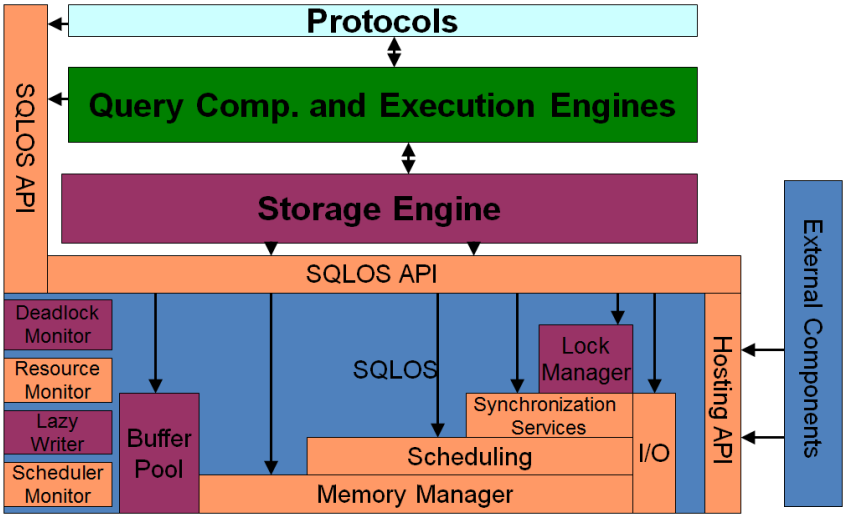

Figure 4.2 SQLOS overview diagram

SQLOS performs the following critical functions for SQL Server [19]:

- *Scheduler and IO completion*. The SQLOS is responsible for scheduling threads for CPU consumption. Most threads in SQL Server are run in cooperative mode, which means the thread is responsible for yielding so that other threads can obtain CPU time. Most IO is asynchronous. The SQLOS is responsible for signaling threads when IO is completed.
- *Synchronization primitives:* The SQL Server is a multi-threaded application, so SQLOS is responsible for managing thread synchronizations.
- *Memory management*: Different components within the SQL Server, e.g., Plan Cache, Common Language Runtime, Lock Manager etc. request memory from the SQLOS.

Therefore, the SQLOS can control how much memory a component within SQL Server is consuming.

- Deadlock detection and management.
- Exception handling framework.
- *Hosting services for external components* such as CLR (Common Language Runtime) and MDAC (Microsoft Data Access Components). The SQL Server will run threads that are associated with external component in preemptive mode. Preemptive mode allows the SQLOS to prevents runaway threads (threads which will not yield and allow other threads to get CPU execution). Also, the SQLOS can keep track of the memory these external components consume. For example, for CLR the SQLOS can invoke garbage collection if the CLR process is taking up too much memory.

4.5. SQL Server Tools

For data management, SQL Server includes the SQL Server Integration Services (SSIS), the SQL Server Data Quality Services, and the SQL Server Master Data Services. To develop databases, the SQL Server provides SQL Server Data tools; and to manage, deploy, and monitor databases the SQL Server has the SQL Server Management Studio (SSMS).

4.5.1. SQL Server Integration Services (SSIS)

Microsoft Integration Services is a platform for building enterprise-level data integration and data transformations solutions. Integration Services is used to solve complex business problems by copying or downloading files, loading data warehouses, cleansing and mining data, and managing SQL Server objects and data. Integration Services can extract and transform data from a wide variety of sources such as XML data files, flat files, and relational data sources, and then load the data into one or more destinations.

The Integration Services includes a rich set of built-in tasks and transformations, graphical tools for building packages, and the Integration Services Catalog database, to store, run, and manage packages.

The Graphical Integration Services tools are used to create solutions without writing a single line of code. The extensive Integration Services object model can be programmed to create packages programmatically and code custom tasks and other package objects.

4.5.2. SQL Server Management Studio (SSMS)

The SQL Server Management Studio (SSMS) is an integrated environment for managing any SQL infrastructure. SSMS is used to access, configure, manage, administer, and develop all components of SQL Server, Azure SQL Database, and SQL Data Warehouse. The SSMS provides a single comprehensive utility that combines a broad group of graphical tools with several script editors to provide access to SQL Server for developers and database administrators of all skill levels.

CHAPTER 5.

MICROSOFT AZURE COSMOS DB

Azure Cosmos DB is Microsoft's globally distributed, horizontally partitioned, multi-model database service. The service is designed to allow customers to elastically (and independently) scale throughput and storage across any number of geographical regions. Azure Cosmos DB offers guaranteed low latency at the 99th percentile, 99.99% high availability, predictable throughput, and multiple well-defined consistency models. As a cloud service, it is designed and engineered with multi-tenancy and global distribution in mind. It is schema-agnostic, horizontally scalable and generally classified as a NoSQL database [20].

5.1. Features

Key features of Azure Cosmos DB are the following:

- *Globally Distributed*: With Azure Cosmos DB, the data can be replicated globally by adding Azure regions with just one click.

- *Linearly Scalable*: Linear Scalability is the ability to handle the increased load by adding more servers to the cluster. Cosmos DB can be scaled horizontally to support hundreds of millions of transactions per second for reads and writes.
- *Schema-Agnostic Indexing*: Cosmos DB's database engine is schema-agnostic, and this enables automatic indexing of the data. Cosmos DB automatically indexes all the data without requiring schema and index management.
- *Multi-Model*: Cosmos DB is a multi-model database, i.e., it can be used for storing data in Key-value Pair, Document-based, Graph-based, Column Family-based databases. Irrespective of which model is chosen for data persistence, global distribution, provisioning throughput, horizontal partitioning, and automatic indexing capabilities are the same.
- *Multi-API and Multi-Language Support*: Microsoft has released SDKs for multiple programming languages (including Java, .NET, Python, Node.js, JavaScript, etc. Cosmos DB has Multi API support, i.e., SQL API, Cosmos DB Table API, MongoDB API, Graph API, Cassandra API, and Gremlin API).
- *Multi-Consistency Support*: Cosmos DB supports five consistency levels, i.e., Eventual, Prefix, Session, Bounded and Strong. Multi Consistency is discussed in detail later in the article.
- *Indexes Data Automatically*: Cosmos DB indexes data automatically on ALL fields in all documents by default without the need for secondary indexes, but someone can still create custom indexes. Azure's automatic indexing capability indexes every property of every record without having to define schemas and indices upfront. This capability works across every data model.
- *High Availability*: Cosmos DB provides 99.999% availability for both reads and writes for multi-region accounts with multi-region writes. Cosmos DB provides 99.99% availability for both reads and writes for single-region accounts. Cosmos DB automatically fails over, if there is a regional disaster. The application may also programmatically failover if there is such a case.
- *Guaranteed Low Latency*: Azure Cosmos DB always guarantees 10 milliseconds latency at the 99th percentile for reads and writes for all consistency levels. Data can be geographically distributed to any number of Azure regions, so the stored data can be

available nearest to the customers, which reduces the possible latency in retrieving the data.

- *Multi-Master Support*: Cosmos DB supports multi-master operation, which means the writes, in addition to reads, can be scaled elastically across any number of Azure regions across the world. With the multi-master feature, somebody can choose that all data servers act as write servers. To enable the multi-master feature for applications, you need to enable multi-region writes. Follow these instructions to configure the multi-master feature.

5.2. Data Model

Internally, Cosmos DB stores "items" in "containers", with these two concepts being interpreted differently depending on the API used. Containers are grouped in "databases", which are analogous to namespaces above containers. Containers are schema-agnostic, which means that no schema is enforced when adding items. Containers can also enforce unique key constraints to ensure data integrity.

After creating an Azure Cosmos DB account under the Azure subscription, one can manage data in their account by creating databases, containers, and items. The following image shows the hierarchy of different entities in an Azure Cosmos DB account:

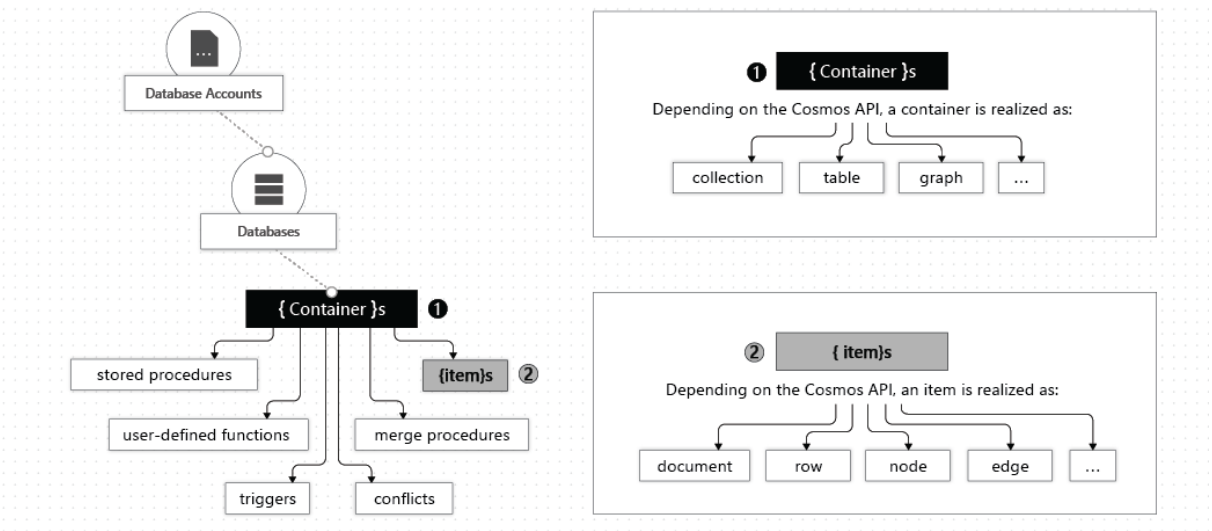

Figure 5.1 Azure Cosmos DB account entities

- *Azure Cosmos databases*: Somebody can create one or multiple Azure Cosmos databases under a simple account. A database is analogous to a namespace. A database is the unit of management for a set of Azure Cosmos containers.
- *Azure Cosmos containers*: An Azure Cosmos container is the unit of scalability both for provisioned throughput and storage. A container is horizontally partitioned and then replicated across multiple regions. Items added to the container and the throughput provisioned on it are automatically distributed across a set of logical partitions based on the partition key. An Azure Cosmos container can scale elastically, whether containers created by using dedicated or shared provisioned throughput modes. An Azure Cosmos container is a schema-agnostic container of items. Items in a container can have arbitrary schemas. For example, an item that represents a person and an item that represents an automobile can be placed in the same container. By default, items added to a container are automatically indexed without requiring explicit index or schema management. Indexing behavior is customized behavior by configuring the indexing policy on a container.

The Time to Live (TTL) can be set on selected items in an Azure Cosmos container or for the entire container to gracefully purge those items from the system. Azure Cosmos DB automatically deletes the items when they expire. It also guarantees that a query performed on the container does not return the expired items within a fixed bound. Change feed is used to subscribe to the operations log that is managed for each logical partition of your container. Change feed provides the log of all the updates performed on the container, along with the before and after images of the items. Retention duration can be configured for the change feed by using the change feed policy on the container. One can register stored procedures, triggers, user-defined functions (UDFs), and merge procedures for an Azure Cosmos container. A unique key constraint can be configured on an Azure Cosmos container. By creating a unique key policy, the uniqueness of one or more values per logical partition key is ensured. If a container is created by using a unique key policy, no new or updated items with values that duplicate the values specified by the unique key constraint can be created.

- *Azure Cosmos items*: Depending on which API you use, an Azure Cosmos item can represent either a document in a collection, a row in a table, or a node or edge in a graph.

5.3. Multi-Model Capabilities

Cosmos DB supports the following data models:

- Key-Value,
- Column-Family,
- Document,
- Graph database models.

Regardless of which data model is used, core content model of the Cosmos DB engine is based on ARS (Atom Record Sequence, which defines persistent layer for key-value pairs) and projects different data models in different APIs. Cosmos DB exposes the data in JSON format.

Depending on the type of application, an appropriate data model is used. If an existing MongoDB or Cassandra Database based application is migrated to Cosmos DB, it requires minimal to no code changes in application code. If an application requires a relationship between entities, then a graph data model works better. Each of the supported data models has an API to integrate with Cosmos DB. The data model is determined based on the selected API for the database. The most relevant API for an application is chosen at the container creation time (i.e. “database instance”). Based on the chosen API, the desired data model (graph, key-value, document or column) is projected on the underlying data store. Because of the way the data is stored and retrieved, only one API can be used against a container; use of multiple APIs is not possible.

5.3.1. Key-Value Pair Data Model (Table API):

In this data model, each entity consists of a key and a value pair. The value itself can be a set of key-value pairs. This is very similar to a table in a relational database where each row has the same set of columns. The Key-Value pair solution is supported by the standard Azure table API.

5.3.2. Column-Family Data Model:

This data model supports the Cassandra API. Existing Cassandra implementations can be easily and quickly moved to Cosmos DB and use the column family format, which is used in

Cassandra. The Column Family data model is similar to the key-value data model except that the items in the data model adhere to the defined schema.

5.3.3. Document Data Model:

This model supports the SQL API and the MongoDB API; both APIs give the document data model. These two APIs are different, though they are similar in data modeling. The SQL API allows building transactional stored procedures, triggers, and user-defined functions. The SQL API stores entities through JSON in a hierarchical key-value document. The MongoDB API stores in BSON (Binary encoded version of JSON, which extends JSON with additional data types and multi-language support).

The SQL API works with Document DB protocols, whereas MongoDB API works with MongoDB APIs. Both these APIs allow interacting with the documents in the database. The max document size in Cosmos DB is 2 MB unlike the max document size of 16MB in MongoDB.

5.3.4. Graph Data Model:

A graph database implements a collection of interconnected entities and relationships. Microsoft has chosen to use Gremlin API from the Apache Tinkerpop open source project. The Gremlin API allows somebody to interact with a Graph database globally scaled and provides a graph traversal language, which enables to efficiently query across many relationships exist in a graph database.

5.3.5. Atom Record Sequence (ARS)

The Azure Cosmos DB natively supports multiple data models including documents, key-value, graph, and column-family. The core content-model of Cosmos DB's database engine is based on atom-record-sequence (ARS). Atoms consist of a small set of primitive types like string, bool, and number. Records are structures composed of these types. Sequences are arrays consisting of atoms, records, or sequences. The database engine can efficiently translate and project different data models onto the ARS-based data model. The core data model of Cosmos DB is natively accessible from dynamically-typed programming languages and can be exposed as-is in JSON.

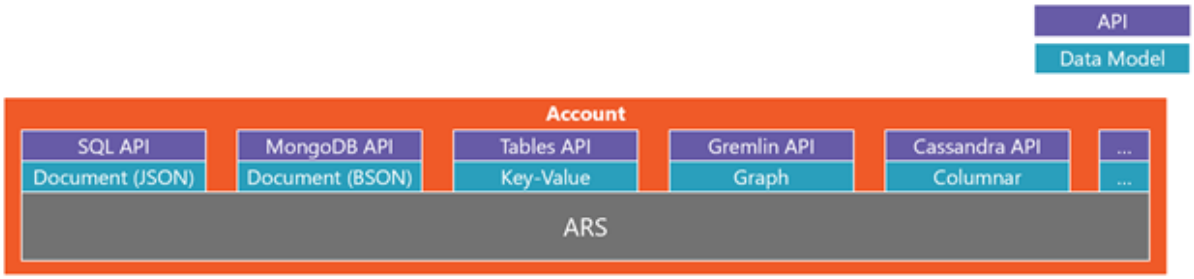

Figure 5.2 Azure Cosmos DB multi model capabilities

5.4. SQL API

The SQL API lets clients create, update and delete containers and items. The items can be queried with a read-only, JSON-friendly SQL dialect. As Cosmos DB embeds a JavaScript engine, the SQL API also enables:

- *Stored procedures*: Functions that bundle an arbitrarily complex set of operations and logic into an ACID-compliant transaction. They are isolated from changes made while the stored procedure is executing and either all write operations succeed or they all fail, leaving the database in a consistent state. Stored procedures are executed in a single partition. Therefore, the caller must provide a partition key when calling into a partitioned collection. Stored procedures can make up for the lack of certain functionality.
- *Triggers*: Functions get executed before or after specific operations (like on a document insertion for example) that can either alter the operation or cancel it. Triggers are only executed on request.
- *User-defined functions (UDF)*: Functions that can be called from and augment the SQL query language making up for limited SQL features.

The SQL API is exposed as a REST API, which itself is implemented in various SDKs that are officially supported by Microsoft and available in .NET, .NET Core, Node.js (JavaScript), Java and Python.

5.5. Cosmos DB Emulator

The Azure Cosmos Emulator provides a local environment that emulates the Azure Cosmos DB service for development purposes [21]. Using the Azure Cosmos Emulator, somebody can develop and test an application locally, without creating an Azure subscription or incurring any costs. When one is satisfied with how the application works in the Azure Cosmos Emulator, can switch to using an Azure Cosmos account in the cloud. Currently, the Data Explorer view in the emulator only fully supports clients for SQL API.

5.5.1. How the emulator works

The Azure Cosmos Emulator provides a high-fidelity emulation of the Azure Cosmos DB service. It supports identical functionality as Azure Cosmos DB, including support for creating and querying data, provisioning and scaling containers, and executing stored procedures and triggers. Somebody can develop and test applications using the Azure Cosmos Emulator, and deploy them to Azure at global scale by just making a single configuration change to the connection endpoint for Azure Cosmos DB.

While emulation of the Azure Cosmos DB service is faithful, the emulator's implementation is different than the service. For example, the emulator uses standard OS components such as the local file system for persistence, and the HTTPS protocol stack for connectivity. Functionality that relies on Azure infrastructure like global replication, single-digit millisecond latency for reads/writes, and tunable consistency levels are not applicable.

5.5.2. Differences between the emulator and the service

Because the Azure Cosmos Emulator provides an emulated environment running on the local developer workstation, there are some differences in functionality between the emulator and an Azure Cosmos account in the cloud:

- Currently the Data Explorer in the emulator supports clients for SQL API. The Data Explorer view and operations for Azure Cosmos DB APIs such as MongoDB, Table, Graph, and Cassandra APIs are not fully supported.
- The Azure Cosmos Emulator supports only a single fixed account and a well-known master key. Key regeneration is not possible in the Azure Cosmos Emulator; however the default key can be changed using the command-line option.

- The Azure Cosmos Emulator is not a scalable service and does not support a large number of containers.
- The Azure Cosmos Emulator does not offer different Azure Cosmos DB consistency levels.
- The Azure Cosmos Emulator does not offer multi-region replication.
- As a copy of the Azure Cosmos Emulator might not always be up to date with the most recent changes in the Azure Cosmos DB service, one should refer to the Azure Cosmos DB capacity planner to accurately estimate the production throughput (RUs) needs of your application.
- When using the Azure Cosmos Emulator, by default, one can create up to 25 fixed size containers (only supported using Azure Cosmos DB SDKs), or 5 unlimited containers using the Azure Cosmos Emulator. For more information about changing this value, see Setting the PartitionCount value.

5.6. Cosmos DB Tools

Cosmos DB offers a full set of tools for supporting important tasks on data manipulation. Next we focus on the Data Explorer and the Data Migration Tool which are commonly used.

5.6.1. Data Explorer

The Cosmo DB Explorer offers a stand-alone, full-screen version of Data Explorer, which provides a graphical tool embedded into the Azure Cosmos DB account blade for managing and viewing content of Azure Cosmos DB account. It also simplifies the process of granting read-only or read and write access (permanently or temporarily, for the 24-hour period) on the database account level to users without Azure AD credentials (or without direct access to the Azure portal). To grant permanent access, one needs to provide these users with either read-only or read-write (depending on the level of access you intend to grant) database account connection string (which one can retrieve from the Keys blade of the database account). To grant temporary access, one first needs to retrieve either the read-write or read only access URL, both of which are revealed by clicking the Open Full Screen button in the Data Explorer interface.

Main features include:

- Named connection
- Create/Read/Update/Delete for every Cosmos DB resources
- Support Partitioned Collection
- Colored Editor (JSON, SQL, JavaScript)
- Customizable layout

5.6.2. Data Migration Tool

The Data Migration tool is an open-source solution that imports data to Azure Cosmos DB from a variety of sources, including:

- JSON files
- MongoDB
- SQL Server
- CSV files
- Azure Table storage
- Amazon DynamoDB
- HBase
- Azure Cosmos containers

While the import tool includes a graphical user interface (dtui.exe), it can also be driven from the command-line (dt.exe). In fact, there's an option to output the associated command after setting up an import through the UI. One can transform tabular source data, such as SQL Server or CSV files, to create hierarchical relationships (subdocuments) during import.

CHAPTER 6.

EXPERIMENTAL FRAMEWORK

Traditional database systems for storage have been based on the relational model. With the increase in accessibility of Internet and the availability of cheap storage, huge amounts of structured, semi-structured, and unstructured data are stored for a variety of applications. Processing such vast amount of data requires speed and flexible schemas. NoSQL databases claim to satisfy these requirements.

Most big-sized companies use these amounts of data for reporting and decision taking. The question that arises is if using a document store will reduce the times needed for getting the reports they want, and if so, what investments in hardware they must make in order to improve their performance. Also, there are questions about moving data stores to cloud storage and technologies. This motivated us to use a product that supports cloud technologies and easy data migration to cloud storage.

The goal of our experimental evaluation is to compare a document store data warehouse implementation with the corresponding SQL one. At this point, we focus on read operations because these are mandatory on reports creating. We also keep some notes on the data warehouse migration issues that we observe.

We will compare two market leading products from the same vendor, Microsoft SQL Server and Microsoft Azure Cosmos DB. We chose Microsoft Azure Cosmos DB document

store because it is also a cloud database solution and provides all the tools needed to move your datastore to the cloud. For practical reasons we used the Microsoft Azure Cosmos DB Emulator version, that lacks some database distribution options, but for our experiments this is not something that matters, because our databases run on premises.

6.1. Dataset Description

The initial dataset used is stored in an Oracle version 11g R2 database. The data used for our tests is a subset of real operational data of one of the biggest dairy companies in Greece, consisting of the last nine years daily sales transactions. Size of the data on disk is about 6 GB and total number of rows 5.365.076.

Table 6.1 Initial Dataset Characteristics

Feature	**Description**
Database Engine	Oracle 11g R2
Initial Size on Disk	48 GB
Final Size in Disk	6 GB
Records No.	5.365.076
Data Format	Structured
Data Subject	Sales data of midsized dairy company from year 2010 and beyond.

A schema subset of the tables in the initial database is presented in the following diagram:

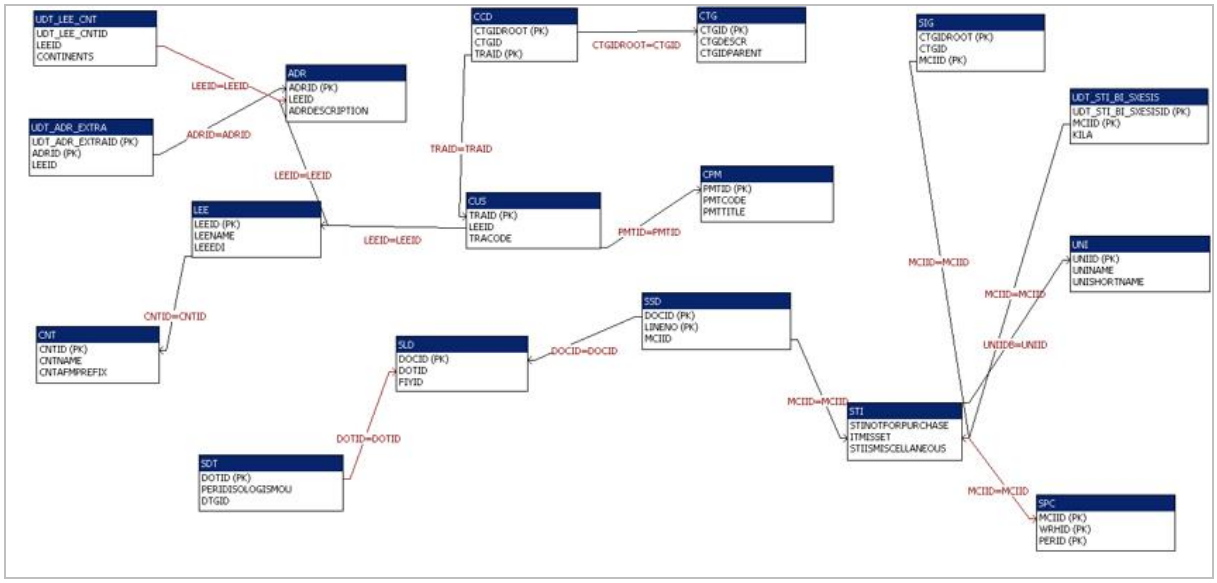

Figure 6.1 Dataset Tables Schema Diagram

A brief description of tables used you can read in the following table:

Table 6.2 Dataset Tables Description

Table Name	**Description**
UNI	Products Units
SDT	Types of Sales Documents
CTG	Categories of Products and Customers
CPM	Way of Payment
CNT	Countries
WRH	Warehouses
CUS	Customers Details
CCD	Customer Categories Details
ADR	Customers Addresses
LEE	Trader Details
SPC	Product Cost per Period
SIG	Products Categories Details
SSD	Sales Document Lines
SLD	Sales Document

STI	Products
UDT_ADR_EXTRA	Extra Traders Addresses Details
UDT_LEE_CNT	Geographical Data of Customers
UDT_STI_BI_SXESIS	Sales Units Convert Ratios

Table below describes attributes used in data migration per table.

Table 6.3 Dataset Tables Attributes Description

Table Name	**Attribute**	**Data Type**	**Description**
UNI	UNIID	Number(10)	Unit ID
	UNINAME	Varchar(2)	Unit Name
SDT	DOTID	Number(10)	Document Type ID
	DOTCODE	Varchar(2)	Document Type Code
	DOTDESCRIPTION	Varchar(2)	Document Type Description
	TDTSIGN	Number(10)	Document Type Value Sign (+/-)
CTG	CTGID	Number(10)	Category ID
	CTGOUTLINE	Varchar(2)	Category Code
	CTGDESCR	Varchar(2)	Category Description
	CTGSUBSYSTEMS	Varchar(2)	Category Subsystem ID
CPM	PMTID	Number(10)	Way of Payment ID
	PMTCODE	Varchar(2)	Way of Payment Code
	PMTTITLE	Varchar(2)	Way of Payment Description
CNT	CNTID	Number(10)	Country ID
	CNTNAME	Varchar(2)	Country Name
WRH	WRHID	Number(10)	Warehouse ID
	WRHCODE	Varchar(2)	Warehouse Code
	WRHNAME	Varchar(2)	Warehouse Name
CUS	TRAID	Number(10)	Customer ID
	LEEID	Varchar(2)	Trader ID
	TRACODE	Varchar(2)	Customer Code
CCD	CTGIDROOT	Number(10)	Customer Categories Details ID
	CTGID	Number(10)	Customer Category ID
	TRAID	Number(10)	Customer ID
ADR	ADRID	Number(10)	Customer Address ID
	ADRCODE	Varchar(2)	Customer Address Code
	LEEID	Number(10)	Customer Address Trader ID

	ADRDESCRIPTION	Varchar(2)	Customer Address Description
LEE	LEEID	Number(10)	Trader ID
	LEENAME	Varchar(2)	Trader Name
	LEEAFM	Varchar(2)	Trader VAT Num
	LEECODE	Varchar(2)	Trader Code
	ADRIDMAIN	Number(10)	Trader Address ID
SPC	MCIID	Number(10)	Product ID
	PERID	Number(10)	Period ID
	FIYID	Number(10)	Year ID
	WRHID	Number(10)	Warehouse ID
	SPCMSTK	Number(22,7)	Cost Price Value
SIG	CTGIDROOT	Number(10)	Products Categories Details
	CTGID	Number(10)	Products Category ID
	MCIID	Number(10)	Product ID
SSD	LINENO	Number(10)	Sales Document Line No
	DOCID	Number(10)	Sales Document Header ID
	MCIID	Number(10)	Sales Document Line Product
	SSDQTYA	Number(22,7)	Sales Document Line Quantity A
	STDQTYB	Number(22,7)	Sales Document Line Quantity B
	SSDNETVALUE	Number(22,7)	Sales Document Line Value
SLD	DOCID	Number(10)	Sales Document ID
	DOTID	Number(10)	Sales Document Type ID
	DOCENIMEROSISDATE	Date	Sales Document Date
	DOCCANCELSTATUS	Varchar(2)	Sales Document Cancel Status
	ADRIDPROORISMOU	Number(10)	Sales Document Delivery Addr.
	TRAID	Number(10)	Sales Document Customer ID
	WRHID	Number(10)	Sales Document Warehouse ID
	PERID	Number(10)	Sales Document Period ID
	FIYID	Number(10)	Sales Document Year ID
STI	MCIID	Number(10)	Product ID
	CODCODE	Varchar(2)	Product Code
	ITMNAME	Varchar(2)	Product Description
	UNIIDA	Number(10)	Product Unit A ID
	UNIIDB	Number(10)	Product Unit B ID
UDT_ADR_ EXTRA	ADRID	Number(10)	Address ID
	POLICY	Varchar(2)	Sales Policy Code
UDT_LEE_ CNT	LEEID	Number(10)	Trader ID
	CONTINENTS	Varchar(2)	Trader Continent

UDT_STI_ BI_SXESIS	MCIID	Number(10)	Product ID
	KILA	Number(15,5)	Kilos Conversion Factor
	TEMAXIA	Number(15,5)	Pieces Conversion Factor

6.2. Data Migration

We had to create two data warehouse databases with equivalent properties and attributes. One in Microsoft SQL Server and one in Microsoft Azure Cosmos DB Emulator. Both data warehouse databases are partitioned by the year values.

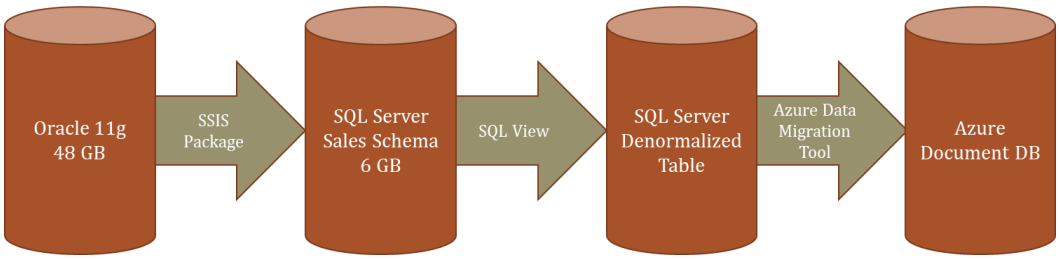

Figure 6.2 Data Migration Steps

Migration from Oracle to SQL Server performed with a SQL Server Integration Services packet. Main steps on migration process include the definition of a Data Control Flow with three steps. The first step will erase all data from the destination tables and the next two data flow tasks will copy data from source to destination database.

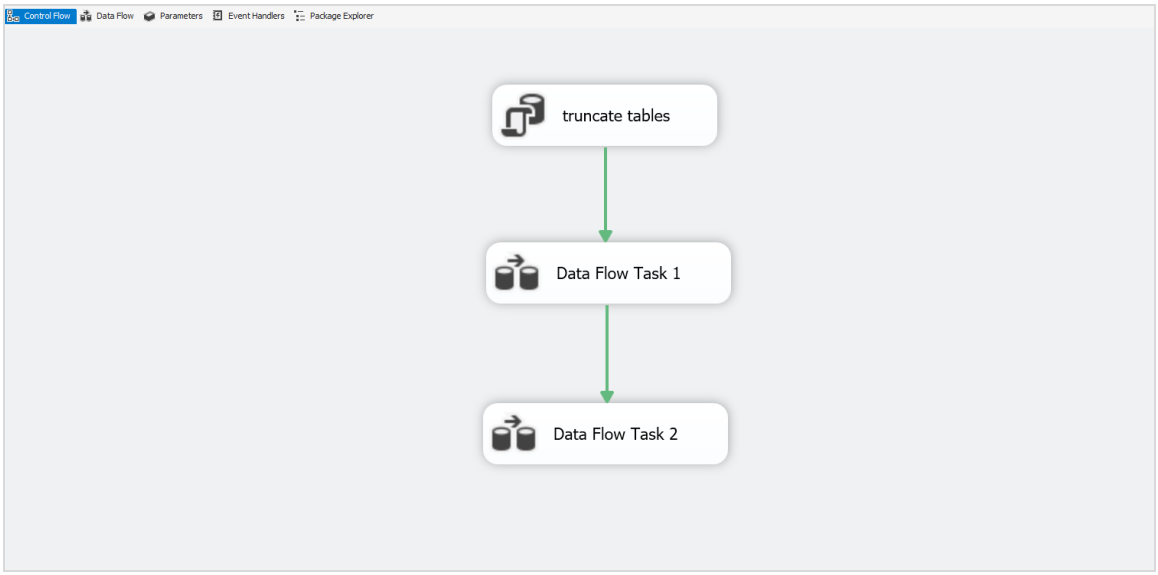

Figure 6.3 SSIS Data Migration Control Flow Diagram

Each Data Flow Task takes as parameters source and destination databases connection properties and SQL statements for selecting source and destination tables and columns.

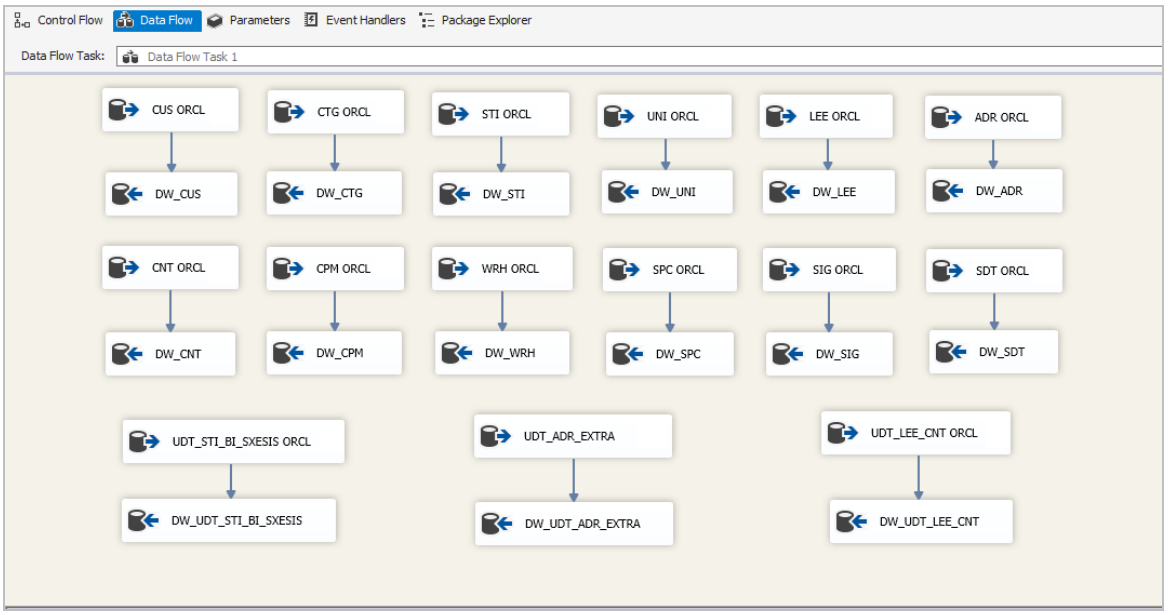

Figure 6.4 SSIS Data Flow from Oracle to SQL Server Diagram

For each source table we had to specify the connection properties and the select statement to filter the data that we wanted to be copied to the destination tables.

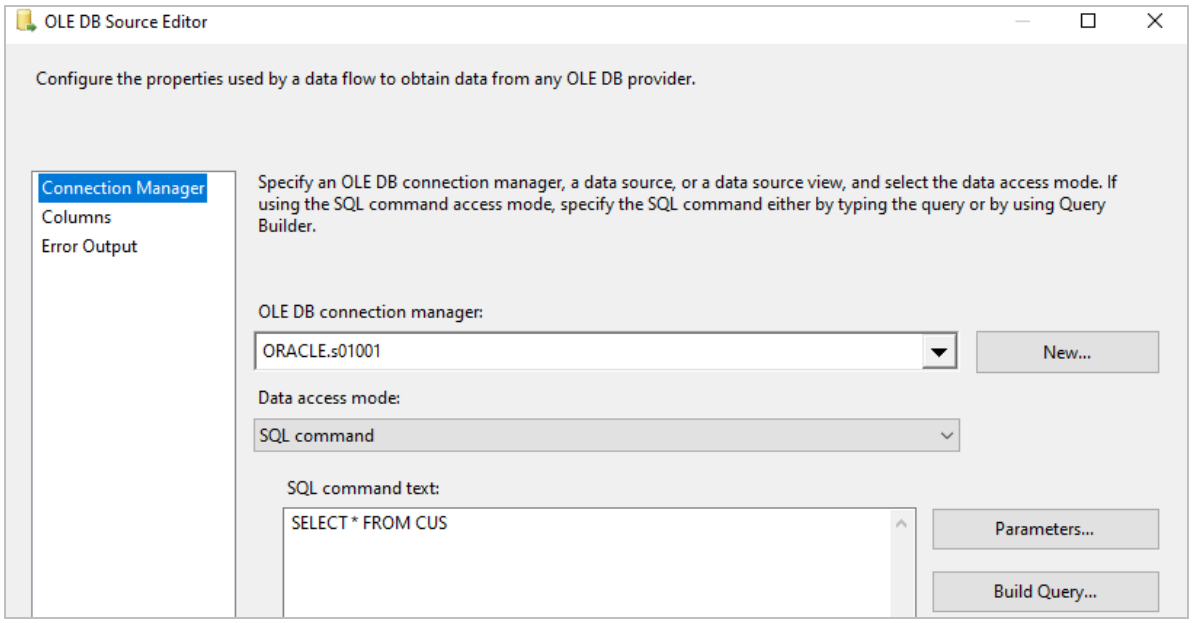

Figure 6.5 SSIS Source Table Properties Editor

For each destination table we had to specify the destination database connection properties and the destination table.

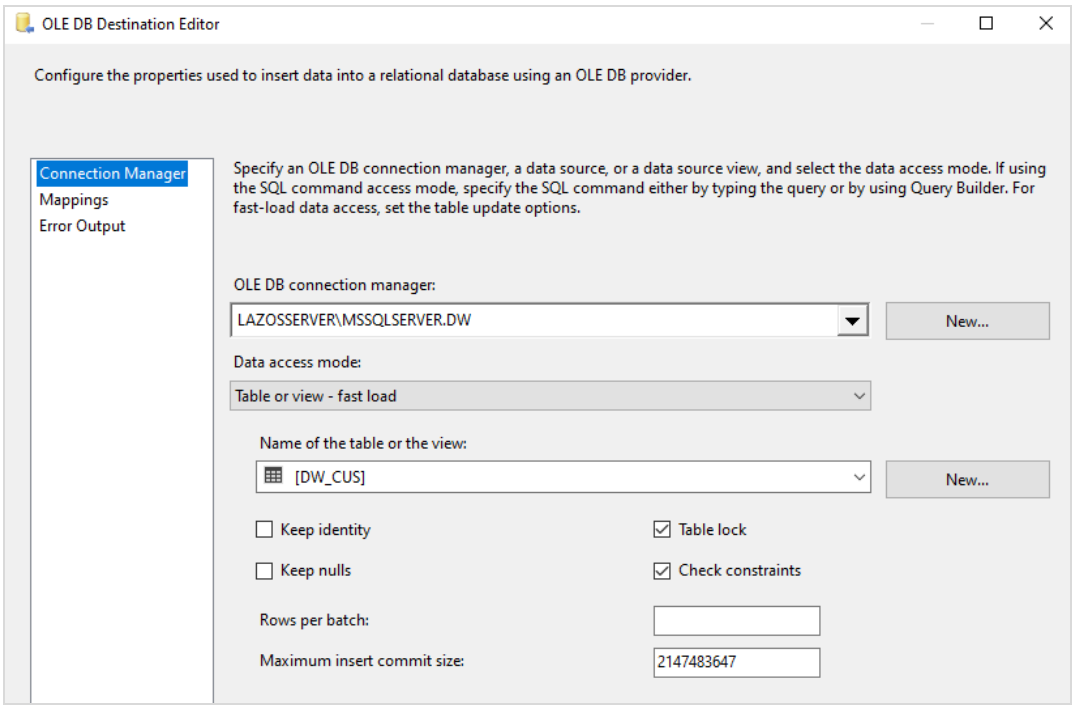

Figure 6.6 SSIS Destination Table Properties Editor

Finally, it is optional to modify the column mappings between source and destination tables.

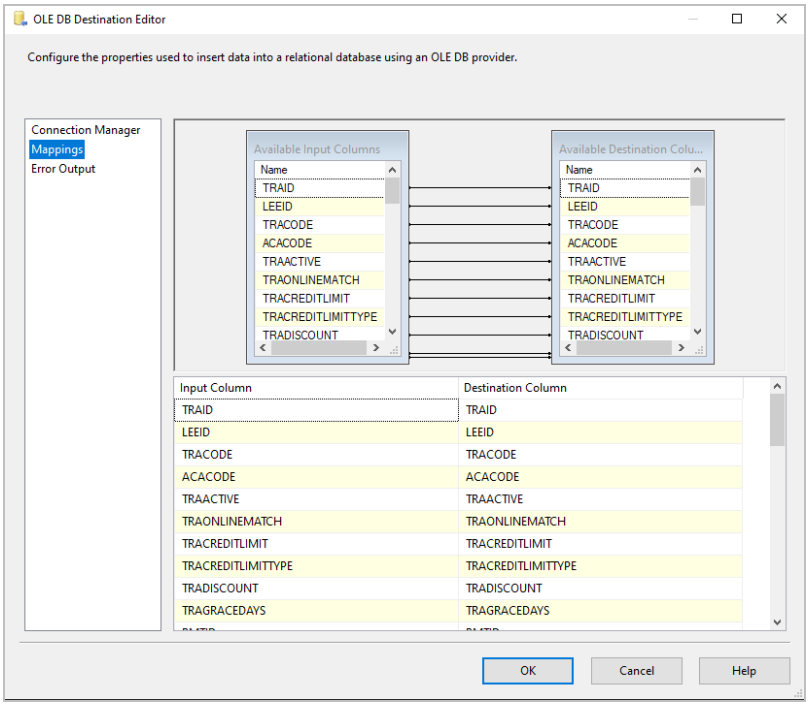

Figure 6.7 SSIS Destination Table Columns Mapping

After finishing the tables copy process, we created a partitioned denormalized table with all columns that were going to need in our final record set. This was important because we had to overcome the following issues:

- Document store does not support joins between collections (SQL tables)
- Document store does not fully support aggregations and grouping.
- Azure Document DB databases are always partitioned.
- We had to compare equivalent datastores in both databases

We used the view described in Table A.1 to create the denormalized table with the use of INSERT INTO SQL command. A brief description of the attributes of the view is presented in the following table.

Table 6.4 Description of Attributes of Denormalization View

Attribute Name	**Description**
CustSalesGroup	Customer Sales Group
CusCode	Customer Code
SITECODE	Sales Policy Code
Customer	Customer Description
ITEMCODE	Product Code
ITEM_DESCRIPTION	Product Description
FULLDATE	Sales Date
FISCALYEAR	Sales Year
FISCALMONTH	Sales Month
ACT_NETVALUE	Net Sales Value
ACTGrossSales	Gross Sales Value
ACTOffInvDiscount	Discounts Values
ACTDistrFee	Distribution Fee Value
ACTPromoNotes	Promotional Fee Value
ACTReturns	Sales Returns Value
ACTExportsCN	Exports Sales Credit Notes
ACTSpecialDisc	Sales Special Discounts Value
ACTOtherServices	Sales Other Services Value
ACTOrders	Orders
[ACT_QTYA]	Sales Quantity A Volume
[ACT_QTYB]	Sales Quantity B Volume
ACT_QtyKGR	Sales Kilos Volume
ACT_OrdersKGR	Order Kilos Volume
ACT_QtyPCS	Sales Pieces Volume

The final action, on SQL Server denormalized table, was the partitioning by year values using the column FISCALYEAR, with the following commands:

```
CREATE PARTITION FUNCTION myYear (int)
AS RANGE RIGHT FOR VALUES ('2012','2013','2014','2015','2016','2017','2018','2019')
GO

CREATE PARTITION SCHEME myPartitionScheme
AS PARTITION myYear ALL TO ([PRIMARY])
GO
```

Figure 6.8 Commands for Creating Partitions on SQL Server

For building Microsoft Azure Cosmos DB Emulator document store database, we used Azure Data Migration Tool with SQL Server's partitioned denormalized table.

The SQL source import option allows one to import from an individual SQL Server database and optionally filter the records to be imported using a query. The format of the connection string is the standard SQL connection string format.

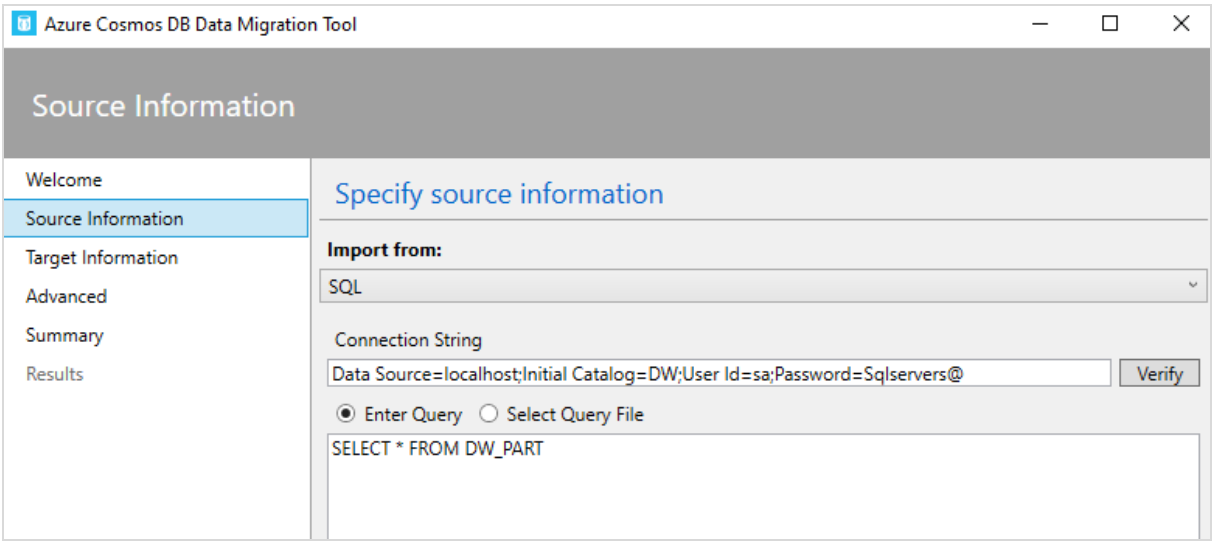

Figure 6.9 Cosmos DB Data Migration Tool Source Info

The Azure Cosmos DB Bulk importer allows you to import from any of the available source options, using an Azure Cosmos DB stored procedure for efficiency. The tool supports import to one single-partitioned Azure Cosmos container. It also supports sharded import whereby data is partitioned across more than one single-partitioned Azure Cosmos container. The tool creates, executes, and then deletes the stored procedure from the target collection(s).

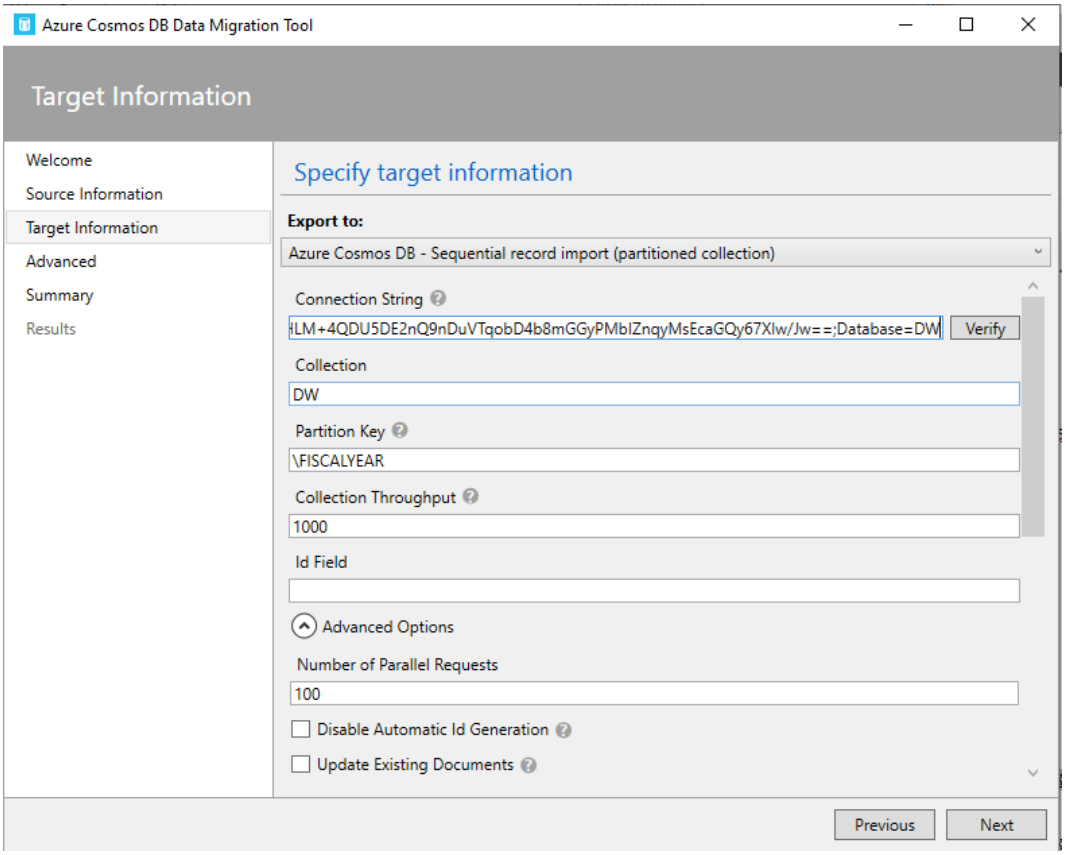

Figure 6.10 Cosmos DB Data Migration Tool Target Info

The Mandatory Target parameters are Collection name, Partition Key (in our case was \FISCLAYEAR) and Number of Parallel Requests, a number that reduces migration time but might increase error during copy of data (we used a value of 100).

Additionally, when importing date types (for example, from SQL Server or MongoDB), someone can choose between three import options:

- *String:* Persist as a string value
- *Epoch:* Persist as an Epoch number value
- *Both:* Persist both string and Epoch number values.

The Azure Cosmos DB Bulk importer has the following additional advanced options:

- *Batch Size*: The tool defaults to a batch size of 50. If the documents that must be imported are large, consider lowering the batch size. Conversely, if the documents that must be imported are small, consider raising the batch size.

- *Max Script Size (bytes)*: The tool defaults to a max script size of 512 KB.
- *Disable Automatic Id Generation*: If every document that must be imported has an ID field, then selecting this option can increase performance. Documents missing a unique ID field are not imported.
- *Update Existing Documents*: The tool defaults to not replacing existing documents with ID conflicts. Selecting this option allows overwriting existing documents with matching IDs. This feature is useful for scheduled data migrations that update existing documents.
- *Number of Retries on Failure*: Specifies how often to retry the connection to Azure Cosmos DB during transient failures (for example, network connectivity interruption).
- *Retry Interval*: Specifies how long to wait between retrying the connection to Azure Cosmos DB in case of transient failures (for example, network connectivity interruption).
- *Connection Mode*: Specifies the connection mode to use with Azure Cosmos DB. The available choices are DirectTcp, DirectHttps, and Gateway. The direct connection modes are faster, while the gateway mode is more firewall friendly as it only uses port 443.

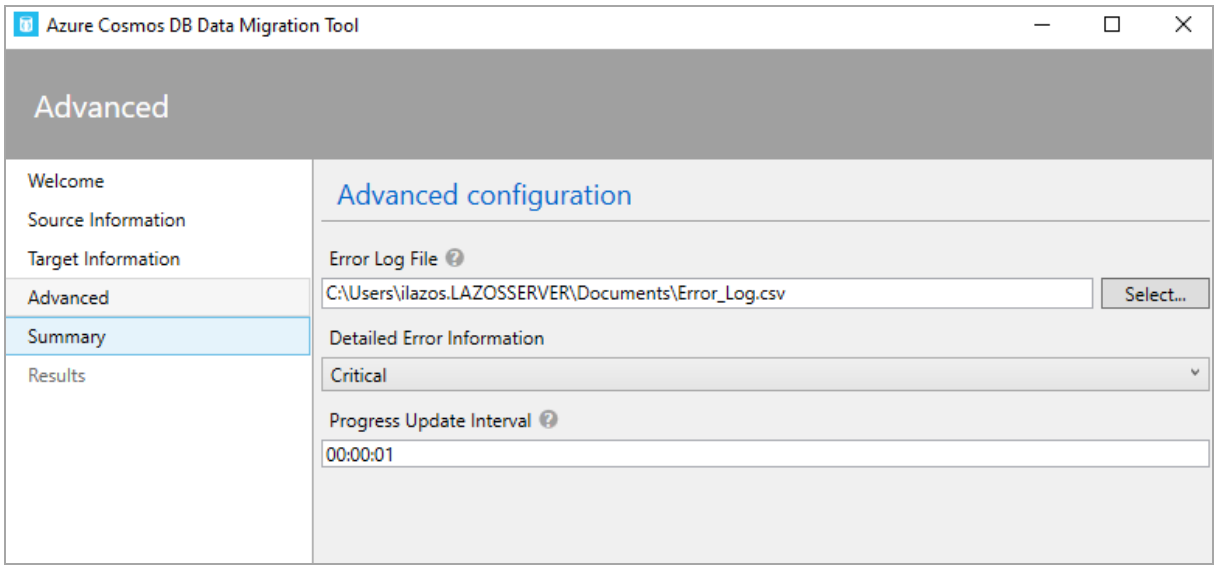

Figure 6.11 Cosmos DB Data Migration Tool Advanced Configuration

After source and target options are chosen, click Import. The elapsed time, transferred count, and failure information (if you did not provide a file name in the Advanced configuration) update as the import is in process. Once complete, the results can be exported (for example, to deal with any import failures).

6.3. Implementation

The goal of our experiments was to measure the execution times of SELECT queries with different syntax on a machine with different hard disk and memory size configuration options, in both the SQL Server and the Azure Cosmos DB Emulator. Note that the times are averaged over three runs. The absolute time values are not significant; what is significant is the time values relative to one another.

We run four different queries on six different hardware configurations concerning available memory (16, 8, 4 GB) and storage device types (HDD: *1TB - 7200 RPM – SATA 6Gb/s - 64MB Cache* and SSD: *1TB – Read 560MB/s – SATA 6Gb/s*). PC used for implementation was an Intel i7-3770 @ 3,40 GHz. Queries used are described in following tables.

Table 6.5 Implementation of Query No. 1

Description	*Query all records*
SQL	`SELECT * FROM DW_PART`
NoSQL	`SELECT * FROM c`

Table 6.6 Implementation of Query No. 2

Description	*Query a specific SKU in a specific customer category*
SQL	`SELECT * FROM DW_PART where ITEMCODE like '1030%' and CustSalesGroup ='37 KEY ACCOUNT DIRECT'`
NoSQL	`SELECT * FROM c where (STARTSWITH(c.ITEMCODE, '1030') and c.CustSalesGroup ='37 KEY ACCOUNT DIRECT')`

Table 6.7 Implementation of Query No. 3

Description	*Calculate the total sales values and volumes per year per customer, of a specific SKU in a specific customer category*
SQL	`SELECT FISCALYEAR, CUSCODE, SUM(ACT_NETVALUE) AS VALUE, SUM(ACT_QtyKGR) AS VOLUME FROM [DW_PART] where ITEMCODE like '1030%' and CustSalesGroup ='37 KEY ACCOUNT DIRECT' GROUP BY FISCALYEAR, CUSCODE`
NoSQL	`SELECT c.FISCALYEAR, c.CusCode, SUM(c.ACT_NETVALUE) AS Sales_Value, SUM(c.ACT_QtyKGR) AS Sales_Volume FROM c where(STARTSWITH(c.ITEMCODE, '1030') and c.CustSalesGroup = '37 KEY ACCOUNT DIRECT') GROUP BY c.FISCALYEAR, c.CusCode`

Table 6.8 Implementation of Query No. 4

Description	*Calculate the total sales values and volumes per year per customer of a specific SKU in a specific customer category*
SQL	`SELECT FISCALYEAR, CUSCODE, SUM(ACT_NETVALUE) AS VALUE, SUM(ACT_QtyKGR) AS VOLUME FROM [DW].[dbo].[DW_PART] GROUP BY FISCALYEAR, CUSCODE`
NoSQL	`SELECT c.FISCALYEAR, c.CusCode, SUM(c.ACT_NETVALUE) AS Sales_Value, SUM(c.ACT_QtyKGR) AS Sales_Volume FROM c GROUP BY c.FISCALYEAR, c.CusCode`

For measuring query execution times in SQL Server, we used Microsoft SQL Server Management Studio query execution panel with the option *Include Client Statistics*. Before each query execution we were clearing SQL Server database cache with the commands:

- DBCC FREEPROCCACHE to clear the procedure cache. Freeing the procedure cache would cause, for example, an ad-hoc SQL statement to be recompiled rather than reused from the cache.

- DBCC DROPCLEANBUFFERS to test queries with a cold buffer cache without shutting down and restarting the server. DBCC DROPCLEANBUFFERS serves to empty the data cache. Any data loaded into the buffer cache due to the prior execution of a query is removed.

For measuring query execution times in Azure Cosmos DB Emulator, we could not use Data Explorer, because query results are displayed with paging, fetching 100 documents per time, so it is not possible to calculate exact query execution times. To overcome this issue, we developed a C#. Net procedure to calculate total query execution time. The basic idea was to use StopWatch class to measure execution time until all records are fetched, without any user interfere. Code of this procedure follows:

```
private async void BtnDocDB_ClickAsync(object sender, EventArgs e)
    {
        FeedResponse<dynamic> result;
        IReadOnlyDictionary<string, QueryMetrics> metrics;
        //Total Fetched Documents Counter
        int doccounter = 0;
        //Total Query Execution Time
        double queryexecutiontime = 0;

        //Database Address
        string EndpointUri = "https://localhost:8081";
        //Connection Key
        string PrimaryKey = "C2y6yDjf5/R+ob0N8A7Cgv30VR
                            DJIWEHLM+4QDU5DE2nQ9nDuVTqob
                            D4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==";
        //Database Name
        string databaseName = "DW";
        //Collection Name
        string CollectionName = "DW";
```

Figure 6.12 C#.Net Procedure - Initialization of Variables

```
//Connection policy parameters
ConnectionPolicy = new ConnectionPolicy ();
connectionPolicy.UserAgentSuffix = " samples-net/3";
connectionPolicy.ConnectionMode = ConnectionMode.Gateway;
connectionPolicy.ConnectionProtocol = Protocol.Https;
connectionPolicy.RequestTimeout = new TimeSpan(0, 60, 00);

DocumentClient cosmosClient;
//Create a new connection to the database
cosmosClient = new DocumentClient(new Uri(EndpointUri), PrimaryKey, connectionPolicy);
Database = await cosmosClient.ReadDatabaseAsync(UriFactory.CreateDatabaseUri(databaseName));
await cosmosClient.OpenAsync();
```

Figure 6.13 C#.Net Procedure - Connection Policy Parameters

```
//Create a new query to the database – The query changes every in every test
IDocumentQuery<dynamic> query = cosmosClient.CreateDocumentQuery(
   UriFactory.CreateDocumentCollectionUri(databaseName, CollectionName),
   "SELECT * FROM c",
 //Query results parameters
 new FeedOptions
   {
      PopulateQueryMetrics = true,
      MaxItemCount = -1,
      MaxDegreeOfParallelism = -1,
      EnableCrossPartitionQuery = true
   }).AsDocumentQuery();
```

Figure 6.14 C#.Net Procedure – Query Initialization

```
    //Loop for fetching query results
    while (query.HasMoreResults)
    {
      //StopWatch start
      queryExecutionTimeEndToEndTotal.Start();
      result = await query.ExecuteNextAsync();
      //StopWatch Stop
      queryExecutionTimeEndToEndTotal.Stop();
      //Count total fetched documents
      doccounter = doccounter + result.Count();
    }

    //Save results in txt file
    System.IO.File.WriteAllText("results.txt", "Doc Counter: " + doccounter.ToString() + " - Elapsed Time: "
    + queryExecutionTimeEndToEndTotal.Elapsed.ToString());
    //Dispose the connection
    cosmosClient.Dispose();
    |
}
```

Figure 6.15 C#.Net Procedure – Fetching Records and Exporting Results

6.4. Azure Cosmos DB emulator issues

Because the Azure Cosmos Emulator provides an emulated environment running on the local developer workstation, we experienced the following issues during our tests:

- When the database size exceeded the size of 25GB on disk, emulator failed to load
- After some runs emulator did not recognize the version of the database and the only solution to that problem was reinstalling the emulator and migrating the database again.
- Sometimes while the emulator was connected to the database, could not display containers and documents. To overcome this issue, we had to create the database again.

- There was no procedure for clearing database cash and buffers between query executions and we had to restart the database instance in order to overcome this issue.

CHAPTER 7.

RESULTS

7.1. Query No. 1 Test Results

Our first experiment measures the time taken to execute query No. 1 on both platforms, with all sizes of installed memory and both HDD and SSD storage options. The number of total records returned (selectivity) is 5.365.076 out of 5.365.076. See Figures 7.1 and 7.2 which summarizes the results of this experiment.

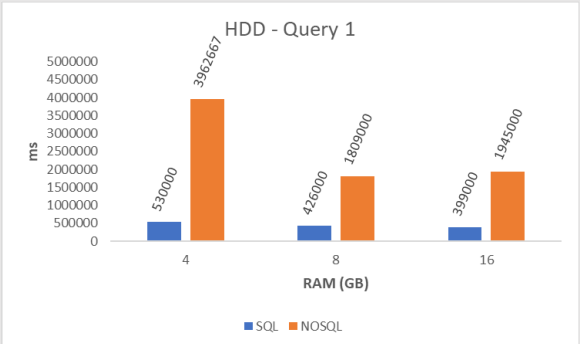

Figure 7.1 Query No.1 with HDD

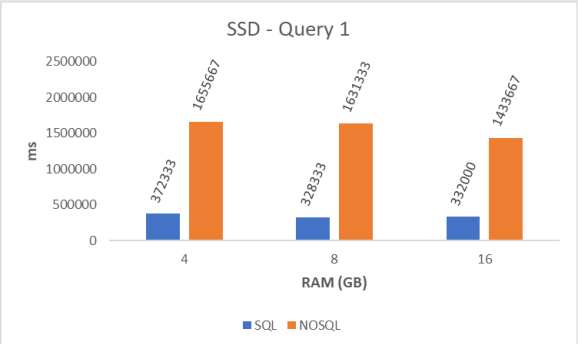

Figure 7.2 Query No.1 with SSD

We observed that:

- The SQL Server has better performance with all configurations
- Query execution times are better when SSD disk is used
- There is a substantial improvement in execution times when RAM is not enough to store all data and SSD disks are used, especially for Azure Cosmos DB.
- The SQL Server is affected less by low memory (4 GB) and shows a stable performance with any of the tested configurations.

7.2. Query No. 2 Test Results

Our second experiment measures the time taken to execute query No. 2 on both platforms, with different sizes of installed memory and both HDD and SSD storage options. The selectivity is 15.037 out of 5.365.076. See Figures 7.3 and 7.4 which summarizes the results of this experiment.

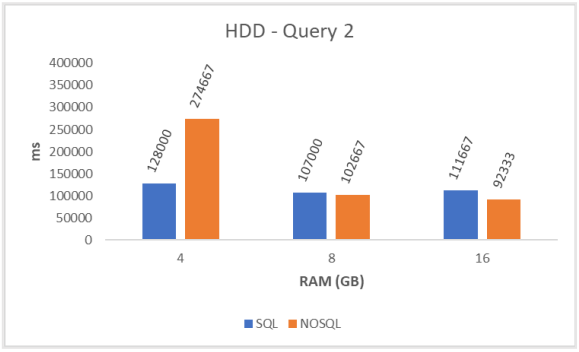

Figure 7.3 Query No.2 with HDD

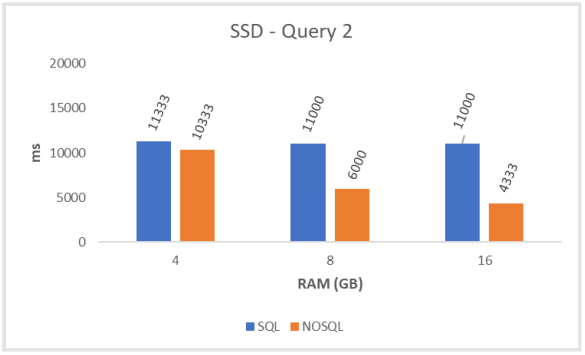

Figure 7.4 Query No.2 with SSD

We observed that:

- Azure Cosmos DB has better performance with all configurations, except with 4 GB of RAM and HDD.
- Both SQL Server and Azure Cosmos DB have better execution times when running on a system with SSD
- There is a substantial improvement in execution times when RAM is not enough to store all data and SSD disks are used, especially for Azure Cosmos DB.

- The SQL Server is affected less by low memory (4 GB) and shows a stable performance with any of the tested configurations.
- When we have a WHERE clause in the SELECT statement, the Azure Cosmos DB has better performance in all cases except one.

7.3. Query No. 3 Test Results

Our third experiment measures the time taken to execute query No. 3 on both platforms, with different sizes of installed memory and both HDD and SSD storage options. The selectivity is 69 out of 5.365.076. See Figures 7.5 and 7.6 which summarizes the results of this experiment.

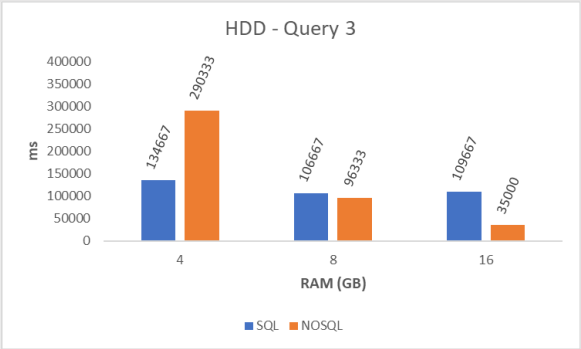

Figure 7.5 Query No.3 with HDD

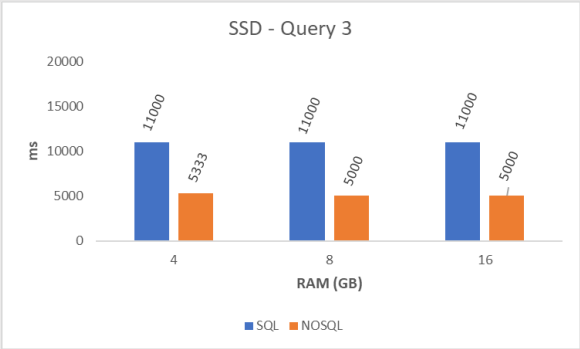

Figure 7.6 Query No.3 with SSD

We observed that:

- Azure Cosmos DB has better performance with all configurations, except with 4 GB of RAM and HDD.
- Both SQL Server and Azure Cosmos DB have better execution times when running on a system with SSD
- There is a substantial improvement in execution times when RAM is not enough to store all data and SSD disks are used, especially for Azure Cosmos DB.
- The SQL Server is affected less by low memory (4 GB) and shows a stable performance with any of the tested configurations.

- When we have a GROUB BY with WHERE clauses in SELECT statement, the Azure Cosmos DB has better performance than SQL Server in all cases except one.

7.4. Query No. 4 Test Results

Our fourth experiment measures the time taken to execute query No. 4 on both platforms, with different sizes of installed memory and both HDD and SSD storage options. Total records returned 3.836 out of 5.365.076. See Figures 7.7 and 7.8 which summarizes the results of this experiment.

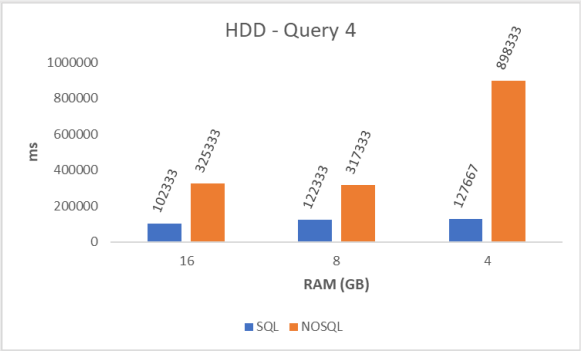

Figure 7.7 Query No.4 with HDD

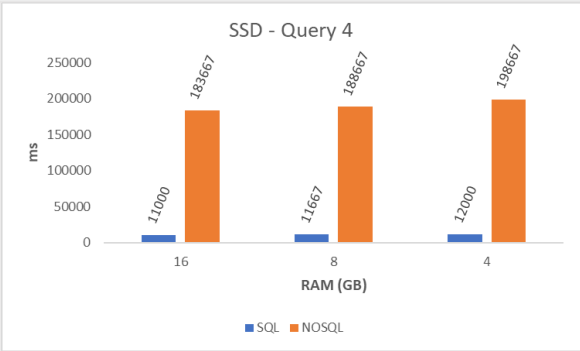

Figure 7.8 Query No.4 with SSD

We observed that:

- Microsoft SQL Server has better performance with all configurations
- Both SQL Server and Azure Cosmos DB have better execution times when running on a system with SSD
- There is a substantial improvement in execution times when RAM is not enough to store all data and SSD disks are used, especially for Azure Cosmos DB.
- SQL Server is affected less by low memory (4 GB) and shows a stable performance with any of the tested configurations.
- When we have a GROUB BY clause in SELECT statement SQL Server has better performance than Azure Cosmos DB in all cases.

CHAPTER 8.

SUMMARY AND CONCLUSION

In this thesis we migrate data from a SQL database to a document store in order to evaluate query execution times using alternative hardware configurations.

Data migration is not a straightforward process between the two data stores. Document store does not support joins between collections, aggregations and grouping, so there is a need of creating a denormalized table with all values precalculated before migrating. The lack of aggregations is a big minus for using document stores as a data warehouse for reporting. The time needed for migration is also another negative parameter because it takes more than two hours to transfer 6 GB of data. Storage space needed for the same number of records is four times more for document stores (24 GB).

After query execution performance evaluation, we observe that the document store does not perform better in all cases than SQL database. There is a big variation in performance based on query type, amount of available system memory and type of storage device.

The SQL Server has a better performance when selecting all records from database and when grouping on all records is used. On the other side the Cosmos DB performs better when there is a WHERE clause in queries.

The SQL Server is affected less by low memory (4 GB) and shows a stable performance with any of the tested configurations. Both SQL Server and Azure Cosmos DB have better execution times when running on a system with SSD. In general, there is a substantial improvement in execution times in case RAM is not enough to load all data and SSD disks are used, especially for Azure Cosmos DB.

From the above we conclude that it does not seem that document store is a better option for creating a data warehouse for reporting. If database distribution is not important then SQL Server is the product to choose.

Like any application software, document store implementations go through changes and thus performance improvements and degradations are likely to happen through these changes. Consequently, one will need to compare databases not only at the application design stage but also at regular intervals to enable switching to the most suitable database implementation.

BIBLIOGRAPHY

[1] K. Kline, SQL in a nutshell, 3rd ed, O'Reilly Media, November 2008

[2] Matt Asay, "NoSQL databases eat into the relational database market", Available:http://www.techrepublic.com/article/nosql-databases-eat-into-therelational-database-market/, March 4, 2015,

[3] P. Litwin, "Fundamentals of relational Database Design" Microsoft Tech-Ed, 1995

[4] Funck, Robin & Jablonski, Stefan, "NoSQL evaluation: A use case oriented survey.", Proc 2011 Int Conf Cloud Serv Computing, 10.1109/CSC.2011.6138544, December 2011

[5] Davoudian, Ali & Chen, Liu & Liu, Mengchi. "A survey on NoSQL stores", ACM Computing Surveys, 51, 1-43, 10.1145/3158661, April 2018

[6] N. Leavitt, "Will NoSQL databases live up to their promise?" Computer, vol. 43, no. 2, pp. 12 –14, feb. 2010

[7] D. Bartholomew, "SQL vs. NoSQL," Linux Journal, no. 195, July 2010

[8] M. Indrawan-Santiago, "Database research: Are we at a crossroad? Reflection on NoSQL," in Network-Based Information Systems (NBiS), 2012 15th International Conference on, sept. 2012, pp. 45 –51

[9] Y. Li and S. Manoharan, "A performance comparison of SQL and NoSQL databases," 2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM), pp. 15-19, 2013.

[10] Gyorodi, Cornelia & Gyorodi, Robert & Sotoc, Roxana, "A Comparative Study of Relational and Non-Relational Database Models in a Web- Based Application",

International Journal of Advanced Computer Science and Applications, 6. 10.14569/IJACSA.2015.061111

[11] S. Gilbert and N. Lynch, "Perspectives on the CAP Theorem," in Computer, vol. 45, no. 2, pp. 30-36, Feb. 2012.

[12] Mohamed A. Mohamed, Obay G. Altrafi, Mohammed O. Ismail, "Relational vs. NoSQL Databases: A Survey", International Journal of Computer and Information Technology (ISSN: 2279 – 0764), Volume 03 – Issue 03, May 2014

[13] C. Mohan et al., "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", ACM Transactions on Database Systems, Volume 17 – No. 1, pp. 94-162, March 1992

[14] Vatika Sharma et al., "SQL and NoSQL Databases", International Journal of Advanced Research in Computer Science and Software Engineering, Volume 02 - Issue 08, pp. 20-27, August- 2012

[15] Christian Bolton et al., "Professional SQL Server 2012 Internals and Troubleshooting". 1st ed, Wrox Press, November 2012.

[16] Microsoft, "Query Processing Architecture Guide", Available: https://docs.microsoft.com/en-us/sql/relational-databases/query-processing-architecture-guide?view=sql-server-ver15, February 2019

[17] Microsoft, "Database Files and Filegroups", Available: https://docs.microsoft.com/en-us/sql/relational-databases/databases/database-files-and-filegroups?view=sql-server-ver15, July 2018

[18] Slava Oks, " A new platform layer in SQL Server 2005 to exploit new hardware capabilities and their trends", Available: https://docs.microsoft.com/el-gr/archive/blogs/slavao/platform-layer-for-sql-server, July 2015

[19] Kalen Delaney et al., "Microsoft SQL Server 2012 Internals", 1st ed., O'Reilly, November 2013

[20] Dharma Shukla, "A technical overview of Azure Cosmos DB", Available: https://azure.microsoft.com/en-us/blog/a-technical-overview-of-azure-cosmos-db/, May 2017

[21] Microsoft, "Use the Azure Cosmos Emulator for local development and testing", Available: https://docs.microsoft.com/en-us/azure/cosmos-db/local-emulator, July 2019

APPENDIX

Table A.1 View Used for Creating the Denormalized Table in SQL Server

```
SELECT
CASE
        WHEN (DW_CUS.TRACODE LIKE '090%') THEN 'EXPORTS'
        WHEN CTG.TRAID IS NULL THEN ''
        ELSE (ctg.ctgoutline) + ' ' + ctg.ctgdescr
END AS CustSalesGroup,
LEFT(DW_CUS.TRACODE,6) AS CusCode,
DW_UDT_ADR_EXTRA.POLICY AS SITECODE,
DW_LEE.LEENAME AS Customer,
LEFT(DW_STI.CODCODE,4) ITEMCODE,
DW_STI.ITMNAME [ITEM_DESCRIPTION],
CASE
        WHEN DW_SDT.DOTCODE IN ('П000','П100') THEN DW_SLD.TDOEKTELESISDATE
        WHEN DW_SDT.DOTCODE IN ('П126F','П126FX','П126','П126X','П127','П127X','П126T','П126TX') AND
                DW_SLD.DOCENIMEROSISDATE IS NULL THEN DW_SLD.TDOEKTELESISDATE
        ELSE DW_SLD.DOCENIMEROSISDATE
END AS [FULLDATE],
CASE
        WHEN DW_SDT.DOTCODE IN ('П000','П100') THEN YEAR(DW_SLD.TDOEKTELESISDATE)
        WHEN DW_SDT.DOTCODE IN ('П126F','П126FX','П126','П126X','П127','П127X','П126T','П126TX') AND
                DW_SLD.DOCENIMEROSISDATE IS NULL THEN YEAR(DW_SLD.TDOEKTELESISDATE)
        ELSE YEAR(DW_SLD.DOCENIMEROSISDATE)
END AS [FISCALYEAR],
CASE
         WHEN DW_SDT.DOTCODE IN ('П000','П100') THEN MONTH(DW_SLD.TDOEKTELESISDATE)
         WHEN DW_SDT.DOTCODE IN ('П126F','П126FX','П126','П126X','П127','П127X','П126T','П126TX') AND
                 DW_SLD.DOCENIMEROSISDATE IS NULL THEN MONTH(DW_SLD.TDOEKTELESISDATE)
         ELSE MONTH(DW_SLD.DOCENIMEROSISDATE)
END AS [FISCALMONTH],
CASE
        WHEN DW_SDT.DOTCODE NOT IN ('П000','П100','П120','П120П','П120Z','П125','П125П','П125Z','П190','П160N','П123','П160Δ','П140')
AND DW_SLD.DOCENIMEROSISDATE IS NOT NULL THEN CAST((DW_SSD.SSDNETVALUE * DW_SDT.TDTSIGN) AS numeric(18,3))
        ELSE CAST(0.0 as numeric(18,3))
END AS [ACT_NETVALUE],
CASE
        WHEN DW_SDT.DOTCODE IN('П113','П113ПЕ','П118','П118ПЕ','П119','П170','П193','П194','П197','П198','П135','П136','П131','П142','П141',
                'П201') AND (DW_SLD.DOCENIMEROSISDATE IS NOT NULL) THEN CAST((DW_SSD.SSDNETVALUE * DW_SDT.TDTSIGN)
                 AS numeric(18,3))
        ELSE CAST(0.000 as numeric(18,3))
END AS ACTGrossSales,
CASE
        WHEN DW_SDT.DOTCODE ='П126F'AND (DW_SLD.DOCENIMEROSISDATE IS NOT NULL) THEN CAST((DW_SSD.SSDNETVALUE *
                 DW_SDT.TDTSIGN) AS numeric(18,3))
        WHEN DW_SDT.DOTCODE ='П126FX'AND (DW_SLD.DOCENIMEROSISDATE IS NOT NULL) THEN CAST((DW_SSD.SSDNETVALUE *
DW_SDT.TDTSIGN) AS numeric(18,3))
        ELSE CAST(0.000 as numeric(18,3))
END AS ACTDistrFee,
CASE
        WHEN DW_SDT.DOTCODE ='П127' AND (DW_SLD.DOCENIMEROSISDATE IS NOT NULL) then CAST((DW_SSD.SSDNETVALUE *
DW_SDT.TDTSIGN) AS numeric(18,3))
         WHEN DW_SDT.DOTCODE ='П127X' AND (DW_SLD.DOCENIMEROSISDATE IS NOT NULL) then CAST((DW_SSD.SSDNETVALUE *
DW_SDT.TDTSIGN) AS numeric(18,3))
        ELSE CAST(0.000 as numeric(18,3)) END AS ACTPromoNotes,
CASE
        WHEN DW_SDT.DOTCODE in ('П114','П114ЕП','П114Z','П115','П195','П199','П130') AND (DW_SLD.DOCENIMEROSISDATE IS NOT
NULL) THEN CAST((DW_SSD.SSDNETVALUE * DW_SDT.TDTSIGN) AS numeric(18,3))
        ELSE CAST(0.000 as numeric(18,3))
END AS ACTReturns,
CASE
        WHEN DW_SDT.DOTCODE ='П196' AND (DW_SLD.DOCENIMEROSISDATE IS NOT NULL) then CAST((DW_SSD.SSDNETVALUE *
DW_SDT.TDTSIGN) AS numeric(18,3))
        WHEN DW_SDT.DOTCODE ='П200' AND (DW_SLD.DOCENIMEROSISDATE IS NOT NULL) then CAST((DW_SSD.SSDNETVALUE *
DW_SDT.TDTSIGN) AS numeric(18,3))
        ELSE CAST(0.000 as numeric(18,3))
END AS ACTExportsCN,
```

```
CASE
		WHEN DW_SDT.DOTCODE ='П116' AND (DW_SLD.DOCENIMEROSISDATE IS NOT NULL) then CAST((DW_SSD.SSDNETVALUE *
DW_SDT.TDTSIGN) AS numeric(18,3))
		ELSE CAST(0.000 as numeric(18,3))
END AS ACTSpecialDisc,
CASE
		WHEN DW_SDT.DOTCODE ='П117' AND (DW_SLD.DOCENIMEROSISDATE IS NOT NULL) then CAST((DW_SSD.SSDNETVALUE *
DW_SDT.TDTSIGN) AS numeric(18,3))
		 WHEN DW_SDT.DOTCODE ='П117П' AND (DW_SLD.DOCENIMEROSISDATE IS NOT NULL) then CAST((DW_SSD.SSDNETVALUE *
DW_SDT.TDTSIGN) AS numeric(18,3))
		ELSE  CAST(0.000 as numeric(18,3))
END AS ACTOtherServices,
CASE
		WHEN DW_SDT.DOTCODE in ('П000','П100') AND DW_SLD.TDOEKTELESISDATE>(SELECT GETDATE() AS CurrentDateTime) then
CAST((DW_SSD.SSDNETVALUE * DW_SDT.TDTSIGN) AS numeric(18,3))
		ELSE CAST(0.000 as numeric(18,3))
END AS ACTOrders,
CASE
		WHEN DW_SDT.DOTCODE NOT IN ('П118','П115','П196','П193','П197','П200','П116','П117','П117П','П000','П190','П170','П100','П126',
			'П126Х','П127','П127Х','П118ПЕ','П141','П126F','П126FX','П126Т','П126ТХ','П201') AND DW_SLD.DOCENIMEROSISDATE
			 IS NOT NULL THEN CAST((DW_SSD.STDQTYA * DW_SDT.TDTSIGN) AS numeric(18,3))
		ELSE CAST(0.0 as numeric(18,3))
END AS [ACT_QTYA],
CASE
		WHEN DW_SDT.DOTCODE NOT IN ('П118','П115','П196','П193','П197','П200','П116','П117','П117П','П000','П190','П170','П100','П126',
			'П126Х','П127','П127Х','П118ПЕ','П141','П126F','П126FX','П126Т','П126ТХ','П201') AND DW_SLD.DOCENIMEROSISDATE
			IS NOT NULL THEN CAST((DW_SSD.STDQTYB * DW_SDT.TDTSIGN) AS numeric(18,3))
		ELSE CAST(0.0 as numeric(18,3))
END AS [ACT_QTYB],
CASE
		 WHEN DW_SDT.DOTCODE NOT IN ('П118','П115','П196','П193','П197','П200','П116','П117','П117П','П000','П190','П170','П100','П126',
			'П126Х','П127','П127Х','П118ПЕ','П141','П126F','П126FX','П126Т','П126ТХ','П201') AND DW_SLD.DOCENIMEROSISDATE
			 IS NOT NULL THEN CAST(ISNULL((DW_UDT_STI_BI_SXESIS.KILA * (DW_SSD.STDQTYA * DW_SDT.TDTSIGN)), 0)
			AS numeric(18,3))
		ELSE CAST(0.0 as numeric(18,3))
 END AS ACT_QtyKGR,
CASE
		WHEN DW_SDT.DOTCODE IN ('П000','П100') AND DW_SLD.TDOEKTELESISDATE>(SELECT GETDATE() AS CurrentDateTime)THEN
CAST(ISNULL((DW_UDT_STI_BI_SXESIS.KILA * (DW_SSD.STDQTYA * DW_SDT.TDTSIGN)), 0) AS numeric (18,3))
		ELSE CAST(0.0 as numeric(18,3))
END AS ACT_OrdersKGR,
CASE
		 WHEN DW_SDT.DOTCODE NOT IN ('П118','П115','П196','П193','П197','П200','П116','П117','П117П','П000','П190','П170','П100','П126',
			'П126Х','П127','П127Х','П118ПЕ','П141','П126F','П126FX','П126Т','П126ТХ','П201') AND DW_SLD.DOCENIMEROSISDATE
			 IS NOT NULL THEN CAST(ISNULL((DW_SSD.STDQTYA * DW_SDT.TDTSIGN) * DW_UDT_STI_BI_SXESIS.TEMAXIA, 0)
			AS numeric(18,3))
		ELSE CAST(0.0 as numeric(18,3))
END AS ACT_QtyPCS,

FROM DW_SSD
INNER JOIN DW_SLD
		with(nolock) ON  DW_SLD.DOCID = DW_SSD.DOCID AND DW_SLD.DOCCANCELSTATUS = 'N' --AND DW_SSD.SUBSYSID = 14

INNER JOIN DW_SDT
		with(nolock) ON  DW_SDT.DOTID = DW_SLD.DOTID

INNER JOIN DW_ADR
		with(nolock) ON DW_SLD.ADRIDPROORISMOU = DW_ADR.ADRID

LEFT JOIN DW_UDT_ADR_EXTRA
		with(nolock) ON DW_ADR.ADRID = DW_UDT_ADR_EXTRA.ADRID

INNER JOIN DW_STI
		with(nolock) ON DW_SSD.MCIID = DW_STI.MCIID

INNER JOIN DW_CUS
		with(nolock) ON DW_CUS.TRAID = DW_SLD.TRAID

LEFT JOIN DW_UNI UNIA
		with(nolock) ON DW_STI.UNIIDA = UNIA.UNIID

LEFT JOIN DW_UNI UNIB
		with(nolock) ON DW_STI.UNIIDB = UNIB.UNIID

LEFT JOIN DW_UDT_STI_BI_SXESIS
		with(nolock) ON DW_STI.MCIID = DW_UDT_STI_BI_SXESIS.MCIID

INNER JOIN DW_LEE
		with(nolock) ON DW_LEE.LEEID = DW_CUS.LEEID
```

```
LEFT JOIN DW_CNT
        with(nolock) ON DW_LEE.CNTID = DW_CNT.CNTID

LEFT JOIN DW_UDT_LEE_CNT
        with(nolock) ON DW_LEE.LEEID = DW_UDT_LEE_CNT.LEEID

LEFT JOIN DW_CPM
        with(nolock) ON DW_CUS.PMTID = DW_CPM.PMTID

INNER JOIN DW_WRH
        with(nolock) ON DW_WRH.WRHID = DW_SLD.WRHID

LEFT JOIN DW_SPC
        with(nolock) ON DW_SSD.MCIID = DW_SPC.MCIID AND DW_SLD.PERID = DW_SPC.PERID AND DW_SLD.FIYID = DW_SPC.FIYID  AND
DW_SPC.WRHID = -1

LEFT JOIN (SELECT DW_CUSWITHOUTLNVIEW.TRAID, DW_CTG.CTGOUTLINE,DW_CTG.CTGDESCR
        FROM DW_CTG INNER JOIN DW_CUSWITHOUTLNVIEW with(nolock) ON      DW_CUSWITHOUTLNVIEW.CTGOUTLINE = DW_CTG.CTGOUTLINE
        WHERE CTGSUBSYSTEMS = '00100000000000000000000000000000000000000' AND (ctgoutlnorder LIKE '003%' OR ctgoutlnorder LIKE
        '0052%') AND CTGIDPARENT = 0) AS CTG ON DW_CUS.TRAID = CTG.TRAID

LEFT JOIN (SELECT DW_CTG.CTGOUTLINE, DW_CTG.CTGDESCR, DW_CTG.CTGID ITEMCTGID, DW_STIWITHOUTLNVIEW.MCIID
         FROM DW_STIWITHOUTLNVIEW INNER JOIN DW_CTG with(nolock) ON DW_CTG.CTGOUTLINE =       DW_STIWITHOUTLNVIEW.CTGoutline
WHERE DW_STIWITHOUTLNVIEW.ctgoutline LIKE '42%') AS ITEMCTG
        ON ITEMCTG.MCIID = DW_STI.MCIID

WHERE (DW_SDT.DOTCODE IN ('П000','П113','П113ПЕ','П118','П118ПЕ','П119','П120','П120П','П120Z','П123','П125','П125П','П125Z','П140',
'П170','П193','П194','П197','П198','П135','П131','П142','П141','П126','П126X','П126F','П126FX','П126T','П126TX','П127','П127X','П114','П114ЕП','П114Z','П115
','П195','П196','П199','П200','П116','П117','П117П','П130','П201')
```

SHORT BIOGRAPHY

Ioannis Lazos was born in Ioannina, Greece in 1975. He received his BSc degree from the Department of Computer Science of Hellenic Open University in 2016. At 2017, he became a MSc student at the Department of Computer Science and Engineering of University of Ioannina, working under the supervision of Associate Prof. Stergios Anastasiadis. He is a professional in Software Developing and Databases with more than 18 years of working experience. He holds the position of IT Manager at DODONI S.A. a Greek dairy company. His research interests are in the area of Databases, Data Management and Analysis.