

Retail Banking Markets (Data Warehouse)

Term
Project

Fariborz Norouzi

Summer 2018

Table of Contents

1. Introduction-----	1
2. Identifying Banking business opportunity or problem: -----	1
3. Perform feasibility study and gather user requirement: -----	2
4. Design (architecture, and databases) -----	3
4-1. Dimensions and Fact Tables-----	4
4-2. Define Measures in Fact Table-----	4
4-3. Create Dimensions Tables-----	6
4-4. Slowly Changing Dimension (SCD) -----	10
4-5. Date Dimension -----	11
4-6. Time Dimension -----	12
4-7. Create Fact Table -----	15
4-8. Create Index-----	17
5. Conclusion-----	22
6. Works Cited-----	22

Table of Figures

Fig 1: Retail Banking Entity Relational Data Model	4
Fig 2: Typical Retail Banking Data Warehouse Diagram	5
Fig 3: Sample Records of transaction dimension table	6
Fig 4: The records of Ref_Account_Type dimension.....	7
Fig 5: The records of Account dimension -----	7
Fig 6: The records of Ref_Transaction_Connection dimension-----	7
Fig 7: The records of Ref_Transaction_Type dimension-----	8
Fig 8: The records of Ref_Payment_Method_Type dimension-----	8
Fig 9: The records of Branch dimension-----	8
Fig 10: The records of Customer dimension-----	9
Fig 11: The records of Customer_Card dimension-----	9
Fig 12: The records of Address dimension-----	10
Fig 13: The sample records of Date dimension-----	11
Fig 14: The sample records of Time dimension-----	14
Fig 15: All dimension keys and measure attributes in Fact table-----	15

Fig 16: The sample records of Fact table-----	15
Fig 17: Rowstore and Column store index-----	17
Fig 18: Creating Non-Cluster Column store index-----	20
Fig 19: The Non-Cluster ColumnStore Index Scan operator-----	21
Fig 20: The Clustered Index Scan operator -----	21

List of Tables

Table 1: Different buckets time of 24 hours -----	15
Table 2: Different between clustered (CCI) and non-clustered (NCCI) columnstore index -----	19

1. Introduction:

In this study, I'm going to develop a data warehouse architecture for analysis of bank and financial transactions using a multi-dimensional model based on the business requirements. Data Warehouse is providing traceability throughout the layers of the information architecture. It provides the data design support through the provision of pre-aggregated fact structures at the center of the star schema design. In retail banks everyday large volume of data is produced as a result of activities, and as a by-product of various transactions. This information about the customers, merchants, employees and their activities can be used for business growth. Data is growing and technology is advancing to keep up with it. Banks are under tremendous economic pressure to succeed. For many banks a data warehouse is an appropriate tools for getting the right analytics and the right insight. The key steps in developing a financial Data Warehouse can be summarized as follows:

- Identifying Banking and Financial business opportunity or problem
- Perform feasibility study and gather user requirement
- Design (architecture, databases and applications)
- Develop data and application models
- Deployment and code data models

2. Identifying Banking business opportunity or problem

Financial institutions are facing a number of compliance challenges such as unstable financial markets and lower performing assets. The biggest challenge according the Basel Committee on Banking Standards (BCBS)¹ is effective risk data aggregation. So making the bank's risk reporting as usual is vital to create high quality, accurate reports to facilitate improved decision making. The bank's risk reporting requirements to enable the bank to measure its performance against its risk tolerance. This includes sorting, merging or breaking down sets of data. Based on BCBS reporting, weak IT and data architectures cause of critical risk issues on financial institute and bank. Therefore technology and data are creating opportunities to drive profitability and reduce risk.

¹ <https://www.bis.org/bcbs/>

To succeed in today's highly competitive financial business, financial companies and banks need to understand the future opportunities of the customers. Data warehouse can help them to better understand customer needs.

3. Perform feasibility study and gather user requirement

The financial services industry has made considerable progress in reducing cost and risk, as well as improvement of securities markets. Clearly defined business terms help standardization and communication within an organization, which these sort of roles led to identify user requirements. In fact user requirements specifies what the user expects the system to be able to do. In this case, users indeed included banks board, senior management, and external bodies such as the national supervisor. Before starting to design data warehouse schema, we need to understand current structure of the typical bank business operations. For this purpose, first we will review entity relationship diagram for general retail bank.

Retail Banks serve diverse customers, from individuals to businesses, and provide an essential financial function by linking depositors and borrowers. This case reviews the basics of retail banking, and financial markets data warehouse support for effective risk data aggregation.

The Retail Banks function is responsible for managing high-frequency tasks such as account maintenance, account terminations, and loan openings. Each customers can perform financial transactions and maintain relationships with personal bankers at brick and mortar retail branch locations.

Following diagram shows a third normal form of bank account data model which represent schema from an OLTP database for a retail banking system.

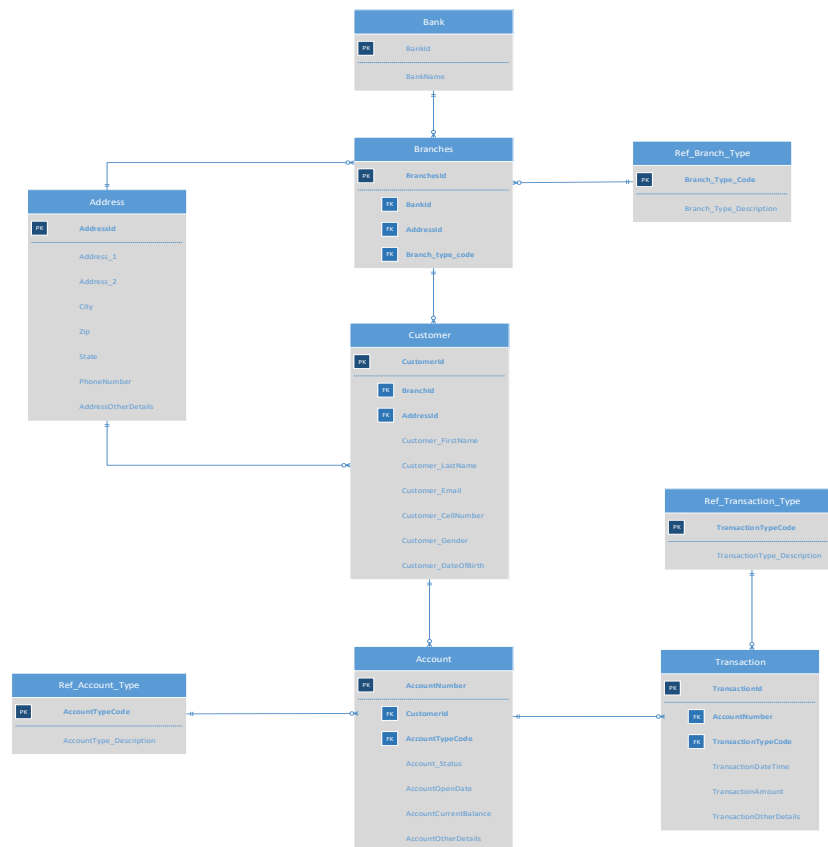


Fig 1: Retail Banking Entity Relational Data Model

4. Design (architecture, and databases)

Design a data warehouse required an understanding of both financial business processes and structures that make up the logical and physical design of a data warehouse. The combination of these skills results in the design of a data model which can Improve the decision-making process throughout the banking organization that most transactional systems can not address efficiency.

The Data Warehouse Model is a subject-oriented, integrated, time-variant and non-volatile collection of

data by storing current and historical data in one single place to use for reporting and data analysis in order to support of management's decision making process. It is derived from the analytical requirements and provides the data design support that is needed to transform the business requirements into business specific and is more easily understood by business users and are optimized for data querying through the provision of pre-aggregated fact structures. Creating an OLAP System from an OLTP System Traditional database design support accurate report but this may not produce a database that is easy to use for ad-hoc reporting.

4-1. Dimensions and Fact Tables

The dimension represents the database applications designed according to the needs of the individual user and it may also have descriptive attributes such as name of the customers, type of account etc. The data warehousing is that the data stored for business analysis can build a systemic data storage environment to support very sophisticated online analysis including multi- dimensional analysis.

Fact table in this study bring dimension keys and measures to facilitate banking business process analysis at a specific grain. Measures in fact table is identified by analyzing what a retail banking business wants to track about specific business process. The primary key of fact table determines the uniqueness of each row and it is composite key made up of all its foreign keys. Also I created a surrogate key as a fact table's primary key which is an auto-incremental numeric field. It is necessary to reduce the size of the table's clustered index when the primary key is based on many foreign keys.

The table dimensionality reflects the granularity of the fact table. The relationship between a fact table's foreign keys and the corresponding dimension primary key is one to many relationship.

4-2. Define Measures in Fact Table

The following are key business requirements for the bank based on data. It needs to allow the bank to produce a variety of different reports such as:

- Transaction details, broken down by date (day/month/year), time of day (morning, afternoon, evening, night, midnight), transaction type, customer details (age, gender, city), and branch details (city, suburb and etc.).
- Average as well as total transaction values and number/value/average of deposits and withdrawals.
- Average balances, broken down by date, customer and branch details.
- Ratio of total transaction value in a month to average balance (over several months or even years).

Design a data warehouse schema and data cube based on the OLTP database that will allow the reports above to be easily produced. Following diagram shows up star database design, because consists of one

fact table referencing any number of dimension tables, which indicates what data items will be in each of the dimension and fact tables.



Fig 2: Typical Retail Banking Data Warehouse Diagram

There are three types of measures inclusive additive or fully-aggregable, semi-additive or semi-aggregable and non-additive or non-aggregable facts. Additive measures are numeric value that can be summed up by all the dimensions in the fact table, for instance Average Transaction Amount. In the other hand semi-additive measures are numeric values that can be summed up by some dimensions in the fact table except time. For example Account Balance can determine the ending balance of an account at the end of the month, but adding up these monthly account balance do not make sense.

4-3. Create Dimensions Tables

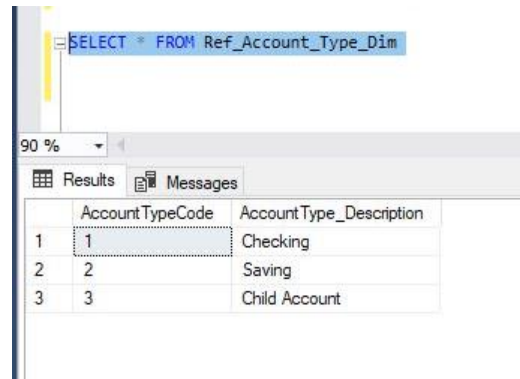
The following assumptions are considered in Data Warehouse design.

- Each Customers can have one or more accounts at a particular bank branch.
- Each Branches be able to process and record all their customers' transactions.

TransactionId	TransactionAmount	TransactionDateTime
307	140.00	2017-03-27 16:23:37.120
308	64.00	2017-03-28 15:25:36.457
309	563.08	2017-03-29 11:28:49.357
310	65.46	2017-03-30 09:23:37.120
311	8.42	2017-03-31 15:12:09.157
312	59.68	2017-04-01 10:28:49.357
313	15.50	2017-04-02 09:23:37.120
314	24.00	2017-04-03 14:25:36.457
315	19.00	2017-04-04 09:28:49.357
316	76.00	2017-04-05 09:23:37.120
317	86.52	2017-04-06 10:13:36.457
318	28.00	2017-04-07 16:18:39.357
319	45.00	2017-04-08 09:23:37.120
320	17.25	2017-04-09 15:25:36.457
321	47.23	2017-04-09 09:28:49.357
322	136.80	2017-04-10 09:23:37.120
323	65.00	2017-04-11 13:15:36.457
324	67.00	2017-04-12 09:28:49.357
325	6.67	2017-04-13 12:23:37.120
326	352.87	2017-04-14 10:25:36.457
327	1345.00	2017-04-15 11:28:49.357
328	58.85	2017-04-16 13:23:37.120
329	111.00	2017-04-17 09:25:36.457
330	29.68	2017-04-18 13:28:49.357

Fig 3: Sample Records of transaction dimension table

- The balance in the corresponding account is updated when a transaction is processed.
- Each customer is linked to census data for the area they live in.
- In Account, account Type indicates checking, saving, and child account.

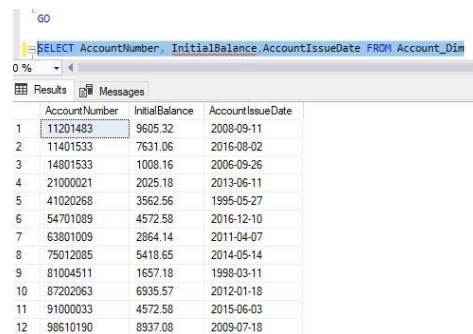


```
SELECT * FROM Ref_Account_Type_Dim
```

	AccountTypeCode	AccountType_Description
1	1	Checking
2	2	Saving
3	3	Child Account

Fig 4: The records of Ref_Account_Type dimension

- Assumed on start date of stored data warehouse processing, current balance of each customer is initial balance and it is updated whenever a transaction is processed.

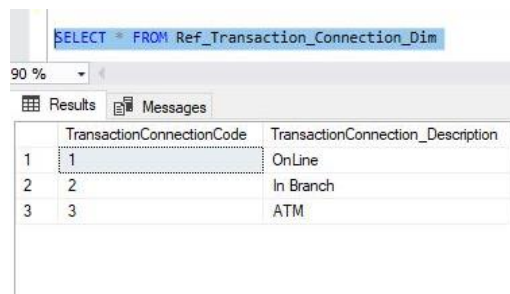


```
SELECT AccountNumber, InitialBalance, AccountIssueDate FROM Account_Dim
```

	AccountNumber	InitialBalance	AccountIssueDate
1	11201483	9605.32	2008-09-11
2	11401533	7631.06	2016-08-02
3	14801533	1008.16	2006-09-26
4	21000021	2025.18	2013-06-11
5	41020268	3562.56	1995-05-27
6	54701089	4572.58	2016-12-10
7	63801009	2864.14	2011-04-07
8	75012085	5418.65	2014-05-14
9	81004511	1657.18	1998-03-11
10	87202063	6935.57	2012-01-18
11	91000033	4572.58	2015-06-03
12	98610190	8937.08	2009-07-18

Fig 5: The records of Account dimension

- In Transaction: Transaction Connection indicates on-line, in branch, ATM.

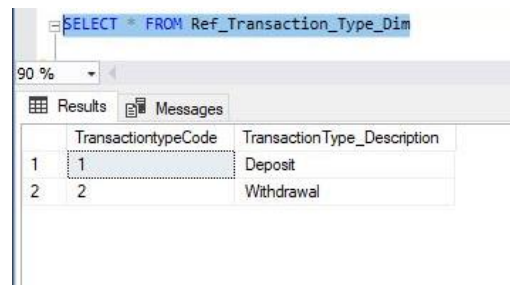


```
SELECT * FROM Ref_Transaction_Connection_Dim
```

	TransactionConnectionCode	TransactionConnection_Description
1	1	OnLine
2	2	In Branch
3	3	ATM

Fig 6: The records of Ref_Transaction_Connection dimension

- Transaction Type tells whether it is a deposit or a withdrawal.

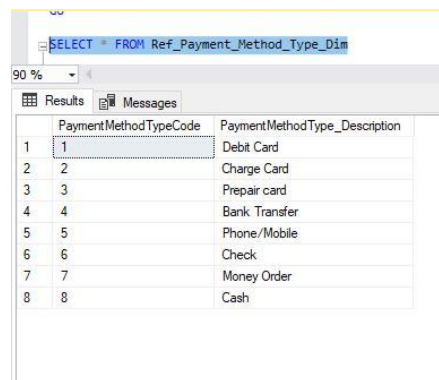


```
SELECT * FROM Ref_Transaction_Type_Dim
```

	TransactionTypeCode	TransactionType_Description
1	1	Deposit
2	2	Withdrawal

Fig 7: The records of Ref_Transaction_Type dimension

- The payment methods types are debit cards, charge cards, prepaid cards, direct debit, bank transfers, phone and mobile payments, checks, money orders and cash payments.

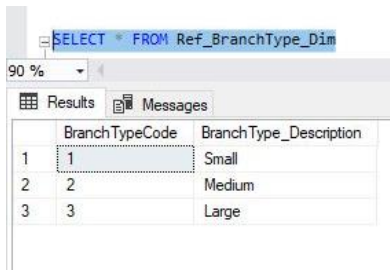


```
SELECT * FROM Ref_Payment_Method_Type_Dim
```

	PaymentMethodTypeCode	PaymentMethodType_Description
1	1	Debit Card
2	2	Charge Card
3	3	Prepaid card
4	4	Bank Transfer
5	5	Phone/Mobile
6	6	Check
7	7	Money Order
8	8	Cash

Fig 8: The records of Ref_Payment_Method_Type dimension

- In Branch: Branch Type indicates large, medium, and small branch.

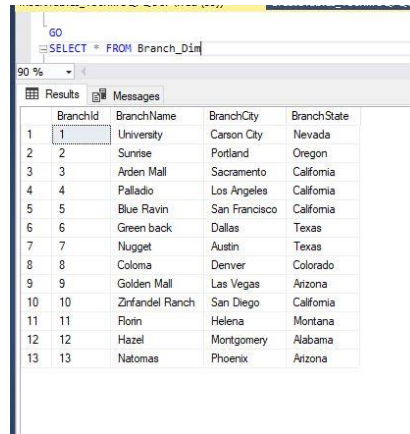


```
SELECT * FROM Ref_BranchType_Dim
```

	BranchTypeCode	BranchType_Description
1	1	Small
2	2	Medium
3	3	Large

Fig 8: The records of Ref_Branch_Type dimension

And branch dimension represents different locations of branches.

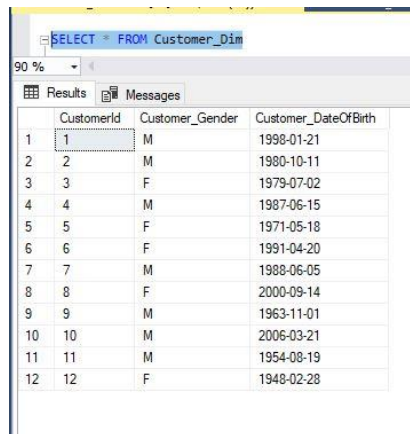


The screenshot shows a SQL query window with the command `SELECT * FROM Branch_Dim`. Below the query, the results are displayed in a table with 5 columns: BranchId, BranchName, BranchCity, and BranchState. The table contains 13 rows of data.

	BranchId	BranchName	BranchCity	BranchState
1	1	University	Carson City	Nevada
2	2	Sunrise	Portland	Oregon
3	3	Arden Mall	Sacramento	California
4	4	Palladio	Los Angeles	California
5	5	Blue Ravin	San Francisco	California
6	6	Green back	Dallas	Texas
7	7	Nugget	Austin	Texas
8	8	Coloma	Denver	Colorado
9	9	Golden Mall	Las Vegas	Arizona
10	10	Zinfandel Ranch	San Diego	California
11	11	Florin	Helena	Montana
12	12	Hazel	Montgomery	Alabama
13	13	Natomas	Phoenix	Arizona

Fig 9: The records of Branch dimension

- Customer dimension indicates gender and date of birth for each individual customer.

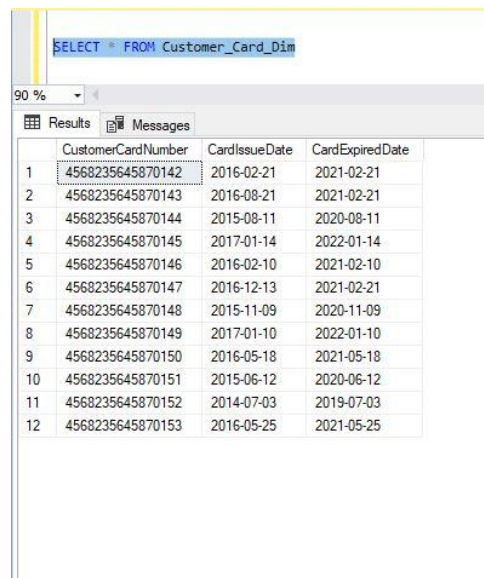


The screenshot shows a SQL query window with the command `SELECT * FROM Customer_Dim`. Below the query, the results are displayed in a table with 3 columns: CustomerId, Customer_Gender, and Customer_DateOfBirth. The table contains 12 rows of data.

	CustomerId	Customer_Gender	Customer_DateOfBirth
1	1	M	1998-01-21
2	2	M	1980-10-11
3	3	F	1979-07-02
4	4	M	1987-06-15
5	5	F	1971-05-18
6	6	F	1991-04-20
7	7	M	1988-06-05
8	8	F	2000-09-14
9	9	M	1963-11-01
10	10	M	2006-03-21
11	11	M	1954-08-19
12	12	F	1948-02-28

Fig 10: The records of Customer dimension

- Customer Card dimension shows sixteen digit customer's card number with issued and expiry date.



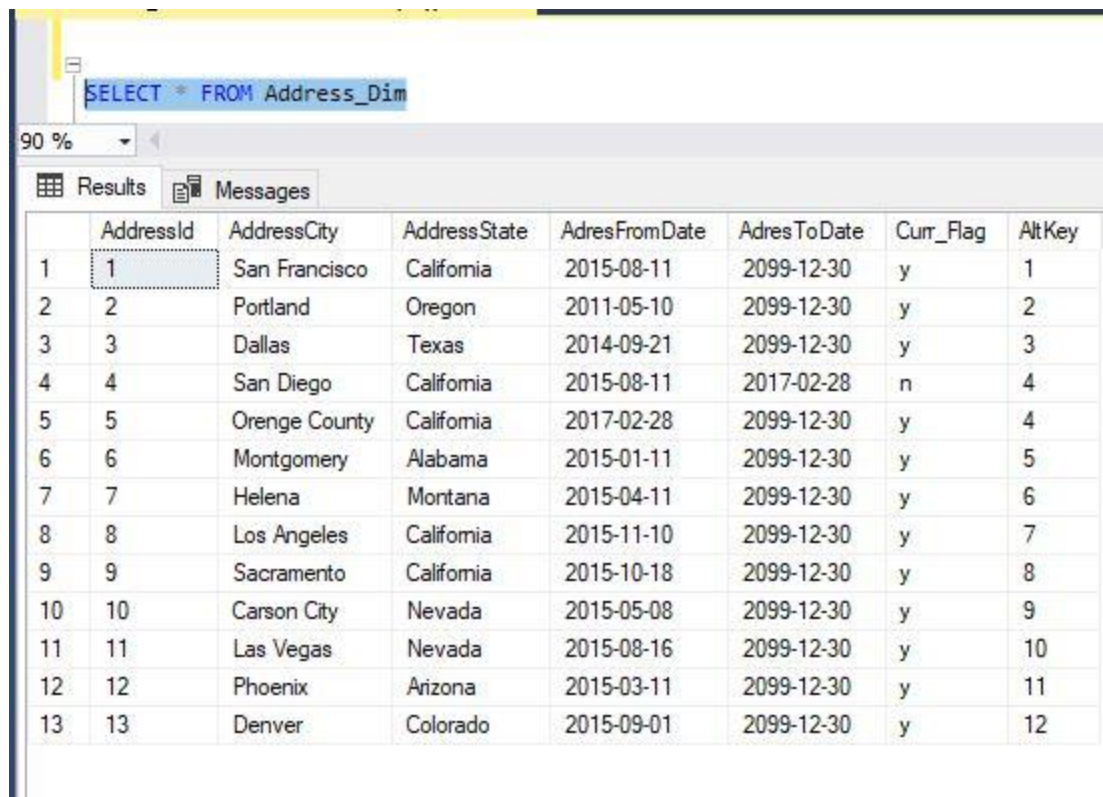
	CustomerCardNumber	CardIssueDate	CardExpiredDate
1	4568235645870142	2016-02-21	2021-02-21
2	4568235645870143	2016-08-21	2021-02-21
3	4568235645870144	2015-08-11	2020-08-11
4	4568235645870145	2017-01-14	2022-01-14
5	4568235645870146	2016-02-10	2021-02-10
6	4568235645870147	2016-12-13	2021-02-21
7	4568235645870148	2015-11-09	2020-11-09
8	4568235645870149	2017-01-10	2022-01-10
9	4568235645870150	2016-05-18	2021-05-18
10	4568235645870151	2015-06-12	2020-06-12
11	4568235645870152	2014-07-03	2019-07-03
12	4568235645870153	2016-05-25	2021-05-25

Fig 11: The records of Customer_Card dimension

4-4. Slowly Changing Dimension (SCD)

Slowly Changing Dimension (SCD) according the definition is a dimension that stores and manages both current and historical data over time in a data warehouse. In this case I have considered type 2 SCDs (Creating another dimension record) in Address dimension related to the customers.

A Type 2 SCD retains the full history of values. In order to track changing record, I added four new fields which included “AdresFromDate”, “AdresToDate”, “Curr_Flag” and “AltKey”. When the value of a chosen attribute changes (Address ID 4), the current record is closed (Curr_Flag set up to ‘n’ and generated same ‘AltKey’ number). A new record is generated with the start data values equal to end date of old record and this new record becomes the current record with a big and too far end date (Curr_Flag set up to ‘y’ and AdresToDate assigned 2099-12-30). Each record contains the effective time and expiration time to identify the time period between which the record was active.



The screenshot shows a SQL query window with the text 'SELECT * FROM Address_Dim'. Below the query, the 'Results' tab is active, displaying a table with 13 rows and 8 columns. The columns are AddressId, AddressCity, AddressState, AdresFromDate, AdresToDate, Curr_Flag, and AltKey. The first row is highlighted with a dotted border.

	AddressId	AddressCity	AddressState	AdresFromDate	AdresToDate	Curr_Flag	AltKey
1	1	San Francisco	California	2015-08-11	2099-12-30	y	1
2	2	Portland	Oregon	2011-05-10	2099-12-30	y	2
3	3	Dallas	Texas	2014-09-21	2099-12-30	y	3
4	4	San Diego	California	2015-08-11	2017-02-28	n	4
5	5	Orenge County	California	2017-02-28	2099-12-30	y	4
6	6	Montgomery	Alabama	2015-01-11	2099-12-30	y	5
7	7	Helena	Montana	2015-04-11	2099-12-30	y	6
8	8	Los Angeles	California	2015-11-10	2099-12-30	y	7
9	9	Sacramento	California	2015-10-18	2099-12-30	y	8
10	10	Carson City	Nevada	2015-05-08	2099-12-30	y	9
11	11	Las Vegas	Nevada	2015-08-16	2099-12-30	y	10
12	12	Phoenix	Arizona	2015-03-11	2099-12-30	y	11
13	13	Denver	Colorado	2015-09-01	2099-12-30	y	12

Fig 12: The records of Address dimension

4-5. Date dimension

Date dimension plays an important role in banking and financial data warehouse designing. It provides the ability to study data by grouping them using various fields of date dimension. For example:

If I want to analyze my data of average balance by each month of the year, or show total transaction by each quarter of the year, or showing which days total transaction takes place more in the entire year or month. In order to achieve this purpose we need to create and fill their Date Dimension with various values of Date, Date Keys, Day Type, Day Name of Week, Month, Month Name, Quarter, etc. Following script generates date dimension with Fiscal Calendar fields like Fiscal Year, and fiscal Quarter.

```

DECLARE @StartDate datetime
DECLARE @EndDate datetime
SET @StartDate = '02/01/2017'
SET @EndDate = '05/30/2017'
DECLARE @LoopDate datetime
SET @LoopDate = @StartDate
WHILE @LoopDate <= @EndDate
BEGIN
INSERT INTO dbo.DimDate VALUES
(
CAST(CONVERT(VARCHAR(8), @LoopDate, 112) AS int), --datekey

```

```

@LoopDate,
Year(@LoopDate), --Calendar Year
datepart(qq,@LoopDate), --Calendar Quarter
Month(@LoopDate), -- Month number of Year
datename(mm, @LoopDate), --Month name
day(@LoopDate), --Day number of Month
datepart(dw, @LoopDate), --Day number of Week
datename(dw, @LoopDate), --Day name of Week
CASE WHEN Month(@LoopDate) < 7 THEN Year(@LoopDate)
ELSE Year(@LoopDate) + 1
END, --Fiscal year
CASE WHEN Month(@LoopDate) IN (1, 2, 3) THEN 3
      WHEN Month(@LoopDate) IN (4, 5, 6) THEN 4
      WHEN Month(@LoopDate) IN (7, 8, 9) THEN 1
      WHEN Month(@LoopDate) IN (10, 11, 12) THEN 2
      END --Fiscal Quarter
)
SET @LoopDate = DateAdd(dd, 1, @LoopDate)
END

```

This script create date dimension data and populate it with all standard values.

	DateKey	DateAltKey	CalendarYear	CalendarQuarter	MonthOfYear	MonthName	DayOfMonth	DayOfWeek	DayName	FiscalYear	FiscalQuarter
55	20170327	2017-03-27 00:00:00.000	2017	1	3	March	27	2	Monday	2017	3
56	20170328	2017-03-28 00:00:00.000	2017	1	3	March	28	3	Tuesday	2017	3
57	20170329	2017-03-29 00:00:00.000	2017	1	3	March	29	4	Wednesday	2017	3
58	20170330	2017-03-30 00:00:00.000	2017	1	3	March	30	5	Thursday	2017	3
59	20170331	2017-03-31 00:00:00.000	2017	1	3	March	31	6	Friday	2017	3
60	20170401	2017-04-01 00:00:00.000	2017	2	4	April	1	7	Saturday	2017	4
61	20170402	2017-04-02 00:00:00.000	2017	2	4	April	2	1	Sunday	2017	4
62	20170403	2017-04-03 00:00:00.000	2017	2	4	April	3	2	Monday	2017	4
63	20170404	2017-04-04 00:00:00.000	2017	2	4	April	4	3	Tuesday	2017	4
64	20170405	2017-04-05 00:00:00.000	2017	2	4	April	5	4	Wednesday	2017	4
65	20170406	2017-04-06 00:00:00.000	2017	2	4	April	6	5	Thursday	2017	4
66	20170407	2017-04-07 00:00:00.000	2017	2	4	April	7	6	Friday	2017	4
67	20170408	2017-04-08 00:00:00.000	2017	2	4	April	8	7	Saturday	2017	4
68	20170409	2017-04-09 00:00:00.000	2017	2	4	April	9	1	Sunday	2017	4
69	20170410	2017-04-10 00:00:00.000	2017	2	4	April	10	2	Monday	2017	4
70	20170411	2017-04-11 00:00:00.000	2017	2	4	April	11	3	Tuesday	2017	4
71	20170412	2017-04-12 00:00:00.000	2017	2	4	April	12	4	Wednesday	2017	4
72	20170413	2017-04-13 00:00:00.000	2017	2	4	April	13	5	Thursday	2017	4
73	20170414	2017-04-14 00:00:00.000	2017	2	4	April	14	6	Friday	2017	4
74	20170415	2017-04-15 00:00:00.000	2017	2	4	April	15	7	Saturday	2017	4
75	20170416	2017-04-16 00:00:00.000	2017	2	4	April	16	1	Sunday	2017	4
76	20170417	2017-04-17 00:00:00.000	2017	2	4	April	17	2	Monday	2017	4
77	20170418	2017-04-18 00:00:00.000	2017	2	4	April	18	3	Tuesday	2017	4

Fig 13: The sample records of Date dimension

4-6. Time Dimension

Time Dimension is another useful tools to create data warehouse. Creating and populating Time data for the entire day with various time buckets can enable us to analysis of data using these time buckets and can do study of trend over the entire day.

Following stored procedure, generates time dimension with seconds accurately over 24 hours.

```

CREATE PROCEDURE dbo.FillDimTime
AS
BEGIN

```

```

DECLARE @Size INTEGER
DECLARE @hour INTEGER
DECLARE @minute INTEGER
DECLARE @second INTEGER
DECLARE @k INTEGER
DECLARE @TimeAltKey INTEGER
DECLARE @TimeInSeconds INTEGER
DECLARE @Time30 varchar(25)
DECLARE @Hour30 varchar(4)
DECLARE @Minute30 varchar(4)
DECLARE @Second30 varchar(4)
DECLARE @HourBucket varchar(15)
DECLARE @HourBucketGroupKey int
DECLARE @DayTimeBucket varchar(100)
DECLARE @DayTimeBucketGroupKey int
SET @hour = 0
SET @minute = 0
SET @second = 0
SET @k = 0
SET @TimeAltKey = 0
WHILE(@hour<= @Size )
BEGIN
IF (@hour <10 )
BEGIN
SET @Hour30 = '0' + cast( @hour as varchar(10))
END
ELSE
BEGIN
SET @Hour30 = @hour
END
--Create Hour Bucket Value
SET @HourBucket= @Hour30+':00' + '-' +@Hour30+':59'
WHILE(@minute <= 59)
BEGIN
WHILE(@second <= 59)
BEGIN
SET @TimeAltKey = @hour *10000 +@minute*100 +@second
SET @TimeInSeconds =@hour * 3600 + @minute *60 +@second
If @minute <10
BEGIN
SET @Minute30 = '0' + cast ( @minute as varchar(10) )
END
ELSE
BEGIN
SET @Minute30 = @minute
END
IF @second <10
BEGIN
SET @Second30 = '0' + cast (@second as varchar(10))
END
ELSE
BEGIN
SET @Second30 = @second
END
--Concatenate values for Time30
SET @Time30 = @Hour30 +':'+@Minute30 +':'+@Second30
--DayTimeBucketGroupKey can be used in Sorting of DayTime Bucket In proper Order
SELECT @DayTimeBucketGroupKey =

```



```

CASE
WHEN (@TimeAltKey >= 00000 AND @TimeAltKey <= 25959) THEN 0
WHEN (@TimeAltKey >= 30000 AND @TimeAltKey <= 65959) THEN 1
WHEN (@TimeAltKey >= 70000 AND @TimeAltKey <= 85959) THEN 2
WHEN (@TimeAltKey >= 90000 AND @TimeAltKey <= 115959) THEN 3
WHEN (@TimeAltKey >= 120000 AND @TimeAltKey <= 135959) THEN 4
WHEN (@TimeAltKey >= 140000 AND @TimeAltKey <= 155959) THEN 5
WHEN (@TimeAltKey >= 50000 AND @TimeAltKey <= 175959) THEN 6
WHEN (@TimeAltKey >= 180000 AND @TimeAltKey <= 235959) THEN 7
WHEN (@TimeAltKey >= 240000) THEN 8
END
--print @DayTimeBucketGroupKey
-- DayTimeBucket Time Divided in Specific Time Zone
-- So Data can Be Grouped as per Bucket for Analyzing as per time of day
SELECT @DayTimeBucket =
CASE
WHEN (@TimeAltKey >= 00000 AND @TimeAltKey <= 25959)
THEN 'Late Night (00:00 AM To 02:59 AM)'
WHEN (@TimeAltKey >= 30000 AND @TimeAltKey <= 65959)
THEN 'Early Morning(03:00 AM To 6:59 AM)'
WHEN (@TimeAltKey >= 70000 AND @TimeAltKey <= 85959)
THEN 'AM Peak (7:00 AM To 8:59 AM)'
WHEN (@TimeAltKey >= 90000 AND @TimeAltKey <= 115959)
THEN 'Mid Morning (9:00 AM To 11:59 AM)'
WHEN (@TimeAltKey >= 120000 AND @TimeAltKey <= 135959)
THEN 'Lunch (12:00 PM To 13:59 PM)'
WHEN (@TimeAltKey >= 140000 AND @TimeAltKey <= 155959)
THEN 'Mid Afternoon (14:00 PM To 15:59 PM)'
WHEN (@TimeAltKey >= 50000 AND @TimeAltKey <= 175959)
THEN 'PM Peak (16:00 PM To 17:59 PM)'
WHEN (@TimeAltKey >= 180000 AND @TimeAltKey <= 235959)
THEN 'Evening (18:00 PM To 23:59 PM)'
WHEN (@TimeAltKey >= 240000) THEN 'Previous Day Late Night _
(24:00 PM to '+cast( @Size as varchar(10)) +':00 PM )'
END
-- print @DayTimeBucket
INSERT into DimTime (TimeKey,TimeAltKey,[Time30] ,[Hour30] ,
[MinuteNumber],[SecondNumber],[TimeInSeconds],[HourlyBucket],
DayTimeBucketGroupKey,DayTimeBucket)
VALUES (@k,@TimeAltKey ,@Time30 ,@hour ,@minute,@Second ,
@TimeInSeconds,@HourBucket,@DayTimeBucketGroupKey,@DayTimeBucket )
SET @second = @second + 1
SET @k = @k + 1
END
SET @minute = @minute + 1
SET @second = 0
END
SET @hour = @hour + 1
SET @minute =0
END
END
Go

```

This stored procedure generates time dimension data and populate it with all standard values.

TimeKey	TimeRKey	Time30	Hour30	MinuteNumber	SecondNumber	TimeInSecond	HourlyBucket	DayTimeBucketGroupKey	DayTimeBucket
86379	86378	23:59:38	23	59	38	86378	23:00:23:59	7	Evening (18:00 PM To 23:59 PM)
86380	86379	23:59:39	23	59	39	86379	23:00:23:59	7	Evening (18:00 PM To 23:59 PM)
86381	86380	23:59:40	23	59	40	86380	23:00:23:59	7	Evening (18:00 PM To 23:59 PM)
86382	86381	23:59:41	23	59	41	86381	23:00:23:59	7	Evening (18:00 PM To 23:59 PM)
86383	86382	23:59:42	23	59	42	86382	23:00:23:59	7	Evening (18:00 PM To 23:59 PM)
86384	86383	23:59:43	23	59	43	86383	23:00:23:59	7	Evening (18:00 PM To 23:59 PM)
86385	86384	23:59:44	23	59	44	86384	23:00:23:59	7	Evening (18:00 PM To 23:59 PM)
86386	86385	23:59:45	23	59	45	86385	23:00:23:59	7	Evening (18:00 PM To 23:59 PM)
86387	86386	23:59:46	23	59	46	86386	23:00:23:59	7	Evening (18:00 PM To 23:59 PM)
86388	86387	23:59:47	23	59	47	86387	23:00:23:59	7	Evening (18:00 PM To 23:59 PM)
86389	86388	23:59:48	23	59	48	86388	23:00:23:59	7	Evening (18:00 PM To 23:59 PM)
86390	86389	23:59:49	23	59	49	86389	23:00:23:59	7	Evening (18:00 PM To 23:59 PM)
86391	86390	23:59:50	23	59	50	86390	23:00:23:59	7	Evening (18:00 PM To 23:59 PM)
86392	86391	23:59:51	23	59	51	86391	23:00:23:59	7	Evening (18:00 PM To 23:59 PM)
86393	86392	23:59:52	23	59	52	86392	23:00:23:59	7	Evening (18:00 PM To 23:59 PM)
86394	86393	23:59:53	23	59	53	86393	23:00:23:59	7	Evening (18:00 PM To 23:59 PM)
86395	86394	23:59:54	23	59	54	86394	23:00:23:59	7	Evening (18:00 PM To 23:59 PM)
86396	86395	23:59:55	23	59	55	86395	23:00:23:59	7	Evening (18:00 PM To 23:59 PM)
86397	86396	23:59:56	23	59	56	86396	23:00:23:59	7	Evening (18:00 PM To 23:59 PM)
86398	86397	23:59:57	23	59	57	86397	23:00:23:59	7	Evening (18:00 PM To 23:59 PM)
86399	86398	23:59:58	23	59	58	86398	23:00:23:59	7	Evening (18:00 PM To 23:59 PM)
86400	86399	23:59:59	23	59	59	86399	23:00:23:59	7	Evening (18:00 PM To 23:59 PM)

Fig 14: The sample records of Time dimension

Day time bucket divided to seven categories that represents different parts of 24 hours. The following table shows the detail of this segmentation.

Group Key	Bucket Name	Bucket Time
0	Late Night	00:00 AM TO 02:59 AM
1	Early Morning	03:00 AM TO 06:59 AM
2	AM Peak	07:00 AM TO 08:59 AM
3	Mid Morning	09:00 AM TO 11:59 AM
4	Lunch	12:00 PM TO 13:59 PM
5	Mid Afternoon	14:00 PM TO 15:59 PM
6	PM Peak	16:00 PM TO 17:59 PM
7	Evening	18:00 PM TO 23:59 PM

Table 1: Different buckets time of 24 hours

4-7. Create Fact Table

And finally fact table is set of important data warehouse structure that store dimension keys and measures to keep track of business process. Following picture shows all dimension keys and measures that used for designing typical retail banking fact table. For this study, I have defined five measures attributes which included Current Balance, Average Balance, Count Transaction, Total Transactions and Average Transaction. It is worth mentioning that total transaction calculated separately for each type of transaction (deposit / Withdraw) related to each individual customer, but average transaction considers both type of transaction type for each customer.

DW_Fact	
PK	BankId
FK	BranchId
FK	BranchTypeCode
FK	CustomerId
FK	AccountNumber
FK	AccountTypeCode
FK	AccountStatusCode
FK	TransactionId
FK	TransactionTypeCode
FK	TransactionConnectionCode
FK	PaymentMethodTypeCode
FK	AddressId
FK	CustomerCardNumber
FK	DateKey
CurrentBalance	
AverageBalance	
Count Transaction	
TotalTransactionAmount	
Average TransactionAmount	

Fig 15: All dimension keys and measure attributes in Fact table

All data of fact table, are coming from group of processes intended for data integration of dimension table's records in Microsoft excel. In other words, ETL processing for extracting, transforming and loading data to Data Warehouse were done by spreadsheets of excel.

CreateTables_TScript...GQFQ08P(fred (54))* X

Select * from DW_Fact

90 %

Results Messages

TransactionCode	PaymentMethodTypeCode	AddressId	CustomerCardNumber	DateKey	TimeKey	CurrentBalance	AverageBalance	Counts Trans	TotalTransactionAmount	Average TransactionAmount
102	7	7	4568235645870147	20170509	36529	12098.19	11821.11	8	128.32	972.78
103	1	6	4568235645870146	20170510	33817	4146.93	4352.56	8	2084.06	447.97
104	4	12	4568235645870153	20170511	44736	12159.36	6762.41	8	7586.78	948.35
105	1	1	4568235645870142	20170512	52129	13939.76	11927.19	17	4737.27	302.36
106	8	10	4568235645870150	20170513	37417	863.88	766.21	9	3101.16	425.67
107	1	2	4568235645870143	20170514	37536	8200.64	7940.08	9	1715.28	317.88
108	4	9	4568235645870150	20170515	34129	8916.45	7492.60	9	3497.80	388.64
109	8	8	4568235645870149	20170516	44617	5134.38	4153.37	9	2483.76	299.70
110	1	1	4568235645870142	20170517	59136	13923.40	12038.08	18	419.19	286.47
111	1	3	4568235645870144	20170518	41329	4740.38	3191.32	9	4223.06	469.23
112	8	13	4568235645870153	20170519	41017	9247.33	9488.82	9	444.96	133.35
113	8	11	4568235645870151	20170520	62736	8091.83	6970.45	9	2002.23	316.47
114	7	5	4568235645870145	20170521	37729	4367.13	3020.30	9	2535.95	303.33
115	7	7	4568235645870147	20170522	51817	12086.19	11850.56	9	140.32	866.03
116	1	6	4568235645870146	20170523	66216	4316.93	4348.60	9	2254.06	417.08
117	4	12	4568235645870153	20170524	55729	13020.44	7457.74	9	8447.86	938.65
118	8	13	4568235645870153	20170525	55417	8157.65	9355.70	10	1534.64	228.99
119	8	11	4568235645870151	20170526	59136	8445.83	7117.99	10	2356.23	320.22
120	7	5	4568235645870145	20170527	52129	4400.21	3158.29	10	2569.03	276.30
121	7	7	4568235645870147	20170527	48217	12226.19	11888.12	10	7793.93	793.43
122	1	6	4568235645870146	20170528	51936	4380.93	4351.83	10	2318.06	381.78
123	4	12	4568235645870153	20170529	70129	12457.36	7957.70	10	563.08	901.09

Query executed successfully. LAPTOP-MGQFQ08P (13.0 SP1) | LAPTOP-MGQFQ08P(fred (54)) | Bank_DW | 00:00:00 | 123 rows

Fig 16: The sample records of Fact table

4-8. Create Index

SQL Server would resort to do a table scan which it means a full read of all rows in a table. A table scan is the most inefficient method of reading data that cause of increasing memory, disk and CPU resource utilization with to perform slowly query. Therefore, for accessing to quickly locate data without having to search every row in table every time, we need a data structure which is used to quickly locate and access the data in a database table which named indexing.

Designing an indexing solution requires the right number of indexes in the data warehouse. Insufficient indexing causes queries to run slow and the other hand too many indexes cause data loads to run slower and database size to increase. Finding the right balance between index performance and index storage (query speed vs update cost) is a very important task for designing a proper data warehouse workload. There are two main type of indexes that SQL Server supports. First type is rowstore indexes which store or view data horizontally based on rows of data. Second main type of index is columnstore indexes where store or view data vertically based on column value. It is a new type of index that introduced in SQL Server 2012.

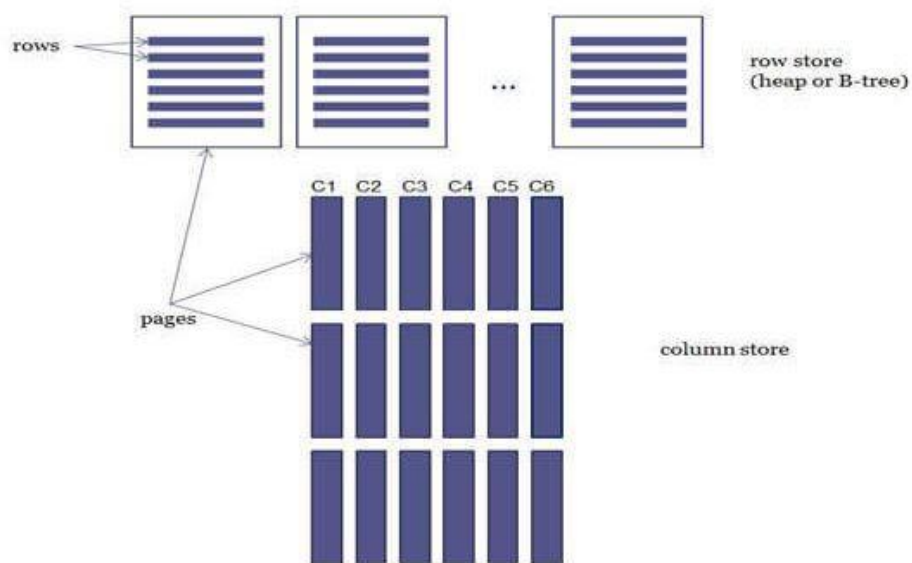


Fig 17: Rowstore and Column store index

There are some advantage and disadvantage of using columnstore indexes over rowstoreindex as follows.

Advantages of using SQL Server ColumnStore Indexes:

- Faster query performance for common data warehouse queries as only required columns/pages in the query are fetched from disk

- Data is stored in a highly compressed form (Vertipaq technology) to reduce the storage space
- Frequently accessed columns (pages that contains data for these columns) remain in memory because a high ratio of compression is used in the pages and less pages are involved
- Enhanced query processing/optimization and execution feature (new Batch Operator or batch mode processing) improves common data warehouse queries' performance

Some disadvantage and Limitations of SQL Server ColumnStore Indexes are:

- Column store index cannot be created on a view or indexed view.
- Column store index cannot include Sparse Column.(1024 columns max)
- Column store index cannot be unique or filtered and cannot be used as primary or foreign key.
- Column store index cannot be created using ASC/DESC or INCLUDE keywords.
- Column store index definition cannot be changed using ALTER INDEX statment. We need to drop and re-create instead.
- ColumnStore Index cannot be created on table which uses features like Replication, Change Tracking, Change Data Capture and Filestream

Rowstore indexes are based on a balanced Tree (B-Tree) structure to find rows of data in a quick and efficient manner. It can be define as cluster index key or non-clustered row index. The clustered index key is usually the table's primary key. Although we can define it, without primary key.

Columnstore indexes are in memory index that provide increased performance benefits over traditional rowstore indexes in data warehousing workloads. SQL Server 2016 provides clustered (CCI) and non-clustered (NCCI) columnstore index. Both indexes are organized as columns but NCCI is created on an existing rowstore table, while a table with CCI does not have a rowstore table. Following table shows lists the main differences between NCCI and CCI.

Differences	NCCI	CCI
Index Columns	Need to specify a list of columns to create NCCI on. A typical usage is to only include columns needed for analytics. If underlying table has <u>has</u> an unsupported column, for example, a spatial datatype or LOB column, you can exclude it	All columns are included. For example, if the underlying table has an unsupported column, for example LOB, you will be required to vertically partition the table. Note, SQL Server team is working to support large datatypes post SQL Server 2016
Storage	Creating NCCI does not save any storage. In fact, it can potentially add approximately 10% additional storage as it is a new index. In many customer cases, once you create an NCCI, you may be able to drop one or more btree nonclustered indices which were created primarily for querying large set of rows.	Creating CCI can compress the data 10x. Of course, the compression achieved will depend upon the schema and the data but 10x is what you can expect. If table was originally PAGE compressed, you can expect around 5x compression
Workload	NCCI is designed to be used for workloads where you have a mix of transactional and analytics workload. In fact, NCCI is the preferred index for real-time operational analytics in SQL Server 2016	CCI is designed for DW workloads. A typical implementation will have FACT tables with CCI and Dimension tables with rowstore. However, there are no hard-and-fast rules but to get benefit of CCI or NCCI, the number of rows must exceed 1 million rows but preferably much higher
Updatable	Yes (starting with SQL Server 2016)	Yes

Table 2: Different between clustered (CCI) and non-clustered (NCCI) columnstore index

Generally the appropriate index design approach for fact table of data warehouse workloads is a mix of both rowstore and columnstore indexes, because their data access patterns are the most controversial. In fact table rowstore indexes are used to cover highly selective queries, while columnstore index in fact tables are used to improve query performance for non-selective queries that aim to speed up scan operations by providing an in memory structure. Columnstore index uses less disk space and compresses database files which is main advantages of this type of index over rowstore index for really big fact table. In addition, creating a non-clustered non-unique index on fact table's foreign keys which are frequently used, could be led to faster queries. Also we don't need to index some dimensions which are rarely used in the queries. We should only index the dimensions which are frequently used when that fact table is queried.

I am going to use the DW_Fact table for this demonstration. I should mention that we can have only one column store index on each table as a non-clustered index. In this case, that should ideally include all the

columns of the table, because all of the columns can be accessed independently from one another. In the following script, I create a ColumnStore Index on the DW_Fact table and including all of the frequently used columns from that table in the index.

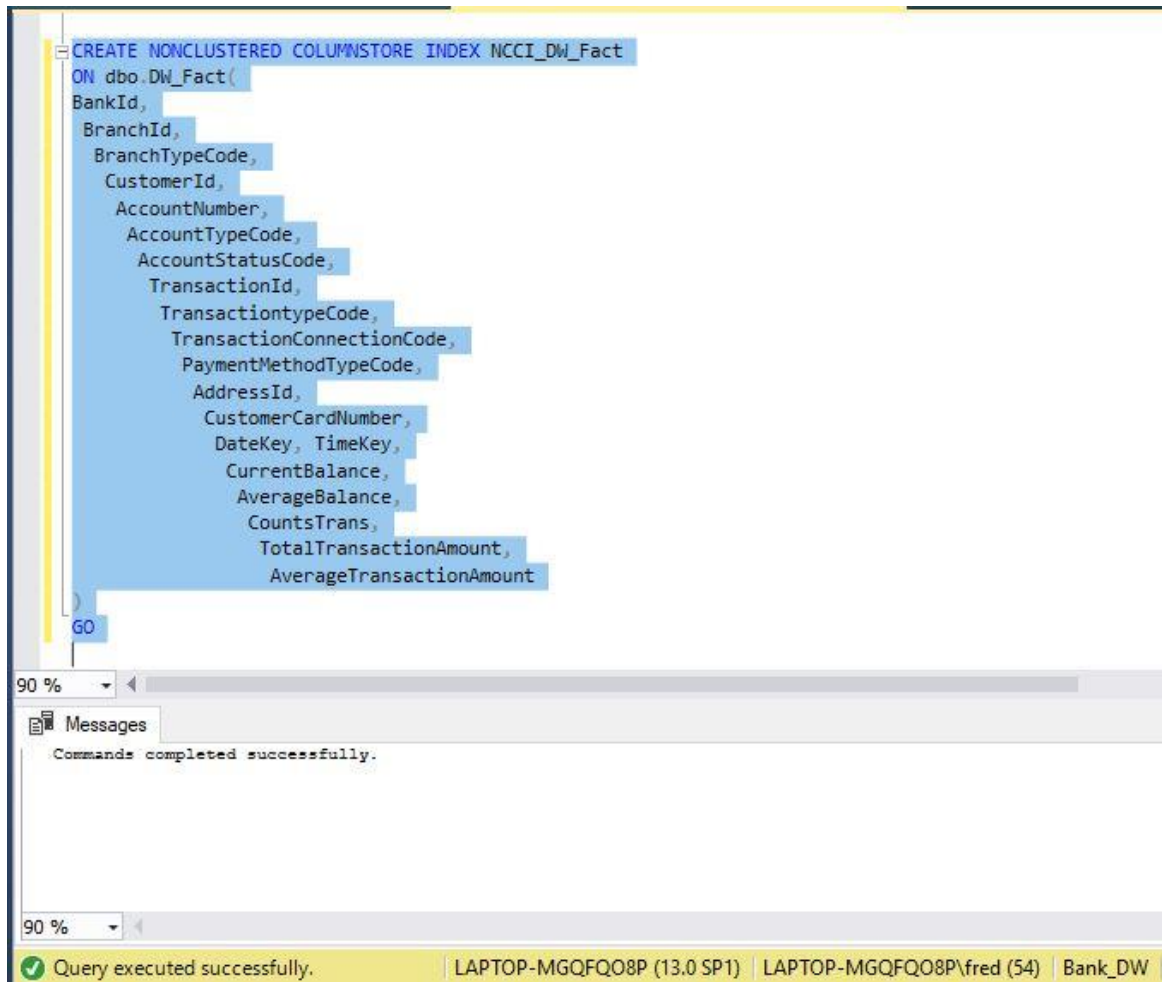


Fig 18: Creating Non-Cluster Column store index

I have provided two sample queries for checking how the query optimizer uses columnstore index and how it improves the query performance. For the first one, the query optimizer will by default use the column store index since all of the required columns' data is available. The second query is same as the first one, but this time I am instructing query optimizer to ignore the ColumnStore Index with the OPTION clause as highlighted below and use the row store index instead (in this case cluster index).

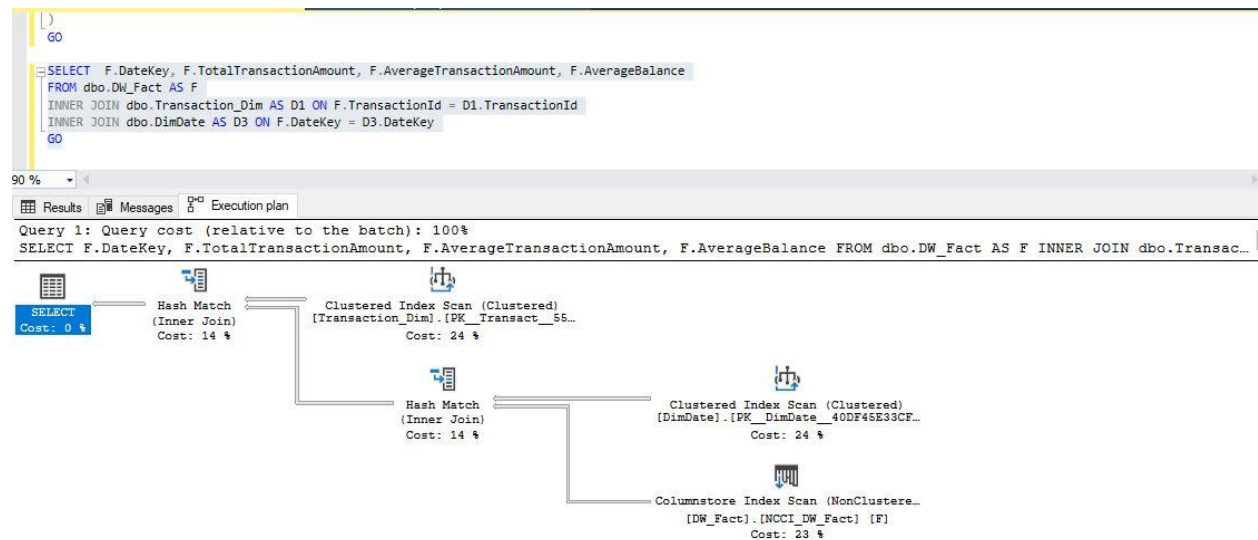


Fig 19: The Non-Cluster ColumnStore Index Scan operator

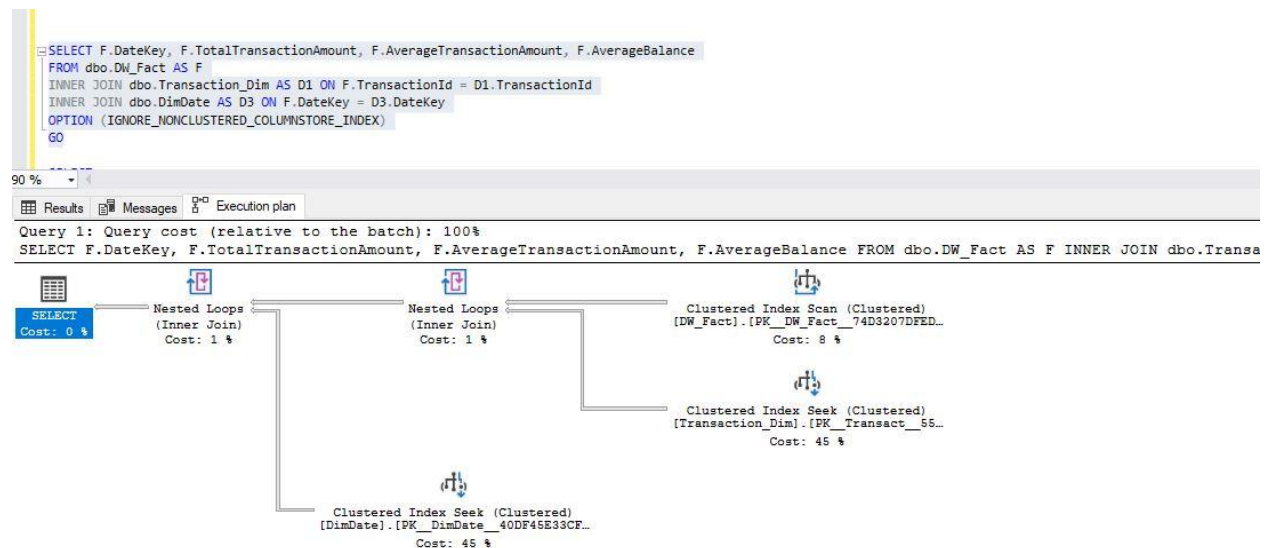


Fig 20: The Clustered Index Scan operator

As you can see in figure 19, 20 both queries are the same result and returns the same set of data. But By execution the first query, the query optimizer uses the ColumnStore Index Scan operator has 23% of the cost relative to DW-Fact table and the second query uses Clustered Index Scan operator (ignore column store index, has 8% of the cost relative to the DW-Fact table. Although in the first query, cluster index scan has 24% cost relative to Date and Transaction dimension, but in second query this percentage change

to 45. According to the results of execution plan of both queries, we can say, Cluster Index is suitable for fact table and non_cluster columnstore index is more perform for dimension table with many rows.

5. Conclusion

From the above work, it can be concluded that a data warehouse can deal with huge amounts of data and it is essential for any retail banking business that wants to profit from sound business decisions. Better decision making and easy access to historic data are main goals common of creating data warehouse for any business like retail banking. The core of the data warehouse concept is transformation and standardization of the initial data in the target data structure. DW provides a strong data view of the past transactions and it provides monitoring to identify such activities like risk and fraud across their daily transactions and help setting up some of those alerts. Also creating proper index is other important aspect of DW in order to access quickly locate data. Making right balance between query speed and update cost is vital to help in enforcing referential integrity.

6. Works Cited

1. (IBM Banking & Financial Markets Data Warehouse). Retrieved from:
<https://www.ibm.com/us-en/marketplace/banking-and-financial-markets-data-w>
2. (The requirement is to design a Database to support BI for Retail Banks). Retrieved from:
http://www.databaseanswers.org/data_models/retail_banks/facts.htm
3. (Mishra S. K. 2011, Design of Data Cubes and Mining for Online Banking System). Retrieved from: <https://pdfs.semanticscholar.org/3d6c/2d7b1e33e0bee395a090a01ff8665c8fa706.pdf>
4. (Data Warehouse Design Techniques – Slowly Changing Dimensions). Retrieved from:
<http://www.nuwavesolutions.com/slowly-changing-dimensions/>
5. (2012, creating a date dimension or calendar table in SQL Server). Retrieved from:
<https://www.mssqltips.com/sqlservertip/2586/sql-server-2012-column-store-index-example/>
6. (Walker A. J. 2013. Date and Time dimension creation and population T-SQL) Retrieved from:
<http://www.sqlservercentral.com/scripts/Data+Warehousing/65762/>

7. (Sheikh M. 2013, Create & Populate Time Dimension with 24 Hour + Values) Retrieved from:
<https://www.codeproject.com/Tips/642912/Create-Populate-Time-Dimension-with-24-Hourplus-Va>
8. (Agarwal S. 2016, Columnstore Index: Differences between Clustered/Nonclustered Columnstore Index) Retrieved from:
<https://blogs.msdn.microsoft.com/sqlserverstorageengine/2016/07/18/columnstore-index-differences-between-clusterednonclustered-columnstore-index/>
9. (2018, SQL Server Index Architecture and Design Guide). Retrieved from:
<https://docs.microsoft.com/en-us/sql/relational-databases/sql-server-index-design-guide?view=sql-server-2017>
10. (Goodman R. 2013, Benefit of Column-store Indexes) Retrieved from:
<https://robertbgoodman.wordpress.com/2013/04/02/benefit-of-column-store-indexes/>
11. (Arshad A. 2012, SQL Server 2012 Column Store Index Example). Retrieved from:
<https://www.mssqltips.com/sqlservertip/2586/sql-server-2012-column-store-index-example/>
12. (Kattunga R. 2016, why data warehousing is important in banking?) Retrieved from:
<https://www.quora.com/Why-data-warehousing-is-important-in-banking>