به نام خدا

حدیث غفوری 9825413

سوال 1

قبل از اپدیت

```sql
select sum(customer_rate),count(rental_id)
from rental inner join inventory using(inventory_id)
inner join film f2 using(film_id)
where film_id =2;
```

Data Output | Explain | Messages | Notifications

| | sum bigint | count bigint |
|---|---|---|
| 1 | 350 | 7 |

```sql
select film_id,rental_id,customer_rate
from rental inner join inventory using(inventory_id)
inner join film f2 using(film_id)
where film_id =2;
```

Data Output | Explain | Messages | Notifications

| | film_id integer | rental_id integer | customer_rate integer |
|---|---|---|---|
| 1 | 2 | 10310 | 50 |
| 2 | 2 | 13421 | 50 |
| 3 | 2 | 4364 | 50 |
| 4 | 2 | 7733 | 50 |
| 5 | 2 | 15218 | 50 |
| 6 | 2 | 10992 | 50 |
| 7 | 2 | 11758 | 50 |

بعد از اپدیت

```sql
select sum(customer_rate),count(rental_id)
 from rental inner join inventory using(inventory_id)
 inner join film f2 using(film_id)
 where film_id =2;

-- select film_id,rental_id,customer_rate
-- from rental inner join inventory using(inventory_id)
-- inner join film f2 using(film_id)
```

Data Output | Explain | Messages | Notifications

| | sum bigint | count bigint |
|---|---|---|
| 1 | 320 | 7 |

```sql
--  update rental
--      set customer_rate = 20
--      where rental_id = 7733;
 select * from film
 where film_id=2;
```

Data Output | Explain | Messages | Notifications

| | score real |
|---|---|
| stound':4 'car':17 'china':20 'databas':8 'epistl':5 'explor':12 'find':15 'goldfing':2 'must':14 | 45 |

سوال 2

A:

If a trigger function executes SQL commands then these commands might fire triggers again. This is known as cascading triggers. There is no direct limitation on the number of cascade levels. It is possible for cascades to cause a recursive invocation of the same trigger; for example, an INSERT trigger might execute a command that inserts an additional row into the same table, causing the INSERT trigger to be fired again. It is the trigger programmer's responsibility to avoid infinite recursion in such scenarios.

Recursion occurs when the same code is executed again and again. It can lead to an infinite loop and which can result in governor limit sometime. Sometime it can also result in unexpected output.

It is very common to have recursion in the trigger which can result in unexpected output or some error. So we should write code in such a way that it does not result to recursion. But sometime we are left with no choice.

For example, we may come across a situation where in a trigger we update a field which in result invoke a workflow. The workflow contains one field update on the same object. So trigger will be executed two times. It can lead us to unexpected output.

Another example is our trigger fires on after update, and it updates some related object, and there is one more trigger on a related object which updates child object. So it can result from too infinite loop.

B:

In Postgres 9.2 or later, use the function pg_trigger_depth().

so that the trigger function is not even executed when called from another trigger (including itself - so also preventing loops).

This typically performs better and is simpler and cleaner:

CREATE TRIGGER set_history

BEFORE UPDATE ON field_data

FOR EACH ROW

WHEN (pg_trigger_depth() < 1)

EXECUTE PROCEDURE gener_history();

The expression pg_trigger_depth() < 1 is evaluated before the trigger function is entered. So it evaluates to 0 in the first call. When called from another trigger, the value is higher and the trigger function is not executed.

سوال 3

```sql
-- drop trigger recommendation
--  on rental;
 insert into rental(rental_date,inventory_id,customer_id,staff_id)
 values(now(),3 ,6,1);
--   select * from rental
--   where customer_id=6
--   order by rental_date desc;
-- select * from customer
--  where customer_id=6;
-- where rent_count >0;
-- select * from inventory;
```

Data Output Explain Messages Notifications

```
INSERT 0 1

Query returned successfully in 92 msec.
```

Query Editor Query History

```sql
--    EXECUTE PROCEDURE give_suggest();

-- drop trigger recommendation
--  on rental;
-- insert into rental(rental_date,inventory_id,customer_id,staff_id)
--    values(now(),3 ,6,1);
--   select * from rental
--   where customer_id=6
--   order by rental_date desc;
select * from customer
 where customer_id=6;
-- where rent_count >0;
-- select * from inventory;
```

Data Output Explain Messages Notifications

| g (45) | last_name character varying (45) | email character varying (50) | address_id smallint | activebool boolean | create_date date | last_update timestamp without time zone | active integer | rent_count integer |
|---|---|---|---|---|---|---|---|---|
| | Davis | jennifer.davis@sakilacustomer.org | 10 | true | 2006-02-14 | 2013-05-26 14:49:45.738 | 1 | 0 |

```sql
--  drop trigger recommendation
--   on rental;
-- insert into rental(rental_date,inventory_id,customer_id,staff_id)
--    values(now(),3 ,6,1);
--   select * from rental
--   where customer_id=6
--   order by rental_date desc;
select * from customer
 where customer_id=6;
-- where rent_count >0;
-- select * from inventory;
```

Data Output | Explain | Messages | Notifications

| ...(45) | last_name character varying (45) | email character varying (50) | address_id smallint | activebool boolean | create_date date | last_update timestamp without time zone | active integer | rent_count integer |
|---|---|---|---|---|---|---|---|---|
| | Davis | jennifer.davis@sakilacustomer.org | 10 | true | 2006-02-14 | 2022-01-03 18:05:59.960236 | 1 | 1 |

```sql
-- insert into rental(rental_date,inventory_id,customer_id,staff_id)
--    values(now(),3 ,6,1);
--   select * from rental
--   where customer_id=6
--   order by rental_date desc;
select * from customer
 where customer_id=6;
-- where rent_count >0;
-- select * from inventory;
```

Data Output | Explain | Messages | Notifications

| ...(45) | last_name character varying (45) | email character varying (50) | address_id smallint | activebool boolean | create_date date | last_update timestamp without time zone | active integer | rent_count integer |
|---|---|---|---|---|---|---|---|---|
| | Davis | jennifer.davis@sakilacustomer.org | 10 | true | 2006-02-14 | 2022-01-03 18:07:14.744296 | 1 | 4 |

بعد از ۴ بار insert کردن دفعه ی ۵م تریگر فعال میشود

```sql
  select * from rental
   where customer_id=6
   order by rental_date desc;
-- select * from customer
--  where customer_id=6;
-- where rent_count >0;
-- select * from inventory;
```

Data Output | Explain | Messages | Notifications

| rental_id [PK] integer | rental_date timestamp without time zone | inventory_id integer | customer_id smallint | return_date timestamp without time zone | staff_id smallint | last_update timestamp without time zone |
|---|---|---|---|---|---|---|
| 16066 | 2022-01-03 18:07:33.120202 | 984 | 6 | [null] | 1 | 2022-01-03 18:07:33.120202 |
| 16065 | 2022-01-03 18:07:33.120202 | 3 | 6 | [null] | 1 | 2022-01-03 18:07:33.120202 |
| 16064 | 2022-01-03 18:07:14.744296 | 3 | 6 | [null] | 1 | 2022-01-03 18:07:14.744296 |
| 16063 | 2022-01-03 18:07:00.25322 | 3 | 6 | [null] | 1 | 2022-01-03 18:07:00.25322 |
| 16062 | 2022-01-03 18:06:31.071843 | 3 | 6 | [null] | 1 | 2022-01-03 18:06:31.071843 |
| 16061 | 2022-01-03 18:05:59.960236 | 3 | 6 | [null] | 1 | 2022-01-03 18:05:59.960236 |

سوال 4

```
WITH group_box AS (
    SELECT
        title,
        rating,
        SUM(amount) sum_amount
from film f2 inner join inventory using(film_id)
inner join rental using(inventory_id)
inner join payment using(rental_id)
group by film_id
)
SELECT
    title,
    rating,
    sum_amount,
```

Data Output Explain Messages Notifications

| | title character varying (255) | rating mpaa_rating | sum_amount numeric | rank_in_all bigint | rank_in_rating bigint | is_in_first_quarter text |
|---|---|---|---|---|---|---|
| 1 | Academy Dinosaur | PG | 33.79 | 718 | 147 | no |
| 2 | Ace Goldfinger | G | 52.93 | 508 | 89 | no |
| 3 | Adaptation Holes | NC-17 | 34.89 | 699 | 137 | no |
| 4 | Affair Prejudice | G | 83.79 | 261 | 47 | no |
| 5 | African Egg | G | 47.89 | 557 | 99 | no |
| 6 | Agent Truman | PG | 111.81 | 118 | 24 | yes |
| 7 | Airplane Sierra | PG-13 | 82.85 | 263 | 64 | no |
| 8 | Airport Pollock | R | 86.85 | 239 | 46 | yes |
| 9 | Alabama Devil | PG-13 | 71.88 | 354 | 80 | no |
| 10 | Aladdin Calendar | NC-17 | 131.77 | 66 | 18 | yes |

سوال 5

```
with group_box as (
       select extract (month from  rental_date) month_d,
    rating,
   sum(amount) amount
     from film
 inner join inventory using(film_id)
 inner join rental r2 using(inventory_id)
 inner join payment using(rental_id)
 where rating is not null
group by rating,extract (month from  rental_date)
order by extract (month from  rental_date)
)

SELECT
   distinct month_d,
```

Data Output | Explain | Messages | Notifications

| | month_d double precision | rating mpaa_rating | amount numeric | previous_month_sales numeric | next_months_sales numeric |
|---|---|---|---|---|---|
| 1 | 2 | NC-17 | 113.56 | [null] | 1666.90 |
| 2 | 2 | PG | 94.69 | [null] | 1659.99 |
| 3 | 2 | PG-13 | 118.59 | [null] | 1857.58 |
| 4 | 2 | R | 82.71 | [null] | 1746.78 |
| 5 | 2 | G | 104.63 | [null] | 1423.60 |
| 6 | 6 | PG | 1658.99 | 94.69 | 5696.52 |
| 7 | 6 | R | 1745.78 | 82.71 | 5436.85 |
| 8 | 6 | PG-13 | 1856.58 | 118.59 | 6521.85 |
| 9 | 6 | G | 1422.60 | 104.63 | 5014.11 |
| 10 | 6 | NC-17 | 1665.90 | 113.56 | 5713.54 |
| 11 | 7 | PG | [illegible] | [illegible] | [illegible] |

سوال 6

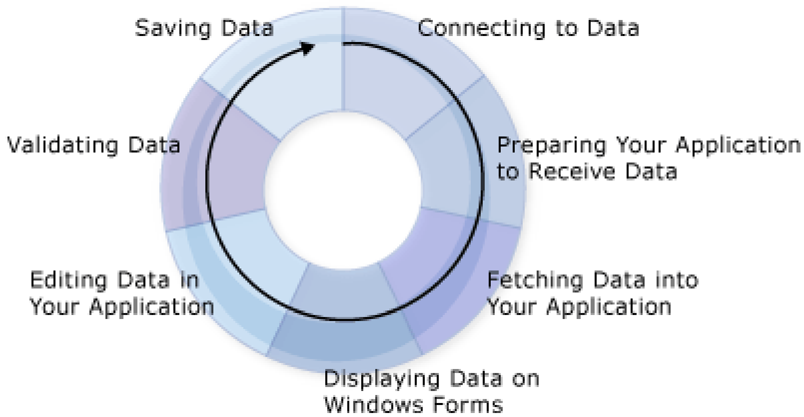
```sql
select film.film_id,city_id , s.store_id,f.staff_id,count(rental_id)
from film,rental r,staff f,store s,address a,inventory i
where f.store_id = s.store_id and
a.address_id = s.address_id and
 r.staff_id = f.staff_id and
i.film_id = film.film_id and
i.inventory_id= r.inventory_id
group by
film.film_id,
cube(city_id , s.store_id,f.staff_id)
order by
count(rental_id) desc
```

Data Output | Explain | Messages | Notifications

| | film_id integer | city_id smallint | store_id integer | staff_id integer | count bigint |
|---|---|---|---|---|---|
| 1 | 103 | [null] | [null] | [null] | 34 |
| 2 | 738 | [null] | [null] | [null] | 33 |
| 3 | 489 | [null] | [null] | [null] | 32 |
| 4 | 767 | [null] | [null] | [null] | 32 |
| 5 | 382 | [null] | [null] | [null] | 32 |
| 6 | 730 | [null] | [null] | [null] | 32 |
| 7 | 331 | [null] | [null] | [null] | 32 |
| 8 | 735 | [null] | [null] | [null] | 31 |
| 9 | 891 | [null] | [null] | [null] | 31 |
| 10 | 621 | [null] | [null] | [null] | 31 |

سوال 7

A:




Connecting to Data

Preparing Your Application to Receive Data

Fetching Data into Your Application

Displaying Data on Forms

Editing Data in Your Application

Validating Data

Saving Data

B:

definition of OLE DB is

a strategic system-level programming interface to data across the organization. OLE DB is an open specification designed to build on the success of ODBC by providing an open standard for accessing all kinds of data.

definition of ODBC is

an industry standard and a component of Microsoft® Windows® Open Services Architecture (WOSA). The ODBC interface makes it possible for applications to access data from a variety of database management systems (DBMSs). ODBC permits maximum interoperability—an application can access data in diverse DBMSs through a single interface. Furthermore, that application will be independent of any DBMS from which it accesses data. Users of the application can add software components called drivers, which create an interface between an application and a specific DBMS

1.OLE DB is a component based specification and ODBC is a procedural based specification

2. SQL is the core of accessing data using ODBC but just one of the means of data access through OLE DB

3. ODBC is constrained to relational data stores;

OLE DB supports all forms of data stores (relational,hierarchical, etc)

4.OLE DB provides Full access to ODBC data sources and ODBC drivers

C:

In short, an ORM is a layer between the server and the database.

The server talks with the ORM and the ORM talks to the database.

The ORM creates objects, that map to the relational data.

It handles your queries, so you don't have to write native SQL, you can query the database with your application language

An object-relational mapper (ORM) is a code library that automates the transfer of data stored in relational database tables into objects that are more commonly used in application code.

## Pros

you don't have to learn/know/write SQL, because the ORM handles it

it will be easier to change your database dialect

your application is less vulnerable to SQL injections

ORMs provide a high-level abstraction upon a relational database that allows a developer to write Python code instead of SQL to create, read, update and delete data and schemas in their database. Developers can use the programming language they are comfortable with to work with a database instead of writing SQL statements or stored procedures.

The ability to write Python code instead of SQL can speed up web application development, especially at the beginning of a project

ORMs also make it theoretically possible to switch an application between various relational databases. For example, a developer could use SQLite for local development and MySQL in production. A production application could be switched from MySQL to PostgreSQL with minimal code modifications.

ORMS for python:

| | | | | |
|---|---|---|---|---|
| web framework | None | Flask | Flask | Django |
| ORM | SQLAlchemy | SQLAlchemy | SQLAlchemy | Django ORM |
| database connector | (built into Python stdlib) | MySQL-python | psycopg | psycopg |
| relational database | SQLite | MySQL | PostgreSQL | PostgreSQL |

## List of ORMs:

sequelize: Postgres, MySQL, MariaDB, SQLite, Microsoft SQL Server

TypeORM: Postgres, MySQL, MariaDB, SQLite, Microsoft SQL Server, Oracle, sql.js, CockroachDB

objection: Postgres, MySQL, MariaDB, SQLite, Microsoft SQL Server, Oracle, Amazon Redshift

سوال 8

A:

**Rowstore**

*Page #1*

| ProductID | ProductName | Price | ... |
|---|---|---|---|
| 1 | Product A | 20 | ... |
| 2 | Product B | 30 | ... |
| 3 | Product C | 15 | ... |

*Page #2*

| ProductID | ProductName | Price | ... |
|---|---|---|---|
| 4 | Product D | 50 | ... |
| 5 | Product E | 20 | ... |
| 6 | Product F | 40 | ... |

**Columnstore**

| *Page #1* | *Page #2* | *Page #3* | *Page ...* |
|---|---|---|---|
| ProductID | ProductName | Price | ... |
| 1 | Product A | 20 | ... |
| 2 | Product B | 30 | ... |
| 3 | Product C | 15 | ... |
| 4 | Product D | 50 | ... |
| 5 | Product E | 20 | ... |
| 6 | Product F | 40 | ... |

Row oriented databases are databases that organize data by record, keeping all of the data associated with a record next to each other in memory. Row oriented databases are the traditional way of organizing data and still provide some key benefits for storing data quickly. They are optimized for reading and writing rows efficiently.

Common row oriented databases:

Postgres , MySQL

Column oriented databases are databases that organize data by field, keeping all of the data associated with a field next to each other in memory. Columnar databases have grown in popularity and provide performance advantages to querying data. They are optimized for reading and computing on columns efficiently.

Common column oriented databases:

Redshift , BigQuery,Snowflake

# Row Oriented Databases

They are optimized to read and write a single row of data which lead to a series of design choices including having a row store architecture.

In a row store, or row oriented database, the data is stored row by row, such that the first column of a row will be next to the last column of the previous row.

For instance

Facebook_Friends

| Name | City | Age |
|---|---|---|
| Matt | Los Angeles | 27 |
| Dave | San Francisco | 30 |
| Tim | Oakland | 33 |

This data would be stored on a disk in a row oriented database in order row by row like this:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Matt | Los Angeles | 27 | Dave | San Francisco | 30 | Tim | Oakland | 33 |

This allows the database write a row quickly because, all that needs to be done to write to it is to tack on another row to the end of the data.

Writing to Row Store Databases

Let's use the data stored in a database:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Matt | Los Angeles | 27 | Dave | San Francisco | 30 | Tim | Oakland | 33 |

If we want to add a new record:

| | | |
|---|---|---|
| Jen | Vancouver | 30 |

We can just append it to the end of the current data:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Matt | Los Angeles | 27 | Dave | San Francisco | 30 | Tim | Oakland | 33 | Jen | Vancouver | 30 |

Reading from Row Store Databases

Say we want to get the sum of ages from the Facebook_Friends data. To do this we will need to load all nine of these pieces of data into memory to then pull out the relevant data to do the aggregation.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Matt | Los Angeles | 27 | Dave | San Francisco | 30 | Tim | Oakland | 33 |

This is wasted computing time.

Let's assume a Disk can only hold enough bytes of data for three columns to be stored on each disk. In a row oriented database the table above would be stored as:

| Disk 1 | | |
|---|---|---|
| Name | City | Age |
| Matt | Los Angeles | 27 |

| Disk 2 | | |
|---|---|---|
| Name | City | Age |
| Dave | San Francisco | 30 |

| Disk 3 | | |
|---|---|---|
| Name | City | Age |
| Tim | Oakland | 33 |

To get the sum of all the people's ages the computer would need to look through all three disks and across all three columns in each disk in order to make this query.

So we can see that while adding data to a row oriented database is quick and easy, getting data out of it can require extra memory to be used and multiple disks to be accessed.

Row oriented databases are fast at retrieving a row or a set of rows but when performing an aggregation it brings extra data (columns) into memory which is slower than only selecting the columns that you are performing the aggregation on. In addition the number of disks the row oriented database might need to access is usually larger.

while adding data to a row oriented database is quick and easy, getting data out of it can require extra memory to be used and multiple disks to be accessed.

# Column Oriented Databases

Data Warehouses were created in order to support analyzing data. These types of databases are read optimized.

the data is stored such that each row of a column will be next to other rows from that same column.

Facebook_Friends

| Name | City | Age |
|---|---|---|
| Matt | Los Angeles | 27 |
| Dave | San Francisco | 30 |
| Tim | Oakland | 33 |

A table is stored one column at a time in order row by row:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Matt | Dave | Tim | Los Angeles | San Francisco | Oakland | 27 | 30 | 33 |

Writing to a Column Store Databases

If we want to add a new record:

| | | |
|---|---|---|
| Jen | Vancouver | 30 |

We have to navigate around the data to plug each column in to where it should be.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Matt | Dave | Tim | Jen | Los Angeles | San Francisco | Oakland | Vancouver | 27 | 30 | 33 | 30 |

If the data was stored on a single disk it would have the same extra memory problem as a row oriented database, since it would need to bring everything into memory. However, column oriented databases will have significant benefits when stored on separate disks.

If we placed the table above into the similarly restricted three columns of data disk they would be stored like this:

| Disk 1 | | |
|---|---|---|
| Name | | |
| Matt | Dave | Tim |

| Disk 2 | | |
|---|---|---|
| City | | |
| Los Angeles | San Francisco | Oakland |

| Disk 3 | | |
|---|---|---|
| Age | | |
| 27 | 30 | 33 |

Reading from a Column store Database

To get the sum of the ages the computer only needs to go to one disk (Disk 3) and sum all the values inside of it. No extra memory needs to be pulled in, and it accesses a minimal number of disks.

by organizing data by column the number of disks that will need to be visited will be reduced and the amount of extra data that has to be held in memory is minimized. This greatly increases the overall speed of the computation.

B:

Row based tables have disadvantages in case of analytic applications where aggregation are used and faster search & processing are required. In row based tables all data in a row has to be read even though the requirement may be there to access data from a few columns.

These are not efficient in performing operations applicable to the entire datasets and hence aggregation in row-oriented is an expensive job or operations.

Typical compression mechanisms which provide less efficient result than what we achieve from column-oriented data stores.

C:

In case of analytic applications, where aggregations are used and faster search & processing are required, row-based storage are not good. In row based tables all data stored in a row has to be read even though the requirement may be there to access data from a few columns. Hence, these queries on huge amounts of data would take lots of times

These are not efficient in performing operations applicable to the entire datasets and hence aggregation in row-oriented is an expensive job or operations.

چون میخاهیم فروش را مقایسه کنیم و انالیز کنیم طبق متن بالا از column based استفاده میکنیم. چون سرعت بیشتری در داده هایی با حجم بالا دارد.

D:

طبق توضیحات بالا و مثال زده شده:

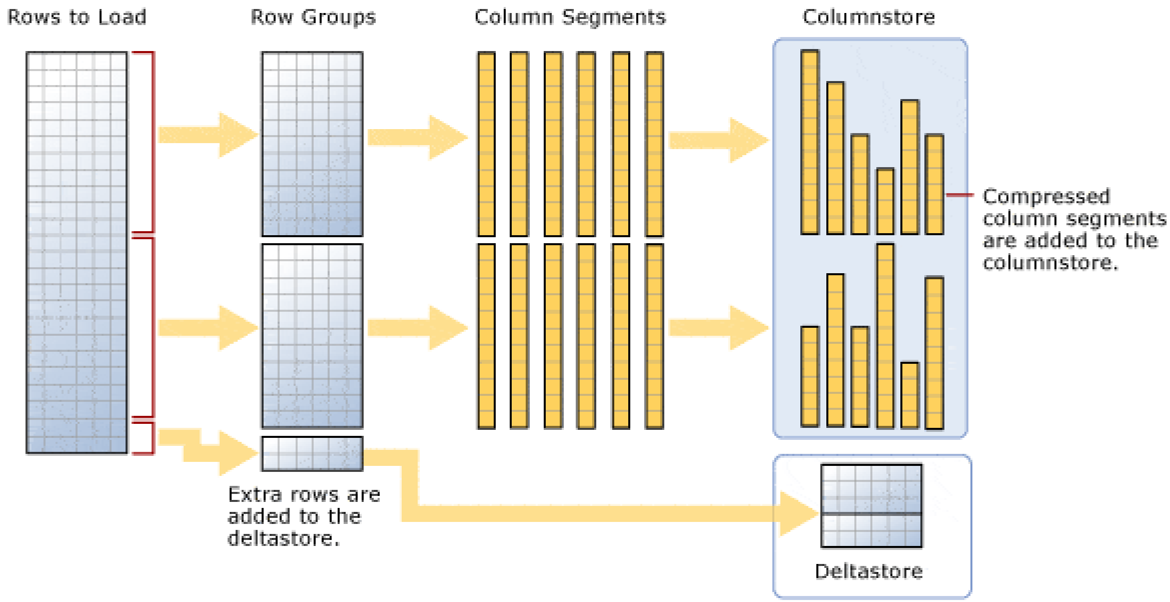
Insert,update در روش rowstore بهینه تر هستند چون فقط نیاز است داده های جدید را به انتهای داده های قبلی اضافه کنیم و

برای عملیات های transaction مناسبت تر است.

E:

Columnstore simply means a new way to store the data in the index. Instead of the normal Rowstore or b-tree indexes where the data is logically and physically organized and stored as a table with rows and columns, the data in columnstore indexes are physically stored in columns and logically organized in rows and columns. Instead of storing an entire row or rows in a page, one column from many rows is stored in that page. It is this difference in architecture that gives the columnstore index a very high level of compression along with reducing the storage footprint and providing massive improvements in read performance.

The index works by slicing the data into compressible segments. It takes a group of rows, a minimum of 102,400 rows with a max of about 1 million rows, called a rowgroup and then changes that group of rows into Column segments. It’s these segments that are the basic unit of storage for a columnstore index, as shown below




Columnstore indexes are designed for large data warehouse workloads, not normal OLTP workload tables.