Fpt University

Chapter 7: Practical issues of database application

Objectives

- Understand **transactions** and their properties (ACID-Atomicity, Consistency, Isolation, Durability)
- Apply transaction in database application programming
- Understand the roles of indexing techniques
- Implement indexes for query optimization
- Understand what views are for and how to use them
- Understand query execution plan for query optimization analysis

Fpt University

Contents

1. Transaction in SQL
2. Indexes in SQL & Query optimization
3. Views

Fpt University

Introduction

- DB User operates on database by querying or modifying the database
- Operations on database are executed one at a time
- Output of one operation is input of the next operation
- So, how the DBMS treats simultaneous operations?

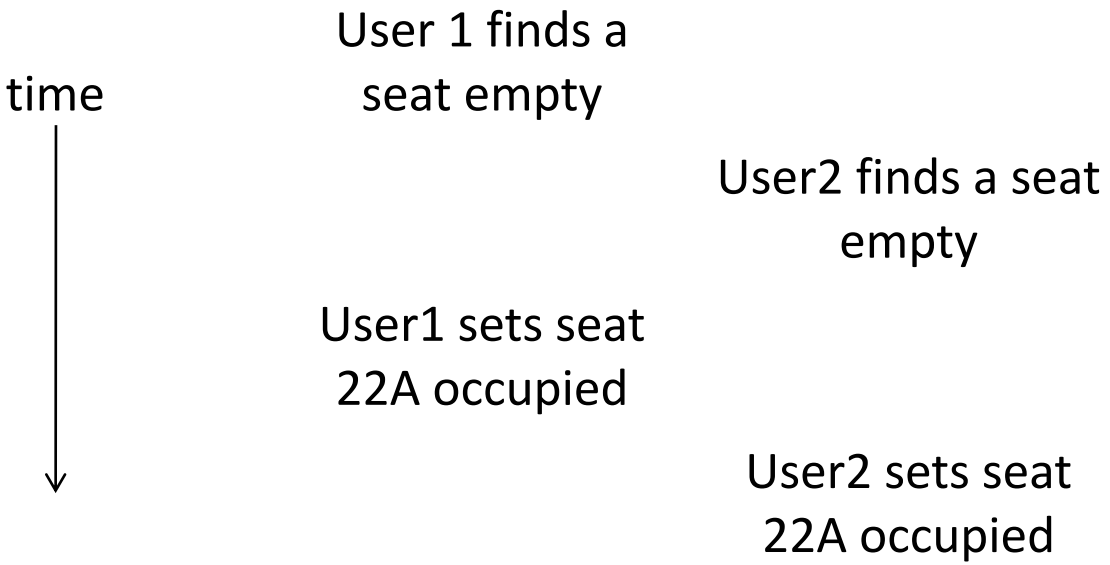
Fpt University

Serializability

- In applications, many operations per second may be performed on database
- These may operate on the same data
- We'll get unexpected results

Serializability

Example: Two users book the same seat of the flight



Fpt University

Serializability

- Transaction is a group of operations that need to be performed together
- A certain transaction must be serializable with respect to other transactions, that is, the transactions run serially – one at a time, no overlap

Fpt University

Atomicity

A certain combinations of database operations need to be done atomically, that is, either they are all done or neither is done

A transaction has only 2 states:

- All transaction changes will not be synchronized to the database.

- All transaction changes are synced to the database.

Fpt University

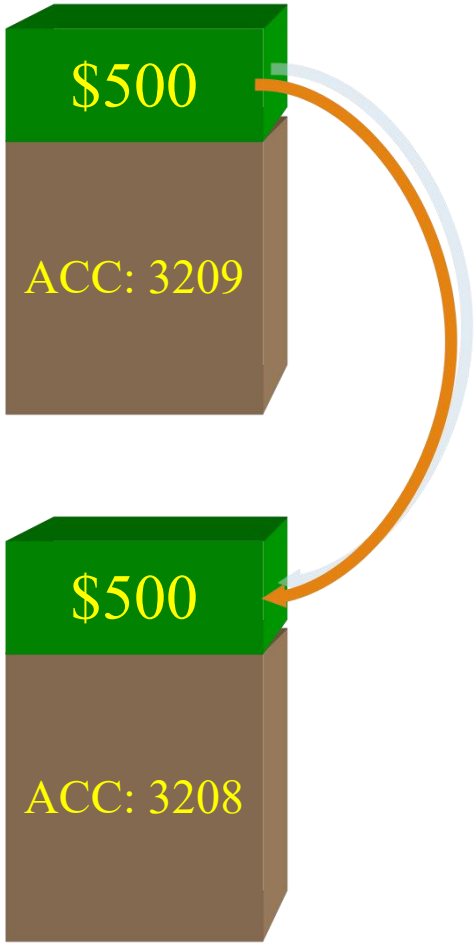
Atomicity

Example: Transfer $500 from the account number 3209 to account number 3208 by two steps

- (1) Subtract $500 from account number 3209
- (2) Add $500 to account number 3208

What happen if there is a failure after step (1) but before step (2)?

Atomicity



A Banking Transaction

```
UPDATE savings_accounts
     SET balance = balance - 500
     WHERE account = 3209;
```

Decrement Savings Account

```
UPDATE checking_accounts
     SET balance = balance + 500
     WHERE account = 3208;
```

Increment Checking Account

```
INSERT INTO journal VALUES
     (journal_seq.NEXTVAL, '1B'
     3209, 3208, 500);
```

Record in Transaction Journal

```
COMMIT WORK;
```

End Transaction

Transaction Ends

Fpt University

Transactions

- Transaction is a collection of one or more operations on the database that must be executed atomically
- That is, either all operations are performed or none are
- In SQL, each statement is a transaction by itself
- SQL allows to group several statements into a single transaction

Fpt University

Transactions

Transaction begins by SQL command START TRANSACTION

Two ways to end a transaction

- The SQL statement COMMIT causes the transaction to end successfully
- The SQL statement ROLLBACK causes the transaction to abort, or terminate unsuccessfully

Fpt University

ACID properties of Transaction

- Atomicity
- Consistency
- Isolation
- Durability

ACID properties of Transaction

Atomicity: a transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all.

- At the end of the transaction, either all statements of the transaction is successful or all statements of the transaction fail.
- If a partial transaction is written to the disk then the *Atomic* property is violated

Before: X : 500	Y: 200
Transaction T	
T1	**T2**
Read (X) X: = X – 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
After: X : 400	Y : 300

Fpt University

ACID properties of Transaction

Consistency: a transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another.

→ If 10$ has been deducted in account A, then in account B must be added 10$.

ACID properties of Transaction

Isolation: a transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing concurrently. That is the execution of a transaction should not be interfered with by any other transactions executing concurrently.

X = 50, Y = 50

- Transaction: T, T”
- When T" reads the value of X
→ T : (X * 100 = 500)
+ Y = 0 → Z= X + Y =

T	T”
Read (X) X: = X*100 Write (X) Read (Y) Y: = Y – 50 Write	Read (X) Read (Y) Z: = X + Y Write (Z)

Fpt University

ACID properties of Transaction

Durability : the changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

Fpt University

Read-Only Transactions

- A transaction can read or write some data into the database
- When a transaction only reads data and does not write data, the transaction may execute in parallel with other transactions
- Many read-only transactions access the same data to run in parallel, while they would not be allowed to run in parallel with a transaction that wrote the same data

Fpt University

Transactions

SQL statement set read-only to the next transaction

- SET TRANSACTION READ ONLY;

SQL statement set read/write to the next transaction

- SET TRANSACTION READ WRITE;

Fpt University

Dirty Reads

- **Dirty data**: data written by a transaction that has not yet committed
- **Dirty read**: read of dirty data written by another transaction

→ Occurs when a transaction performs a read on a data unit, which is being updated by another transaction but the update has not been confirmed.

- Problem: the transaction that wrote it may eventually abort, and the dirty data will be removed
- Sometimes the dirty read matters, sometimes it doesn't

Fpt University

Dirty Reads

EX:

- Betty: has $78.00 in her bank account
- The payment withdraws $45 for the electric bill.
- Betty check her bank account on the ATM --> $33 in bank account.
- If the electric bill payment transaction is rollbacked: the bank account balance will be turned to $78.00 again

--> so the data read by Betty is dirty data

Fpt University

Dirty Reads

We can specify that dirty reads are acceptable for a given transaction

- SET TRANSACTION READ WRITE
  ISOLATION LEVEL READ UNCOMMITTED;

→Can be used to prevent Dirty Write errors

Fpt University

Other Isolation Levels

SQL provides isolation levels: Read Uncommitted, Read Committed, Repeatable Read, Serializable:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITED;
SET TRANSACTION ISOLATION LEVEL READ  COMMITTED;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Fpt University

Other Isolation Levels

- Read Uncommitted

--> Access records being updated by another transaction and get data at that time even though that data has not been committed (uncommited data)
--> If for some reason the original transaction rolls back the updates, the data will revert to the old value --> Then the second transaction gets the wrong data (No matter if the data is being updated or not, give the data available right now)

- Read Committed:

This is the default isolation level of SQL Server
→ Transaction will not read the data being updated but must wait until the update is done
--> So it avoids dirty read

Fpt University

Other Isolation Levels

- Repeatable read

- Works like a read commit level, but takes it a step further by preventing the transaction from writing to data being read by another transaction until that other transaction completes.

--> Ensure reads in the same transaction give the same result

--> in other words, the data being read will be protected from updating by other transactions.

→ However, it does not protect data from insert or delete.

- Serializable

Lock the entire range of records that might be affected by another transaction

Index overview

- An index on attribute A is a data structure that makes it efficient to find those tuples that have a fixed value for attribute A
- Can dramatically speed up certain operations:
  - Find all R tuples where R.A = v
  - Find all R and S tuples where R.A = S.B
  - Find all R tuples where R.A > v (sometimes, depending on index type)

Index overview

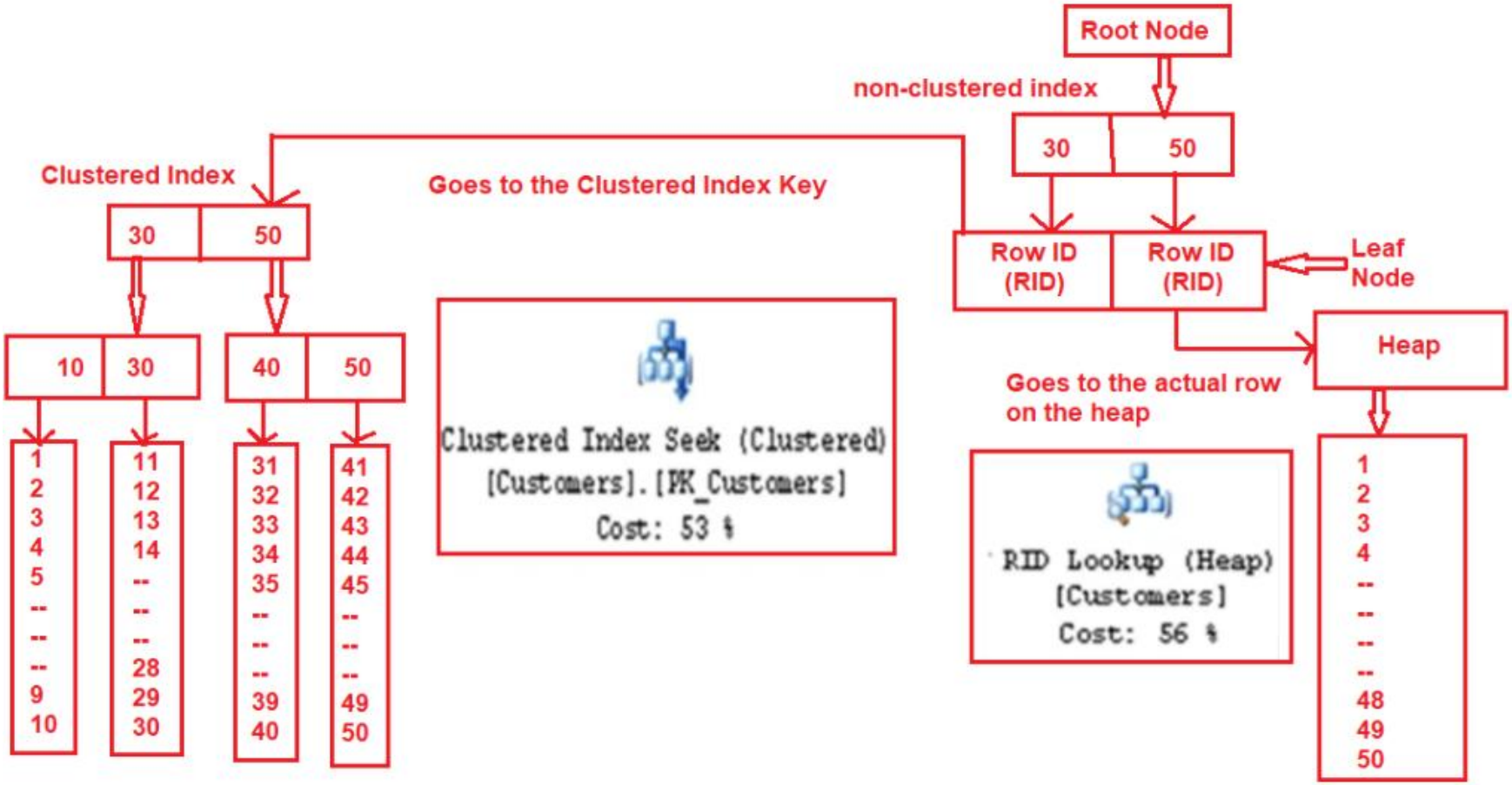
- Example

```
SELECT *
FROM Student
WHERE name = 'Mary’
```

- Without index: Scan all Student tuples
- With index: Go "directly" to tuples with name='Mary'

Indexes are built on single attributes or combinations of attributes.

Type of indexes



Indexes implementation on SQL

```
CREATE CLUSTERED INDEX index_name ON
dbo.Tablename(ColumnName1,ColumnName2...)
```

Table: users

-id (Primary Key, Auto Increament)

- username

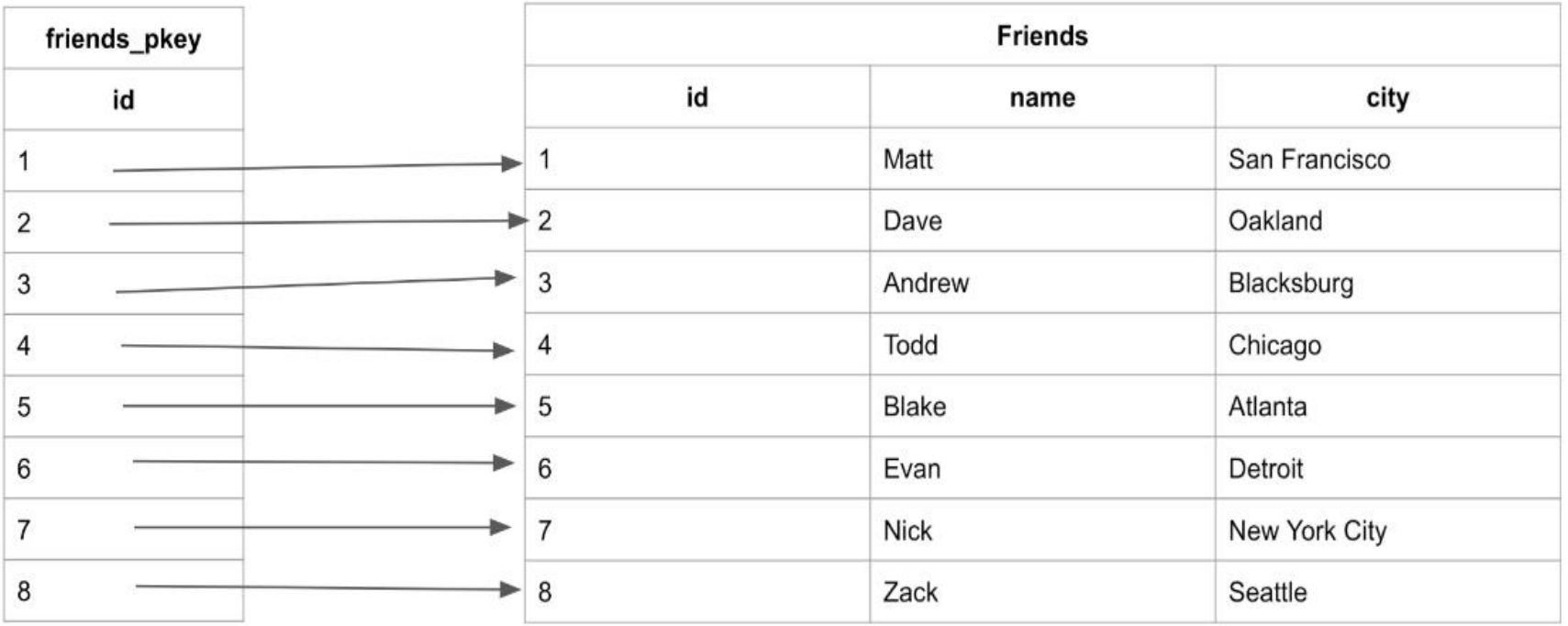
- email

- password

"username" --> We want to type INDEX for it to make the query faster

```
CREATE CLUSTERED INDEX idx_username ON users (username);
```

Indexes implementation on SQL



friends_pkey
id
1
2
3
4
5
6
7
8

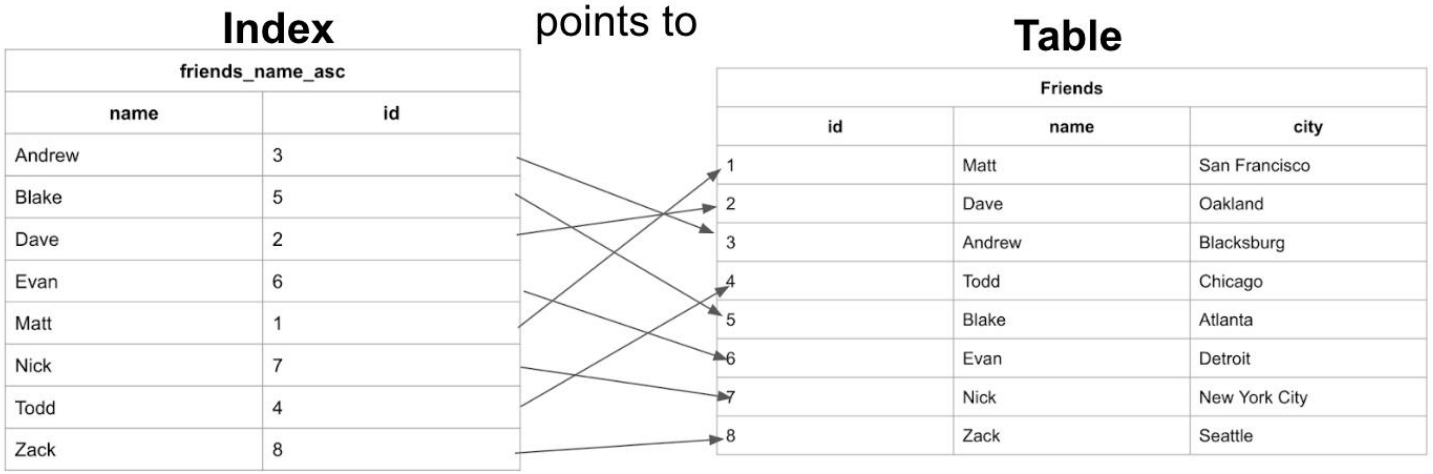
Friends		
id	name	city
1	Matt	San Francisco
2	Dave	Oakland
3	Andrew	Blacksburg
4	Todd	Chicago
5	Blake	Atlanta
6	Evan	Detroit
7	Nick	New York City
8	Zack	Seattle

Indexes implementation on SQL

CREATE NONCLUSTERED INDEX index_name ON dbo.Tablename(ColumnName1, ColumnName2...)

DROP INDEX index_name

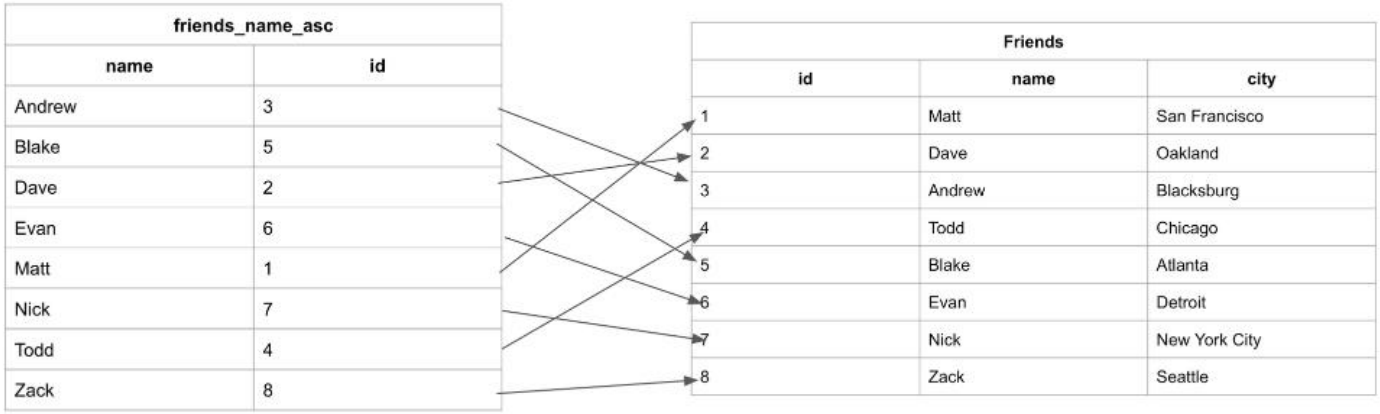
//demo required



Indexes implementation on SQL

```
CREATE NONCLUSTERED INDEX friends_name_asc
 ON friends(name ASC);
```

Create an index named “friends_name_asc”, which stores the name “friends” and sorts it in ascending order.



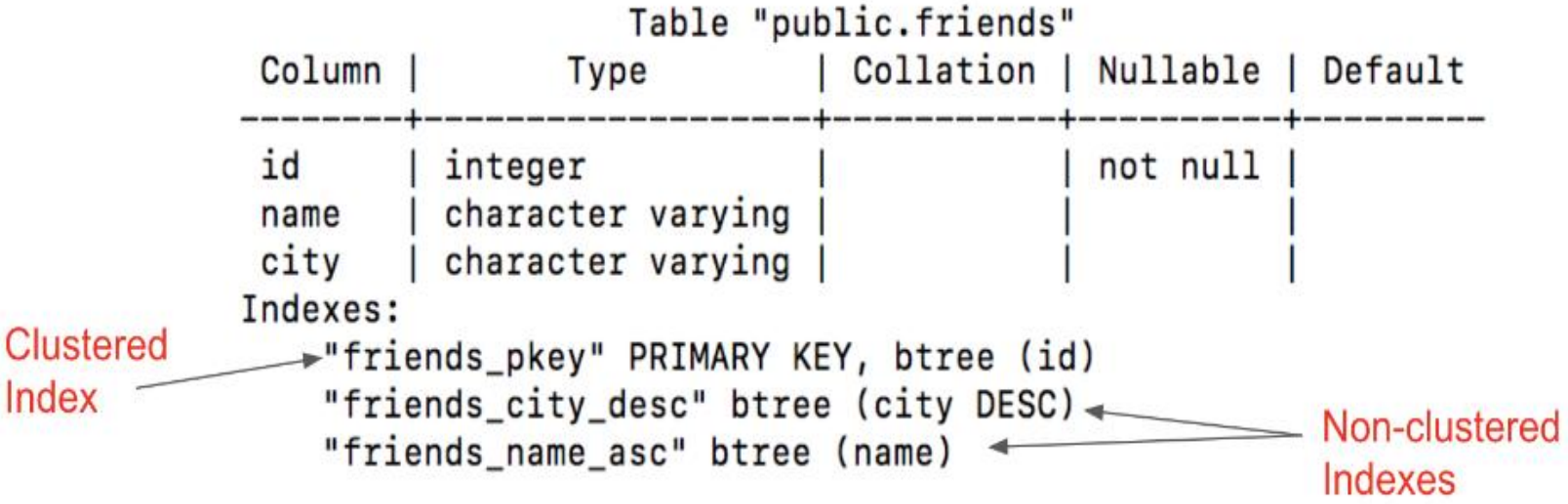
friends_name_asc

name	id
Andrew	3
Blake	5
Dave	2
Evan	6
Matt	1
Nick	7
Todd	4
Zack	8

Friends

id	name	city
1	Matt	San Francisco
2	Dave	Oakland
3	Andrew	Blacksburg
4	Todd	Chicago
5	Blake	Atlanta
6	Evan	Detroit
7	Nick	New York City
8	Zack	Seattle

Indexes implementation on SQL

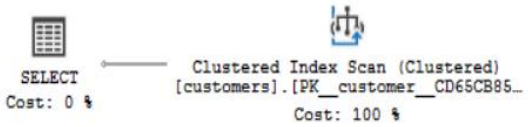


Indexes implementation on SQL

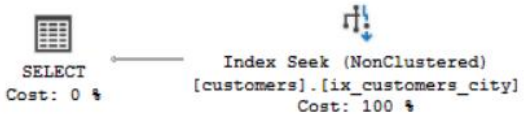
sales.customers
* customer_id
first_name
last_name
phone
email
street
city
state
zip_code

To search for customers in the city address "atwater"

```
SELECT customer_id, city
FROM sales.customers
WHERE city='Atwater'
```




```
CREATE NONCLUSTERED INDEX ix-
customer_city
ON sales.customers(city);

SELECT customer_id, city
FROM sales.customers
WHERE city='Atwater'
```



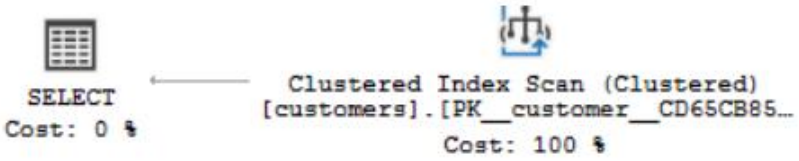
Indexes implementation on SQL

Create non-clustered index for multiple columns in SQL

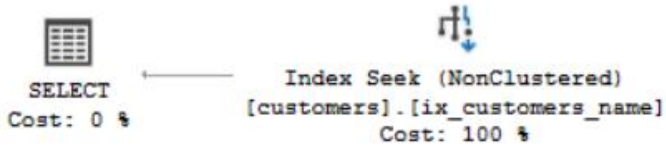
- Search for customers with last name 'Berg' and first name 'Monika'

sales.customers
* customer_id
first_name
last_name
phone
email
street
city
state
zip_code

```
SELECT customer_id, first_name, last_name
FROM sales.customers
WHERE last_name ='Berg' and first_name
='Monika'
```




```
CREATE NONCLUSTERED INDEX ix-
customer_name
ON sales.customers(last_name,
first_name);

SELECT customer_id, first_name, last_name
FROM sales.customers
WHERE last_name ='Berg' and first_name
='Monika'
```



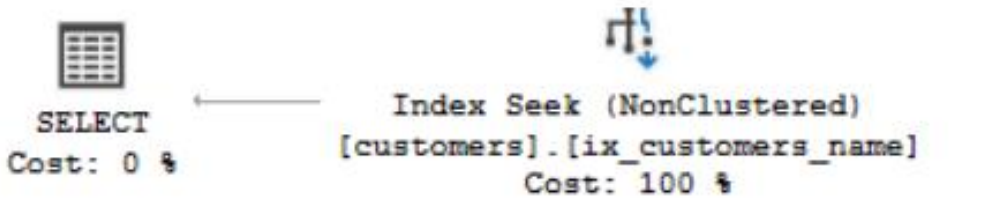
Indexes implementation on SQL

Create non-clustered index for multiple columns in SQL

- Search for customers with last name ‘Albert'

sales.customers
* customer_id
first_name
last_name
phone
email
street
city
state
zip_code

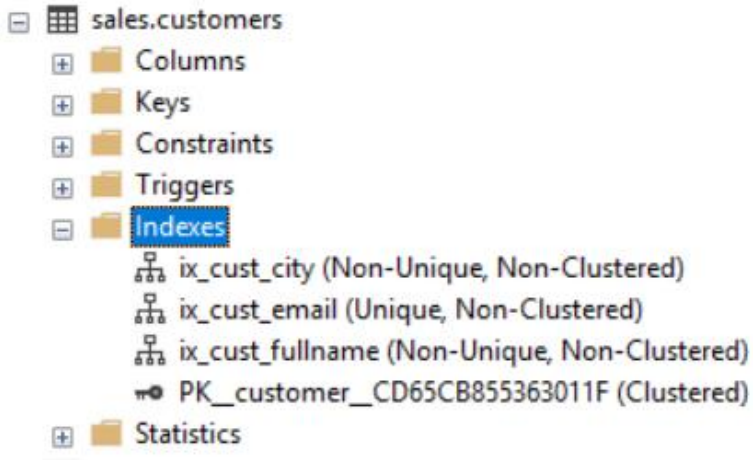
```
CREATE NONCLUSTERED INDEX ix-customer_name
ON sales.customers(last_name, first_name);

SELECT customer_id, first_name, last_name
FROM sales.customers
WHERE last_name =‘Albert’
```



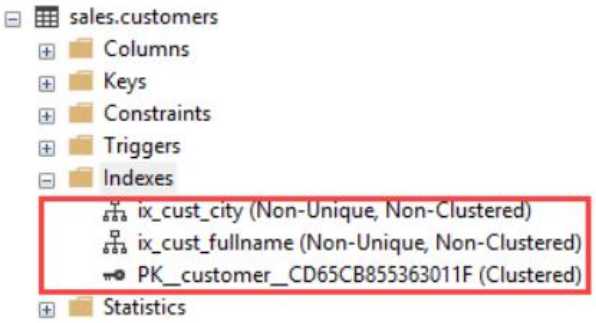
Indexes implementation on SQL

DROP INDEX index_name
//demo required

sales.customers
* customer_id
first_name
last_name
phone
email
street
city
state
zip_code



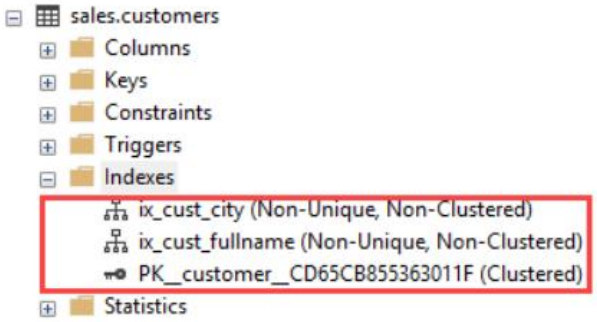
DROP INDEX ix_cust_email
ON sales.customers



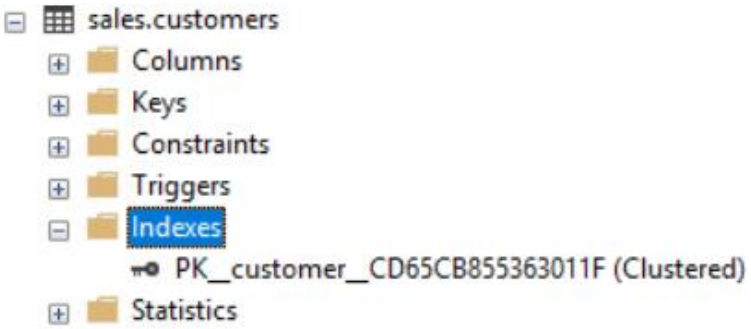
Indexes implementation on SQL

DROP INDEX index_name
//demo required

sales.customers
* customer_id
first_name
last_name
phone
email
street
city
state
zip_code



DROP INDEX
ix_cust_city ON sales.customers,
Ix_cust_fullname ON sales.customers;



Index design guidelines

- Choosing which indexes to create is a difficult and very important design issue. The decision depends on size of tables, data distributions, and most importantly query/update load.
  - Table Size: enough large. Why?
  - Column Types:
    - Small size (INT, BIGINT). Should create clustered index on Unique and NOT NULL field.
    - Identity field (automatically increment)
    - Static field
  - Number of indexes
  - Storage Location of Indexes
  - Index Types
  - Query design

Fpt University

Relations vs. Views

Relations

- Actual exist in database in some physical organization
- Defined with a CREATE TABLE statement
- Exist indefinitely and not to change unless explicit request

Virtual views

- Do not exist physically
- Defined by an expression like a query
- Can be queried and can even be modified

Views

- A view just a relation, but we store a definition rather than a set of tuples

- When do we use view?

- Restrict access to Tables -> only allow viewing through View

- Restrict access to the Column of the Table -> When accessing through the View, they cannot know the name of the Column that the View accesses.

- Linking Columns from many Tables -> into a new Table shown through View.

- Presenting aggregated information (eg using functions like COUNT, SUM, ...)

Views

- Syntax:

```
CREATE VIEW ViewName
AS
SELECT * / RequiredColumnNames
FROM TableName
```

The simplest form of view definition is

CREATE VIEW <view-name> AS <view-definition>;

→The <view-definition> is a SQL query

Types of views in SQL Server:

- Simple view or Updatable views: single table, can INSERT, UPDATE, DELETE through view.
- Complex view or non-updatable views: multi tables

Virtual Views

```
create view v_Student_AgeGreaterThan18
as
select * from Student
where Age > 20

select * from v_Student_AgeGreaterThan18
```

76 %

Results | Messages

	Id	Name	Age	DateOfBirth	Gender	Province1	District
1	2	Nguyen Van B	21	1990-09-13	1	NULL	NULL

Virtual Views

Example 1:

- Create a view to list all employees who are in Department number 1

```
CREATE VIEW Employee_Dep1 AS
SELECT * FROM tblemployee WHERE depnum=1;
```

Renaming Attributes

Example 3:

- Create view for all employees of Department number 1, including: SSN, Fullname, Age, Salary, Sex

```
CREATE VIEW Employee_Dep1 AS
SELECT te.empSSN AS 'Mã số nhân viên',te.empName AS 'Họ và tên',
        YEAR(GETDATE())-YEAR(te.empBirthdate) AS 'Tuổi',
        te.empSalary AS 'Lương',
        CASE WHEN te.empSex ='F' THEN N'Nữ' ELSE N'Nam' END AS 'Giới tính'
FROM tblEmployee te  WHERE te.depNum=1;
```

Fpt University

Modifying Views

- With *updatable views*, the modification is translated into an equivalent modification on a base table

The modification can be done to the base table

Fpt University

Updatable Views

Example 4:

- Create view from table Employee
- Do changes on Employee and review created view
- Do changes on created view and review Employee

Fpt University

Update on table effects on view

```
CREATE VIEW Employee_Dep1v2 AS
SELECT te.empSSN,te.empName,te.empSalary,te.empSex
FROM tblemployee te WHERE te.depnum=1;


INSERT INTO tblEmployee (empSSN, empName, empSalary, empSex, depNum)
    VALUES (100000,N'Lê Văn Tám',100000,'M',1)

SELECT * FROM tblEmployee te WHERE te.depNum=1

SELECT * FROM Employee_Dep1v2
```

Update on view effects on table with unexpected result

```
CREATE VIEW Employee_Dep1v2 AS
SELECT te.empSSN,te.empName,te.empSalary,te.empSex
FROM tblemployee te WHERE te.depnum=1;


INSERT INT Employee_Dep1v2 VALUES (100001,N'Lê Văn Bảy',100000,'M')

SELECT * FROM tblEmployee te WHERE te.depNum=1

SELECT * FROM Employee_Dep1v2
```

Update on view raises error on table

```
CREATE VIEW Employee_Dep1v3 AS
SELECT te.empSSN AS 'Mã số nhân viên',te.empName AS 'Họ và tên',
        YEAR(GETDATE())-YEAR(te.empBirthdate) AS 'Tuổi',
        te.empSalary AS 'Lương',
        CASE WHEN te.empSex ='F' THEN N'Nữ' ELSE N'Nam' END AS 'Giới tính'
FROM tblEmployee te  WHERE te.depNum=1;

INSERT INTO Employee_Dep1v3 VALUES (100002,N'Lê Văn Chín',30,90000,N'Nam')
GO
```

Update on view raises error on table

EX:

```
create view v_Student_AgeGreaterThan18
as
select * from Student
where Age > 20

select * from v_Student_AgeGreaterThan18
```

76 %

Results | Messages

	Id	Name	Age	DateOfBirth	Gender	Province1	District
1	2	Nguyen Van B	21	1990-09-13	1	NULL	NULL

Update on view raises error on table

Select * from v_Student_AgeGreaterThan18

Id	Name	Age	DateOfBirth	Gender	Province1	District
1	Nguyen Van A	20	1990-09-13	1	NULL	NULL
2	Nguyen Van B	21	1990-09-13	1	NULL	NULL
3	Nguyen Van C	19	1990-09-13	1	NULL	NULL

```
Update  v_Student_AgeGreaterThan18
Set Province1='Ha Noi', District='Cau Giay'
Where Id=1
```

Id	Name	Age	DateOfBirth	Gender	Province1	District
1	Nguyen Van A	20	1990-09-13	1	Ha Noi	Cau Giay
2	Nguyen Van B	21	1990-09-13	1	NULL	NULL
3	Nguyen Van C	19	1990-09-13	1	NULL	NULL

Update on view raises error on table

Id	Name	Age	DateOfBirth	Gender	Province1	District
1	Nguyen Van A	20	1990-09-13	1	Ha Noi	Cau Giay
2	Nguyen Van B	21	1990-09-13	1	NULL	NULL
3	Nguyen Van C	19	1990-09-13	1	NULL	NULL

Insert into v_Student_AgeGreaterThan18
Values(4,'Nguyen Van D',22,'1992-01-12',1,'Ha Noi','Cau Giay'

Id	Name	Age	DateOfBirth	Gender	Province1	District
1	Nguyen Van A	20	1990-09-13	1	Ha Noi	Cau Giay
2	Nguyen Van B	21	1990-09-13	1	NULL	NULL
3	Nguyen Van C	19	1990-09-13	1	NULL	NULL
4	Nguyen Van D	22	1992-01-12	1	Ha Noi	Cau Giay

Delete from v_Student_AgeGreaterThan18
Where Id=4

Id	Name	Age	DateOfBirth	Gender	Province1	District
1	Nguyen Van A	20	1990-09-13	1	Ha Noi	Cau Giay
2	Nguyen Van B	21	1990-09-13	1	NULL	NULL
3	Nguyen Van C	19	1990-09-13	1	NULL	NULL

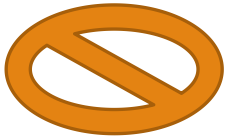
Fpt University

View Removal

As we know, Employee_Dep1 is associated to tplEmployee relation

DROP VIEW Employee_Dep1;

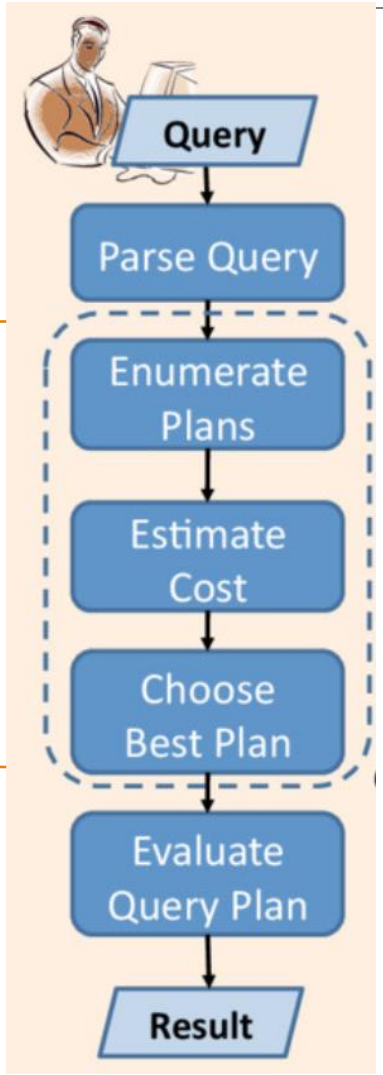
- Delete the definition of the view
- Does not effect on tplEmployee relation



DROP TABLE tplEmployee;

- Delete tplEmployee relation
- Make the view Employee_Dep1 unusable

Query optimization



- In practice:

1. Define the requirements: *Who? What? Where? When? Why?*
2. SELECT fields instead of using SELECT *
3. Avoid SELECT DISTINCT
4. Indexing
5. Create joins with INNER JOIN (not WHERE)
6. To check the existence of records, use EXISTS() rather than COUNT()
7. Ignore linked subqueries
8. Use of temp table
9. Don't run queries in a loop
10. Limit your working data set size