UNIVERSITY OF MINES AND TECHNOLOGY

ADVANCED DATABASE
CE 280

PRESENTED BY: DR ERIC AFFUM

Query Optimization Process

- **Query processing** refers to the range of activities involved in extracting data from a database to process the query and generate result.

- The steps involved in processing a query are :-

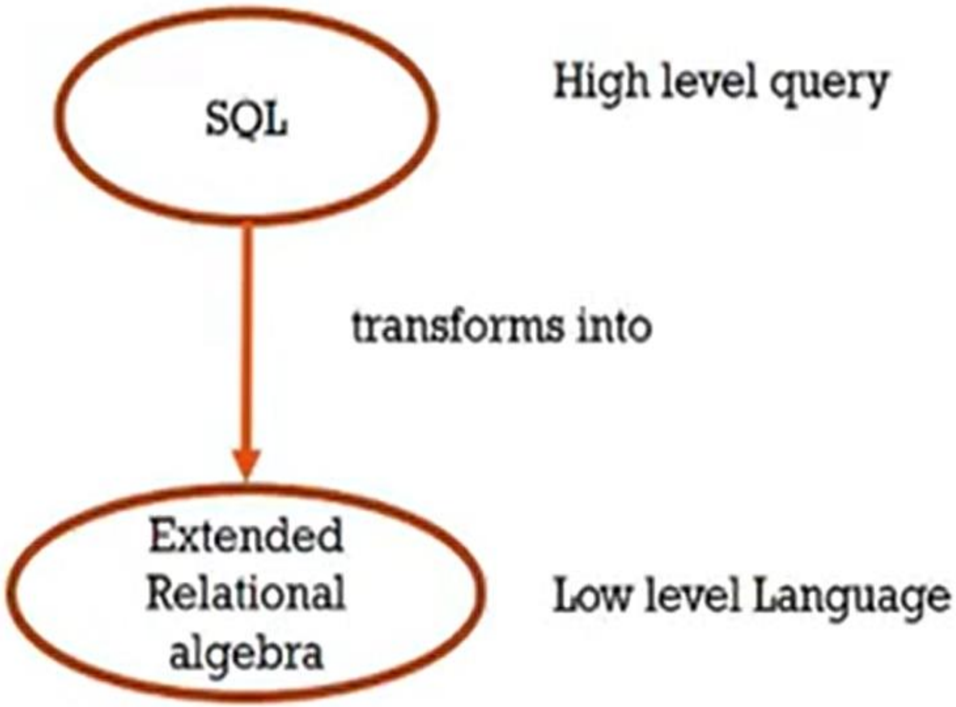
✓ **1.** Parsing and translation

✓ **2.** Optimization

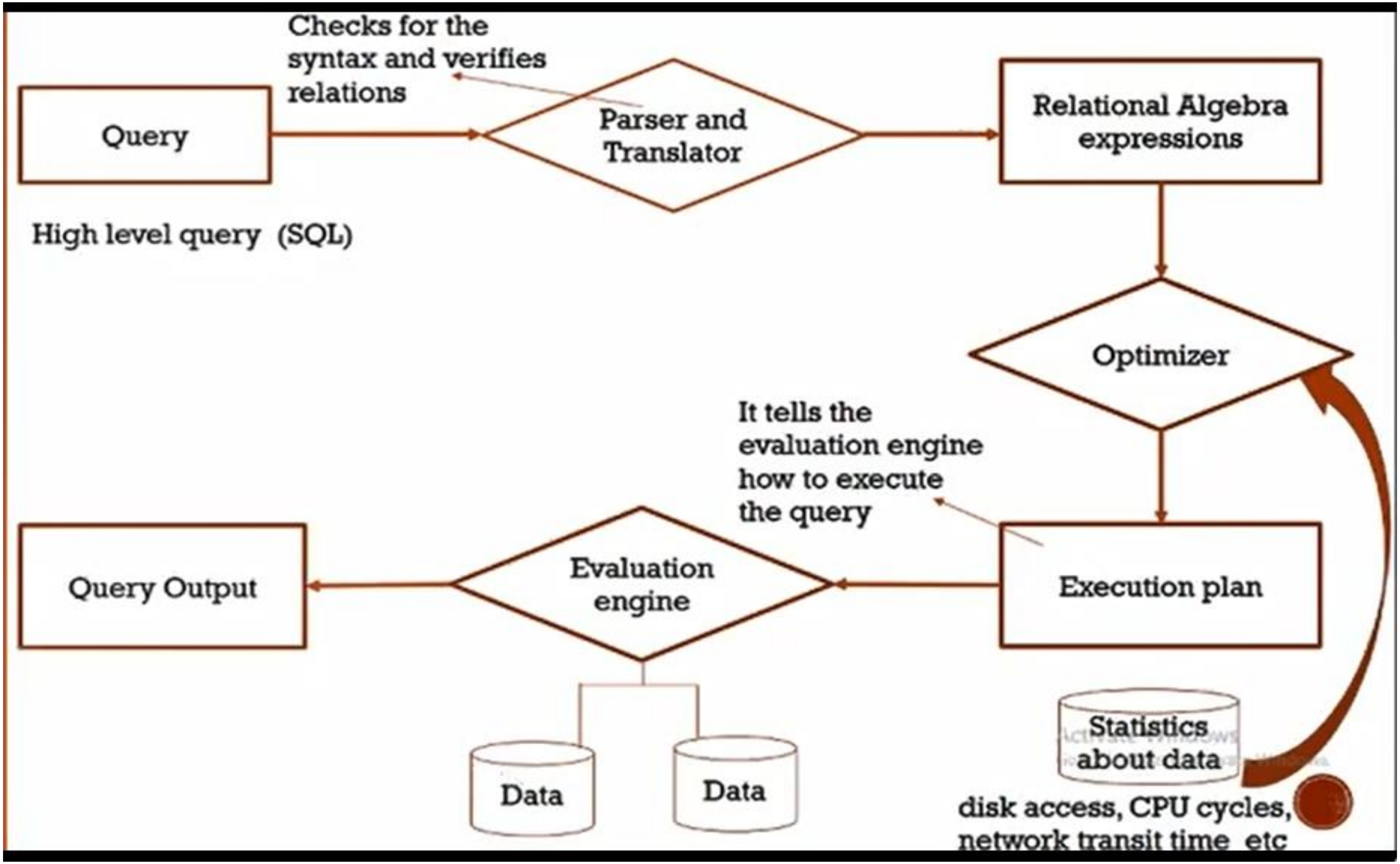
✓ **3.** Evaluation

Query Optimization Process

- Before processing the query (which is the SQL query) , the system must translate the query into a usable form(language which system can understand).
- A more useful internal representation is based on the extended relational algebra.



Query Optimization Process



Query Optimization Process

- **The parser** checks the syntax of the user's query, e.g verifies that the relation names appearing in the query are names of the relations in the database, and so on

- **Query Execution Plan** - sequence of primitive operations used to evaluate a query

$$\pi_{balance}$$

$$\sigma_{balance < 2500}$$

account

- **The query-execution engine** takes a query-evaluation plan, executes that plan, and returns the answers to the query

Query Optimization

As an illustration, consider the query

select *balance*
from *account*
where *balance* < 2500;

This query can be translated into either of the following relational-algebra expressions:

- $\sigma_{balance<2500} (\Pi_{balance} (account))$
- $\Pi_{balance} (\sigma_{balance<2500} (account))$

The query which takes less CPU time, CPU cycles, or disk access will have less cost and will be executed

Query Optimization

- The different evaluation plans created during query processing step for a given query can have different costs.

- We do not expect users to write their queries in a way that suggests the most efficient evaluation plan.

- Rather, it is the responsibility of the system to construct a query-evaluation plan that minimizes the cost* of query evaluation.

This is where query optimization comes into play

*Cost of the Query - **Optimizers make use of statistical information about the relations, such as relation sizes and index depths, to make a good estimate of the cost of a plan. Disk access, which is slow compared to memory access, usually dominates the cost of processing a query.**

Query Optimization

Consider the relational-algebra expression for the query :-

"Find the names of all customers who have an account at any branch located in Brooklyn."

$$\Pi_{customer\text{-}name}\ (\sigma_{branch-city\ =\ \text{"Brooklyn"}}\ (branch\ X\ (account\ X\ depositor)))$$

whereas: -

$$\Pi_{customer\text{-}name}\ (\ (\sigma_{branch\text{-}city\ =\ \text{"Brooklyn"}}\ (\mathbf{branch}))\ \mathbf{X}\ (\mathbf{account\ X\ depositor}))$$

Query Optimization

Consider the relational-algebra expression for the query :-

"Find the names of all customers who have an account at any branch located in Brooklyn."

$$\Pi_{customer\text{-}name}\,(\sigma_{branch-city = \text{"Brooklyn"}}\,(\overset{100}{branch} \times (\overset{100}{account} \times \overset{100}{depositor})))$$

whereas: -

$$\Pi_{customer\text{-}name}\,(\,(\overset{50}{\sigma_{branch\text{-}city = \text{"Brooklyn"}}\,(branch)}) \times (\overset{100}{account} \times \overset{100}{depositor}))$$

Query Optimization

Consider the relational-algebra expression for the query :-

“Find the names of all customers who have an account at any branch located in Brooklyn.”

$$\Pi_{customer\text{-}name}\ (\sigma_{branch\text{-}city\ =\ \text{“Brooklyn”}}\ (\overset{100}{branch}\ X\ (\overset{100}{account}\ X\ \overset{100}{depositor})))$$

This expression constructs **a large intermediate relation**, *branch X account X depositor*

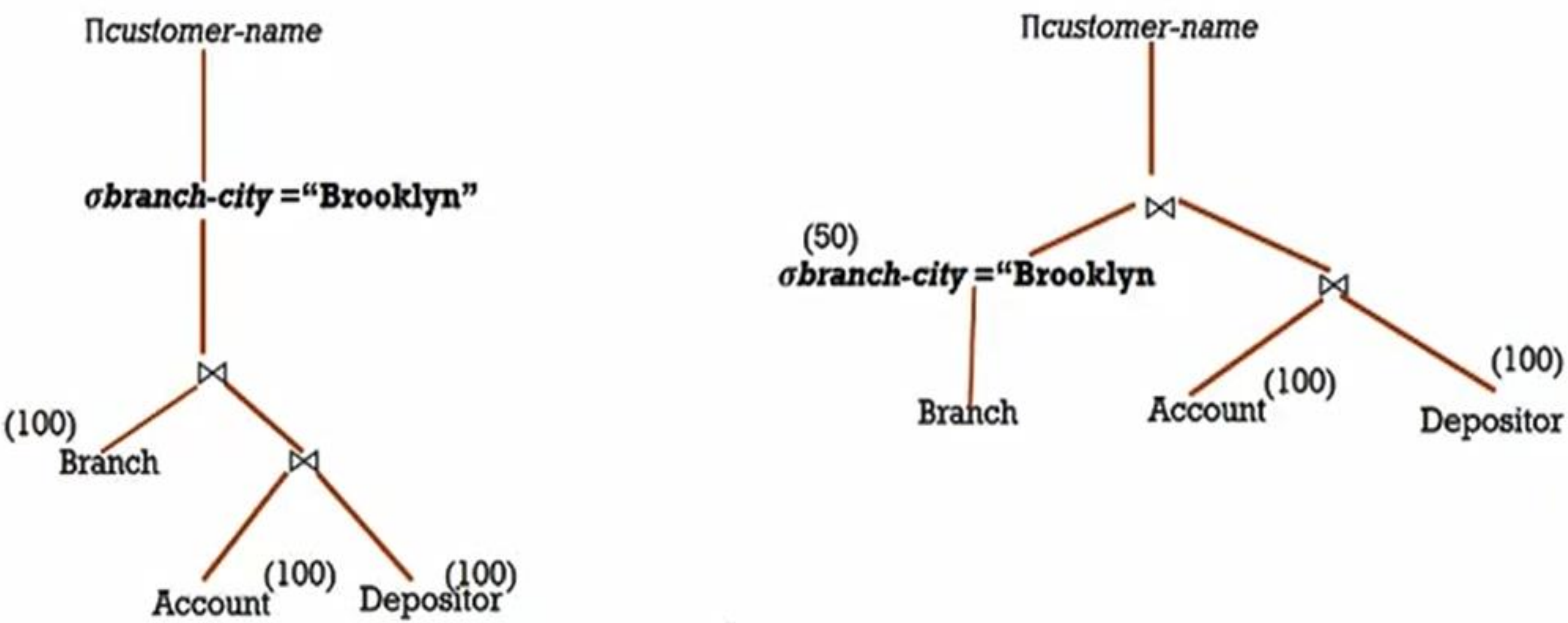
whereas: -

$$\Pi_{customer\text{-}name}\ (\ (\sigma_{branch\text{-}city\ =\ \overset{50}{\text{“Brooklyn”}}}\ (\mathbf{branch}))\ \mathbf{X}\ (\overset{100}{\mathbf{account}}\ \mathbf{X}\ \overset{100}{\mathbf{depositor}}))$$

which is equivalent to our original algebra expression, but which generates **smaller intermediate relation**

Query Optimization

Figure below depicts the initial expression and the final expression after all these transformations



a) Initial expression tree

b) expression tree after several transformations

Equivalent Expressions

Query Optimization

Generation of query-evaluation plans involves two steps:

(1) generating expressions that are **logically equivalent** to the given expression,

(2) estimating **the cost of each** evaluation plan.

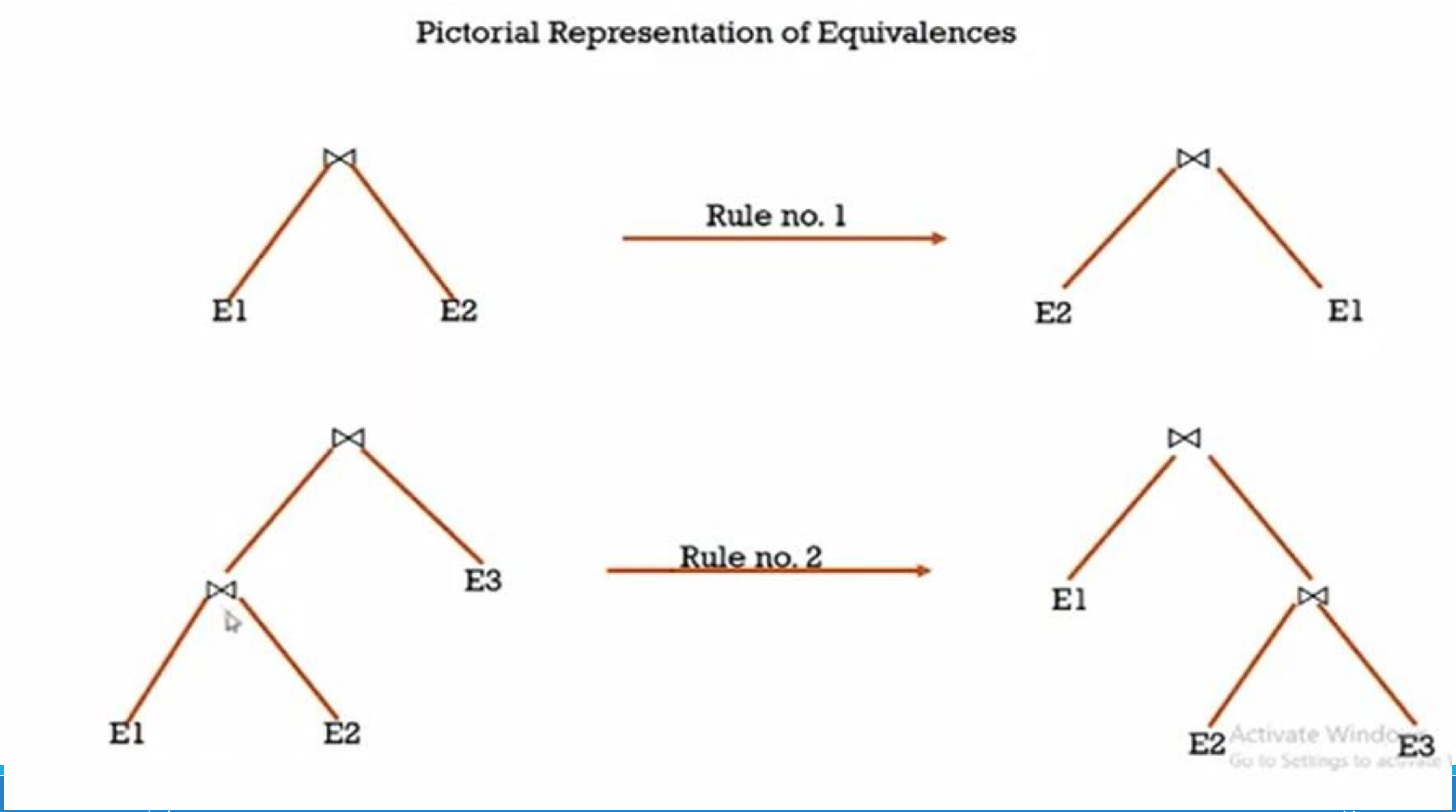
To implement the first step, **the query optimizer** must generate expressions equivalent to a given expression. It does so by means of ***equivalence rules*** that specify how to transform an expression into a logically equivalent one

Query Optimization

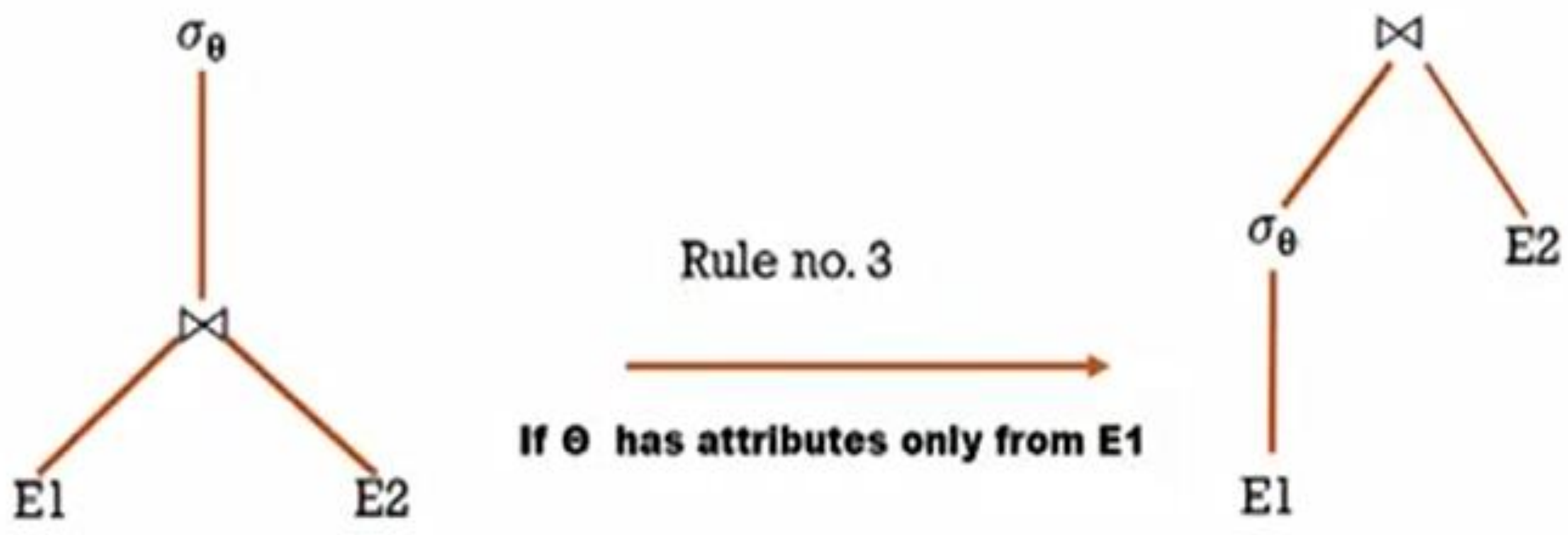
Equivalence Rules

An **equivalence rule** says that expressions of two forms are equivalent. We can replace an expression of the first form by an expression of the second form, or vice versa

Query Optimization: Equivalence Rule



Query Optimization: Equivalence Rule



Pictorial Representation of Equivalences

Query Optimization: Example

Examples of Transformations

We now illustrate the use of the equivalence rules. We use bank example with the relation schemas:

Example -1

- *Branch-schema = (branch-name, branch-city, assets)*
- *Account-schema = (account-number, branch-name, balance)*
- *Depositor-schema = (customer-name, account-number)*

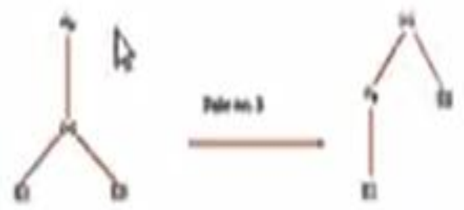
Query Optimization: Example

The relations *branch*, *account*, and *depositor* are instances of these schemas. The expression

Π*customer-name*(σ*branch-city* ="Brooklyn"(***branch* X (*account* X *depositor*)))**

↓

Appling rule no.3



Π*customer-name*((σ***branch-city* ="Brooklyn"(*branch*)) X (*account* X *depositor*))**

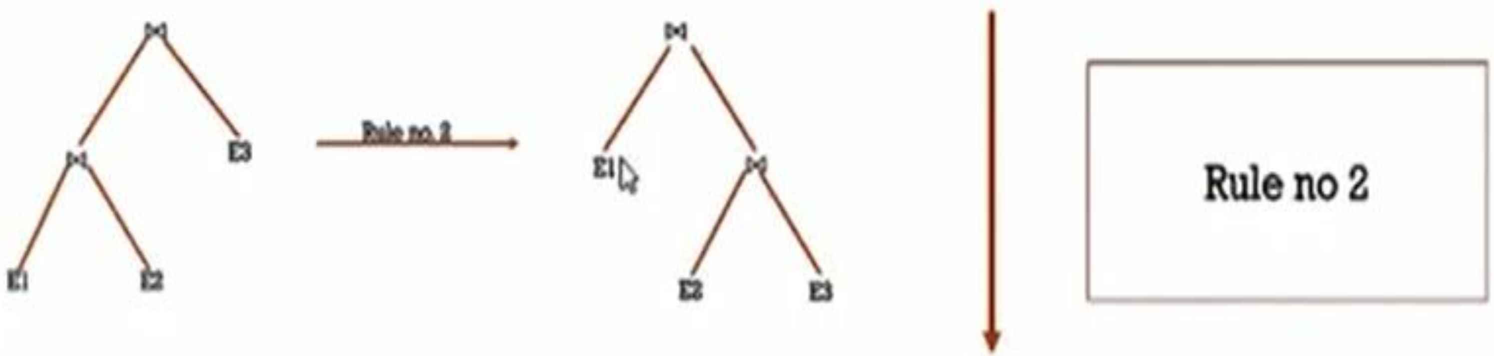
which is equivalent to our original algebra expression, but generates smaller intermediate relations.

Query Optimization: Example

Suppose that we modify our original query to restrict attention to customers who have a balance over $1000.

The new relational-algebra query is :-

Π*customer-name* (σ*branch-city* ="Brooklyn" $\wedge$ *balance* >1000 (**branch X (account X depositor**)))



***branch* X (*account* X *depositor*)** → **(*branch* X *account*) X *depositor*:**

Query Optimization: Example

Query after rule no 2

100 100 100

$\Pi_{customer\text{-}name}$ ($\sigma_{branch\text{-}city}$ ="Brooklyn" $\wedge$ *balance* >1000 ((**branch X account**) **X depositor**))

Using rule no 3

20

200 X 100

$\Pi_{customer\text{-}name}$ ((σ**branch-city ="Brooklyn"** $\wedge$ **balance>1000 (branch X account)**) X *depositor*)

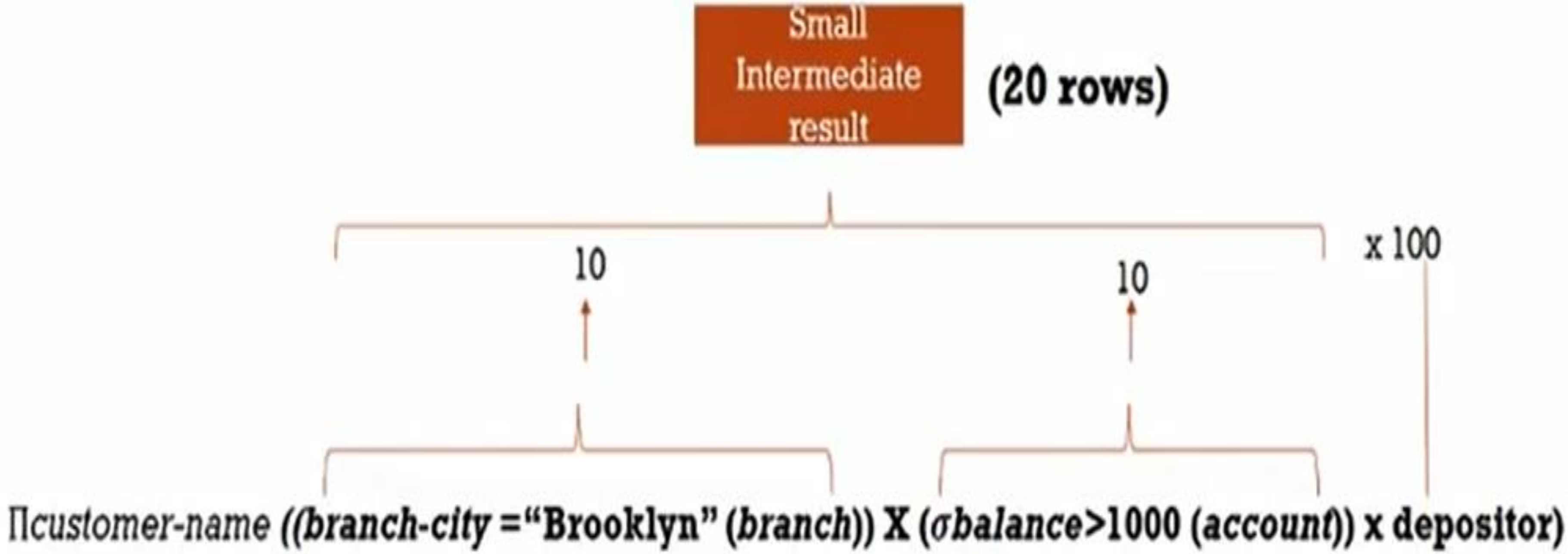
We can break the selection into two selections, to get the following subexpression:

10 10

***σbranch-city* ="Brooklyn" (*branch*)** **X** ***σbalance*>1000 (account)**

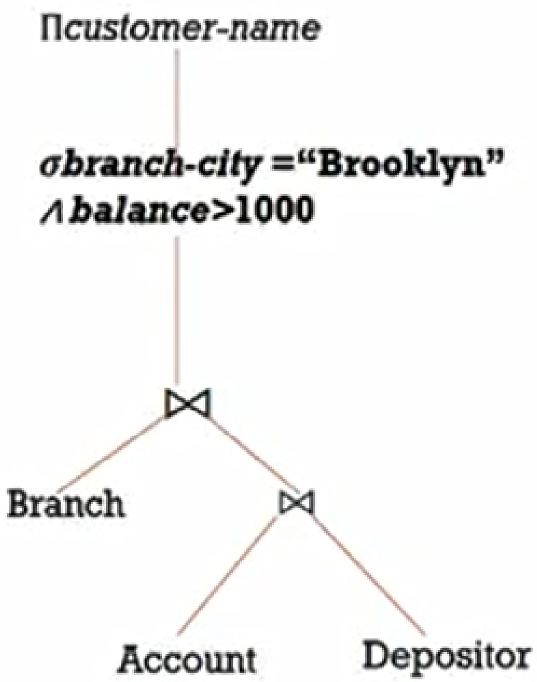
Both of the preceding expressions select tuples with *branch-city* = "Brooklyn" and *balance* > 1000. However, the latter form of the expression provides a new opportunity to apply the **"perform selections early" rule**.

Query Optimization: Example

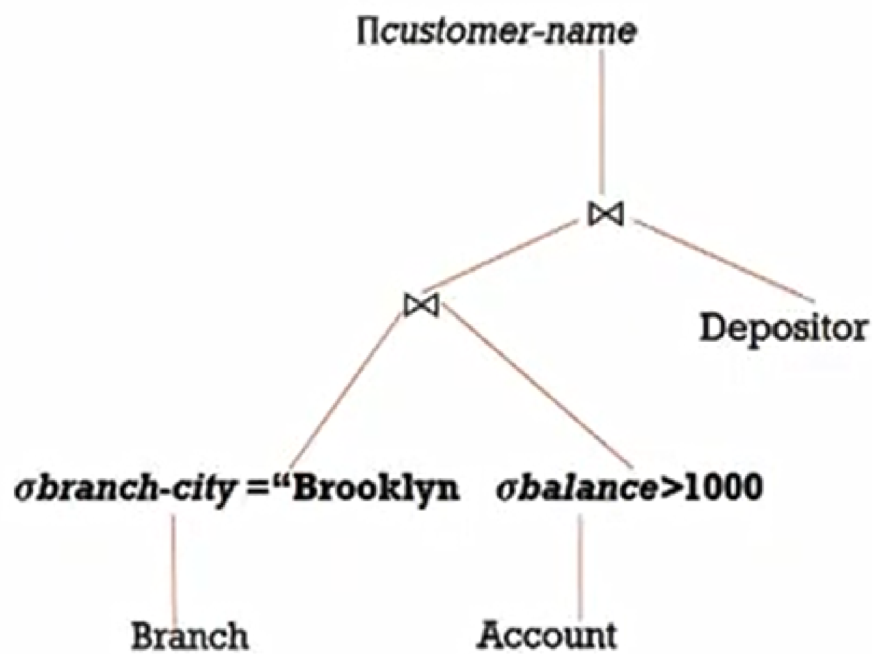


Query Optimization: Example

Figure below depicts the initial expression and the final expression after all these transformations



a) Initial expression tree



b) expression tree after several transformations

Query Optimization Process

- The **different evaluation plans** for a given query can have **different costs**.
- The query having **least cost is** selected by optimizer.
- **Query evaluation plan is generated** by Query evaluation plan module.
- Once the query plan is chosen, the query is evaluated with that plan by **query evaluation engine**, and the result of the query is output.

Some Ways to Optimize SQL Queries

Use varchar/nvarchar instead of char/nchar whenever possible

Inefficient example:

```
`deptName` char(100) DEFAULT NULL
```

efficient example:

```
`deptName` varchar(100) DEFAULT NULL
```

Use numeric fields as much as possible. If the fields only contain numeric information, try not to design them as a character type.

Inefficient Example:

`Emp_id` varchar（20） NOT NULL;

Efficient example:

`Emp_id` int(11) NOT NULL;

Try to replace union with union all

If there are no duplicate records in the search results, it is recommended to replace union with union all.

Inefficient example:

```
select * from Emp where Empid=1
union
select * from Emp where age = 10
```

Efficient example:

```
select * from Emp where Empid=1
union all
select * from Emp where age = 10
```

Use the distinct keyword with caution

.

Inefficient example:

```
SELECT DISTINCT * from  user;
```

Efficient example:

```
select DISTINCT name from user;
```

Try not to use select * to query SQL, but select specific fields.

Inefficient example:

```
select * from employee;
```

Efficient example:

```
select id, name from employee;
```

If you know that there is only one query result, it is recommended to use limit 1

Inefficient example:

select id, name from employee where name='lucky';

Efficient example:

select id, name from employee where name='lucky' limit 1;

Optimize your like statement

In daily development, if you use fuzzy keyword queries, it is easy to think of like, but like is likely to invalidate your index.

Inefficient example:

```
select userId,   name from user where userId like '%123';
```

Efficient example:

```
select userId,   name from user where userId like '123%';
```

You should avoid using the != or <> operator in the where clause as much as possible, otherwise the engine will give up using the index and perform a full table scan

Inefficient example:

```
select age,name  from user where age <>18;
```

Efficient example:

```
//You can consider separate two sql write
select age,name  from user where age <18;
select age,name  from user where age >18;
```