PHYSICAL DB ISSUES, INDEXES, QUERY OPTIMIZATIONS

Introduction to Database Systems

Mohammad Tanhaei
Ilam University

IN THIS LECTURE

- Physical DB Issues
  - RAID arrays for recovery and speed
  - Indexes and query efficiency
- Query optimization
  - Query trees
- For more information
  - Connolly and Begg chapter 21 and appendix C.5

PHYSICAL DESIGN

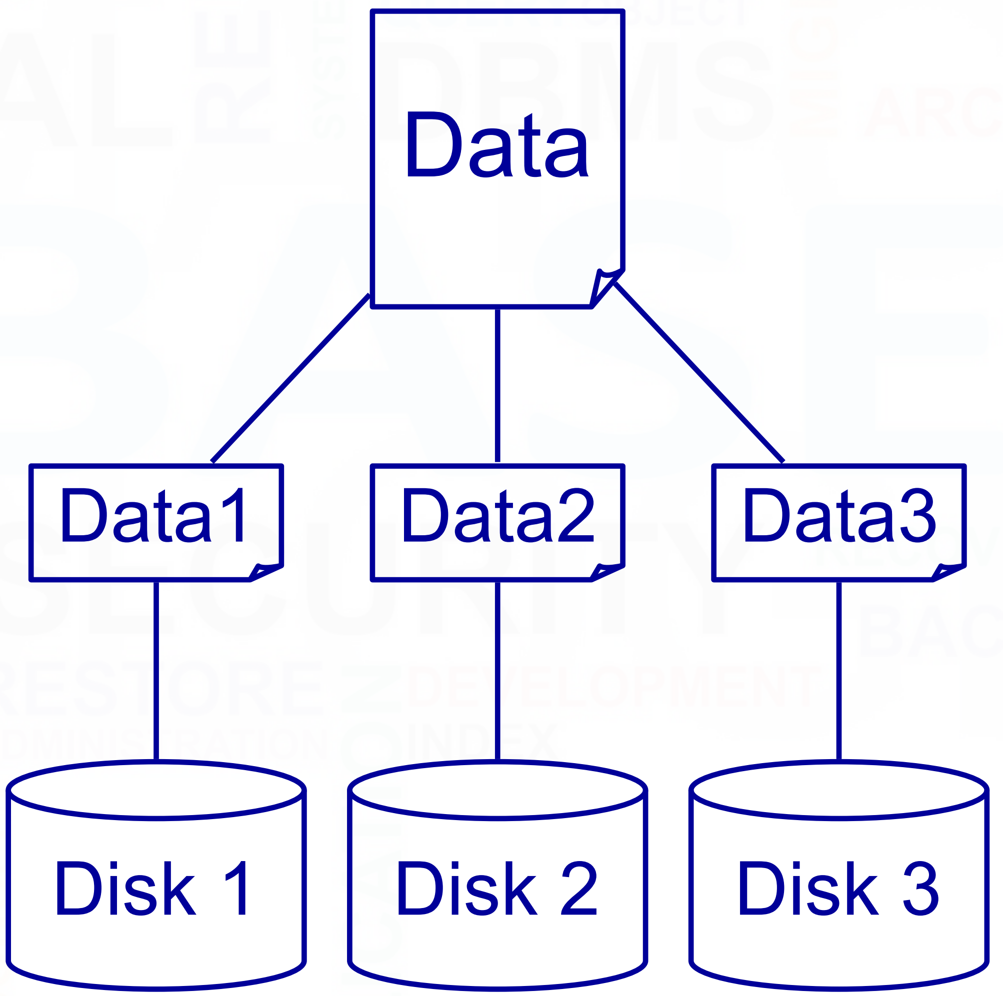
- Design so far
  - E/R modeling helps **find** the requirements of a database
  - Normalization helps to **refine** a design by removing data redundancy
- Physical design
  - Concerned with storing and accessing the data
  - How to deal with media failures
  - How to access information efficiently

RAID ARRAYS

- **RAID** - **R**edundant **A**rray of **I**ndependent (inexpensive) **D**isks
  - Storing information across more than one physical disk
  - **Speed** - can access more than one disk
  - **Robustness** - if one disk fails it is OK
- RAID techniques
  - **Mirroring** - multiple copies of a file are stored on separate disks
  - **Striping** - parts of a file are stored on each disk
  - Different levels (RAID 0, RAID 1…)

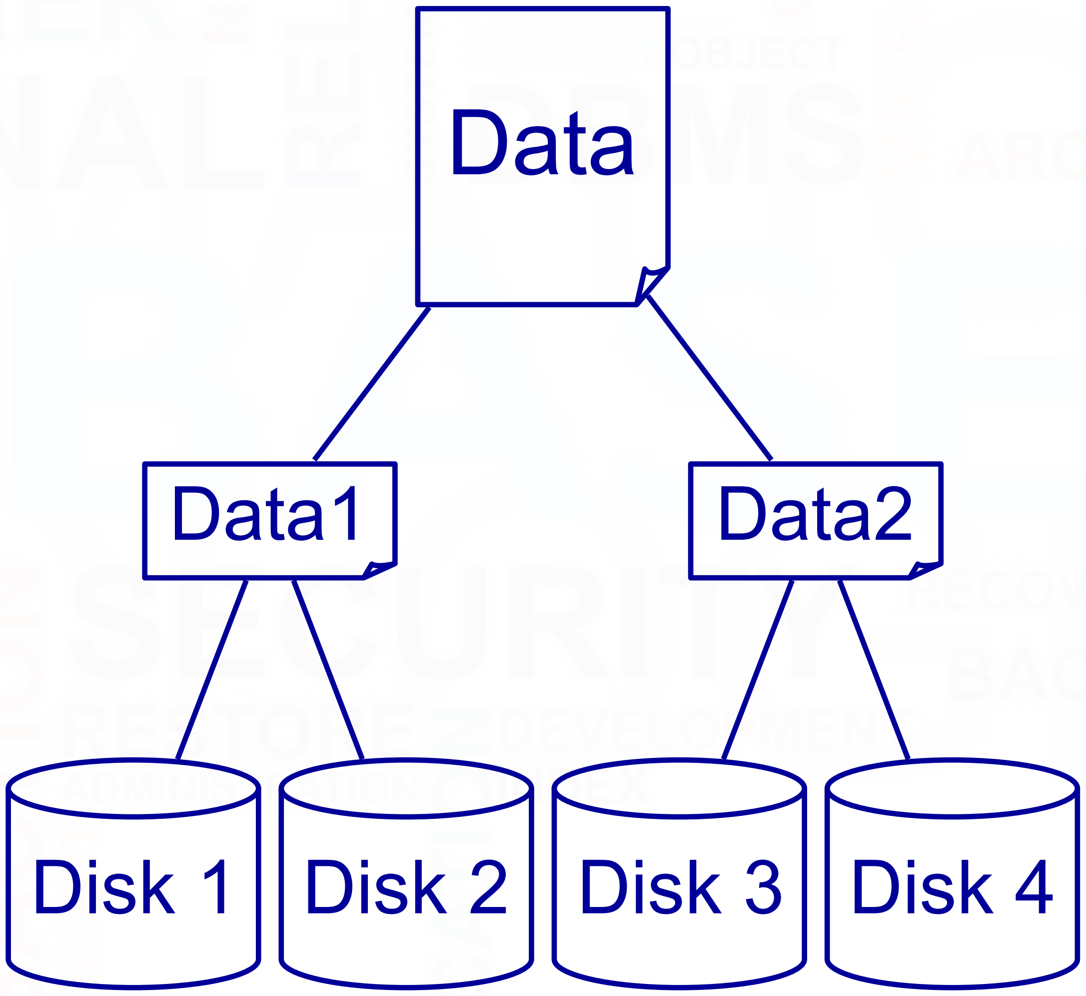
RAID LEVEL 0

- Files are **split** across several disks
  - For a system with **n** disks, each file is split into **n** parts, one part stored on each disk
  - **Improves speed**, but **no redundancy**



RAID LEVEL 1

- As RAID 0 but with **redundancy**
  - Files are split over multiple disks
  - Each disk is mirrored
  - For **n** disks, split files into **n/2** parts, each stored on 2 disks
  - **Improves speed**, **has redundancy**, but **needs lots of disks**



PARITY CHECKING

- We can use parity checking to reduce the number of disks
  - **Parity** - for a set of data in binary form we count the number of **1s** for each bit across the data
  - If this is even the parity is 0, if odd then it is 1

1	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1
1	0	1	0	1	0	0	1
0	1	1	0	1	1	1	0
0	1	0	0	0	1	1	1

RECOVERY WITH PARITY

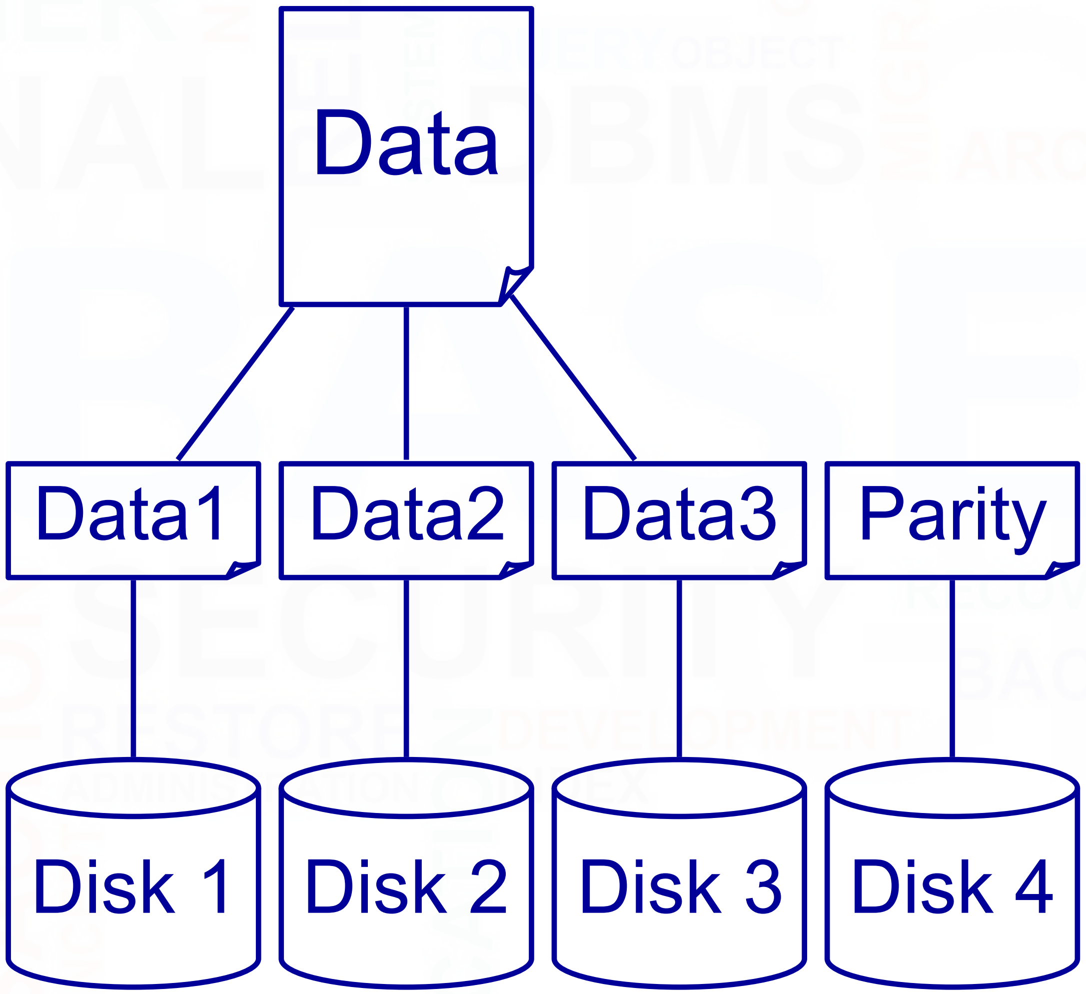
- If one of our pieces of data is lost we can **recover it**
  - Just compute it as the parity of the remaining data and our original parity information

```
1 0 1 1 0 0 1 1
0 0 1 1 0 0 1 1

0 1 1 0 1 1 1 0
_______________
0 1 0 0 0 1 1 1
```

RAID LEVEL 3

- Data is **striped** over disks, and a **parity disk** for **redundancy**
  - For **n** disks, we split the data in **n-1** parts
  - Each part is stored on a disk
  - The final disk stores parity information



OTHER RAID ISSUES

- Other RAID levels consider
  - How to split data between disks
  - Whether to store parity information on one disk, or spread across several
  - How to deal with multiple disk failures
- Considerations with RAID systems
  - Cost of disks
  - Do you need speed or redundancy?
  - How reliable are the individual disks?
  - ‘Hot swapping’
  - Is the disk the weak point anyway?

INDEXES

- Indexes are to do with ordering data
  - The relational model says that order doesn't matter
  - From a practical point of view it is very important
- Types of indexes
  - **Primary** or **clustered** indexes affect the **order** that the data is **stored** in a file
  - **Secondary** indexes give a **look-up table** into the file
  - Only one primary index, but many secondary ones

INDEX EXAMPLE

- A telephone book
  - You store people's addresses and phone numbers
  - Usually you have a name and want the number
  - Sometimes you have a number and want the name
- Indexes
  - A clustered **index** can be made on **name**
  - A secondary **index** can be made on **number**

INDEX EXAMPLE

As a Table

Name	Number
John	925 1229
Mary	925 8923
Jane	925 8501
Mark	875 1209

Order does not really concern us here

As a File

Jane, 9258501
John, 9251229
Mark, 8751209
Mary, 9258923

Most of the time we look up numbers by name, so we sort the file by name

Secondary Index

8751209
9251229
9258501
9258923

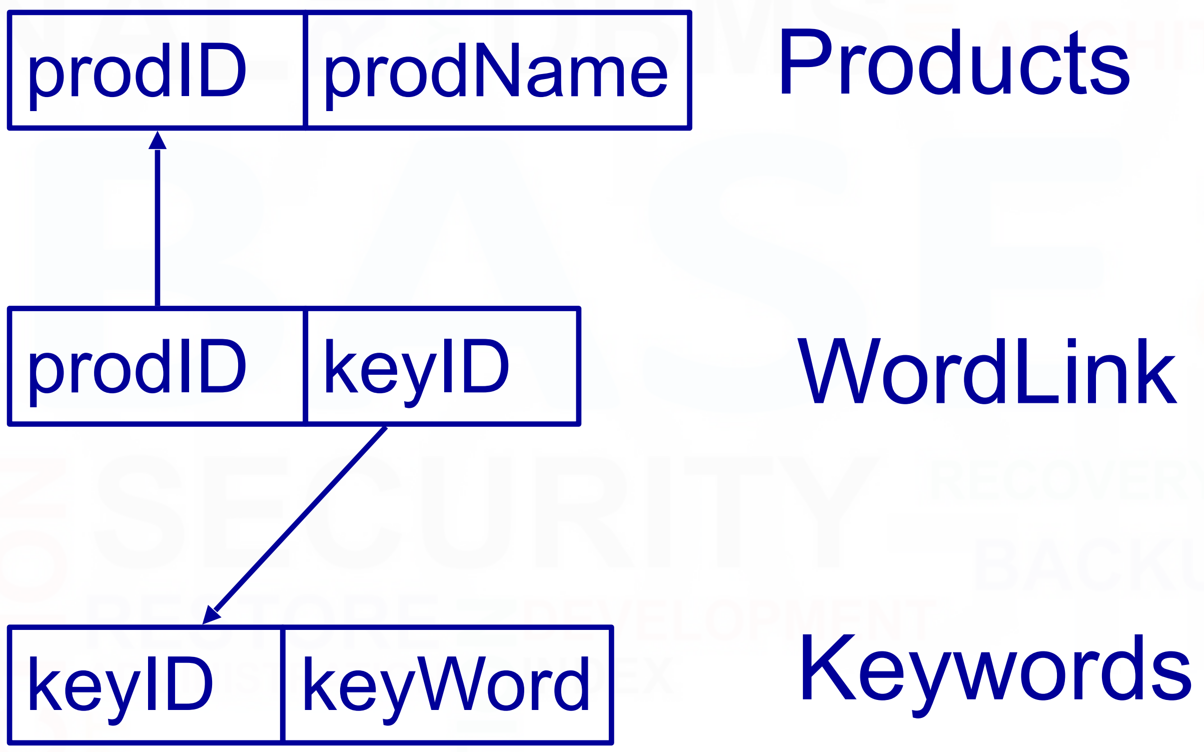
Sometimes we look up names by number, so we index number

CHOOSING INDEXES

- You can only have **one primary index**
  - The most frequently looked-up value is often the best choice
  - Some DBMSs assume the primary key is the primary index, as it is usually used to refer to rows
- Don’t create too many indexes
  - They can **speed up queries**, but **they slow down inserts**, **updates** and **deletes**
  - Whenever the data is changed, the index may need to change

INDEX EXAMPLE

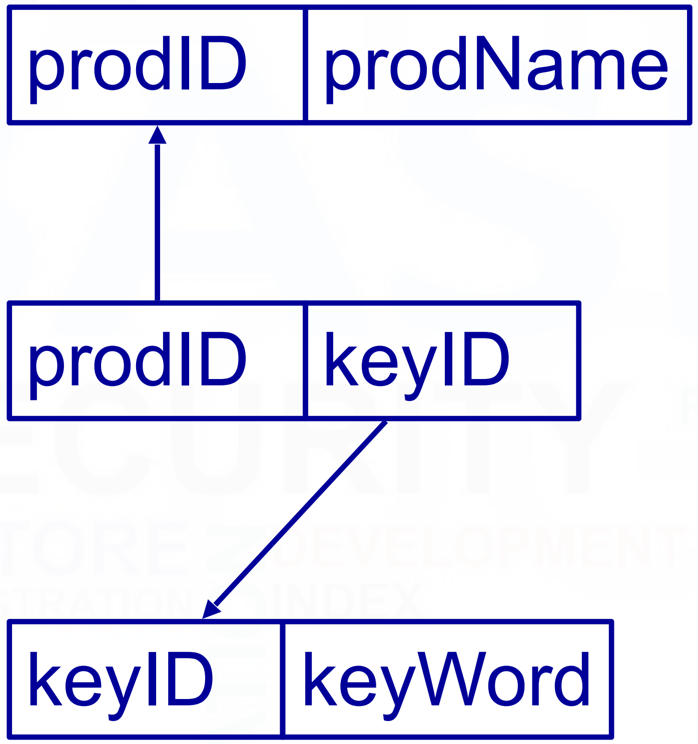
- A product database, which we want to search by keyword
  - Each product can have many keywords
  - The same keyword can be associated with many products



INDEX EXAMPLE

- To search the products given a keyWord value

  1. We look up the keyWord in Keywords to find its keyID
  2. We look up that keyID in WordLink to find the related prodIDs
  3. We look up those prodIDs in Products to find more information about them



CREATING INDEXES

- In SQL we use **CREATE INDEX**:

```
CREATE INDEX
  <index name>
  ON <table>
  (<columns>)
```

- Example:

```
CREATE INDEX keyIndex ON
  Keywords (keyWord)
CREATE INDEX linkIndex
  ON WordLink(keyID)
CREATE INDEX prodIndex
  ON Products (prodID)
```

QUERY PROCESSING

- Once a database is designed and made we can query it
  - A query language (such as SQL) is used to do this
  - The query goes through several stages to be executed
- Three main stages
  - **Parsing** and **translation** - the query is put into an internal form
  - **Optimization** - changes are made for efficiency
  - **Evaluation** - the optimized query is applied to the DB

PARSING AND TRANSLATION

- SQL is a good language for people
  - It is quite high level
  - It is non-procedural
- Relational algebra is better for machines
  - It can be reasoned about more easily
- Given an SQL statement we want to find an equivalent relational algebra expression
- This expression may be represented as a tree - the query tree

SOME RELATIONAL OPERATORS

- Product ×
  - Product finds all the combinations of one tuple from each of two relations
  - R1 × R2 is equivalent to

```
SELECT DISTINCT *
  FROM R1, R2
```

- Selection σ
  - Selection finds all those rows where some condition is true
- σ_{cond} R is equivalent to

```
SELECT DISTINCT *
  FROM R
 WHERE <cond>
```

SOME RELATIONAL OPERATORS

- Projection π
  - Projection chooses a set of attributes from a relation, removing any others
- $\pi_{A1,A2,\ldots}$ R is equivalent to

```
SELECT DISTINCT
  A1, A2, ...
  FROM R
```

- Projection, selection and product are enough to express queries of the form

```
SELECT <cols>
  FROM <table>
 WHERE <cond>
```

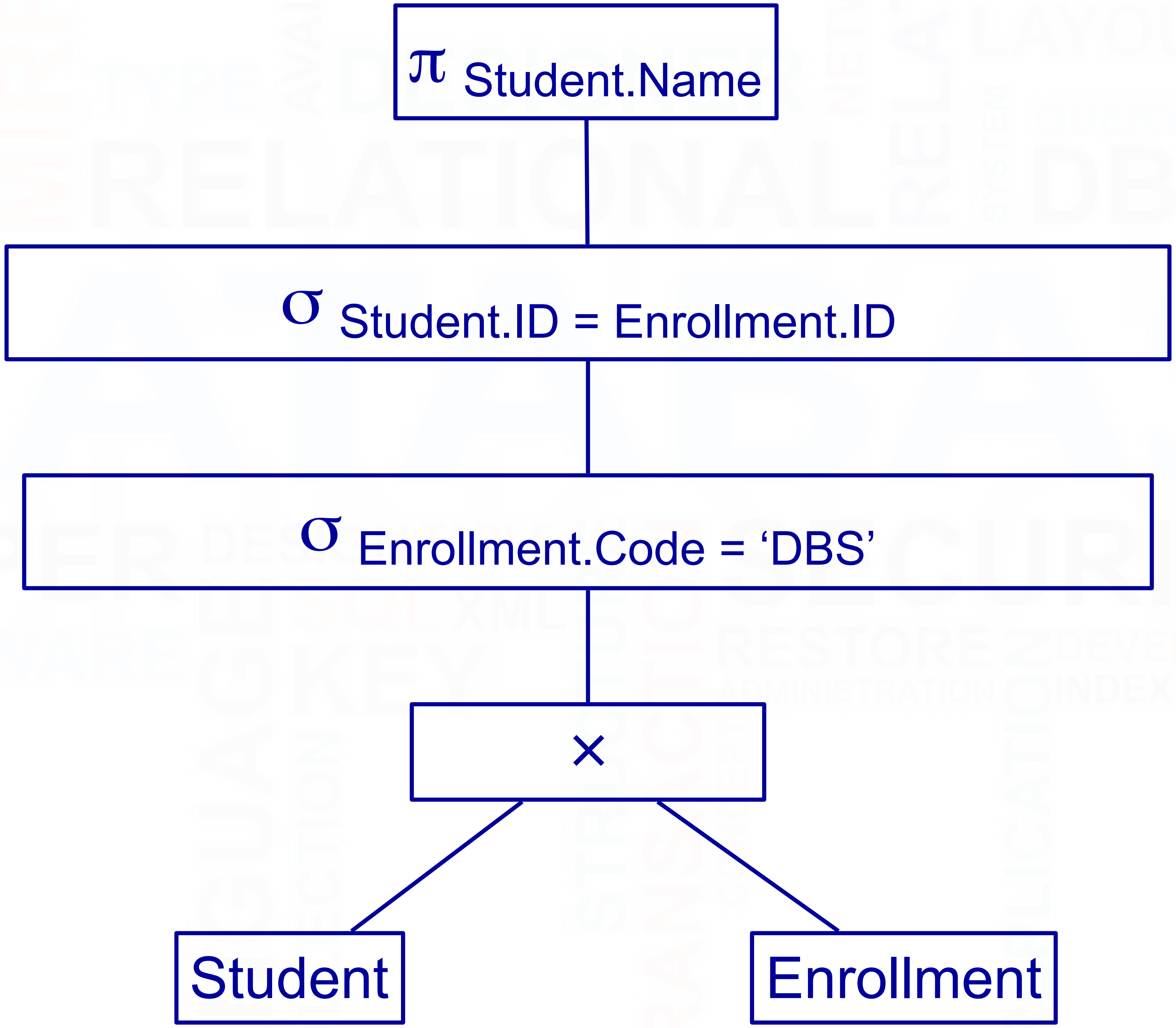
SQL → RELATIONAL ALGEBRA

- SQL statement

```
SELECT
  Student.Name
FROM
  Student,Enrollment
WHERE
Student.ID = Enrollment.ID
    AND
Enrollment.Code = 'DBS'
```

- Relational Algebra
  - Take the product of Student and Enrollment
  - Select tuples where the IDs are the same and the Code is DBS
  - Project over Student.Name

QUERY TREE


π Student.Name
σ Student.ID = Enrollment.ID
σ Enrollment.Code = 'DBS'
×
Student
Enrollment

OPTIMIZATION

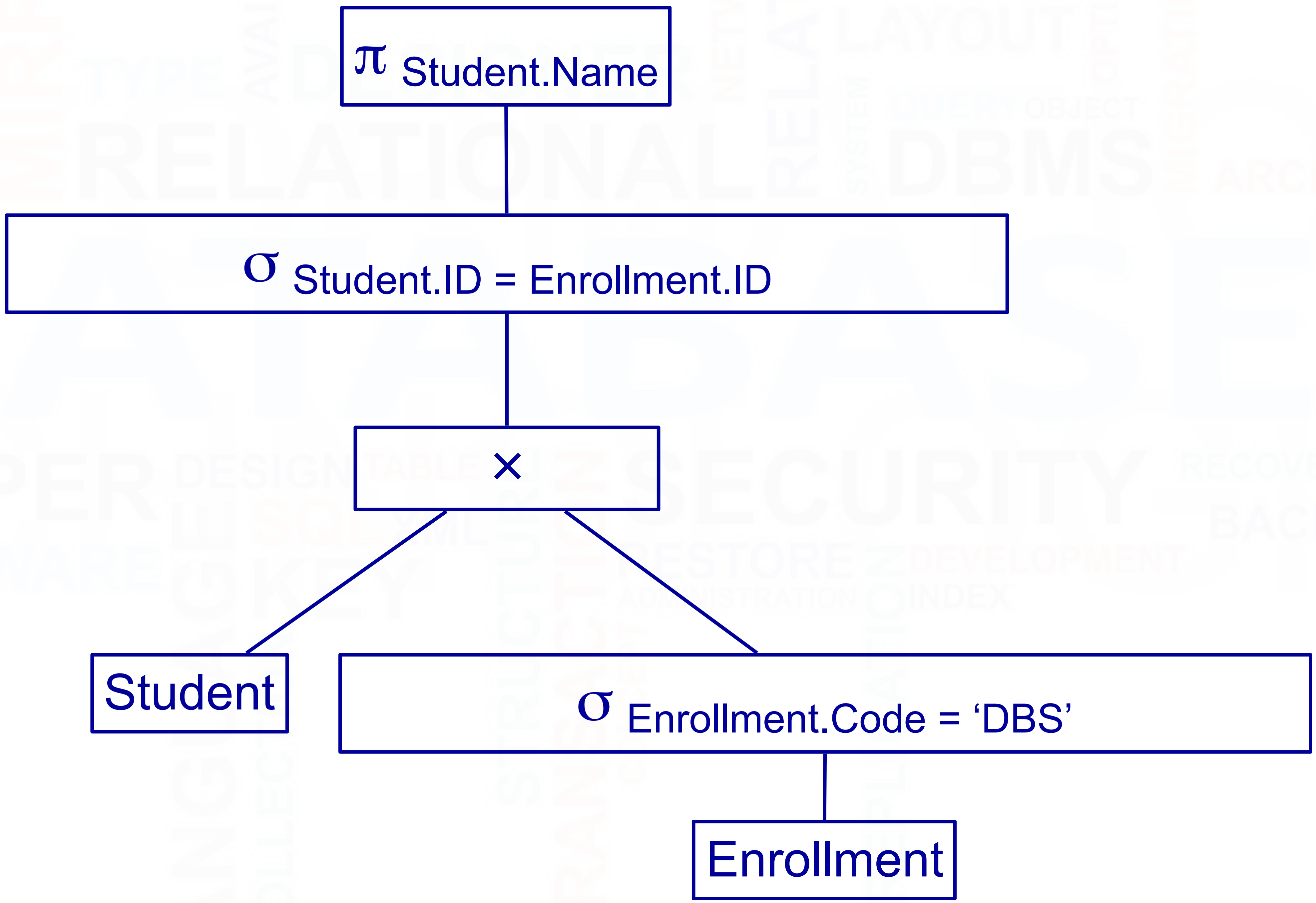
- There are often many ways to express the same query
- Some of these will be **more efficient** than others
- Need to find a good version

- Many ways to optimize queries
  - Changing the query tree to an equivalent but more efficient one
  - Choosing efficient implementations of each operator
  - Exploiting database statistics

OPTIMIZATION EXAMPLE

- In our query tree before we have the steps
  - Take the product of Student and Enrollment
  - Then select those entries where the Enrollment.Code equals ‘DBS’
- This is equivalent to
  - Selecting those Enrollment entries with Code = ‘DBS’
  - Then taking the product of the result of the selection operator with Student

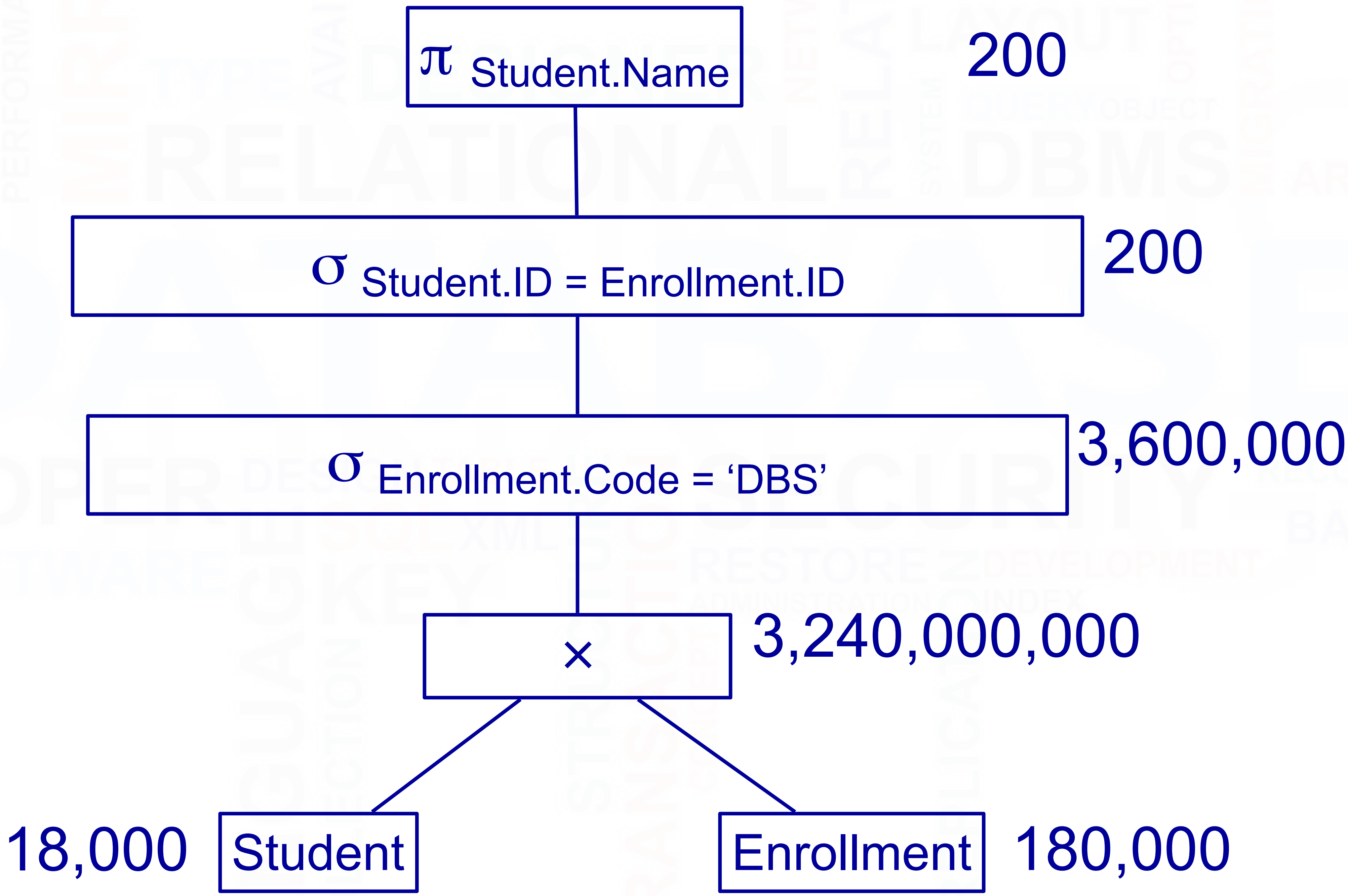
OPTIMIZED QUERY TREE



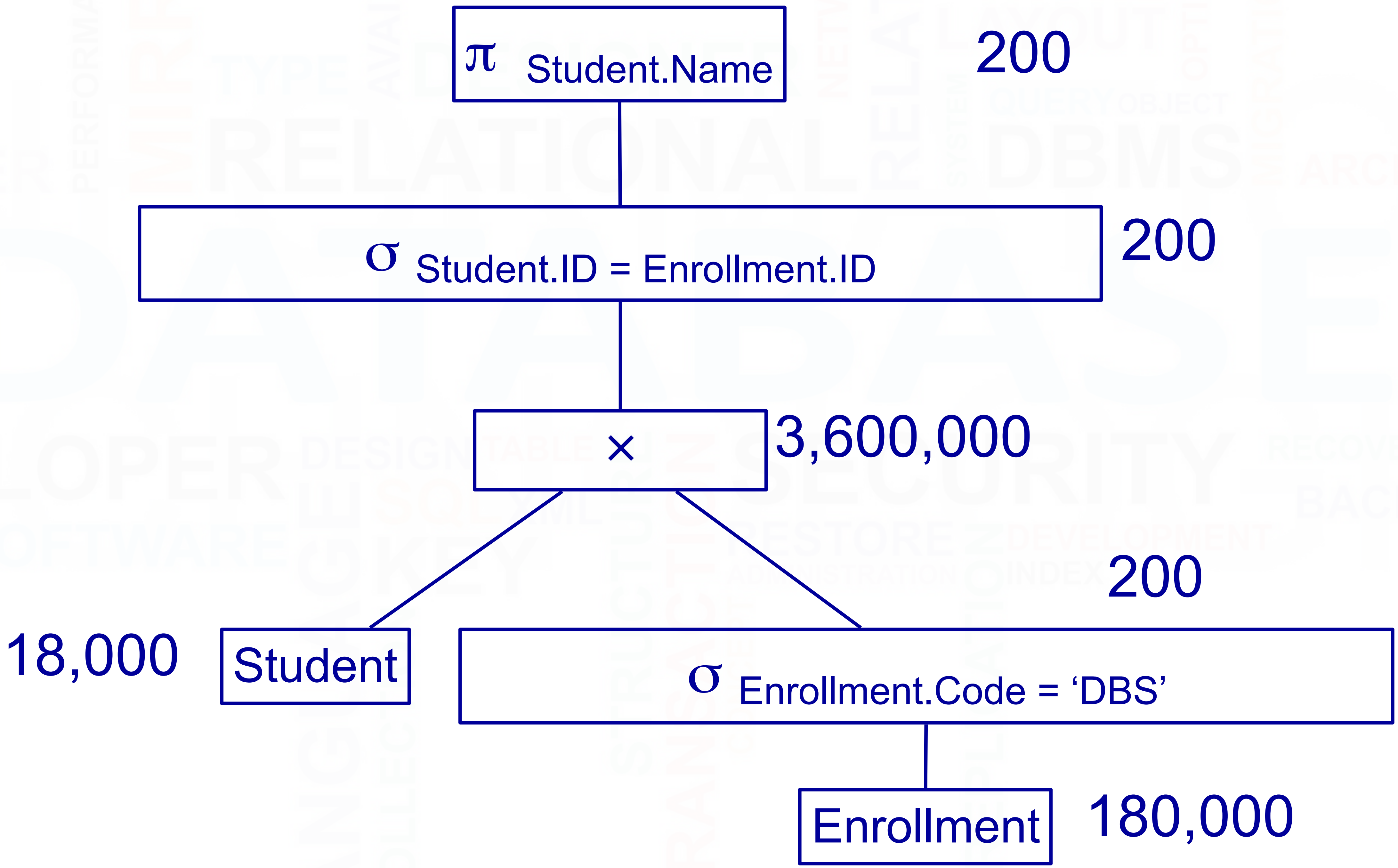
OPTIMIZATION EXAMPLE

- To see the benefit of this, consider the following statistics
  - Tehran Uni. has around 18,000 full time students
  - Each student is enrolled in at about 10 modules
  - Only 200 take DBS
- From these statistics we can compute the sizes of the relations produced by each operator in our query trees

ORIGINAL QUERY TREE



OPTIMIZED QUERY TREE



OPTIMIZATION EXAMPLE

- The original query tree produces an intermediate result with 3,240,000,000 entries
- The optimized version at worst has 3,600,000
- **A big improvement!**
- There is much more to optimization
  - In the example, the product and the second selection can be combined and implemented efficiently to avoid generating all Student-Enrollment combinations

OPTIMIZATION EXAMPLE

- If we have an index on Student.ID we can find a student from their ID with a binary search
- For 18,000 students, this will take at most 15 operations
- For each Enrollment entry with Code ‘DBS’ we find the corresponding Student from the ID
- 200 x 15 = 3,000 operations to do *both* the product and the selection.

END

But take a look at next slide!

NEXT LECTURE

- Transactions
  - ACID properties
  - The transaction manager
- Recovery
  - System and Media Failures
- Concurrency
  - Concurrency problems
- For more information
  - Connolly and Begg chapter 20
  - Ullman and Widom chapter 8.6