Physical Design

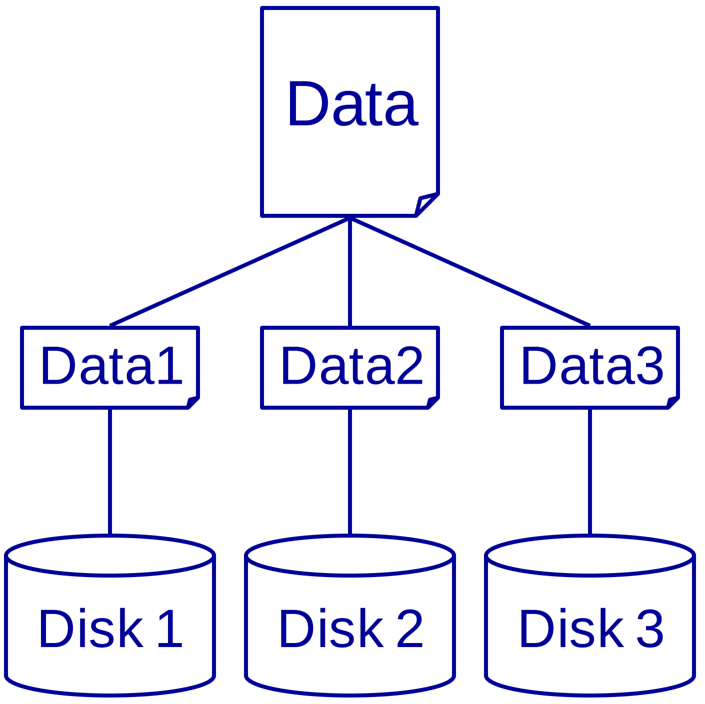
- Design so far
  - E/R modelling helps find the requirements of a database
  - Normalisation helps to refine a design by removing data redundancy
- Physical design
  - Concerned with storing and accessing the data
  - How to deal with media failures
  - How to access information efficiently

RAID Arrays

- RAID - redundant array of independent (inexpensive) disks
  - 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…)

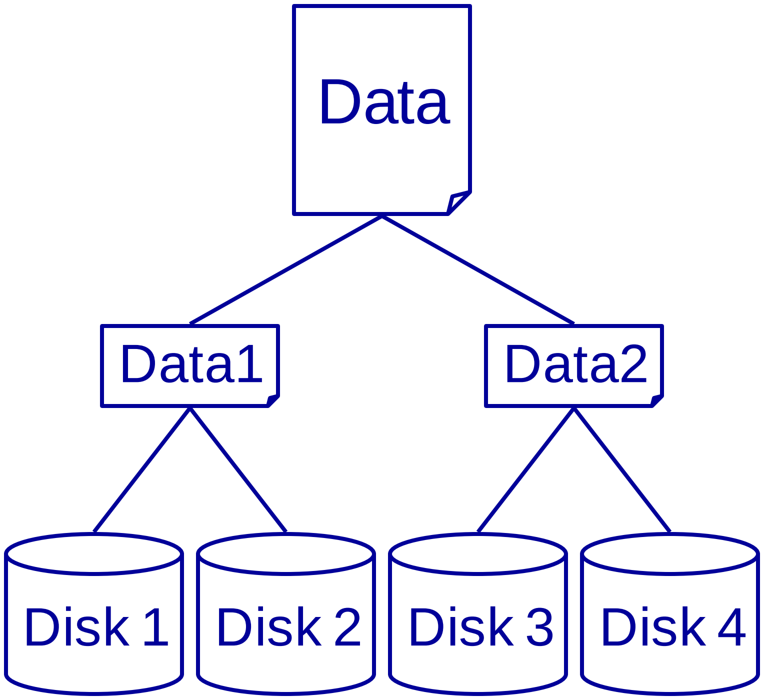
RAID Level 0

- 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

- 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

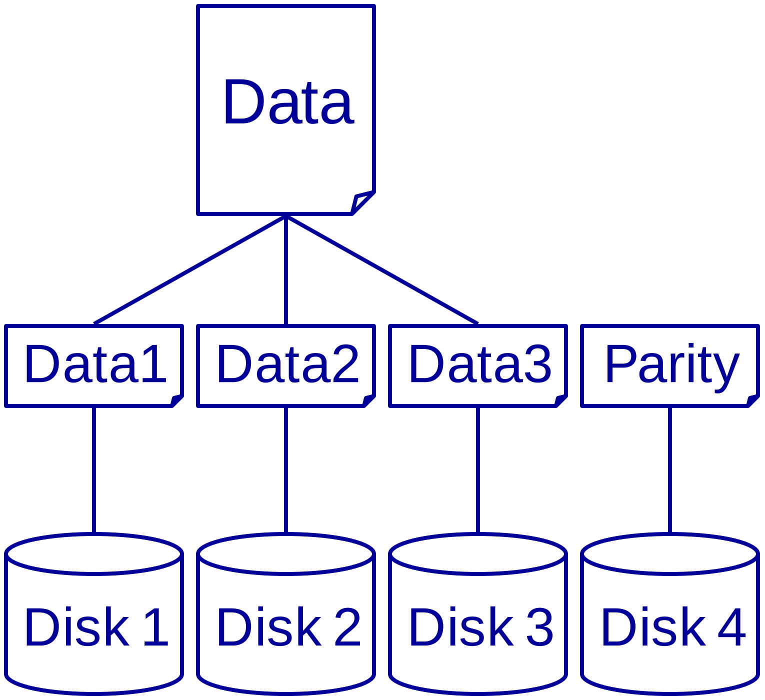
- If one of our pieces of data is lost we can recover it
  - Just compute it as the parity of the remaining data and our original parity information

```
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

Indexes (discussed)

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
  - Optimisation - changes are made for efficiency
  - Evaluation - the optimised 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

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

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

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

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

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

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

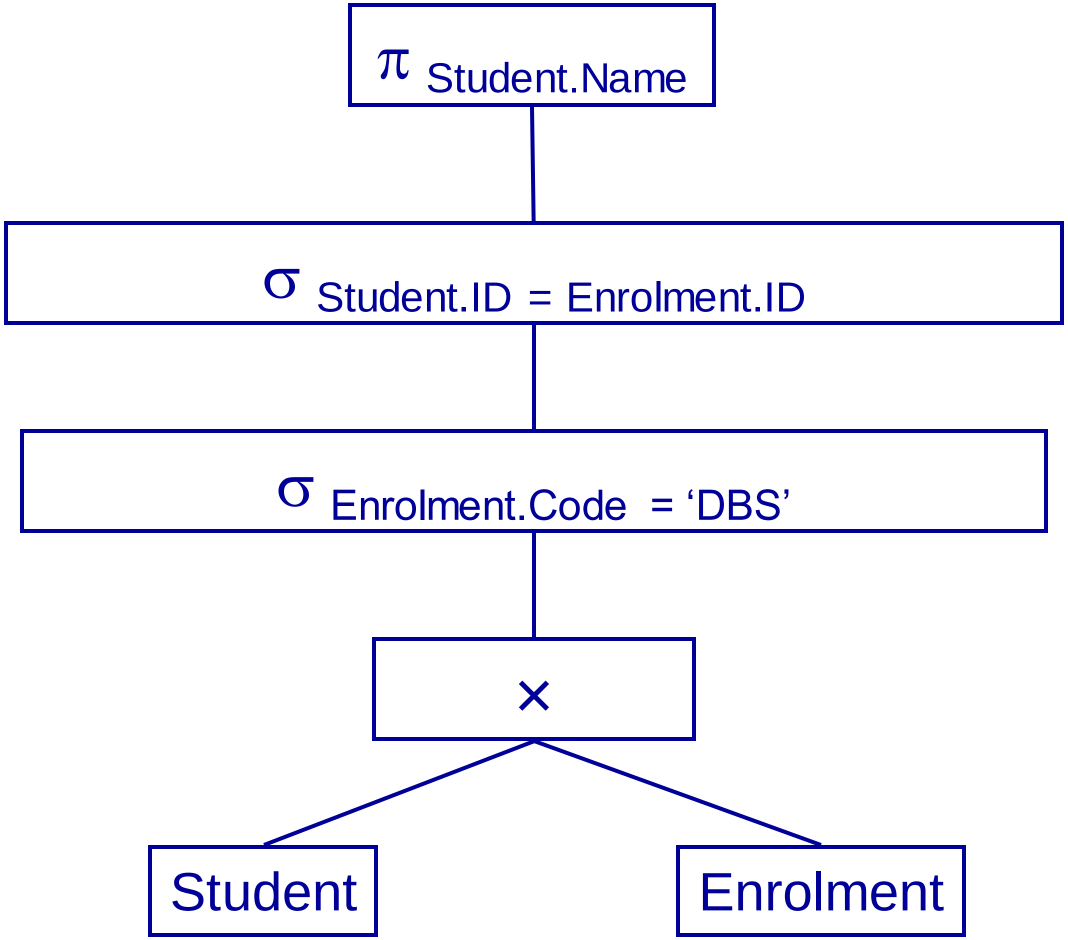
SQL → Relational Algebra

- SQL statement

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

- Relational Algebra
  - Take the product of Student and Enrolment
  - 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 = Enrolment.ID
σ Enrolment.Code = 'DBS'
×
Student
Enrolment

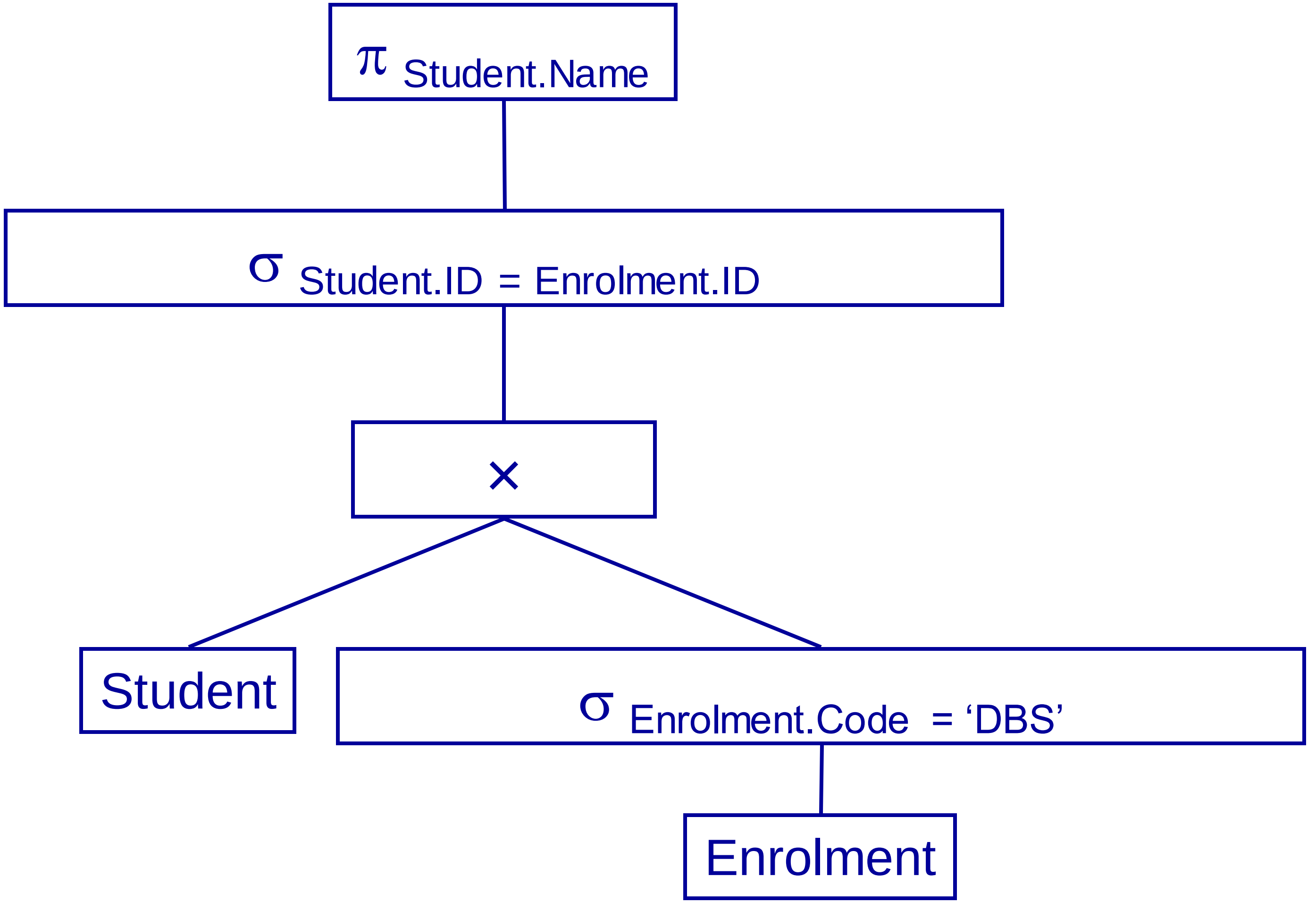
Optimisation

- 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 optimise queries
  - Changing the query tree to an equivalent but more efficient one
  - Choosing efficient implementations of each operator

Optimisation Example

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

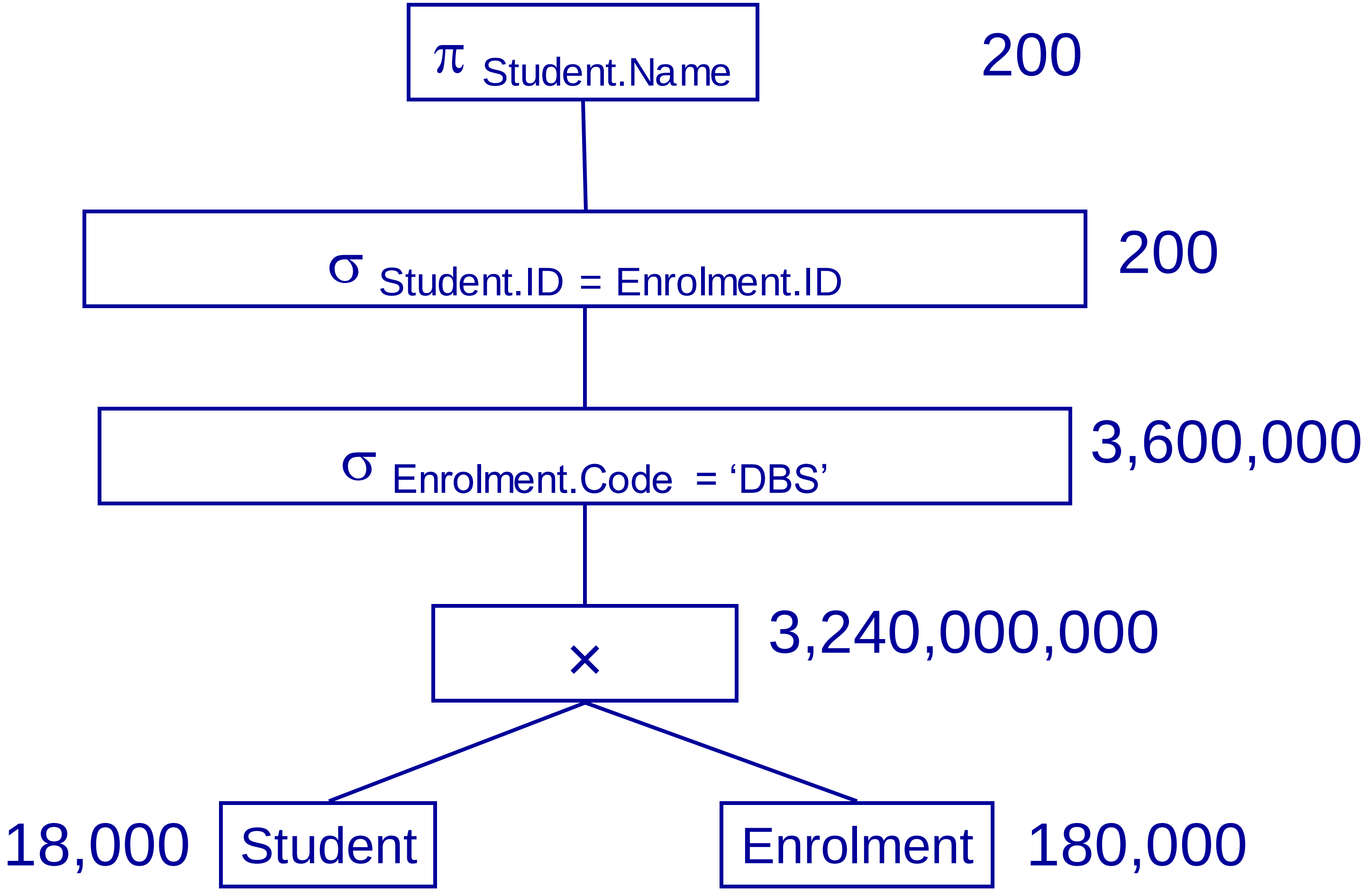
Optimised Query Tree



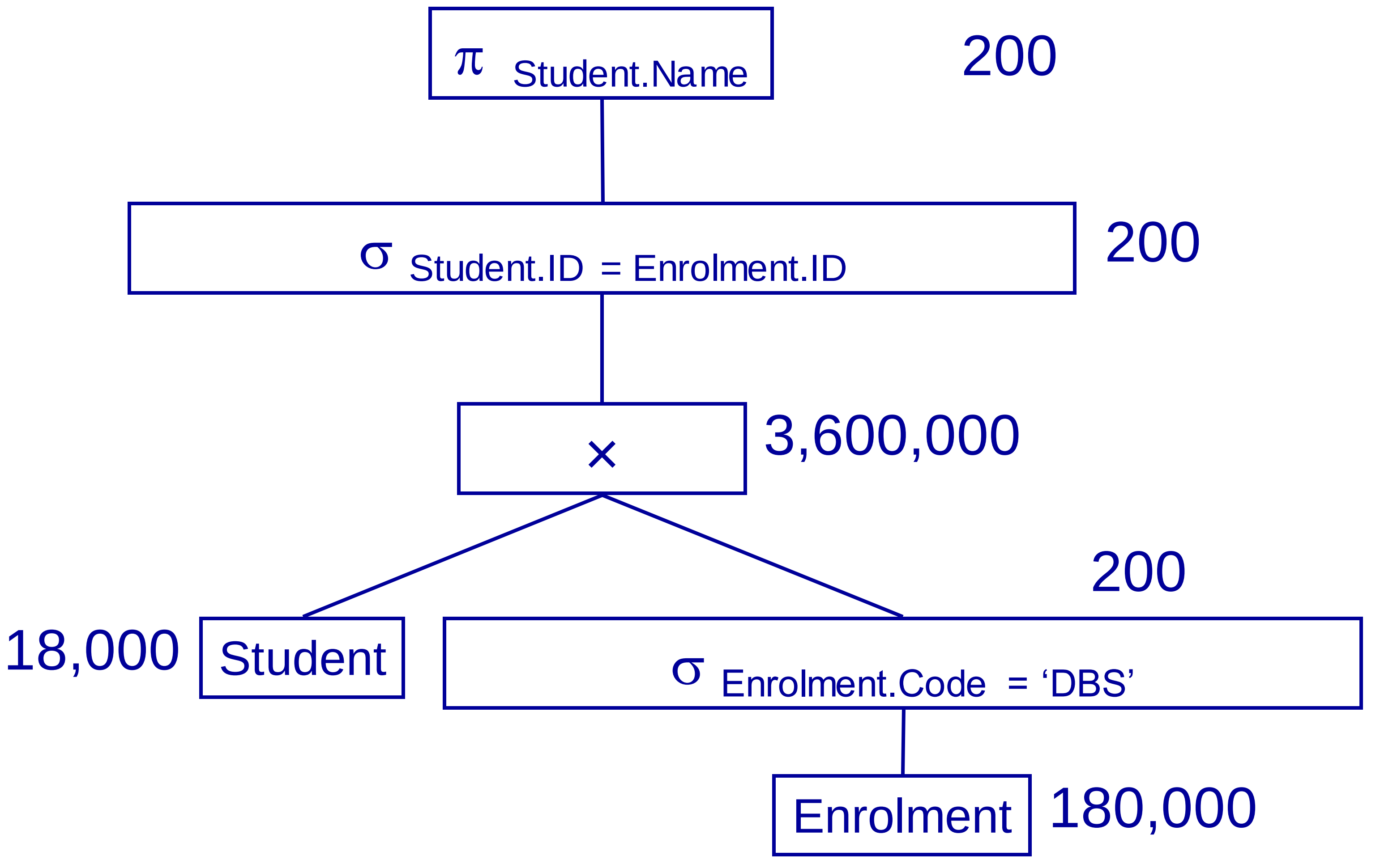
Optimisation Example

- To see the benefit of this, consider the following statistics
  - Nottingham 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



Optimised Query Tree



Optimisation Example

- The original query tree produces an intermediate result with 3,240,000,000 entries
- The optimised version at worst has 3,600,000
- A big improvement!

- There is much more to optimisation
  - In the example, the product and the second selection can be combined and implemented efficiently to avoid generating all Student-Enrolment combinations

Optimisation 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 Enrolment 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.