Өгөгдлийн сангийн үндэс (CSII202 - 3 кр)
Database Systems

Lecture 11: Physical DB Issues, Indexes, Query Optimisation

МУИС, ХШУИС, МКУТ-ийн багш

Маг. Довдонгийн Энхзол

Хүн мэдэхгүй юмныхаа дайсан.

Д.Намдаг

In This Lecture

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

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 history

RAID (Redundant Array of Independent (inexpensive) Disks) was invented by **David Patterson, Garth A. Gibson,** and **Randy Katz** at the University of California, Berkeley in 1987.

- June 1988 paper "A Case for Redundant Arrays of Inexpensive Disks (RAID)", presented at the SIGMOD conference

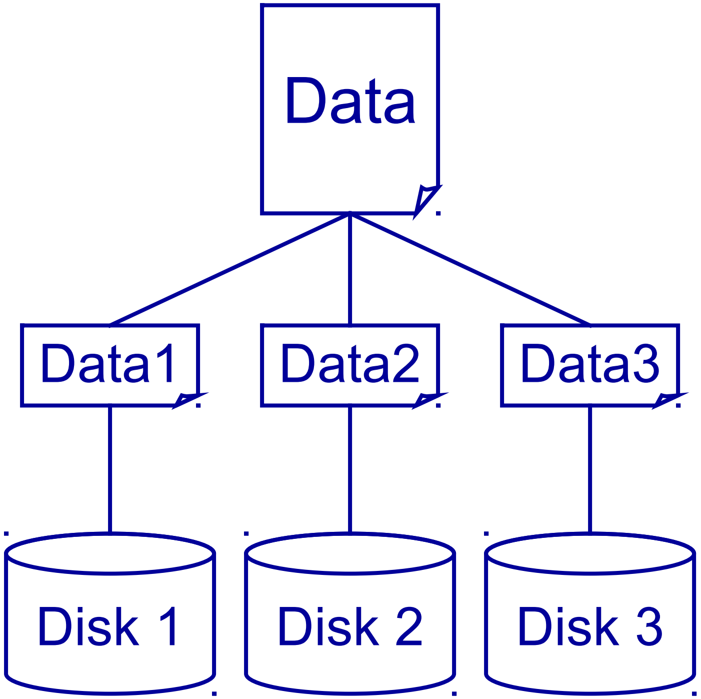
RAID Arrays

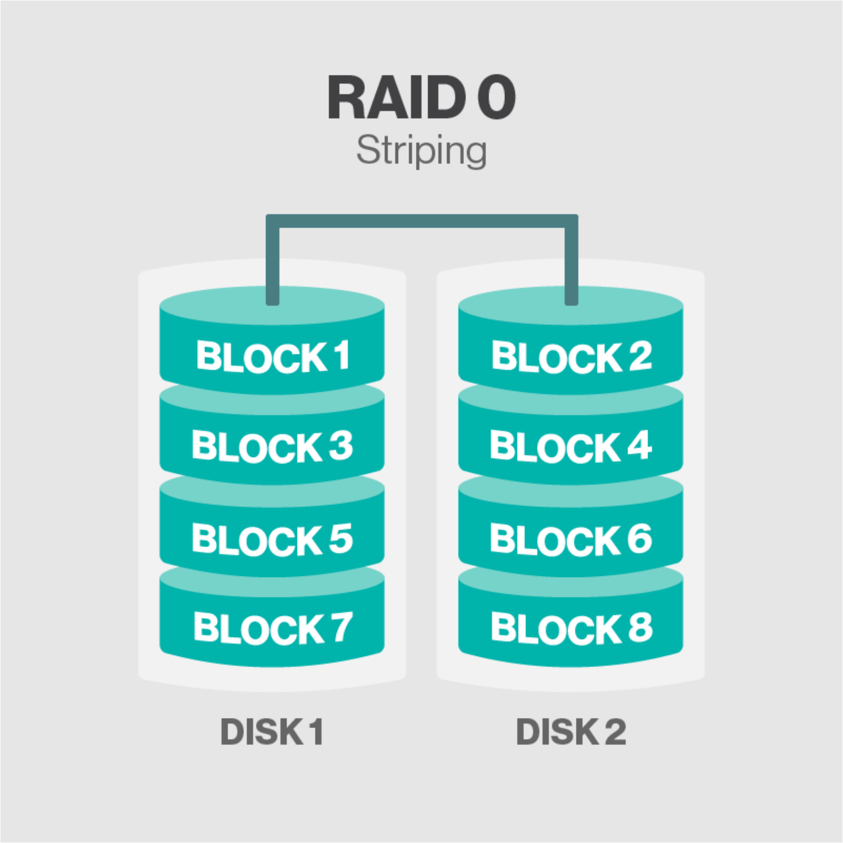
- RAID
  - Storing information across more than one physical disk
  - Speed - can access more than one disk
  - Robustness - if one disk fails it is OK

- 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

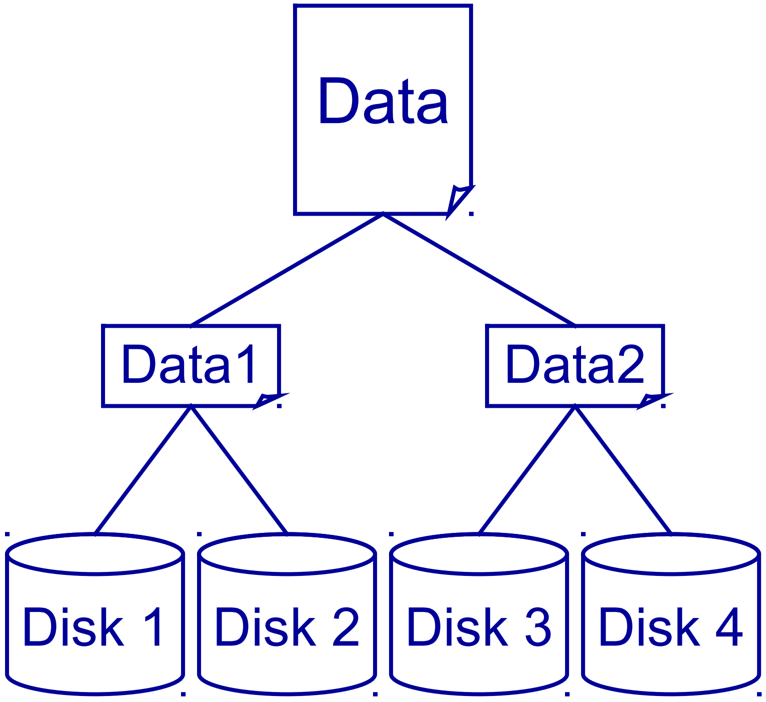


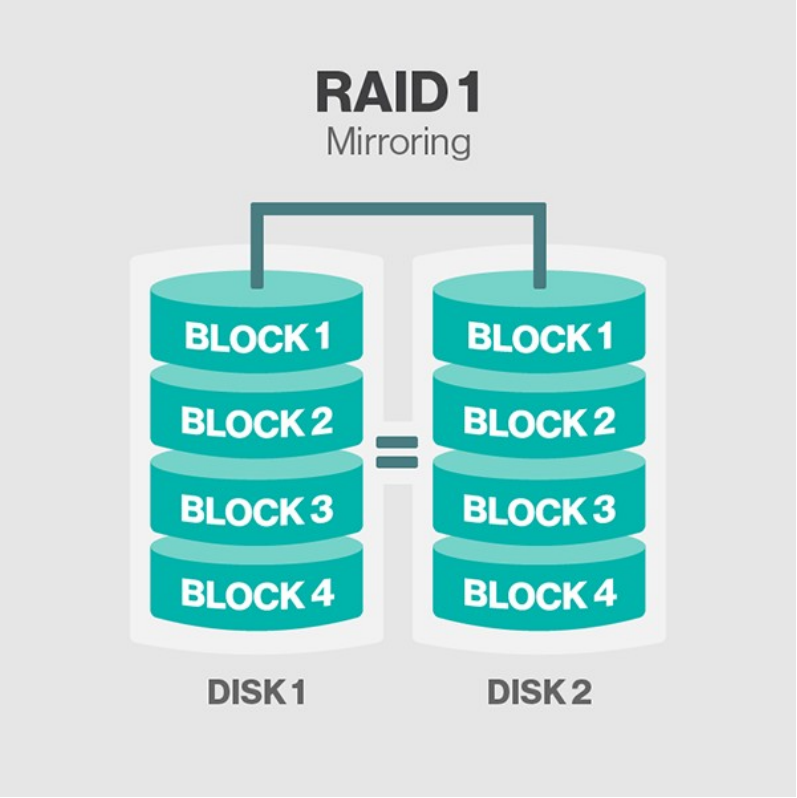


- Minimum number of drives: *2*
- Strengths: *Highest performance.*
- Weaknesses: *No data protection; One drive fails, all data is lost.*

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





- Minimum number of drives: *2*
- Strengths: *Very high performance; Very high data protection; Very minimal penalty on write performance.*
- Weaknesses: *High redundancy cost overhead; Because all data is duplicated, twice the storage capacity is required.*

RAID2 - Parity Checking

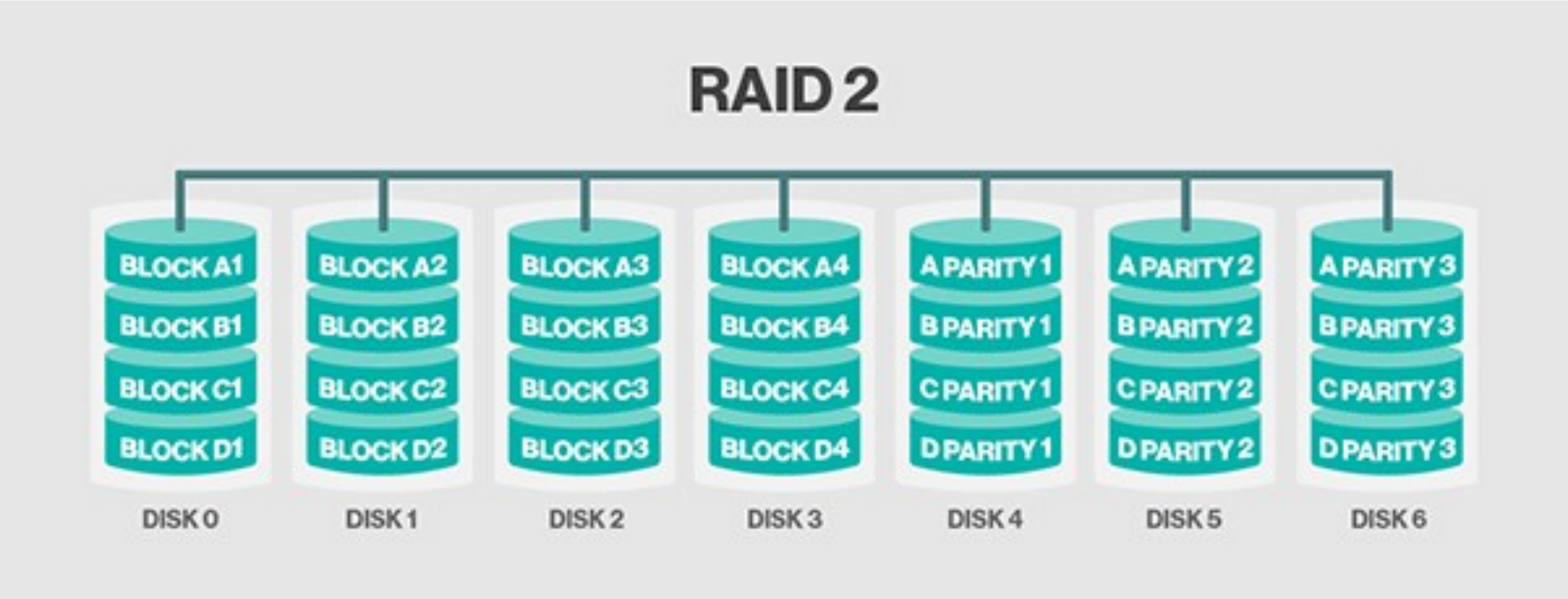
- We can use parity checking to reduce the number of disks
  - Parity - for a set of data in binary form we count the number of 1s for each bit across the data
  - If this is even the parity is 0, if odd then it is 1

$$\begin{array}{cccccccc}
1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\
\hline
0 & 1 & 0 & 0 & 0 & 1 & 1 & 1
\end{array}$$

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

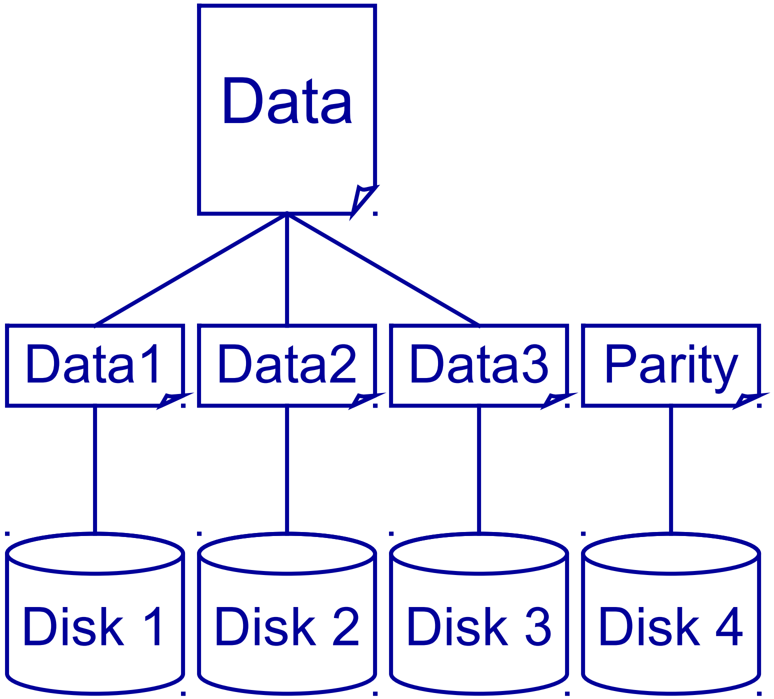
```
1 0 1 1 0 0 1 1
0 0 1 1 0 0 1 1


0 1 1 0 1 1 1 0
---------------
0 1 0 0 0 1 1 1
```

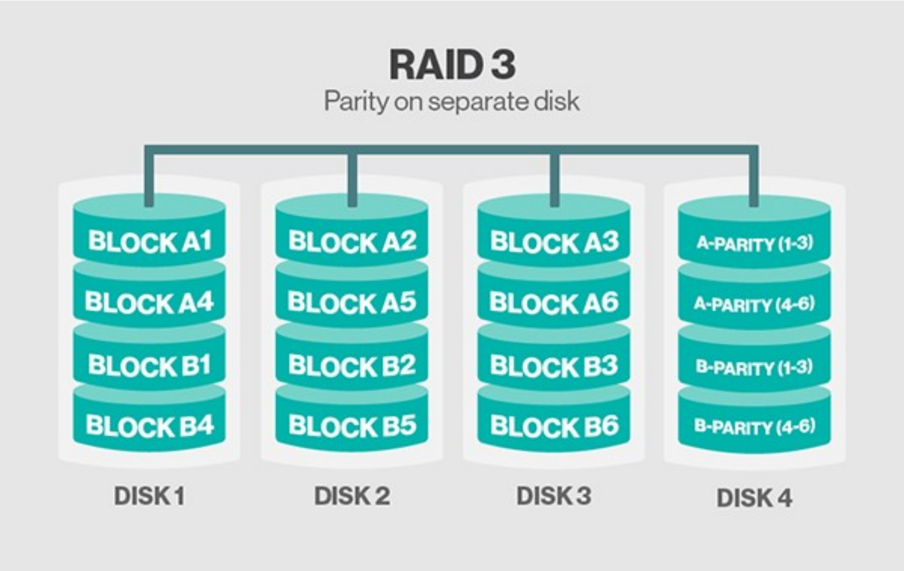


- Minimum number of drives: *Not used in LAN*
- Strengths: *Previously used for RAM error environments correction (known as Hamming Code) and in disk drives before he use of embedded error correction.*
- Weaknesses: *No practical use; Same performance can be achieved by RAID 3 at lower cost.*

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

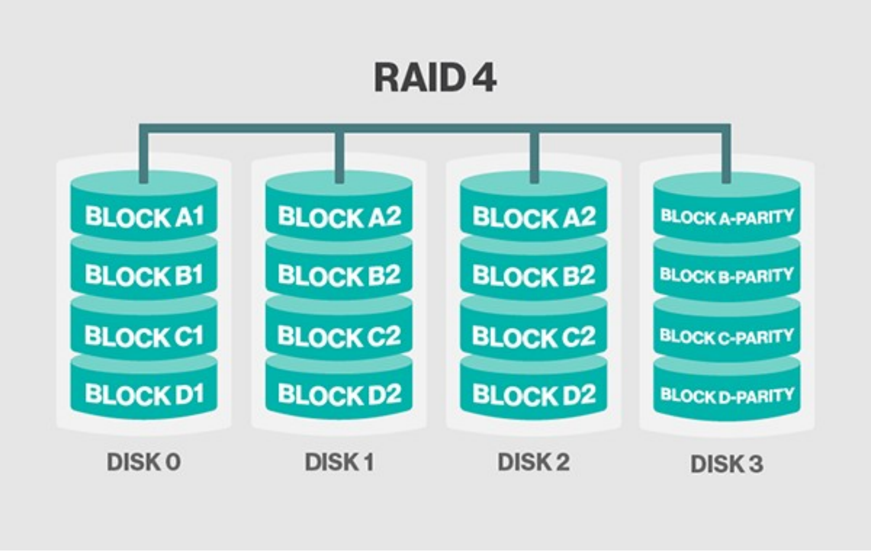




- Minimum number of drives: 3
- Strengths: Excellent performance for large, sequential data requests.
- Weaknesses: Not well-suited for transaction-oriented network applications; Single parity drive does not support multiple, simultaneous read and write requests.

RAID Level 4

This level uses large stripes, which means you can read records from any single drive. This allows you to use overlapped I/O for read operations. Since all write operations have to update the parity drive, no I/O overlapping is possible. RAID 4 offers no advantage over RAID 5.



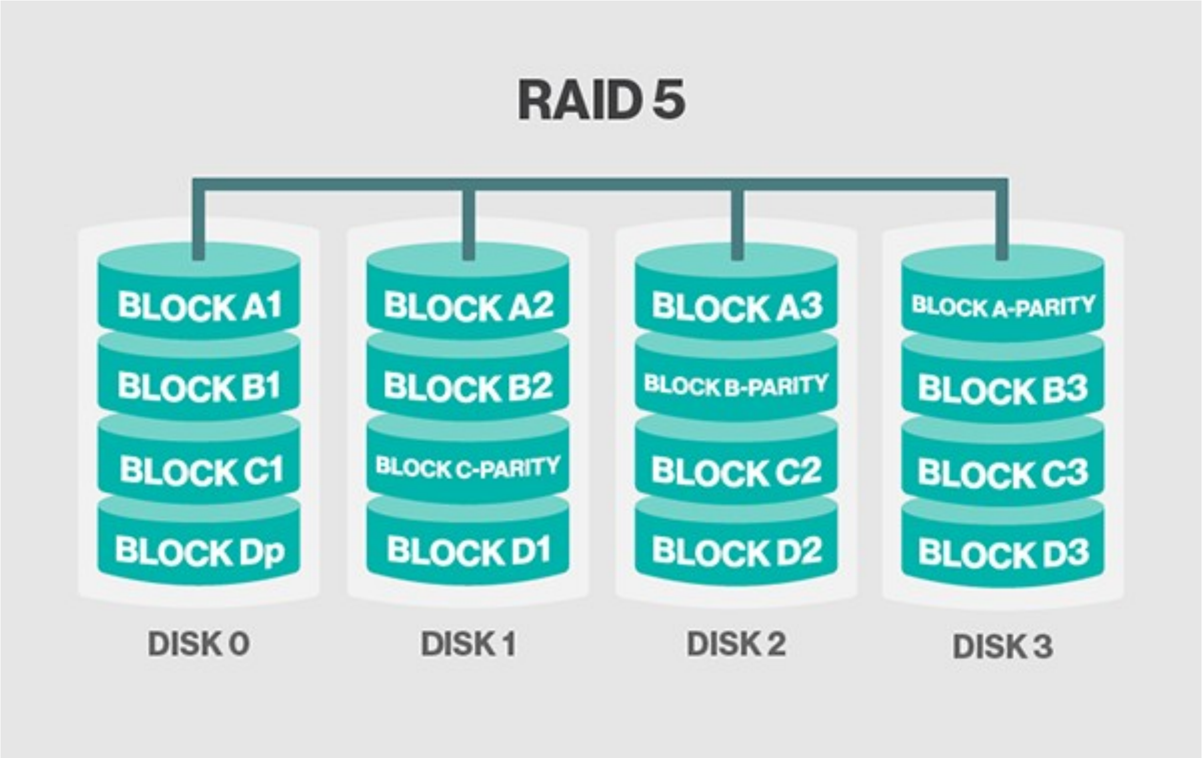
- Minimum number of drives: *3 (Not widely used)*
- Strengths: *Data striping supports multiple simultaneous read requests.*
- Weaknesses: *Write requests suffer from same single parity-drive bottleneck as RAID 3; RAID 5 offers equal data protection and better performance at same cost.*

RAID Level 5

This level is based on block-level striping with parity. The parity information is striped across each drive, allowing the array to function even if one drive were to fail. The array's architecture allows read and write operations to span multiple drives. This results in performance that is usually better than that of a single drive, but not as high as that of a RAID 0 array. RAID 5 requires at least three disks, but it is often recommended to use at least five disks for performance reasons.

RAID Level 5

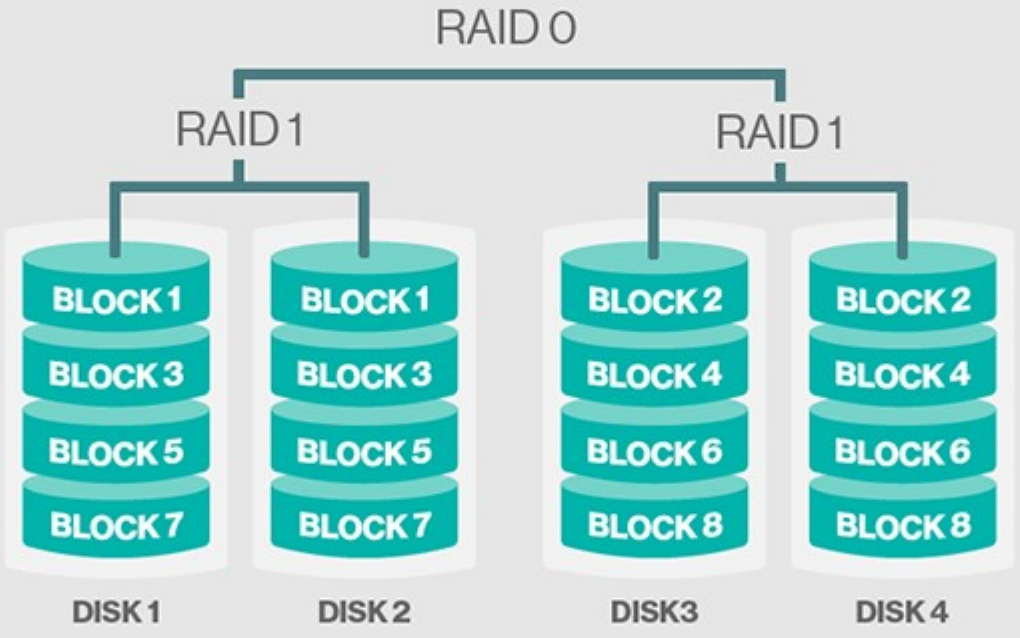
RAID 5 arrays are generally considered to be a poor choice for use on write-intensive systems because of the performance impact associated with writing parity information. When a disk does fail, it can take a long time to rebuild a RAID 5 array. Performance is usually degraded during the rebuild time, and the array is vulnerable to an additional disk failure until the rebuild is complete.



- Minimum number of drives: 3
- Strengths: Best cost/performance for transaction-oriented networks; Very high performance, very high data protection; Supports multiple simultaneous reads and writes; Can also be optimized for large, sequential requests.
- Weaknesses: Write performance is slower than RAID 0 or RAID 1.

RAID 10 (RAID 1+0)

Stripe + Mirror



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?

More RAID

- https://adaptec.com/nr/rdonlyres/8c58de73-377d-47ea-bd7a-87b54fd7c93b/0/raid_e.pdf
- Types Of RAID: Software or Hardware solutions
-

Fault tolerance

- MTDL:

  Mean Time to Data Loss. The average time before the failure of an array component causes data to be lost or corrupted.
- MTDA:

  Mean Time between Data Access (or availability). The average time before non-redundant components fail, causing data inaccessibility without loss or corruption.
- MTTR:

  Mean Time To Repair. The average time required to bring an array storage subsystem back to full fault tolerance.
- MTBF:

  Mean Time Between Failure. Used to measure computer component average reliability/life expectancy. MTBF is not as well-suited for measuring the reliability of array storage systems as MTDL, MTTR or MTDA (see below) because it does not account for an array's ability to recover from a drive failure. In addition, enhanced enclosure environments used with arrays to increase uptime can further limit the applicability of MTBF ratings for array solutions.

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

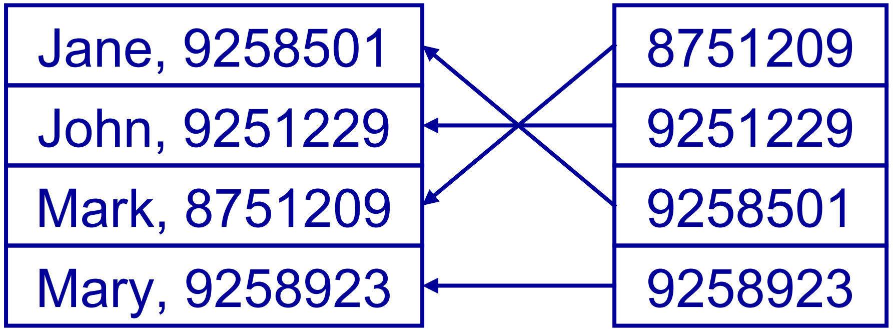
As a Table

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

Order does not really concern us here

As a File

Secondary Index



As a File	Secondary Index
Jane, 9258501	87751209
John, 9251229	9251229
Mark, 8751209	9258501
Mary, 9258923	9258923

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

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

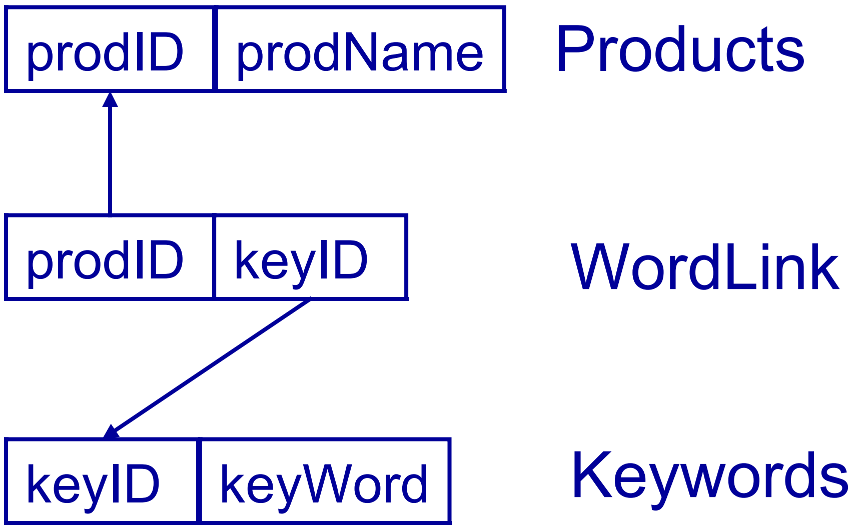
Choosing Indexes

- 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



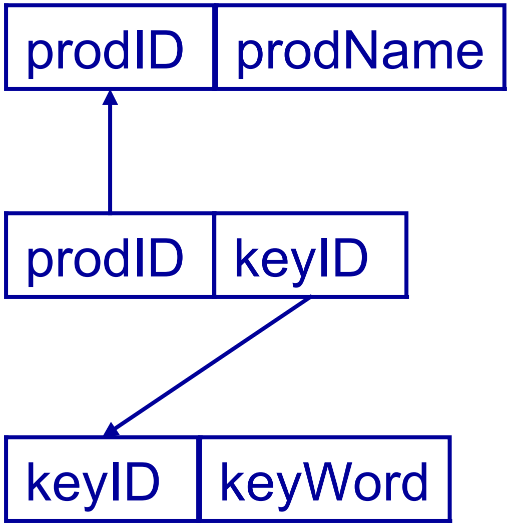
prodID | prodName — Products

prodID | keyID — WordLink

keyID | keyWord — Keywords

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
  - 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
- 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>
```

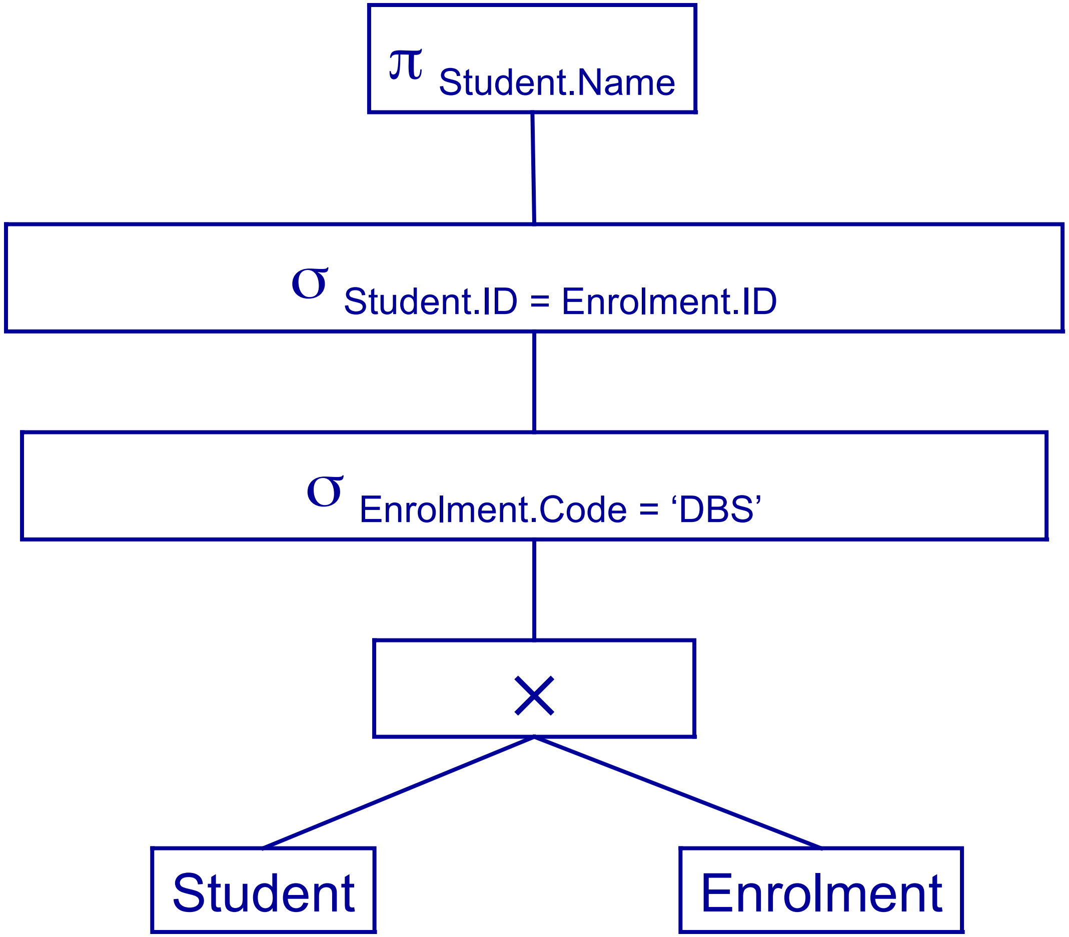
SQL → Relational Algebra

- SQL statement

```
SELECT Student.Name
  FROM Student,
       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



$\pi_{\text{Student.Name}}$

$\sigma_{\text{Student.ID = Enrolment.ID}}$

$\sigma_{\text{Enrolment.Code = 'DBS'}}$

×

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

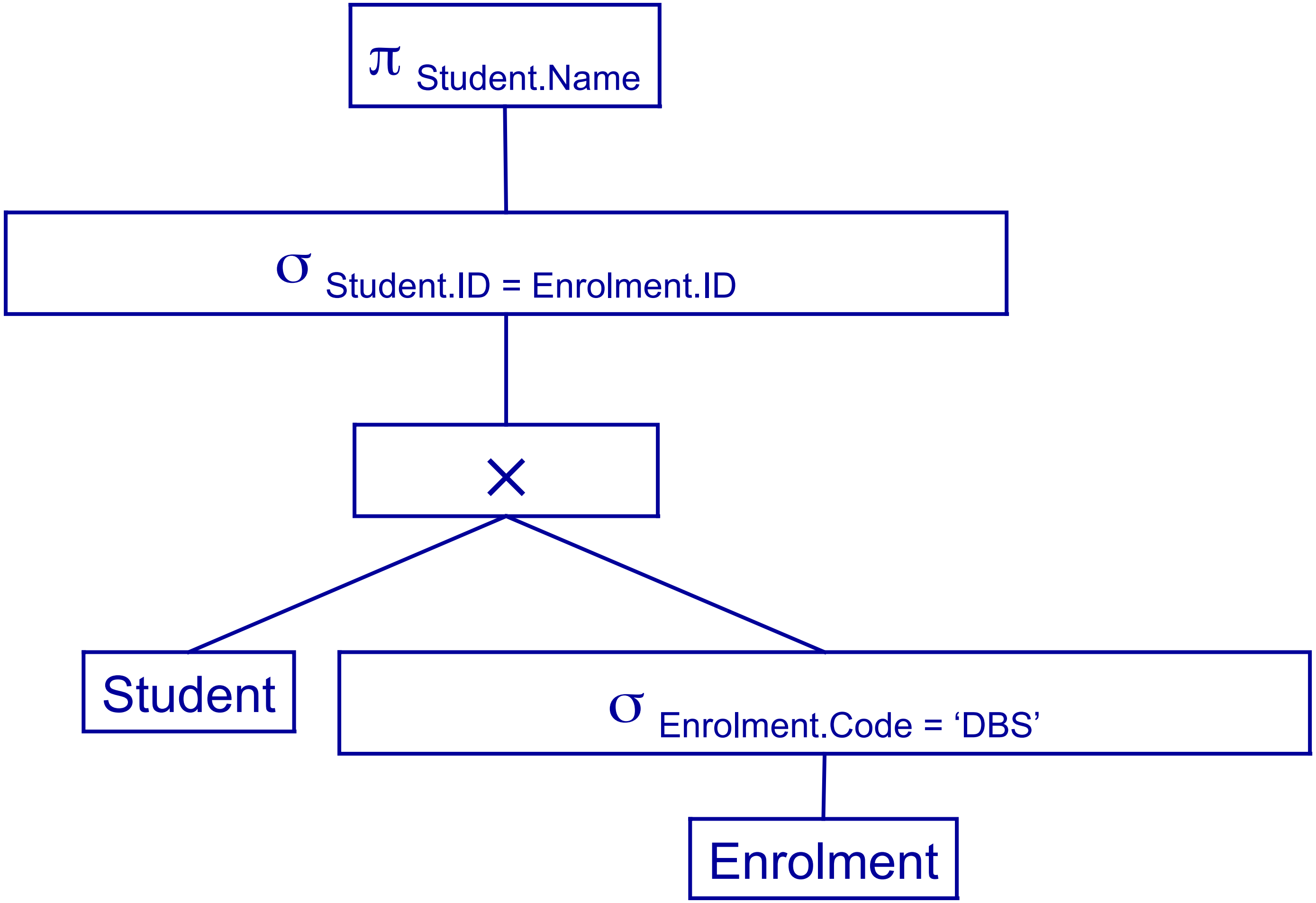
- Many ways to optimise queries
  - Changing the query tree to an equivalent but more efficient one
  - Choosing efficient implementations of each operator
  - Exploiting database statistics

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



$\pi_{\text{Student.Name}}$

$\sigma_{\text{Student.ID = Enrolment.ID}}$

$\times$

Student

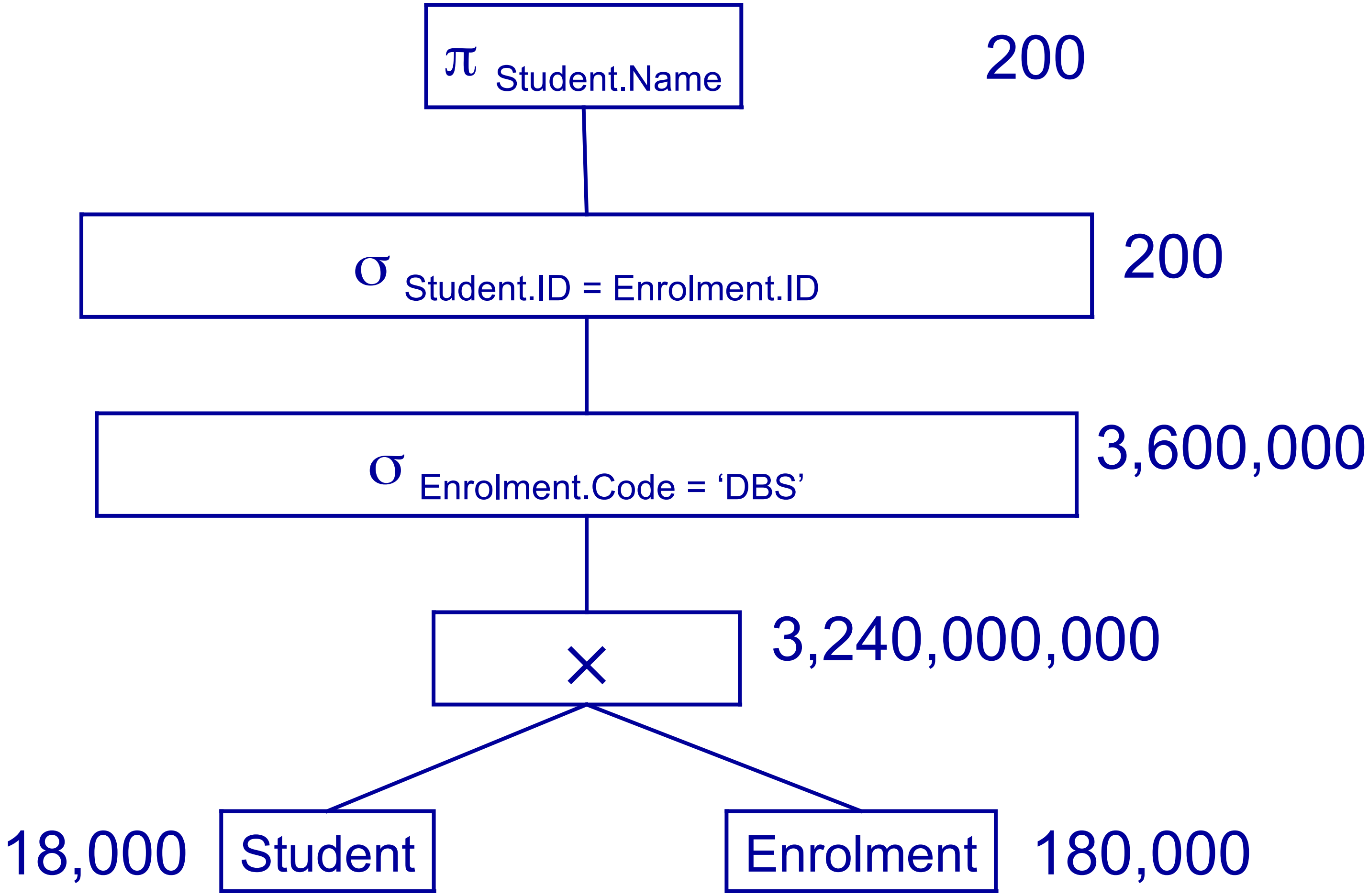
$\sigma_{\text{Enrolment.Code = 'DBS'}}$

Enrolment

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



π Student.Name 200

σ Student.ID = Enrolment.ID 200

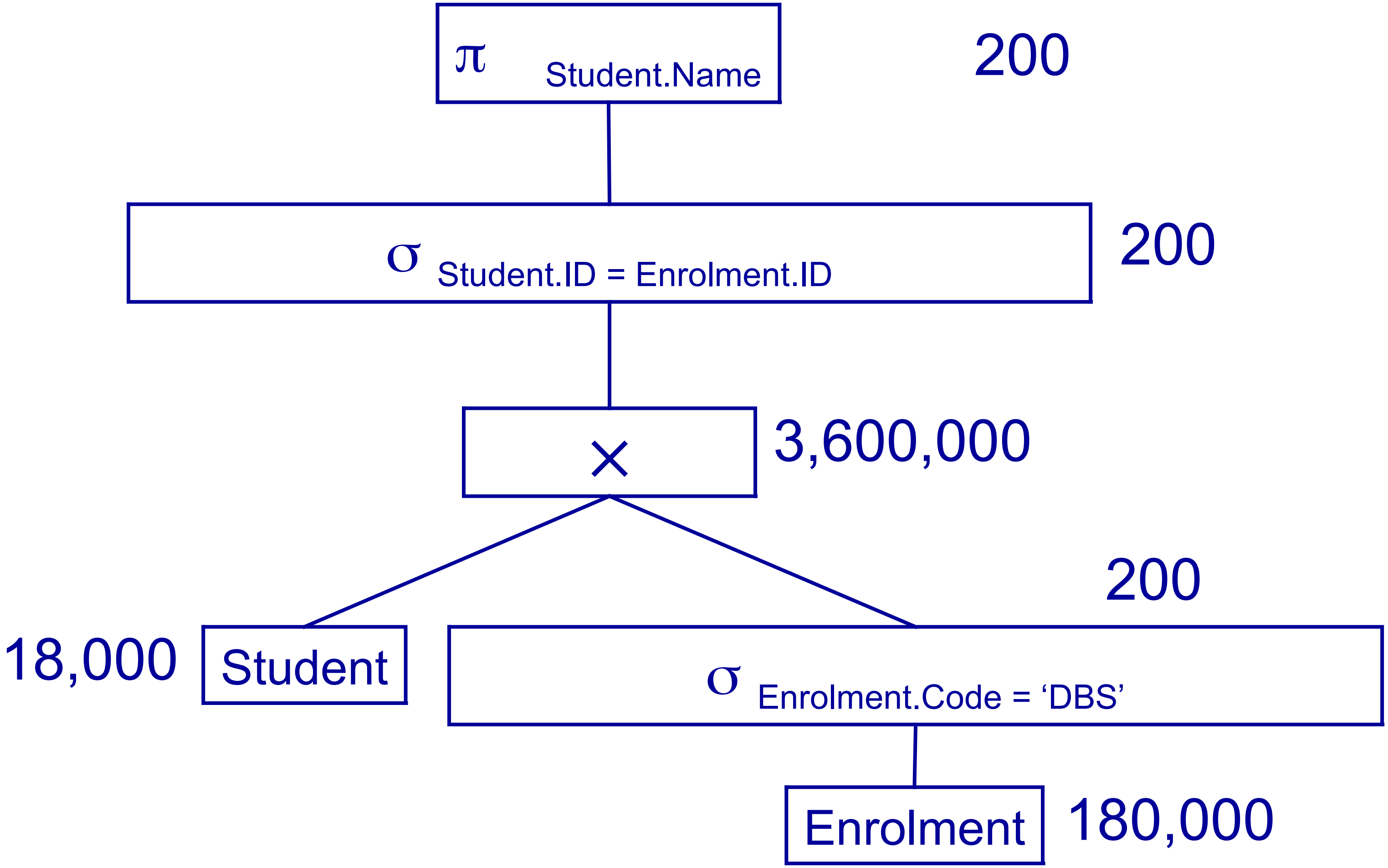
σ Enrolment.Code = ‘DBS’ 3,600,000

× 3,240,000,000

18,000 Student

Enrolment 180,000

Optimised Query Tree



$\pi_{\text{Student.Name}}$ 200

$\sigma_{\text{Student.ID = Enrolment.ID}}$ 200

× 3,600,000

18,000 Student

$\sigma_{\text{Enrolment.Code = 'DBS'}}$ 200

Enrolment 180,000

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.