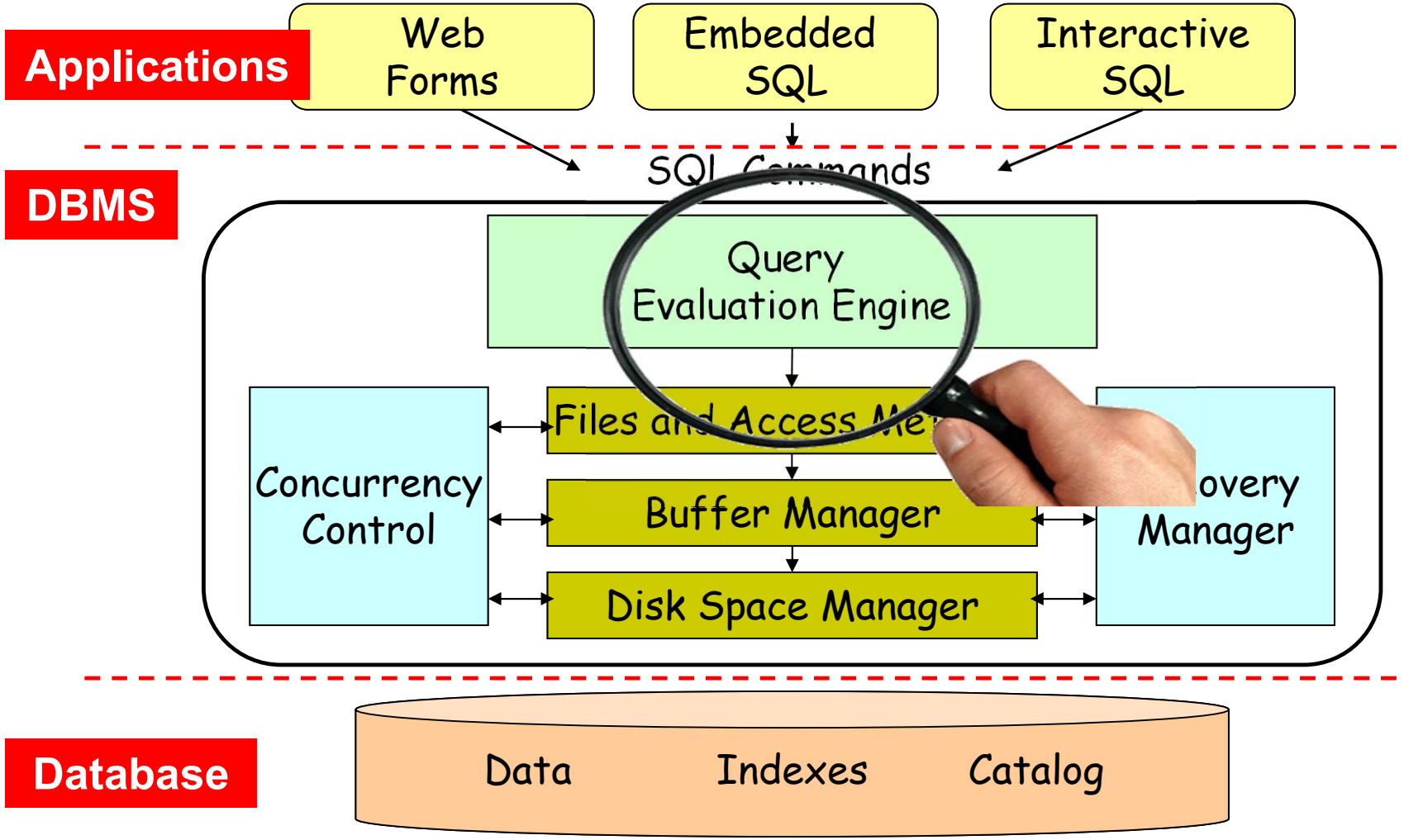
Comp2411 Database Systems

Query Optimization

Database Management System (DBMS)



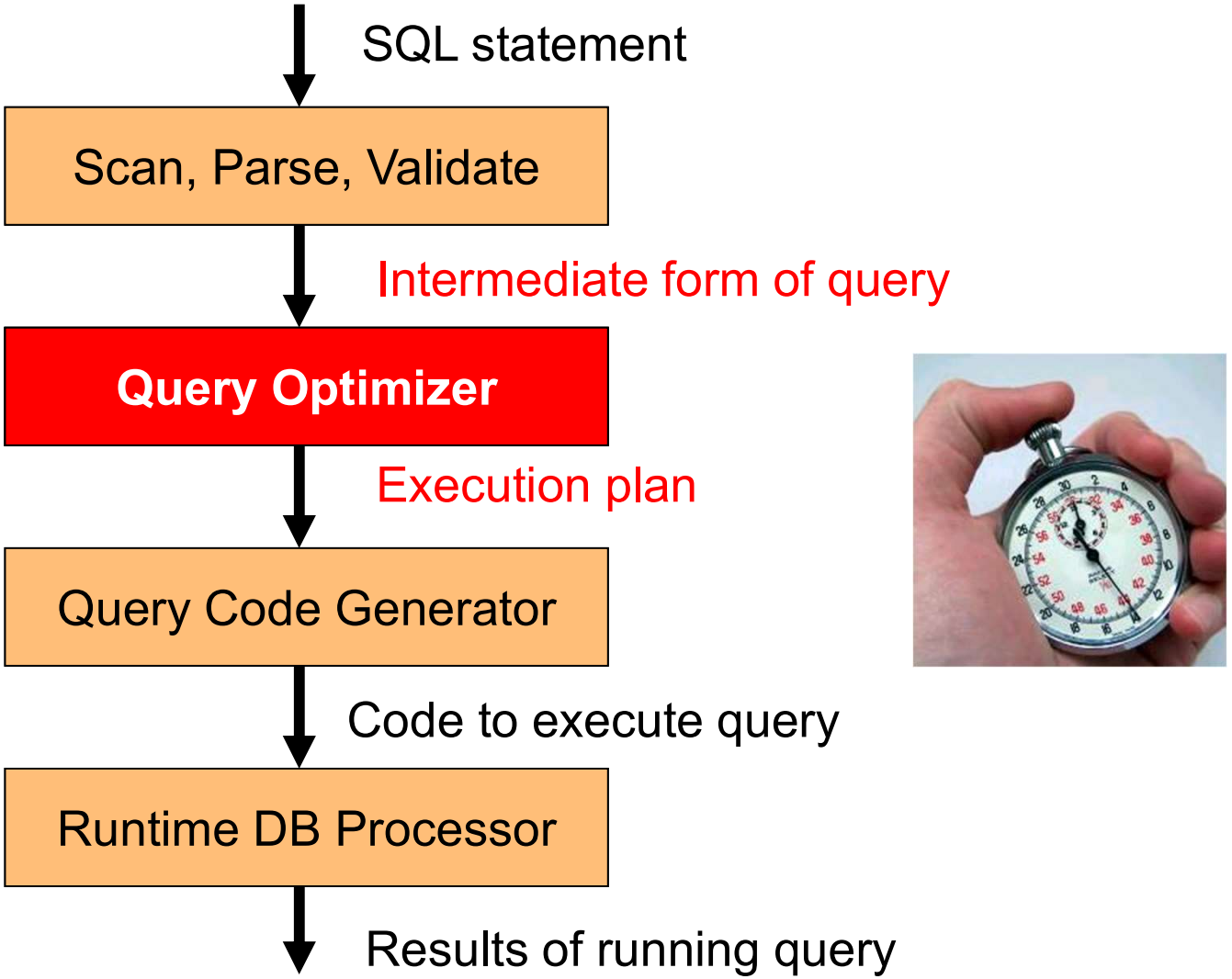
Basic *SQL* Query

SELECT [DISTINCT] *select-list*
FROM *from-list*
WHERE *qualifications*

SPJ Query

- Unary Relational Operations:
  - Project and Select
- Binary Relational Operations:
  - Cartesian Product and Join

Steps in Processing a Query



Intermediate Query Form

- An SQL query is translated into **equivalent**:

Relational Algebra Expression

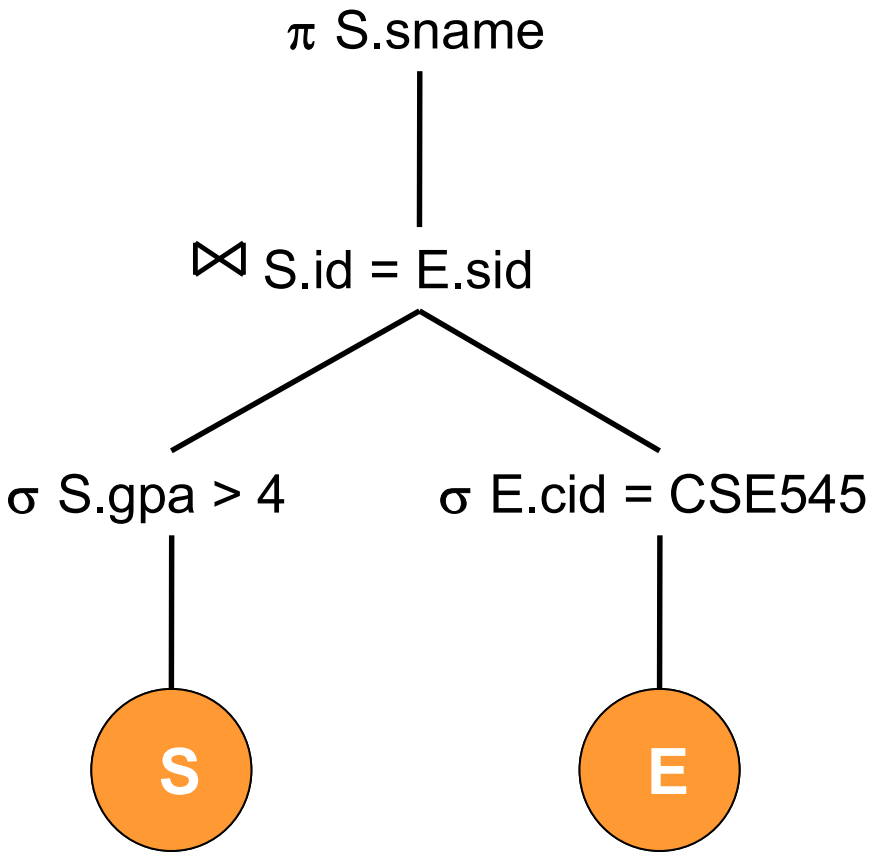
- Represented as a:

Query Tree

Query Tree

- **Query tree**: a data structure that corresponds to a relational algebra expression
  - Input relations of the query as leaf nodes
  - Relational algebra operations as internal nodes
- An **execution** of the query tree consists of executing internal node operations

Query Tree




```
SELECT  S.sname
FROM    Student S, Enrolled E
WHERE   S.id = E.sid
        AND S.gpa > 4
        AND E.cid = CS545
```

$$\pi_{sname}(\sigma_{gpa>4}(S) \bowtie_{id=sid} \sigma_{cid=CS545}(E))$$

Query Execution Plan

Query Execution Plan

- An **execution plan** for a relational algebra query consists of a combination of:
  1. The optimized relational algebra **query tree**
  2. Information about the **access methods** to be used for each relation
     - E.g., table scan, B+ tree index, Bitmap index, etc.
  3. The **algorithms** to be used in computing the relational operators stored in the tree
     - E.g., Nested Loop Join, Single Loop Join, etc.

Query Tree Optimization

- Get the **initial** Query Tree
  - Conceptual evaluation strategy (discussed later)

- Transform it into an **equivalent** Query Tree
  - Represents a **different** relational algebra expression
  - Gives the **same** result
  - More **efficient** to execute

Operator Evaluation

- Several alternative algorithms are available for **implementing** each relational operator
  - No algorithm is universally superior
  - Several factors influence which algorithm performs **best**:
    - Size of the tables
    - Existing index and sort orders
    - Size of available memory
    - ….

Relational Algebra Operations

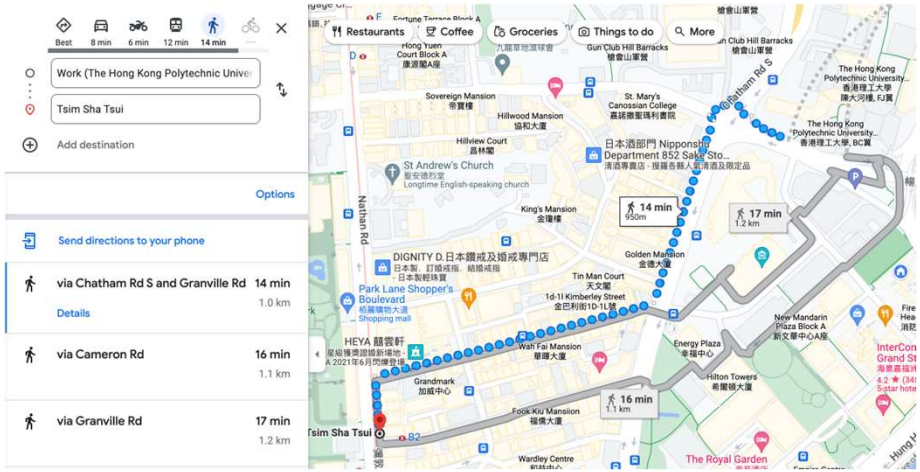
- Unary Relational Operations:
  - Project and Select
- Binary Relational Operations:
  - Cartesian Product and Join

How to Implement Projection (π)?

- Consider two scenarios:
  - Project on key attribute?
    - Table scan
  - Project on non-key attribute?
    - Sorting
    - Table scan
- What if there is an index on the attribute?

How to Implement Selection (σ)?

- Depends on:
  - **Type of Query:**
    1. Point query,
    2. Range query,
    3. Conjunction
    4. ~~Disjunction~~



  - **Type of available Access Path:**
    1. Index,
    2. Sorted File
    3. None!

Algorithms for Selection (σ)

Let's start with the form: $\sigma_{\text{R.attr <op> value}}(R)$

- This form can represent:
  - **Point Query:**
    - <op> is: "**=**" (i.e., equality)
    - Eg.: $\sigma_{\text{Age=21}}$ (Students)
  - **Range Query:**
    - <op> is: "**>**", "**≥**", "**<**", "**≤**"
    - Eg.: $\sigma_{\text{Age>21}}$(Students)
- Algorithms: *in the next slides..*

Selection (σ): No Index, Unsorted Data

$\sigma_{\text{R.attr op value}}(R)$

(1) Linear search (brute force):

1. Retrieve every record in the file, and
2. Test whether its attribute values satisfy the selection condition (R.attr op value)

- Records are grouped into blocks:
  - Each block is read from disk and search (i.e., test) is done in main memory

Selection (σ): No Index, Sorted Data

$\sigma_{\text{R.attr op value}}(R)$

(2) Binary search:

- If the data file is sorted on the **condition attribute** (R.attr)
- Works for **Point** Query ($\sigma_{\text{Age=21}}$)
- Works for **Range** Query ($\sigma_{\text{Age>21}}$)
  - Find the first record that satisfies the condition, then scan until the condition is no longer satisfied
- More efficient than linear search

Selection (σ): Index (B+ tree Index)

$$\sigma_{\text{R.attr op value}}(R)$$

(3) Using an Index (e.g., B+ tree)

- If an index exists on the **condition attribute** (R.attr)
- Flexible solution; can create a B+ tree index on:
  - key/non-key attribute
  - Sorted or unsorted data file
- Works for both **Point** and **Range** Query
  - Find the first record then follow the next pointer

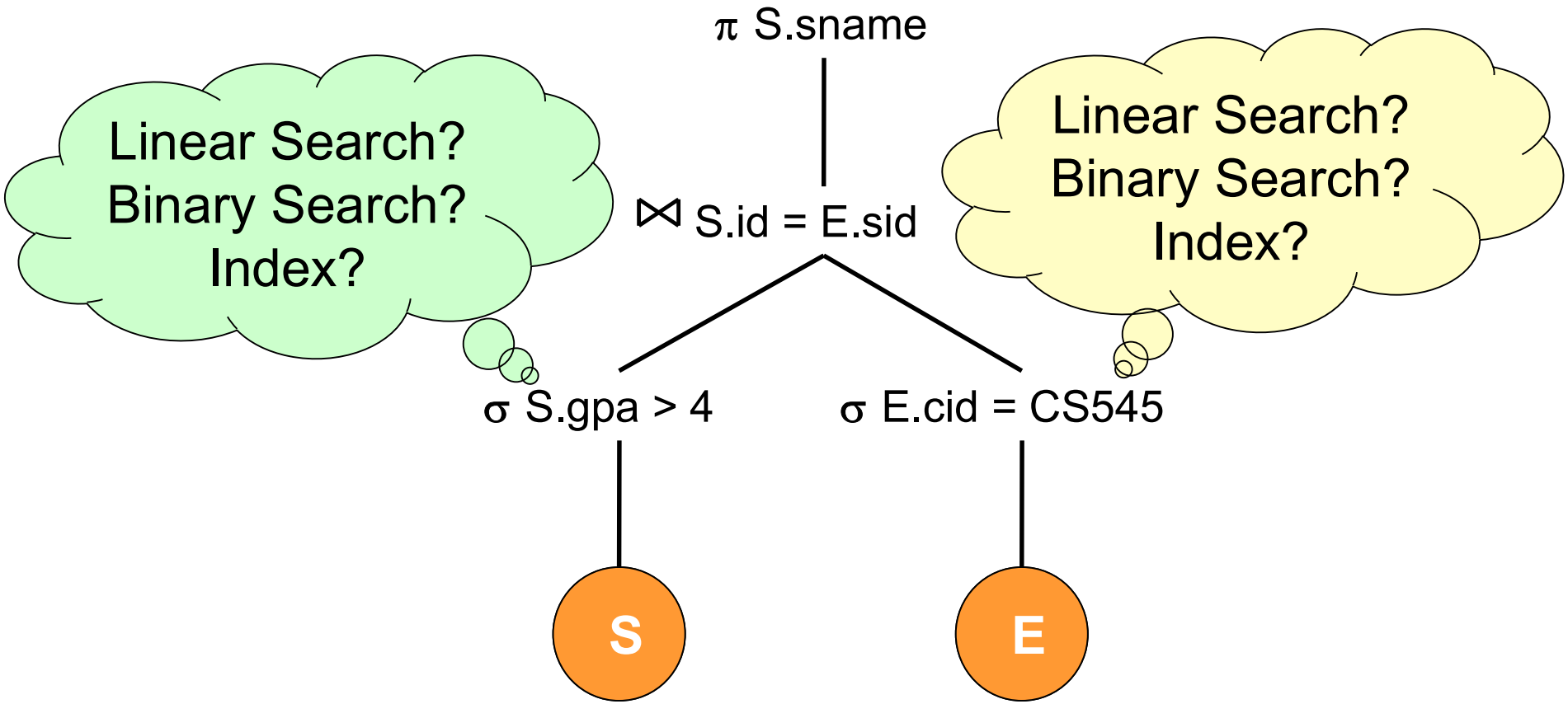
Putting it Together

For a **single condition:** $\sigma_{\text{R.attr op value}}(R)$

1. Use the **index** on the condition attribute if available, else
2. Use **Binary search** if the file is sorted on the condition attribute, else
3. Use the "brute force" **linear search** approach

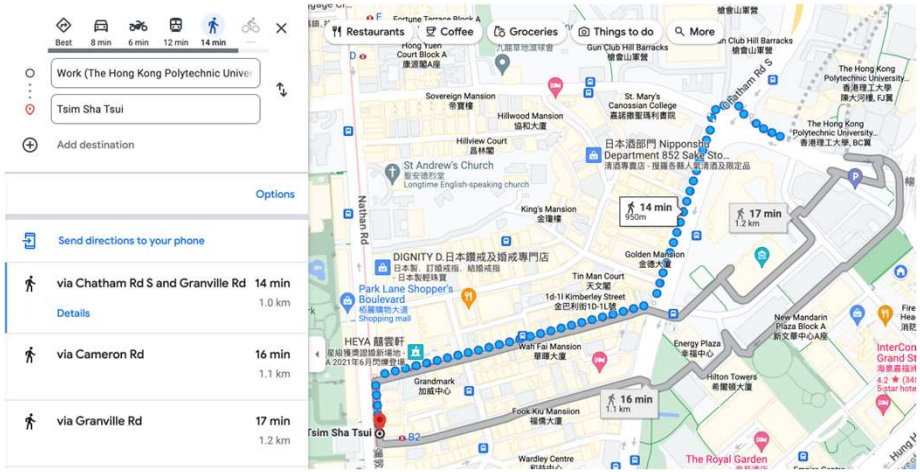
Heuristic-based vs. Cost-based?

Putting it Together



How to Implement Selection (σ)?

- Depends on:
  - **Type of Query:**
    1. Point query,
    2. Range query,
    3. Conjunction
    4. ~~Disjunction~~



  - **Type of available Access Path:**
    1. Index,
    2. Sorted File
    3. None!

Algorithms for SELECT with Conjunctions

$\sigma_{\text{Dept=COMP } \textbf{AND} \text{ Gender=F}}$ (Employee)

- **Using a composite index**:
  - If two or more attributes are involved in equality conditions in the conjunction, and
  - A composite index exists on the combined field, we can use the composite index directly
    - Example: Bitmap index
    - Works for point query but what about a range query?

Algorithms for SELECT with Conjunctions

σ Age<25 **AND** Salary>60000 **AND** Gender=F (Employee)

- **Conjunctive selection**:
  - If an attribute in the conjunctive condition has an index, then:

1. Use that index to retrieve the records
   - Eg: use the index on "age" to fetch all *age<25* records
2. For each retrieved record, **check** if it satisfies the remaining conditions in the conjunction
   - Eg: for each retrieved record with *age<25*, **check** if: *salary>60,000* and *gender=F*

Which Index?

- For **conjunctive selection** conditions:
  - If there are multiple indexes, query optimization should:

Choose the index that retrieves the fewest records

- $\sigma_{\text{Age<25 AND Salary>60,000}}$ (Employee)
  - Which index to choose: *Age* or *Salary*?
  - The optimizer should choose the access path (i.e., index) that retrieves the fewest records:
    1. Less blocks to read from disk
    2. Less checks to do in main memory

Selectivity

- In a movies database:
  - $\sigma_{\text{type="Drama" AND actor="Jackie Chan"}}$ (Movies)
  - Which index to choose: *type* or *actress*?

- From IMDb:
  - Total: **2,000,000** movies
  - Drama: **500,000** movies
  - Jackie Chan: **142** movies

  - The optimizer should:

1. Retrieve all *"Jackie Chan"* movies first, and then
2. Check which of the 142 movies is *"Drama"*

Selectivity

- The optimizer uses **selectivity** to choose!
- **Selectivity (S)**:
  - The **ratio** of: the number of records that satisfy a condition to the total number of records
  - Example: for $\sigma_{\text{type=“Drama”}}$, **S** = 500K/2M = 0.25
  - ***S*** between 0.0 and 1.0
    - ***S* = 0:** no records satisfy the condition
    - ***S* = 1:** all records satisfy the condition
  - Selectivity estimates are stored in DBMS **Catalog**

Selectivity

- Equality condition on a key attribute
  - Eg.: $\sigma_{\text{title="Police Story"}}$
    - Number of records that satisfy the condition:
      - Only one!
    - S = **1**/|R|, where |R| is the number of records in R

Selectivity

- Equality on an attribute with "x" distinct values
  - Eg.: $\sigma_{\text{gender="female"}}$
    - Distinct values: female and male (then x=2)
    - Number of records that satisfy the condition:
      - |R|/x (assuming uniform distribution)
    - S = (|R|/x)/|R| = 1/x (if x=2, then S=½)
  - Eg.: $\sigma_{\text{type="Drama"}}$
    - Distinct values: Drama, Comedy, Action
    - S = (|R|/x)/|R| = 1/3

Selectivity

- The actual distribution of selectivity is kept in the catalog in the form of a **histogram**
  - To get more accurate estimate of the number of records that satisfy a particular condition
    - Drama: 0.25
    - Comedy: 0.23
    - Documentary: 0.1
    - Musicals: 0.01
- The estimated number of records satisfying condition with selectivity **S** equals: **|R| * S**
  - **Eg.:** for $\sigma_{\text{type="documentary"}}$, 2M * 0.1 = 200K records
- Apply the condition with smallest estimate first

Relational Algebra Operations

- Unary Relational Operations:
  - Project and Select

- Binary Relational Operations:
  - Cartesian Product and Join

How to Implement Join (⋈)?

1. Nested-loop join
2. Single-loop join
3. ~~Sort-merge join~~
4. ~~Hash-join~~

(Equi) Join ⋈

		SID	Name	Age
Block 1	Record 1	546007	Peter	18
	Record 2	546100	Bob	19
Block 2	Record 3	546200	Ann	21
	Record 4	546207	Jane	20

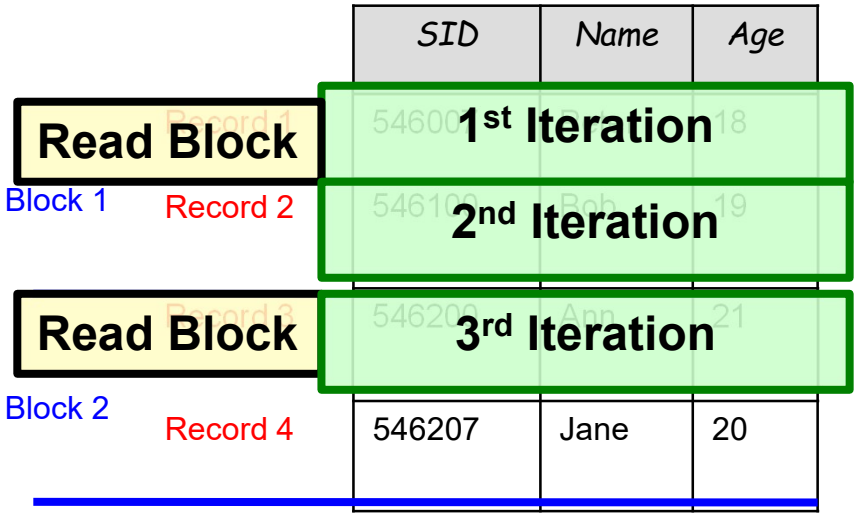
		SID	CID	GPA
Block 1	Record 1	546007	INFS	
	Record 2	546007	MMDS	
Block 2	Record 3	546200	INFS	
	Record 4	546100	ENGG	
Block 3	Record 5	546007	ENGG	
	Record 6	546200	MMDS	
Block 4	Record 7	546007	ELEC	
	Record 8	546200	ELEC	

Nested-loop Join

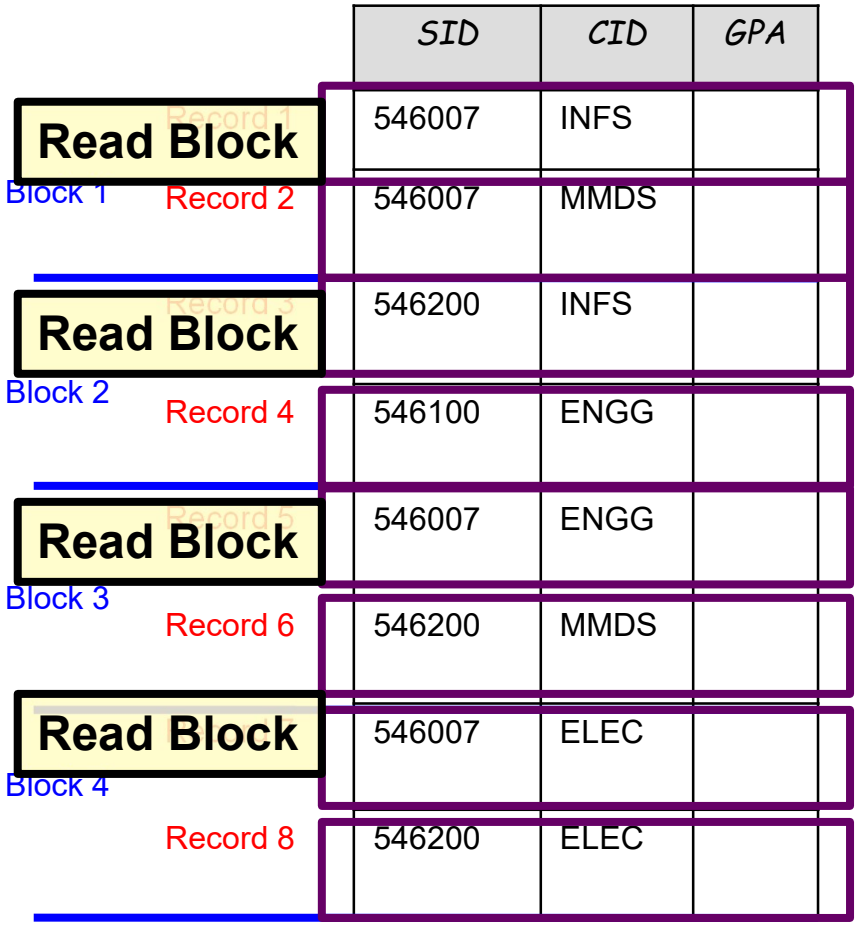
- Join relations T and S
  - A is the common attribute in T,
    B is the common attribute in S

```
For each record t in T  /*outer loop*/
{
    For  each record s in S  /*inner loop*/
    {
        if (t.A = s.B)
                add  <t,s>  to result
    }
}
```

Nested-loop Join



Relation T:
N_T records in B_T blocks



Relation S:
N_S records in B_S blocks

Total number of block reads =

B_T + Number of iterations (??) × B_S

$$B_T + N_T \times B_S$$

Nested-loop Join

- Join relations T and S
  - T: $\mathbf{N_T}$=50 records stored in $\mathbf{B_T}$=10 disk blocks
  - S: $\mathbf{N_S}$=6000 records stored in $\mathbf{B_S}$=2000 disk blocks
- Relation T (outer loop) is scanned one time
  - I/O cost = B_T
- Relation S (inner loop) is scanned **N_T times**
  - I/O cost = N_T x B_S
- **Tuple-based Implementation:**
  - I/O Cost = $B_T + N_T B_S$ = 10 + 50 * 2000 block access
  - If block access = 10 ms, total cost = 16.6 mins!

Nested-loop Join

- In tuple-based Implementation:

  For each **record** t in T

  For each record s in S

  Compare record s with record t /*in-memory operation*/

  - However, because of block access:
    - An entire block of records is already read from T in memory

- **Idea:** compare tuple s with the:
  - Entire block of records from T ➔ reduce iterations

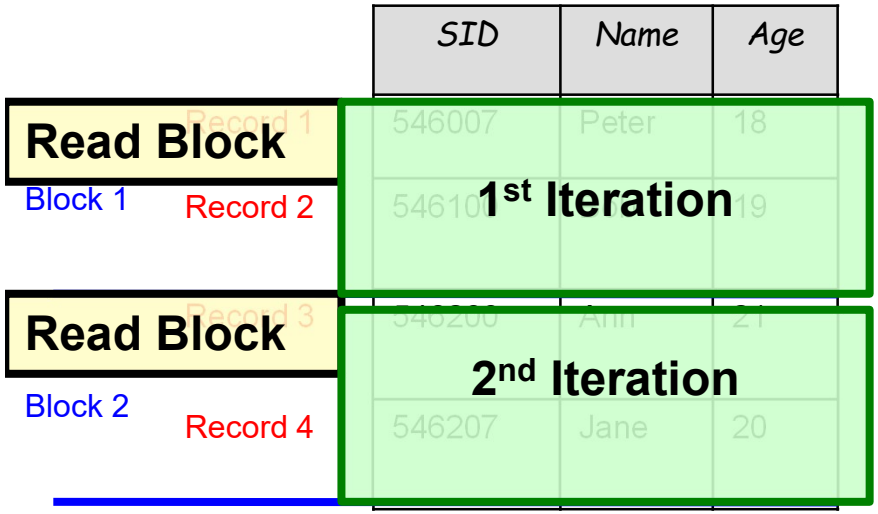
- **Page-oriented** implementation *Next slide…*

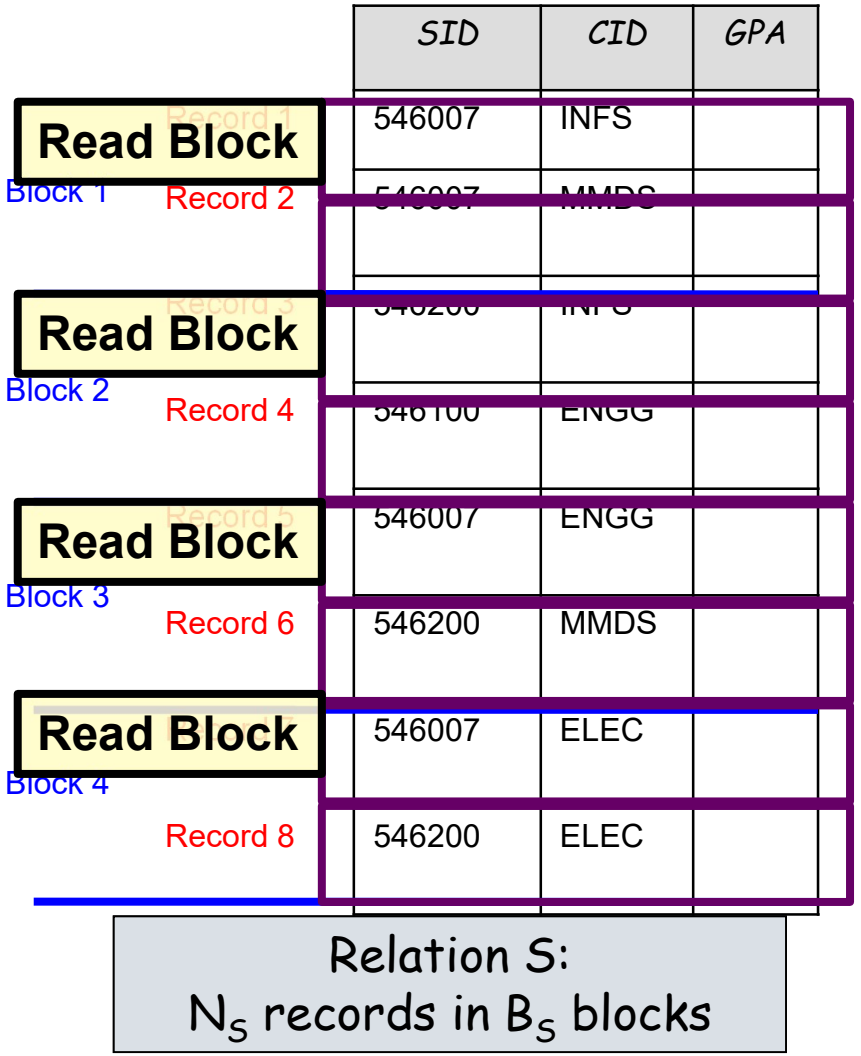
Nested-loop Join: Page-oriented

```
For each block T_block in T {  /*outer loop*/
    For  each block S_block in S {  /*inner loop*/

        For each record t in T_block { /* in-memory matching*/
            For each record s in S_block {
                if (t.A = s.B)
                    add  <t,s>  to result
            }
        }
    } /*end inner loop*/
} /*end outer loop*/
```

Nested-loop Join: Page-oriented





Total number of block reads =

B_T + Number of iterations (??) × B_S

$\mathbf{B_T + B_T \times B_S}$

Nested-loop Join: Page-oriented

- Join relations T and S
  - T: $\mathbf{N_T}$=50 records stored in $\mathbf{B_T}$=10 disk blocks
  - S: $\mathbf{N_S}$=6000 records stored in $\mathbf{B_S}$=2000 disk blocks
- Relation T (outer loop) is scanned one time
  - I/O cost = B_T
- Relation S (inner loop) is scanned **B_T times**
  - I/O cost = B_T x B_S
- **Page-oriented Implementation:**
  - I/O Cost = $B_T + B_T B_S$ = 10 + 10 * 2000 block access
  - If block access = 10 ms, total cost = 3.3 mins!

Block Nested-loop Join

- **Observation:** having more tuples from T in memory reduces the number of iterations ☺
- However, In both of the previous implementations:
  - We read only one block from T ☹

- **Idea:** Read a chunk of blocks from T instead of only one!
  - But how many?!

Block Nested-loop Join

- Let buffer size be $\mathbf{N_B}$ memory blocks, we need:
  - One block for buffering result, and
  - One block for reading from the inner file (i.e., S)
  - Then, remaining $\mathbf{N_B}$**-2**
    - Read as much as:

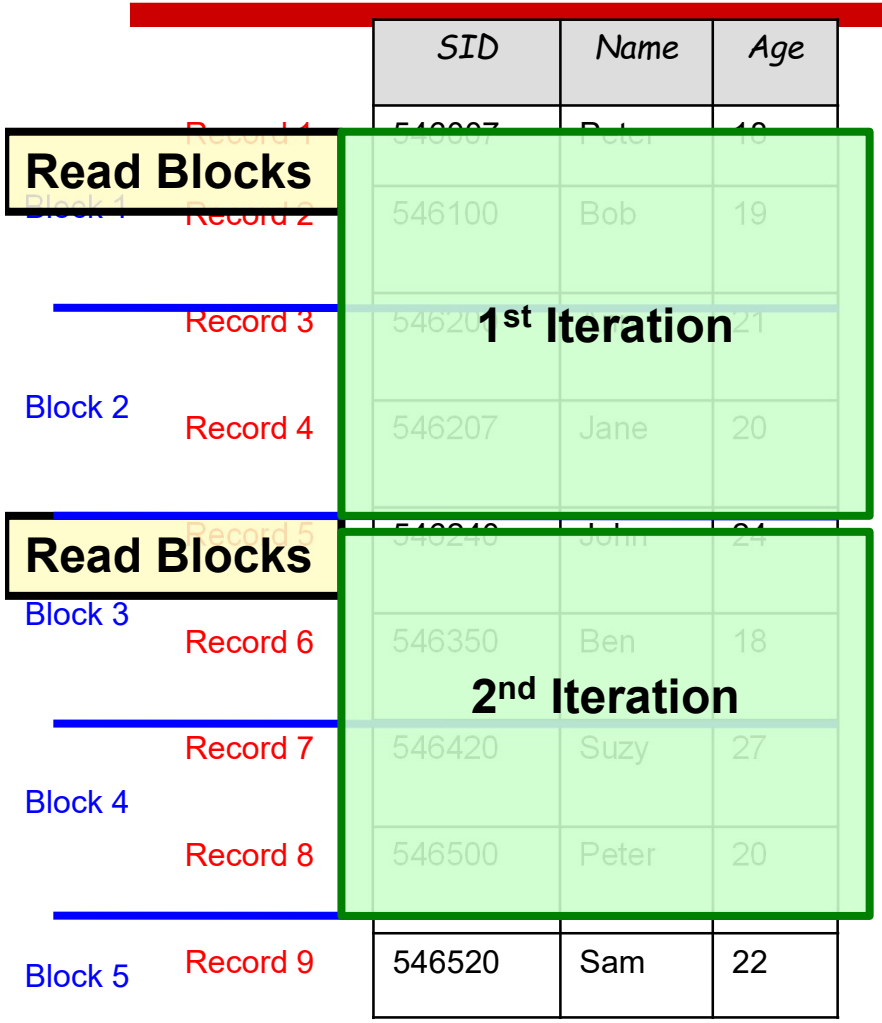
      N_B-2 blocks <u>at a time</u> from the outer file (i.e., T)

Block Nested-loop Join

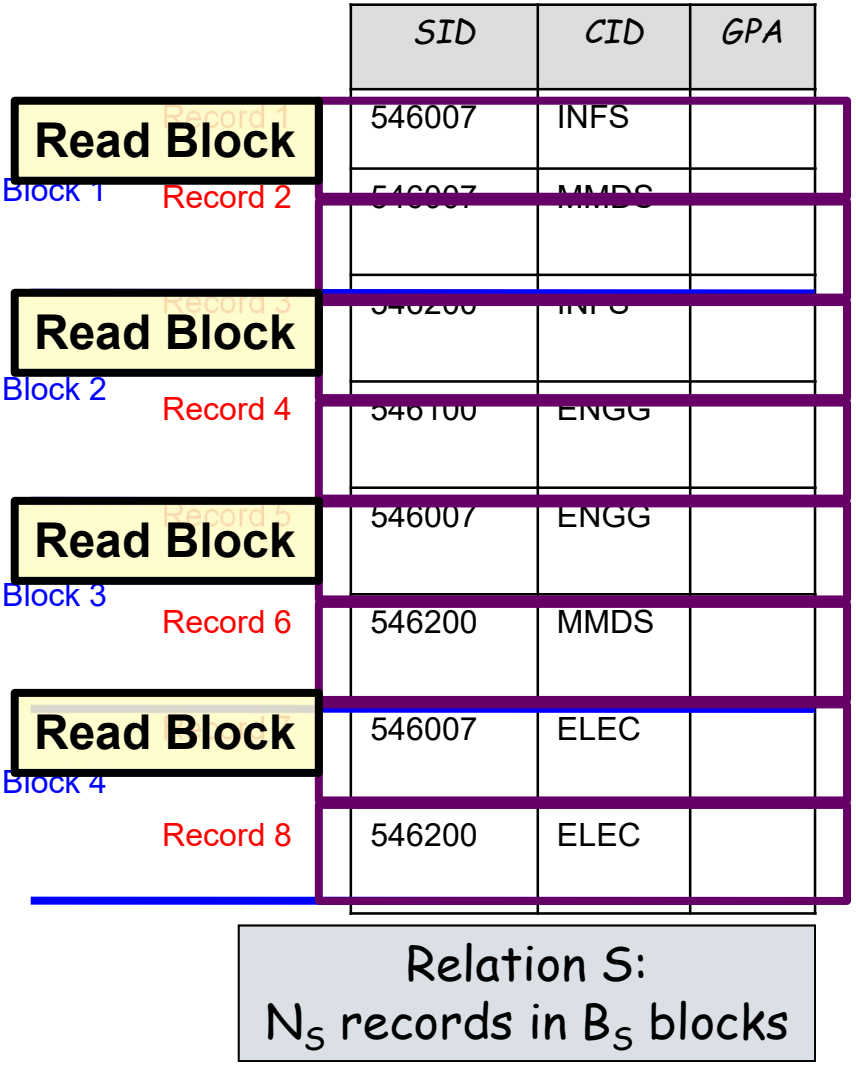
```
For each N_B-2 blocks T_blocks in T {  /*outer loop*/
    For  each block S_block in S {  /*inner loop*/

        For each record t in T_blocks { /* in-memory matching*/
            For each record s in S_block {
                if (t.A = s.B)
                    add <t,s>  to result
            }
        }
    } /*end inner loop*/
} /*end outer loop*/
```

Block Nested-loop Join

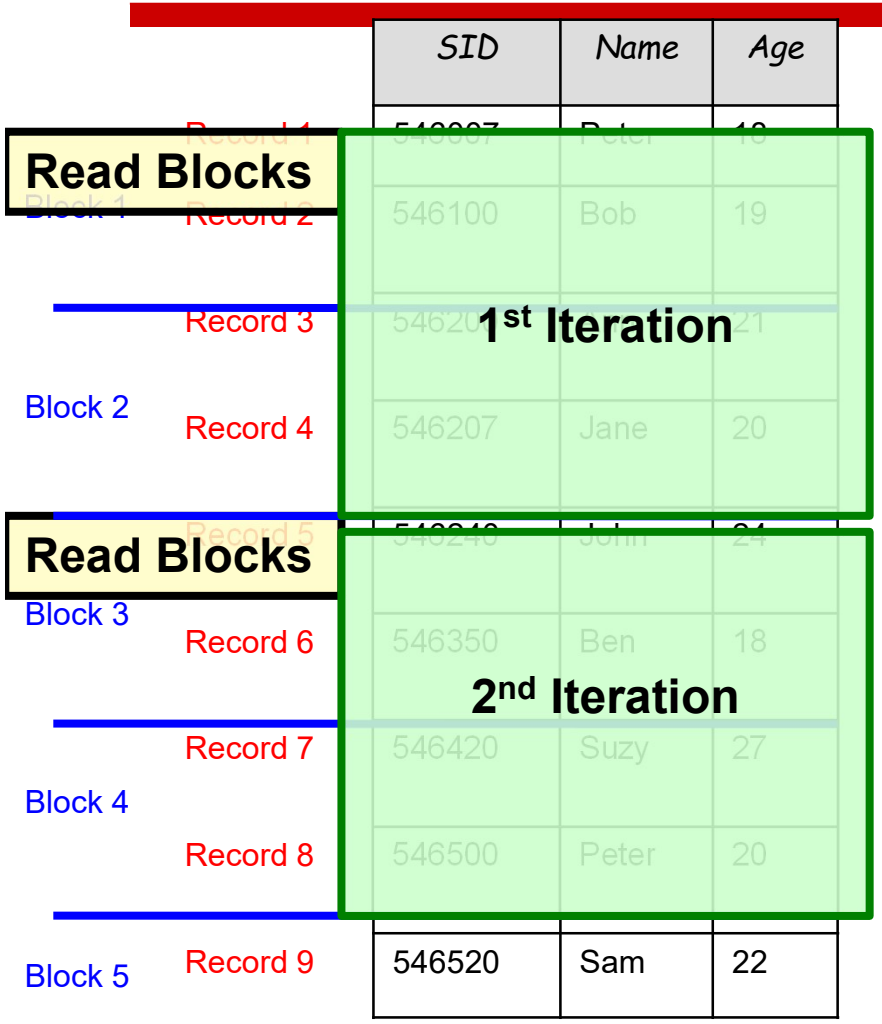


Relation T:
N_T records in B_T blocks



Relation S:
N_S records in B_S blocks

Block Nested-loop Join



Relation T:
N_T records in B_T blocks

Total number of block reads =

B_T + Number of iterations (??) × B_S

Number of iterations
= number of chunks = $\mathbf{B_T / (N_B-2)}$

Total = $\mathbf{B_T + (B_T /(N_B-2)) \times B_S}$

Order Matters!

- When T: outer & S: inner
  - $B_T + B_S * (B_T/(N_B-2))$
- When T: inner & S: outer
  - $B_S + B_T * (B_S/(N_B-2))$
- In general:

$$\mathbf{B_{Outer} + B_{Inner} \times (B_{Outer}/(N_B-2))}$$

- If $B_{Outer} < B_{inner}$:
  - Has no impact on the term: $B_{Inner} * (B_{Outer}/(N_B-2))$
  - Reduces the term: B_{outer}
- Overall, the smaller file should be the outer file

Query Processing: Natural Joins

1. Nested-loop join
2. Single-loop join
3. ~~Sort-merge join~~
4. ~~Hash-join~~

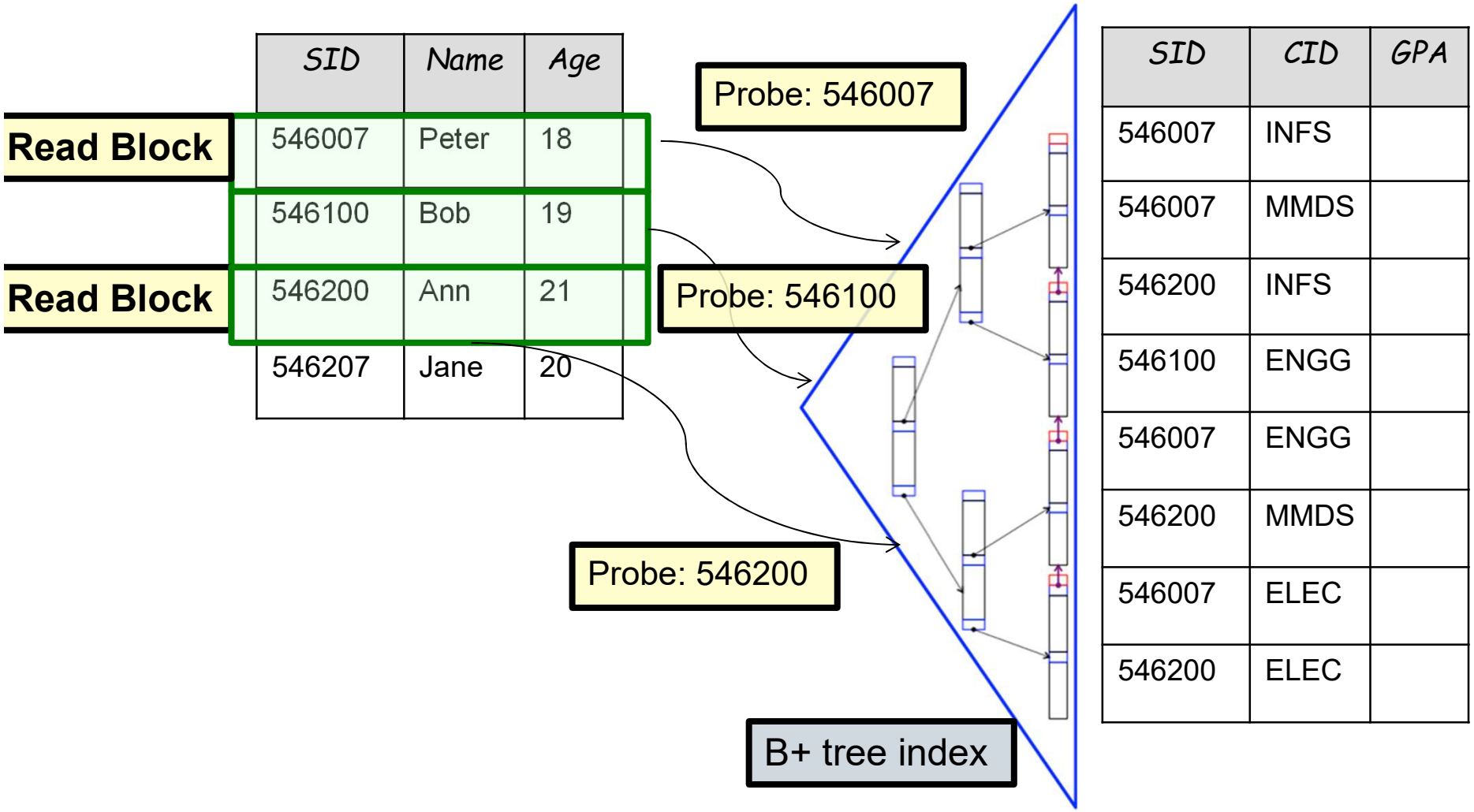
Single-loop Join

- Join relations T and S
  - A is the common attribute in T,
    B is the common attribute in S

- Only works if an **index** exists for one of the two join attributes (A or B).
  - Assume an index on attribute B in relation S, then:

```
For each record t in T  /*outer loop*/
{
    Locate records s from S, that satisfy:
                t.A = s.B
}
```

Single-loop Join



Single-loop Join Performance

- For outer relation:
  - Number of blocks B_{Outer} ,and
  - Number of records N_{Outer}
- For inner relation:
  - Number of index levels L_{Index}
- Cost in number of block accesses:

$$B_{Outer} + (N_{Outer} \times (L_{Index} + 1))$$

  - Order matters for single-loop join as well

Operator Evaluation

- Several alternative algorithms are available for **implementing** each relational operator
  - No algorithm is universally superior
  - Several factors influence which algorithm performs **best**:
    - Size of the tables
    - Existing index and sort orders
    - Size of available memory…
  - Heurist-based vs. Cost-based

Query Tree Optimization

- Get the **initial** Query Tree
  - Conceptual evaluation strategy (next slides)

- Transform it into an **equivalent** Query Tree
  - Represents a **different** relational algebra expression
  - Gives the **same** result
  - More **efficient** to execute

Conceptual Evaluation Strategy

```
SELECT    [DISTINCT] select-list
FROM      from-list
WHERE     qualifications
```

- Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:
  1. Compute the cross-product of ***from-list***
  2. Discard resulting tuples that fail ***qualifications***
  3. Delete attributes that are not in ***select-list***
  4. If `DISTINCT` is specified, eliminate duplicates

Example of Conceptual Evaluation

```
SELECT   S.name
FROM     Sailors S, Reserves R
WHERE    S.sid=R.sid AND R.bid=103
```

SID	Name	Rating	Age
22	Dustin	7	45
31	Lubber	8	55
58	Rusty	10	35

Sailors

SID	BID	Day
22	101	10/10/96
58	103	11/12/96

Reserves

Example of Conceptual Evaluation

```
SELECT   S.name
FROM     Sailors S, Reserves R
WHERE    S.sid=R.sid AND R.bid=103
```

1. Compute the cross-product of *from-list*

S.SID	Name	Rating	Age	R.SID	BID	Day
22	Dustin	7	45	22	101	10/10/96
22	Dustin	7	45	58	103	11/12/96
31	Lubber	8	55	22	101	10/10/96
31	Lubber	8	55	58	103	11/12/96
58	Rusty	10	35	22	101	10/10/96
58	Rusty	10	35	58	103	11/12/96

Example of Conceptual Evaluation

```
SELECT    S.name
FROM      Sailors S, Reserves R
WHERE     S.sid=R.sid AND R.bid=103
```

2. Discard resulting tuples that fail *qualifications*

S.SID	Name	Rating	Age	R.SID	BID	Day
22	Dustin	7	45	22	101	10/10/96
22	Dustin	7	45	58	103	11/12/96
31	Lubber	8	55	22	101	10/10/96
31	Lubber	8	55	58	103	11/12/96
58	Rusty	10	35	22	101	10/10/96
58	Rusty	10	35	58	103	11/12/96

Example of Conceptual Evaluation

```
SELECT   S.name
FROM     Sailors S, Reserves R
WHERE    S.sid=R.sid AND R.bid=103
```

3. Delete attributes that are not in *select-list*

S.SID	Name	Rating	Age	R.SID	BID	Day
22	Dustin	7	45	22	101	10/10/96
22	Dustin	7	45	58	103	11/12/96
31	Lubber	8	55	22	101	10/10/96
31	Lubber	8	55	58	103	11/12/96
58	Rusty	10	35	22	101	10/10/96
58	Rusty	10	35	58	103	11/12/96

Conceptual Evaluation Strategy

- This strategy is probably the least efficient way to compute a query!

- An "**optimizer**" will find more efficient strategies to compute the same answers

Query Tree Optimization

- Get the **initial** Query Tree
  - Conceptual evaluation strategy (next slides)

- Transform it into an **equivalent** Query Tree
  - Represents a **different** relational algebra expression
  - Gives the **same** result
  - More **efficient** to execute
  - Heuristic (rule)-based

Example Database

ID	Name	Gender	Age	Height
1	…			
2	Nicole Kidman	F	….	…
3				
…				

Actors

General Transformation Rules

- Cascade of σ
  - A conjunctive selection condition can be **broken up** into a cascade of individual σ operations
  - $\sigma_{c1\ AND\ c2\ AND\ \ldots\ AND\ cn}(R) = \sigma_{c1}(\sigma_{c2}(\ \ldots\ (\sigma_{cn}(R))\ \ldots\ ))$

$\sigma_{A.gender = \text{'F'}\ \textbf{AND}\ A.age > 25}(A)$ $= \sigma_{A.gender = \text{'F'}}(\sigma_{A.age > 25}(A))$

$\sigma_{A.gender = \text{'F'}\ \textbf{OR}\ A.age > 25}(A)$ $= \sigma_{A.gender = \text{'F'}}(\sigma_{A.age > 25}(A))$ ✗

General Transformation Rules

- Commutativity of σ
  - The σ is commutative
  - $\sigma_{c1}(\sigma_{c2}(R)) = \sigma_{c2}(\sigma_{c1}(R))$

$\sigma_{A.gender = \text{'F'}}(\sigma_{A.age > 25}(A))$ $= \sigma_{A.age > 25}(\sigma_{A.gender = \text{'F'}}(A))$

When is it beneficial ?

$\sigma_{A.height > 1.8m}(\sigma_{A.age > 18}(A))$ $= \sigma_{A.age > 18}(\sigma_{A.height > 1.8m}(A))$

Execute the one that returns fewest records first
→ Smaller **intermediate result** to store and process

General Transformation Rules

- Cascade of Π
  - In a cascade of of π operations, all but the last one can be **ignored**
  - $\Pi_{list1}\ (\Pi_{list2}\ (\ldots\ (\Pi_{listN}\ (R))\ \ldots\ )) = \Pi_{list1}\ (R)$

$\Pi_{A.name}\ (\Pi_{A.name,\ A.age}\ (A))$ $= \Pi_{A.name}\ (A)$

$\Pi_{A.name}\ (\Pi_{A.age}\ (A))$ ✗ $= ??$

General Transformation Rules

- Commuting σ with Π
  - $\Pi_{A1, A2, \ldots, An}(\sigma_c(R)) = \sigma_c(\Pi_{A1, A2, \ldots, An}(R))$
  - If c involves only attributes in A_1, A_2, …, An, then the two operations can be commuted

$\Pi_{A.name, A.age}(\sigma_{A.age > 18}(A))$ $= \sigma_{A.age > 18}(\Pi_{A.name, A.age}(A))$

$\Pi_{A.name, A.age}(\sigma_{A.height>1.8m}(A))$ $= \sigma_{A.height >1.8m}(\Pi_{A.name, A.age}(A))$ ✗

Example Database

ID	Name	Gender	Age	Height
1	…			
2	Nicole Kidman	F	….	…
3				
…				

Actors

ID	Movie	Year	Award
2	The Hours	2003	Best Actress

Oscars

General Transformation Rules

- Commutativity of join and cross product
  - $R \bowtie S = S \bowtie R$
  - $R \times S = S \times R$

$$\text{Actors} \bowtie_{\text{Actors.id=Oscars.id}} \text{Oscars}$$

$$= \text{Oscars} \bowtie_{\text{Actors.id=Oscars.id}} \text{Actors}$$

General Transformation Rules

- Commuting σ with $\bowtie$
  - $\sigma_c (R \bowtie S) = (\sigma_c (R)) \bowtie S$
    - If <u>all the attributes</u> in the selection condition "**c**" involve <u>only</u> the attributes of <u>one</u> of the joined relations (e.g., R), the two operations can be commuted

$$\sigma_{\text{Actors.age} < 18} (\text{Actors} \bowtie_{\text{Actors.id=Oscars.id}} \text{Oscars}) =$$

$$= (\sigma_{\text{Actors.age} < 18} (\text{Actors})) \bowtie_{\text{Actors.id=Oscars.id}} \text{Oscars}$$

General Transformation Rules

- If "**c**" can be written as: **c_1 AND c_2**, where
  - c_1 involves only the attributes in R, and
  - c_2 involves only the attributes in S:
  - $\sigma_c (R \bowtie S) = (\sigma_{c1} (R)) \bowtie (\sigma_{c2} (S))$

$$\sigma_{\text{Actors.age <18 AND Oscars.year >2000}} (\text{Actors} \bowtie_{\text{Actors.id=Oscars.id}} \text{Oscars})$$

$$=(\sigma_{\text{Actors.age<18}} (\text{Actors})) \bowtie_{\text{Actors.id=Oscars.id}} (\sigma_{\text{Oscars.year>2000}} (\text{Oscars}))$$

General Transformation Rules

- Commuting Π with $\bowtie$
  - Project list L = {A_1, A_2, …, A_n, B_1, B_2, …, B_m)
  - **Suppose:** A_1, A_2, …, A_n are attributes of relation R, and B_1, B_2, …, B_m are attributes of relation S
  - $\Pi_L (R \bowtie_c S) = (\Pi_{A1, A2, \ldots, An} (R)) \bowtie_c (\Pi_{B1, B2, \ldots, Bm} (S))$

$$\Pi_{\text{Actors.name,Actors.id,Oscars.id}} (\text{Actors} \bowtie_{\text{Actors.id=Oscars.id}} \text{Oscars})$$

$$=(\Pi_{\text{Actors.name,Actors.id}} (\text{Actors})) \bowtie_{\text{Actors.id=Oscars.id}} (\Pi_{\text{Oscars.id}} (\text{Oscars}))$$

General Transformation Rules

$\Pi_{\text{Actors.name,Actors.id,Oscars.year}}$ (Actors $\bowtie_{\text{Actors.id=Oscars.id}}$ Oscars)

=($\Pi_{\text{Actors.name,Actors.id}}$(Actors)) $\bowtie_{\text{Actors.id=Oscars.id}}$($\Pi_{\text{Oscars.year}}$ (Oscars))

=($\Pi_{\text{Actors.name,Actors.id}}$(Actors)) $\bowtie_{\text{Actors.id=Oscars.id}}$
($\Pi_{\textbf{Oscars.id}, \text{Oscars.year}}$ (Oscars))

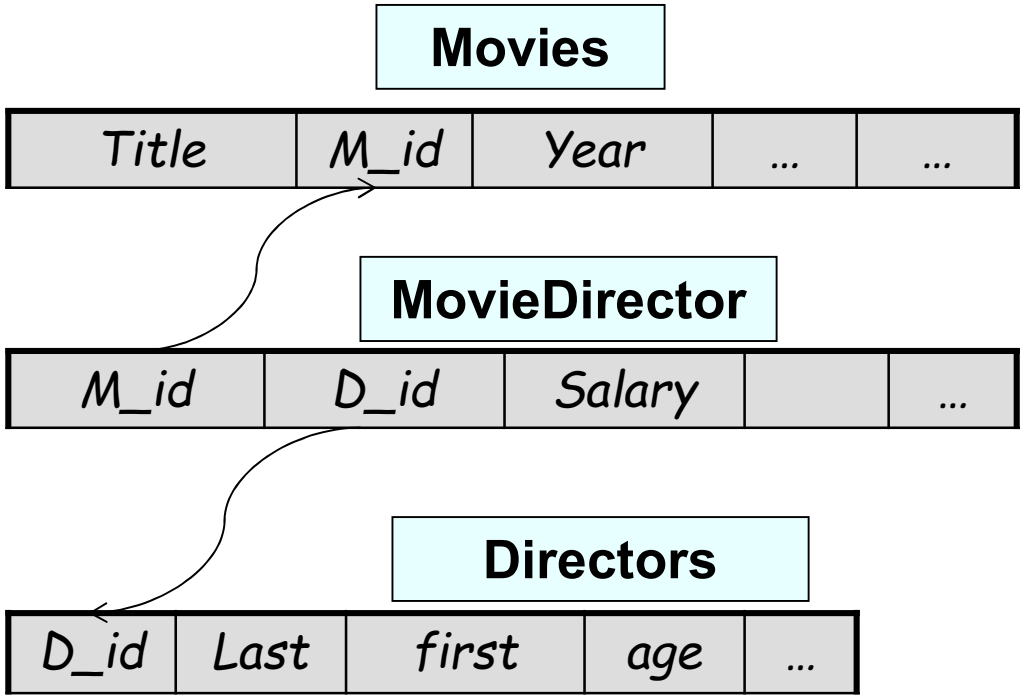
- Π_L (R $\bowtie_c$ S) = ($\Pi_{A1, A2, \ldots, An}$ (R)) $\bowtie_c$ ($\Pi_{B1, B2, \ldots, Bm}$ (S))
  - "**c**" must only involve attributes in L
  - What if "**c**" contains additional attributes?

General Transformation Rules

- Associativity of join and cross product
  - $(R * S) * T = R * (S * T)$
    - Where * can be $\bowtie$ or x
- Converting a (σ, x) sequence into a join operation
  - If c corresponds to a join condition, then
    - $\sigma_c (R \times S) = R \bowtie_c S$
- More transformations…


IKEA Job Interview
Please have a seat
CANAAy PETE

Example Schema



For every movie produced in "1990", **retrieve**:
movie title, movie number,
and the director's last name,
first name, and age

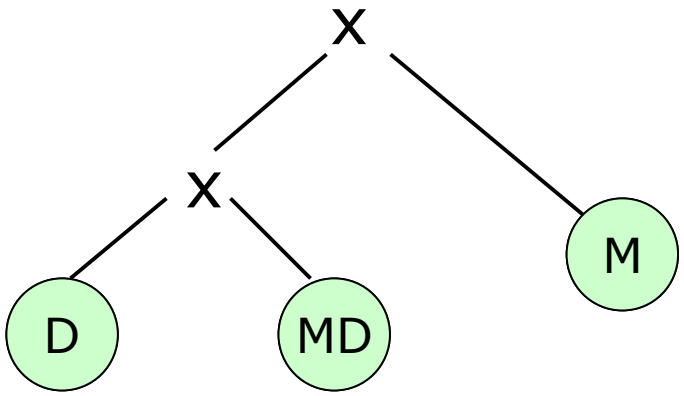
```
SELECT  M.title, M.M_id, D.last, D.first, D.age
FROM    Movies as M, MovieDirector as MD, Directos as D
WHERE   M.M_id=MD.M_id AND  MD.D_id=D.D_id AND  M.year= '1990'
```

Query Tree Optimization

- Get **Initial** Query Tree
  1. Apply Cartesian Product of relations in FROM
  2. Selection and Join conditions of WHERE are applied
  3. Project on attributes in SELECT

- **Transform** it into an **Equivalent** Query Tree
  - Represents a different relational expression
  - Gives the same result
  - More efficient to execute
  - Heuristic (rule)-based

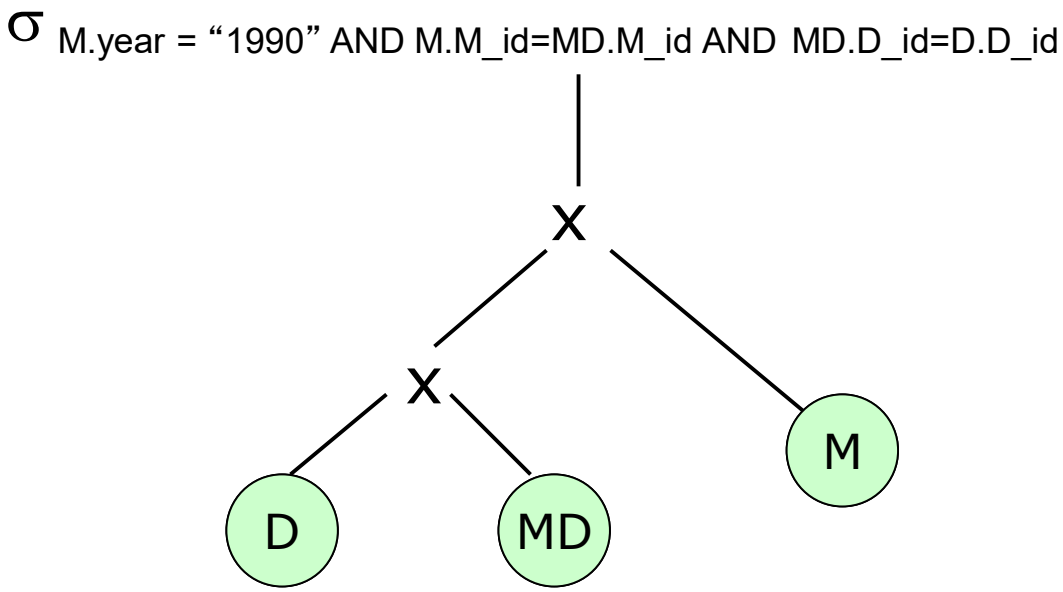
Example Query Tree



1

```
SELECT  M.title, M.M_id, D.last, D.first, D.age
FROM    Movies as M, MovieDirector as MD, Directors as D
WHERE   M.M_id=MD.M_id AND  MD.D_id=D.D_id AND  M.year= '1990'
```

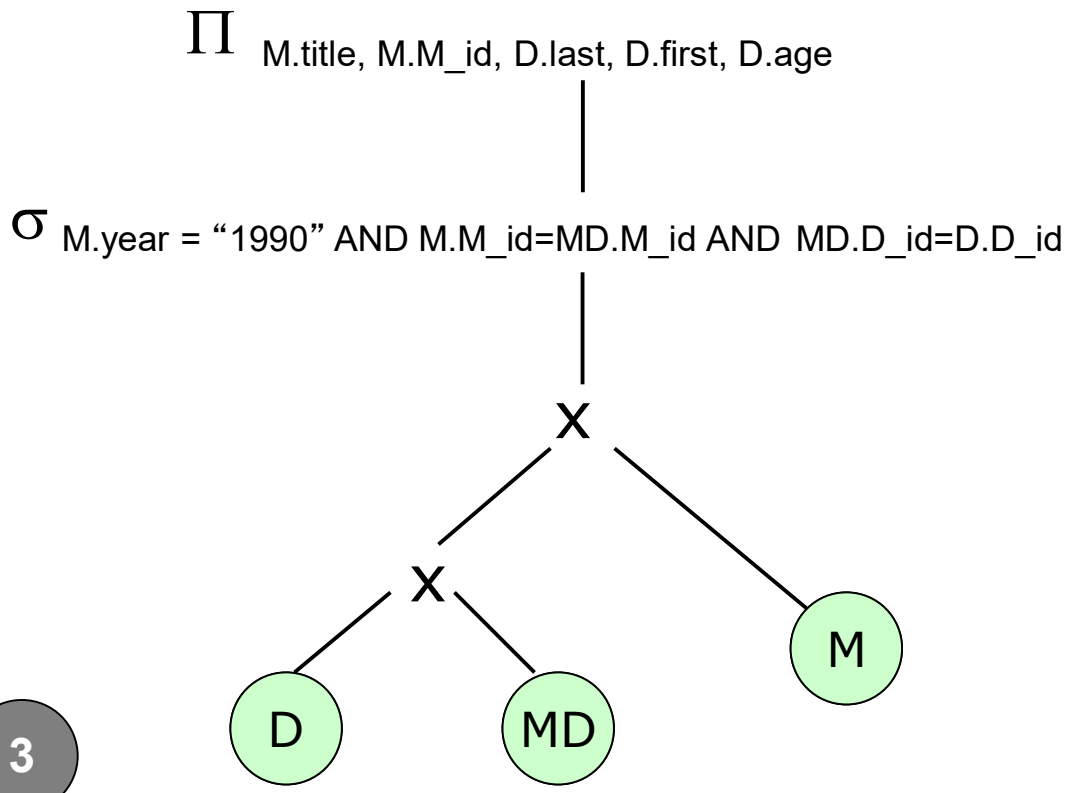
Example Query Tree

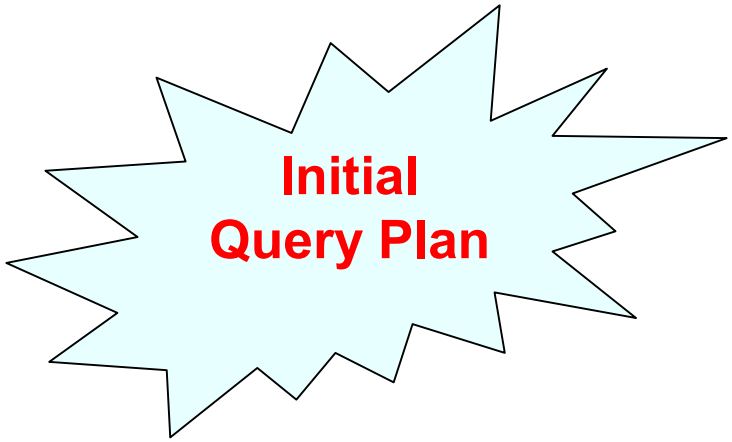


2

```
SELECT  M.title, M.M_id, D.last, D.first, D.age
FROM    Movies as M, MovieDirector as MD, Directors as D
WHERE   M.M_id=MD.M_id AND MD.D_id=D.D_id AND M.year= ‘1990’
```

Example Query Tree





3

```
SELECT  M.title, M.M_id, B.last, B.first, B.age
FROM    Movies as M, MovieDirector as MD, Directors as D
WHERE   M.M_id=MD.M_id AND  MD.D_id=D.D_id AND  M.year= '1990'
```

Query Tree Optimization

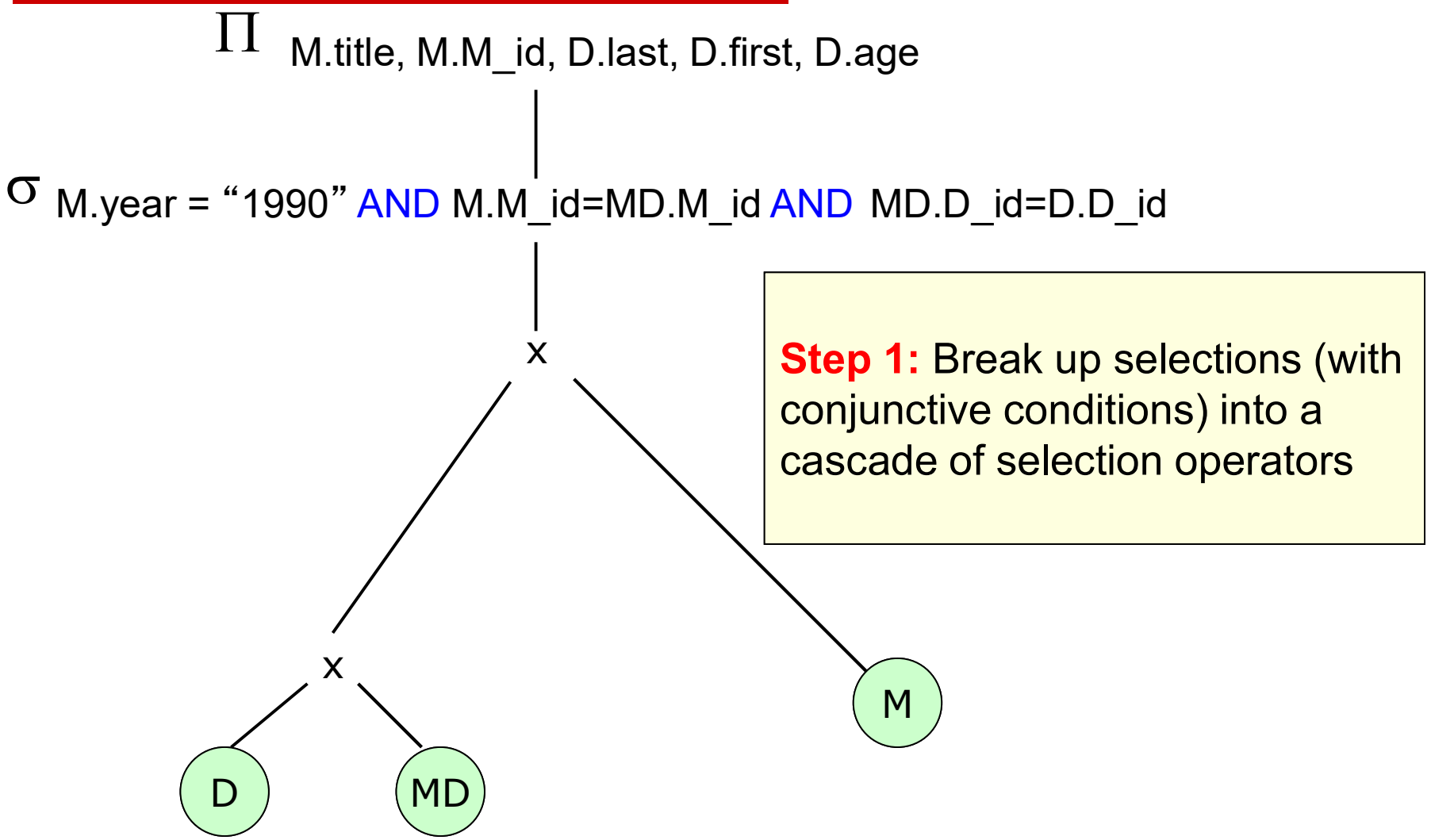
- Get **Initial** Query Tree
  1. Apply Cartesian Product of relations in FROM
  2. Selection and Join conditions of WHERE are applied
  3. Project on attributes in SELECT

- **Transform** it into an **Equivalent** Query Tree
  - Represents a different relational expression
  - Gives the same result
  - More efficient to execute
  - Heuristic (rule)-based (next slides)

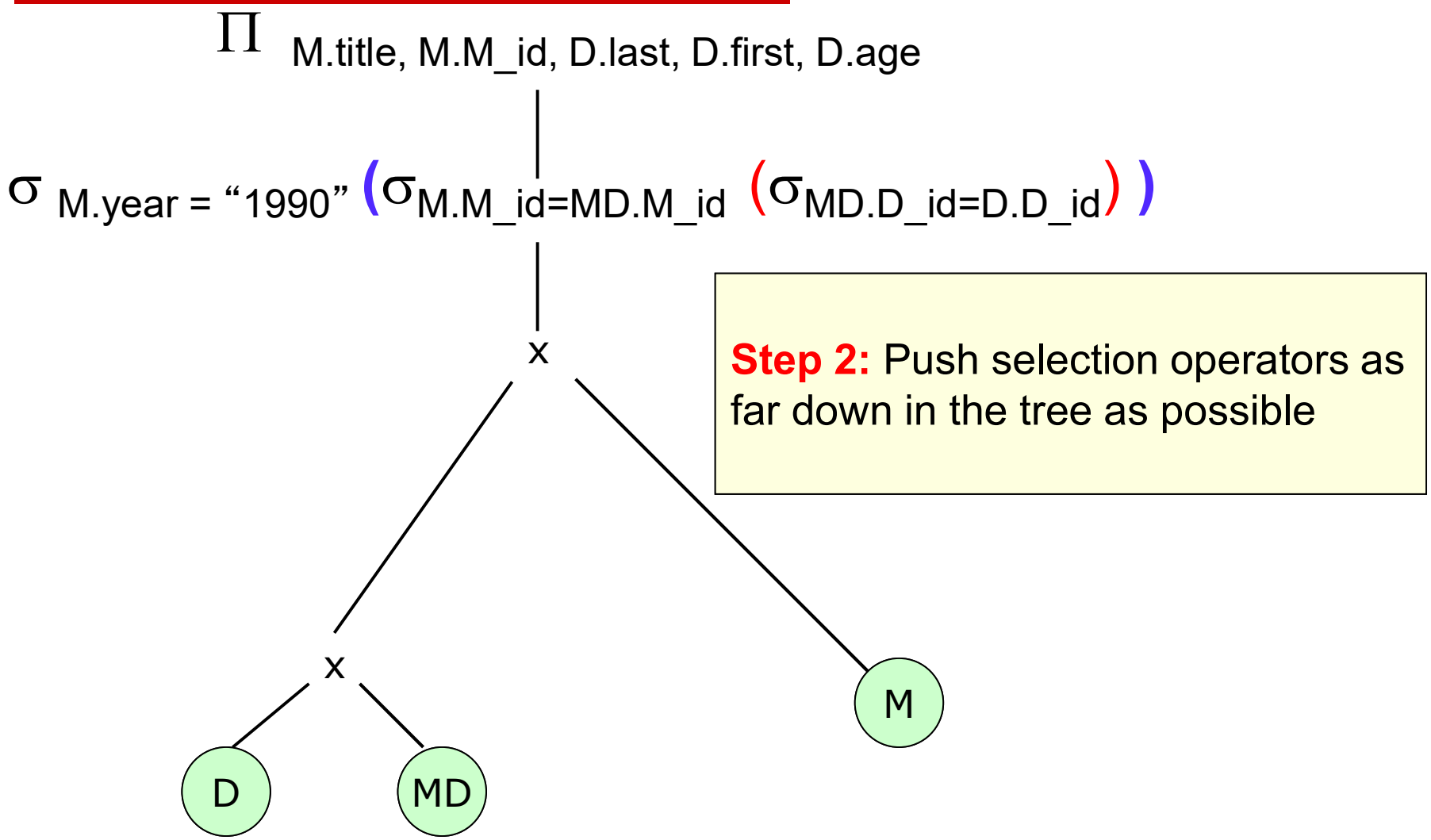
Outline of Query Tree Optimization

1. **Break up** selections (with conjunctive conditions) into a cascade of selection operators
2. **Push** selection operators as far down in the tree as possible
3. **Convert** Cartesian products into joins
4. **Rearrange** leaf nodes so that to:
   - Execute first the most restrictive select operators
     - What is restrictive? (Fewest tuples or Smallest size)
5. **Move** projections as far down as possible

Example of Query Tree (Step 0)

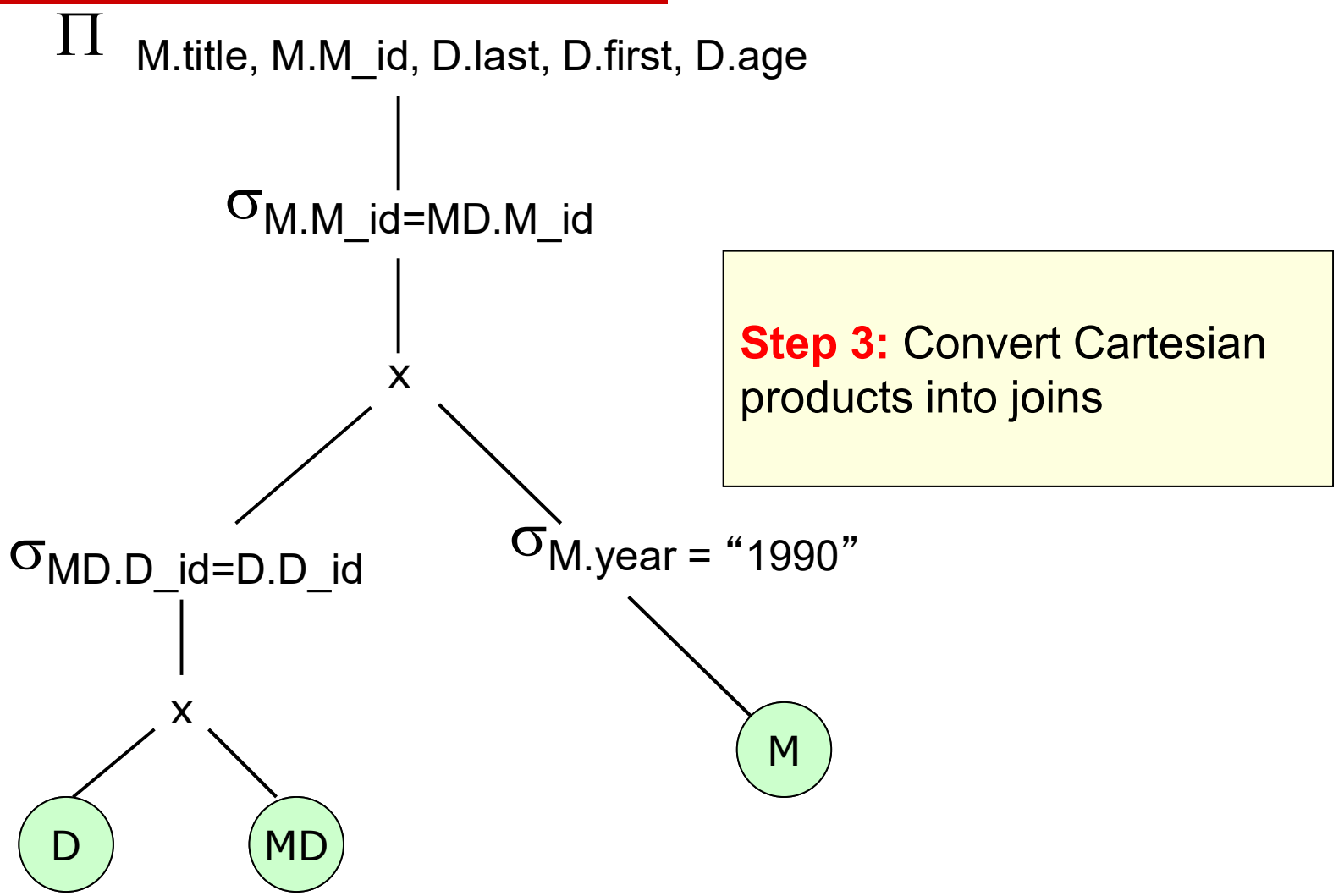


Example of Query Tree (Step 1)



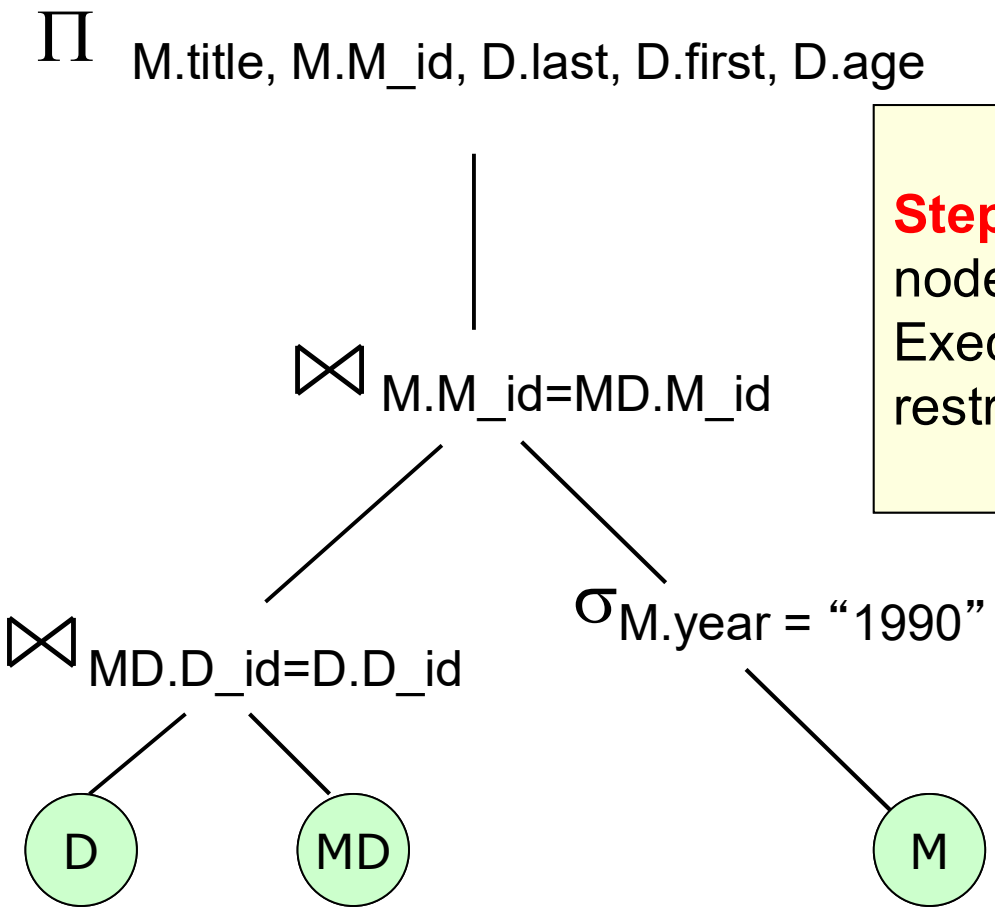
Step 2: Push selection operators as far down in the tree as possible

Example of Query Tree (Step 2)



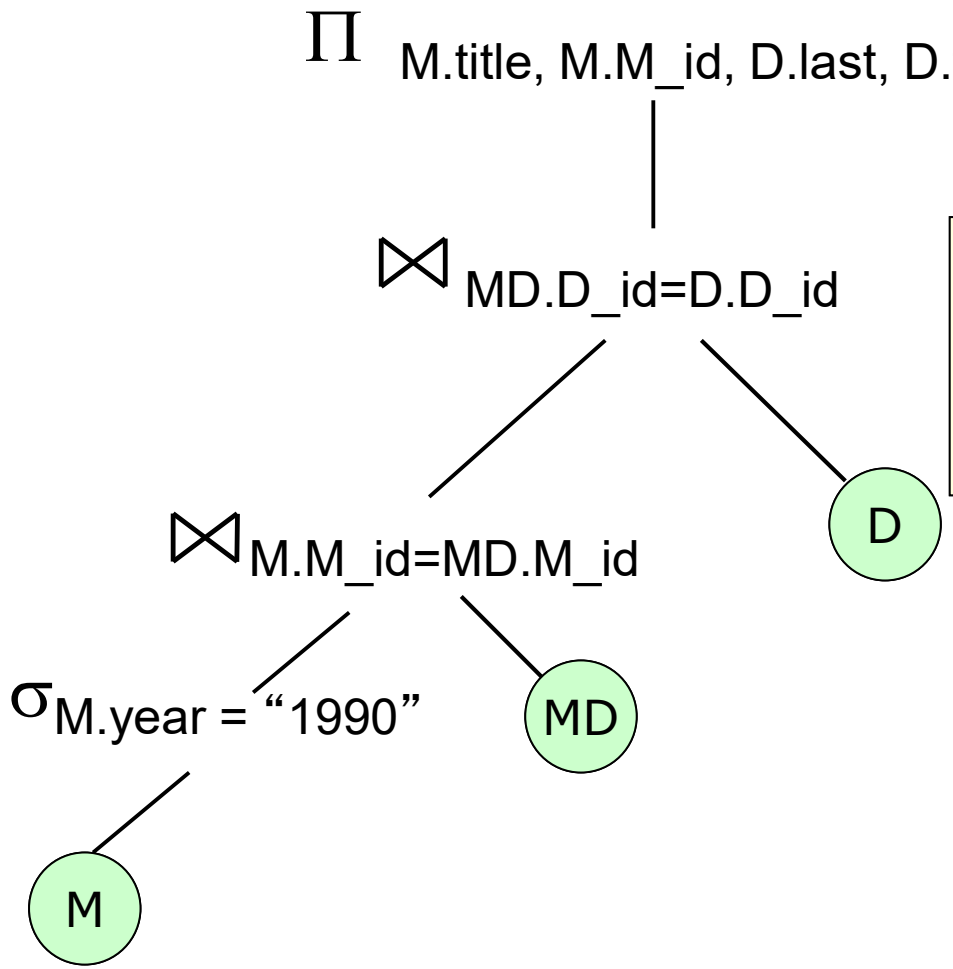
Step 3: Convert Cartesian products into joins

Example of Query Tree (Step 3)



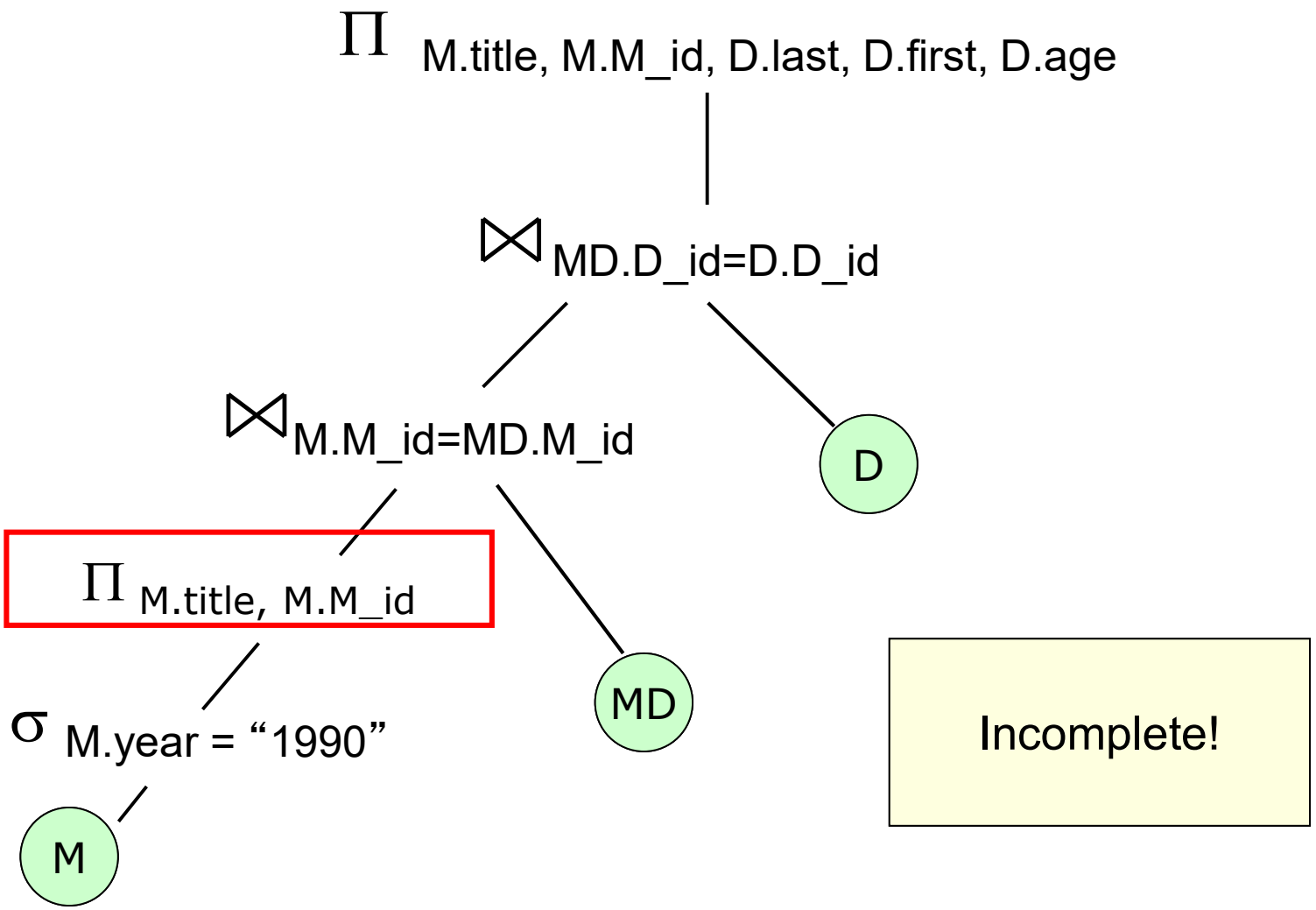
Step 4: Rearrange leaf nodes so that to:
Execute first the most restrictive select operators

Example of Query Tree (Step 4)

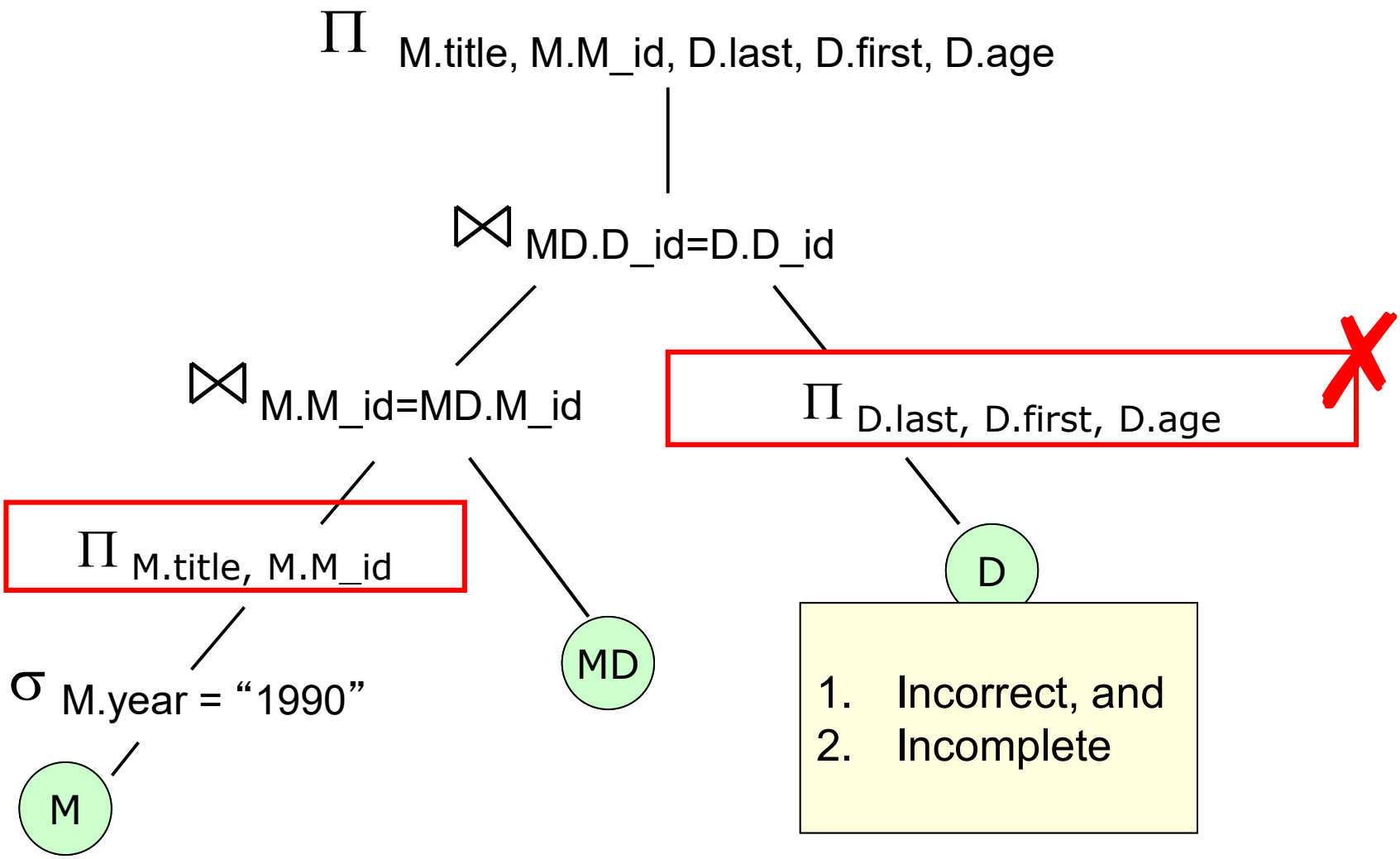


Step 5: Move projections as far down as possible

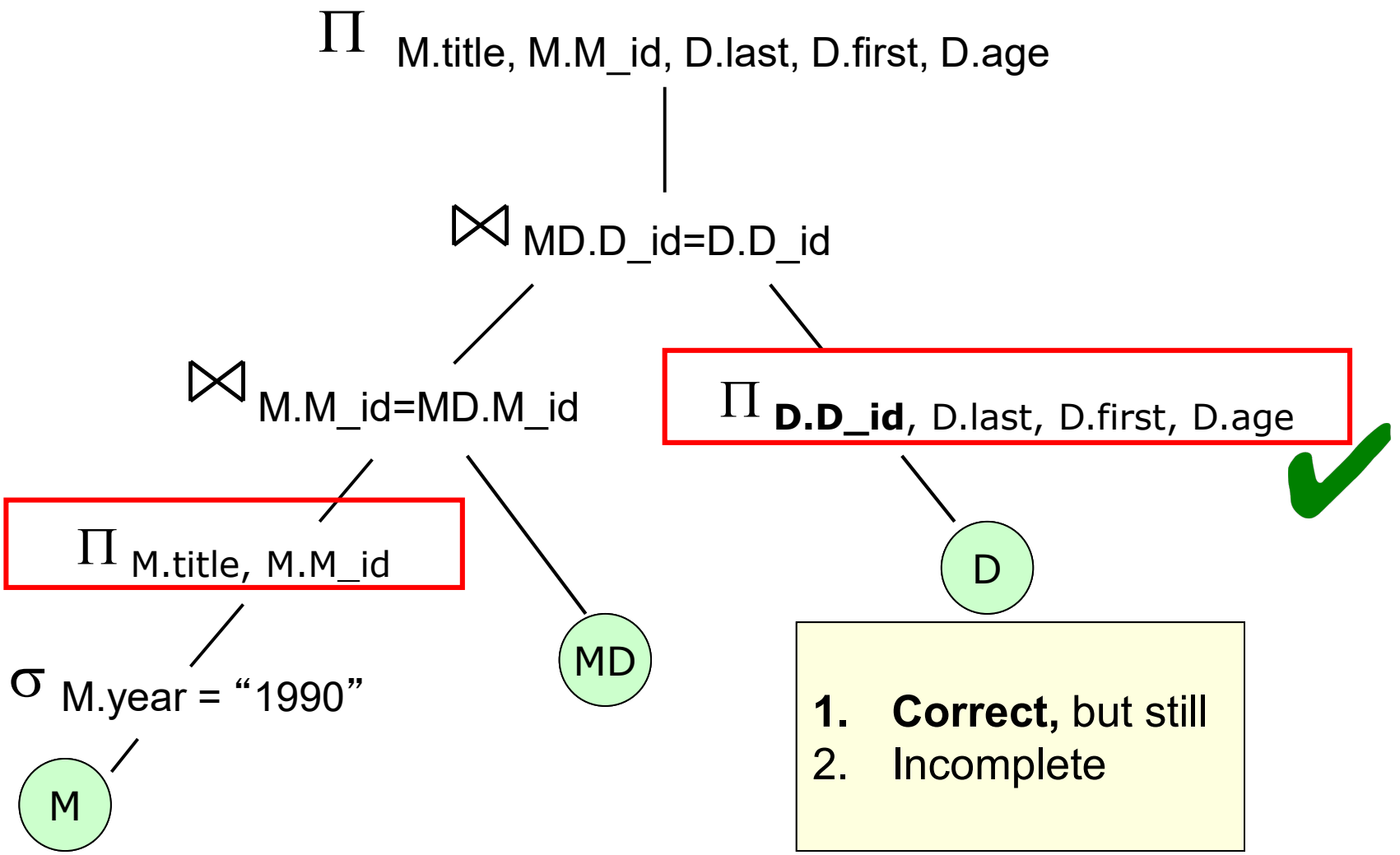
Example of Query Tree (Step 5)



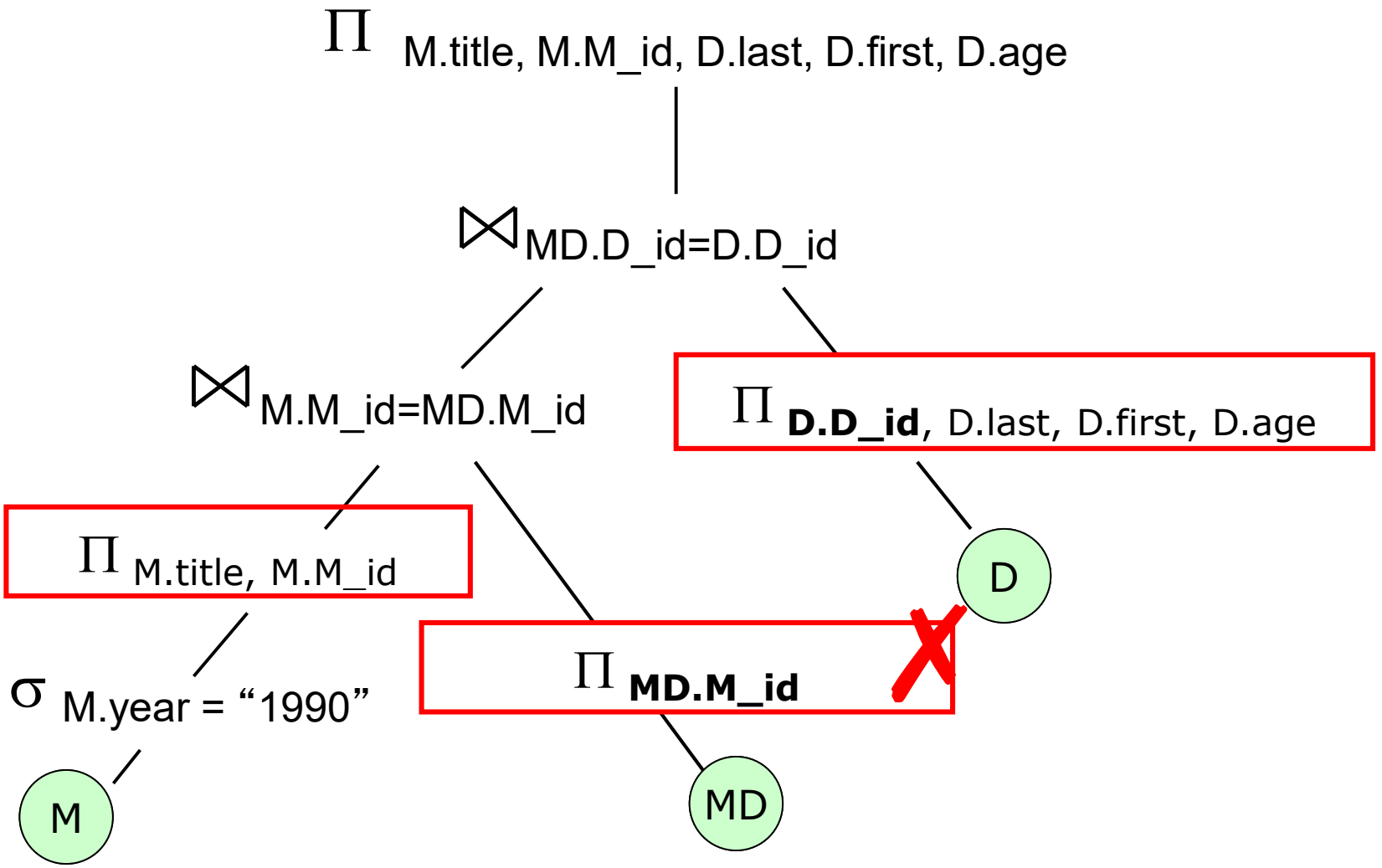
Example of Query Tree (Step 5)



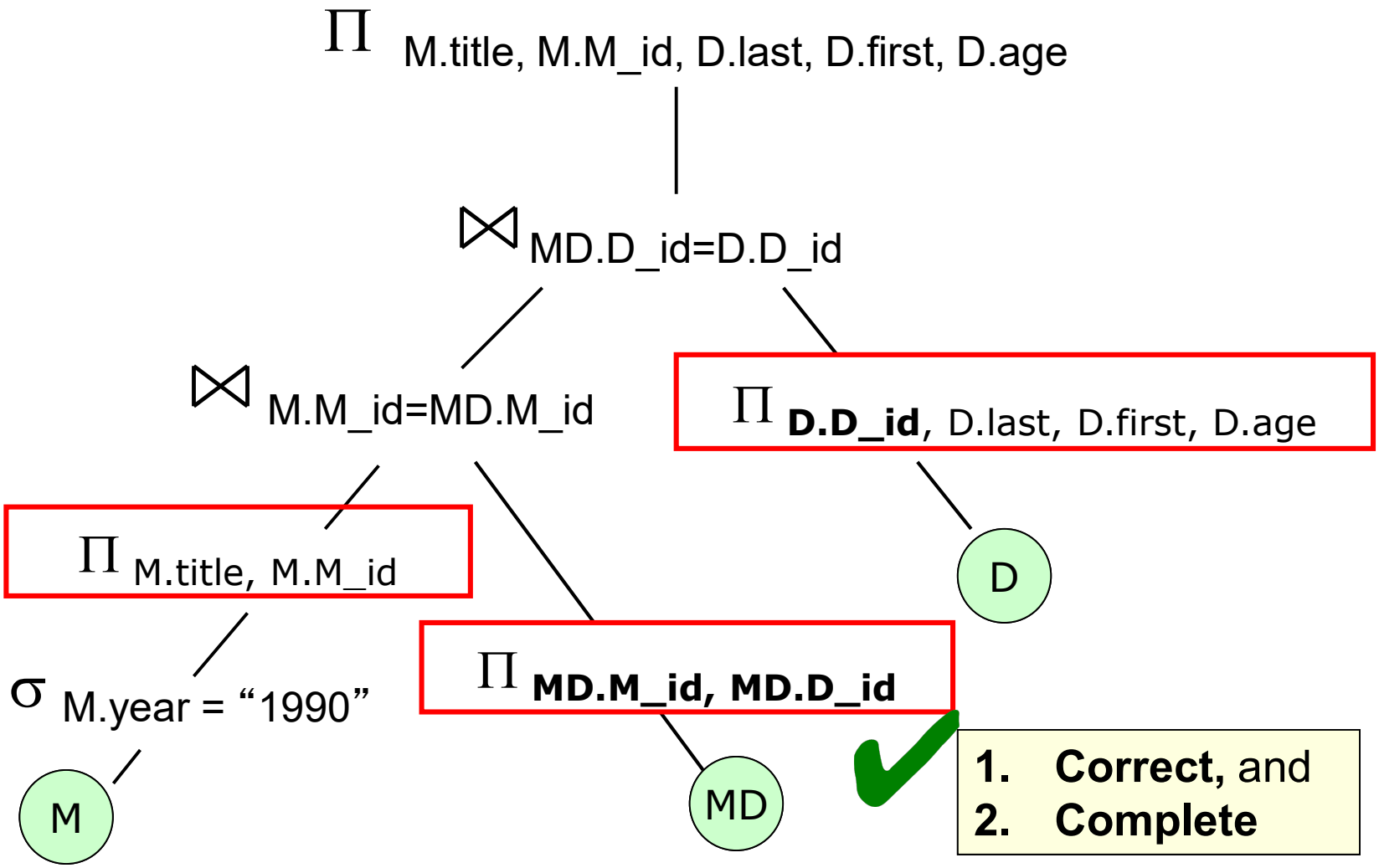
Example of Query Tree (Step 5)



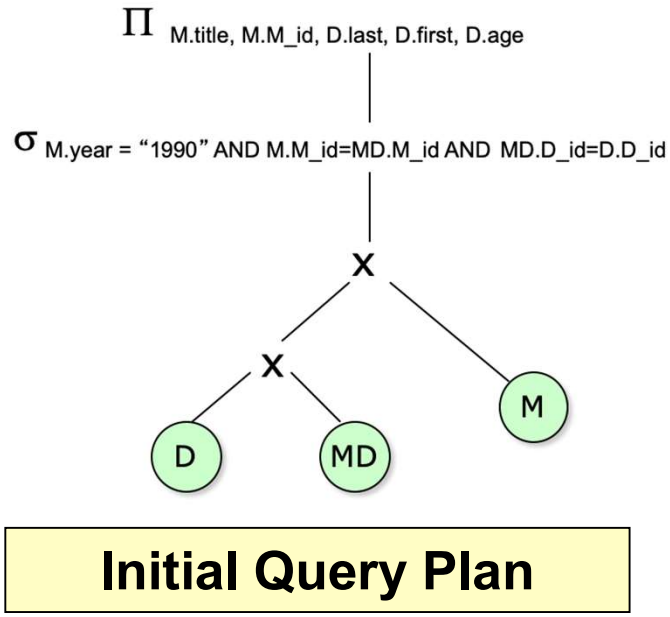
Example of Query Tree (Step 5)



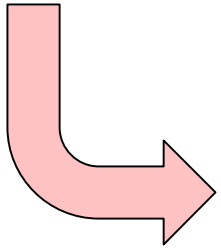
Example of Query Tree (Step 5)



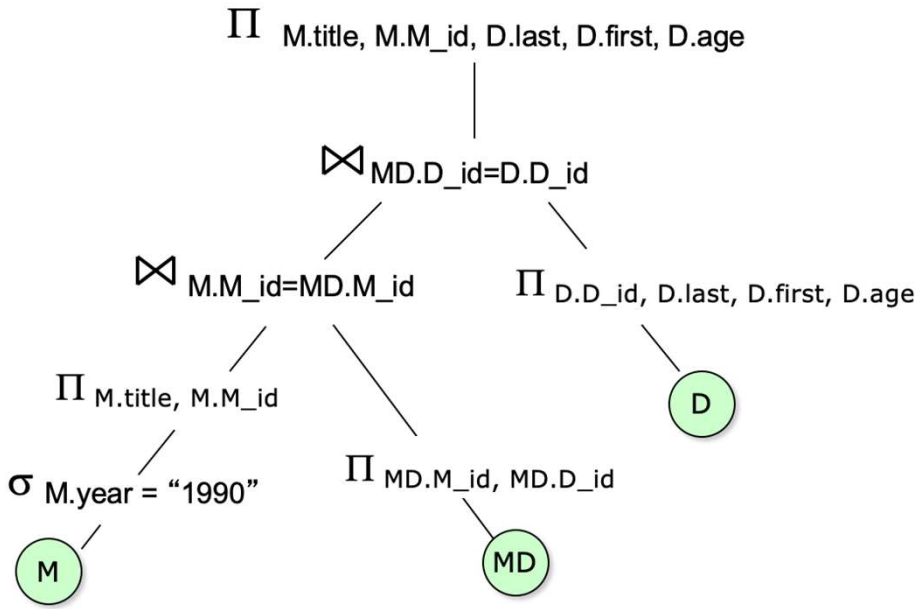
Transform into Equivalent Query Tree



Initial Query Plan



Transform



Equivalent Query Plan

Heuristics for Query Tree Optimization

- The main idea is to apply first the operations that reduce the size of **intermediate** results

  1. Perform **select** operations as early as possible to reduce the number of tuples, and

  2. Perform **project** operations as early as possible to reduce the number of attributes

     - This is done by **moving** select and project operations as far down the tree as possible

  3. The select and join operations that are most **restrictive** should be executed before other similar operations

     - This is done by **rearranging** leaf nodes and adjusting the rest of the tree appropriately

Cost-based Query Optimization (vs. Heuristic)

- **Cost-based query optimization**:
  1. **Estimate** and compare the costs of executing a query using different execution strategies, and
  2. **Choose** the strategy with the lowest cost estimate
     - Compare to heuristic query optimization

- Issues
  - Cost function
  - Number of execution strategies to be considered

Cost-based Query Optimization

- **Cost Components** for Query Execution
  1. Access cost to secondary storage
  2. Storage cost
  3. Computation cost
  4. Memory usage cost
  5. Communication cost

- Different database systems may **focus** on different cost components