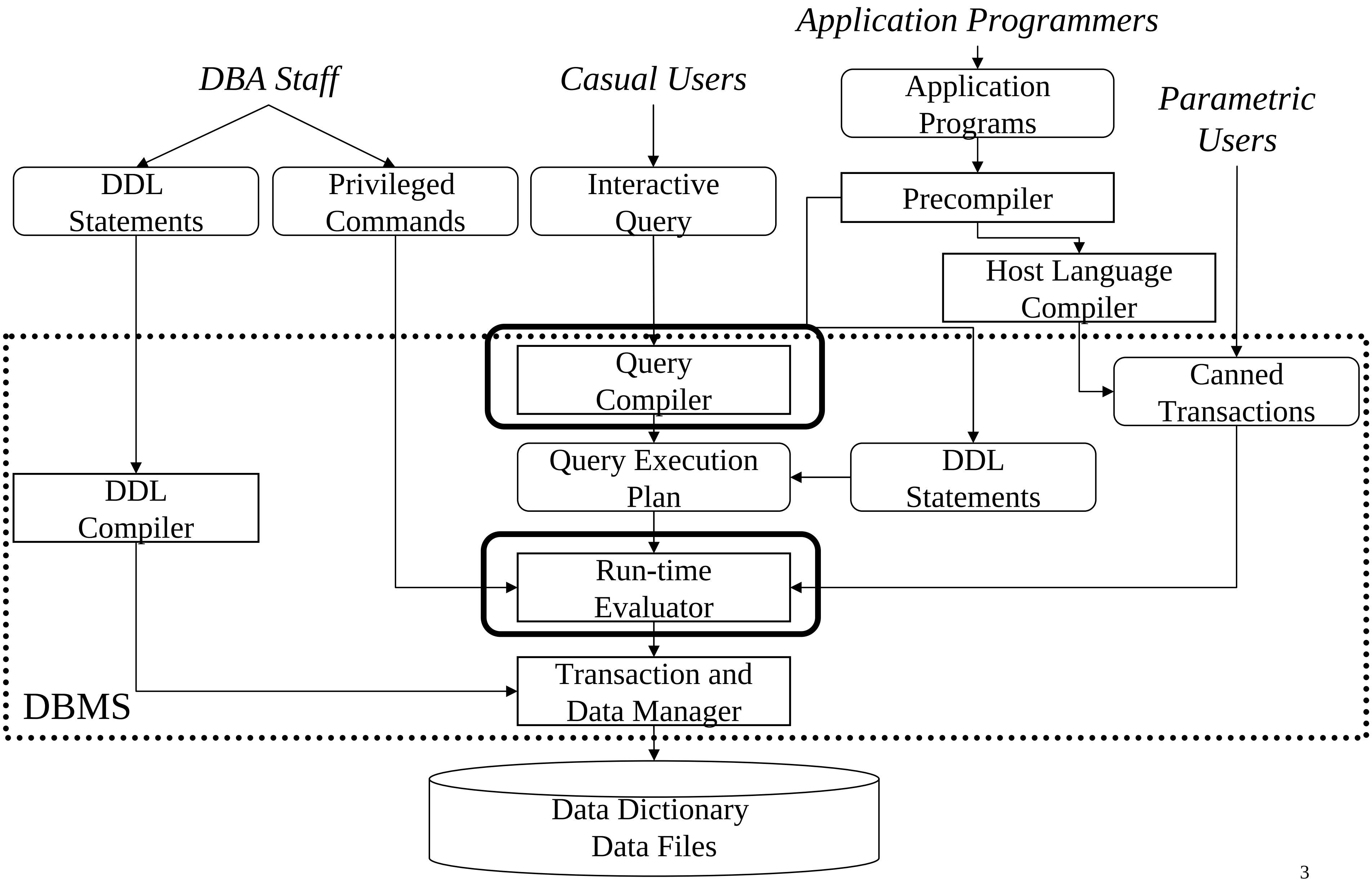
Query Processing + Optimization: Outline

- Operator Evaluation Strategies
  - Query processing in general
  - Selection
  - Join
- Query Optimization
  - Heuristic query optimization
  - Cost-based query optimization
- Query Tuning

Query Processing + Optimization

- Operator Evaluation Strategies
  - Selection
  - Join
- Query Optimization
- Query Tuning

Architectural Context

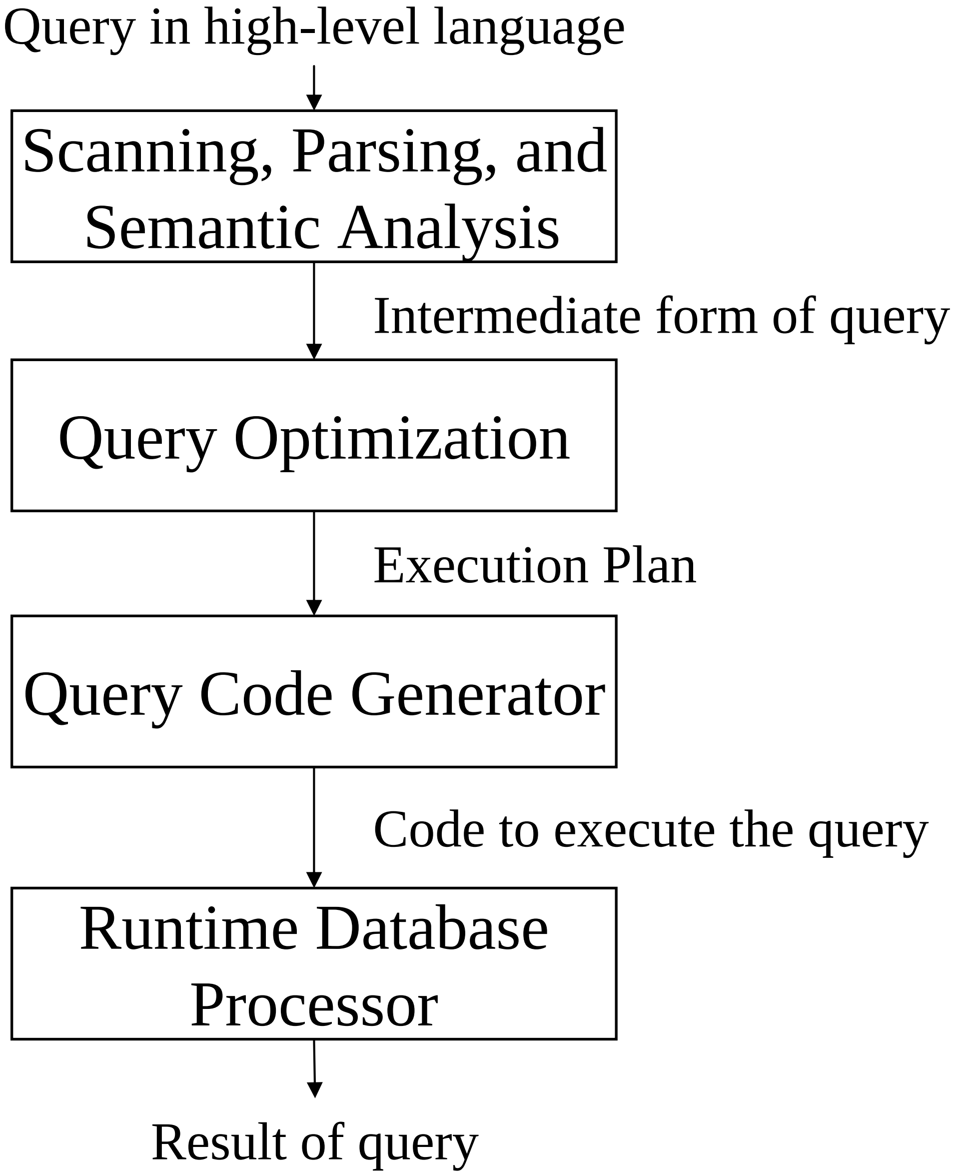


Evaluation of SQL Statement

- The query is evaluated in a different order.
  - The tables in the from clause are combined using Cartesian products.
  - The where predicate is then applied.
  - The resulting tuples are grouped according to the group by clause.
  - The having predicate is applied to each group, possibly eliminating some groups.
  - The aggregates are applied to each remaining group.
    The select clause is performed last.

Overview of Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation
4. Execution



Query in high-level language

Scanning, Parsing, and Semantic Analysis

Intermediate form of query

Query Optimization

Execution Plan

Query Code Generator

Code to execute the query

Runtime Database Processor

Result of query

Selection Queries

- Primary key, point

$$\sigma_{FilmID = 2}\,(Film)$$

- Point

$$\sigma_{Title = \text{`Terminator'}}\,(Film)$$

- Range

$$\sigma_{1 < RentalPrice < 4}\,(Film)$$

- Conjunction

$$\sigma_{Type = \text{`M'} \wedge Distributor = \text{`MGM'}}\,(Film)$$

- Disjunction

$$\sigma_{PubDate < 2004 \vee Distributor = \text{`MGM'}}\,(Film)$$

Selection Strategies

- Linear search
  - Expensive, but always applicable.
- Binary search
  - Applicable only when the file is appropriately ordered.
- Hash index search
  - Single record retrieval; does not work for range queries.
  - Retrieval of multiple records.
- Clustering index search
  - Multiple records for each index item.
  - Implemented with single pointer to block with first associated record.
- Secondary index search
  - Implemented with dense pointers, each to a single record.

Selection Strategies for Conjunctive Queries

- Use any available indices for attributes involved in simple conditions.
  - If several are available, use the most selective index. Then check each record with respect to the remaining conditions.
- Attempt to use composite indices.
  - This can be very efficient.
- Do intersection of record pointers.
  - If several indices with record pointers are applicable to the selection predicate, retrieve and intersect the pointers. Then retrieve (and check) the qualifying records.
- Disjunctive queries provide little opportunity for smart processing.

Joins

- Join Strategies
  - Nested loop join
  - Index-based join
  - Sort-merge join
  - Hash join

- Strategies work on a per block (not per record) basis.
  - Need to estimate #I/Os (block retrievals)

- Relation sizes and *join selectivities* impact join cost.
  - Query selectivity = #tuples in result / #candidates
    - ‘More selective’ means smaller ‘selectivity value’
  - For join, #candidates is the size of Cartesian product

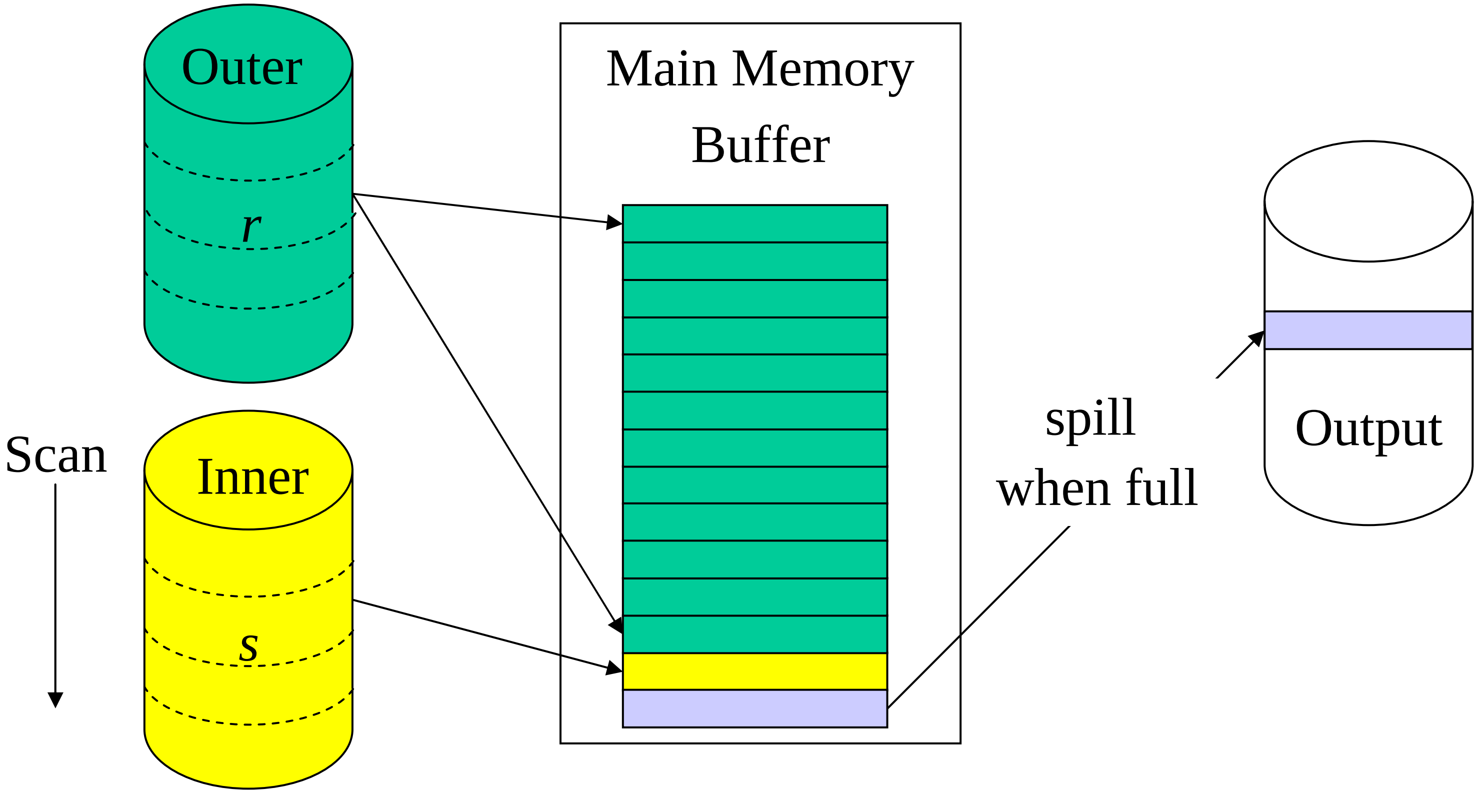
Nested Loop Join and Index-Based Join

- Nested loop join
  - Exhaustive comparison (i.e., brute force approach)
  - The ordering (outer/inner) of files and allocation of buffer space is important.

- Index-based join
  - Requires (at least) one index on a join attribute.
  - At times, a temporary index is created for the purpose of a join.
  - The ordering (outer/inner) of files is important.

Nested Loop

- Basically, for each block of the outer table (*r*), scan the entire inner table (*s*).
  - Requires quadratic time, $O(n^2)$
  - Improved when buffer is used.



Example of Nested-Loop Join

$$Customer \bowtie_{C.CustomerID\ =\ CO.EmpId} CheckedOut$$

- Parameters

  $r_{CheckedOut} = 40.000$ $r_{Customer} = 200$
  $b_{CheckedOut} = 2.000$ $b_{Customer} = 10$
  $n_B = 6$ (size of main memory buffer)

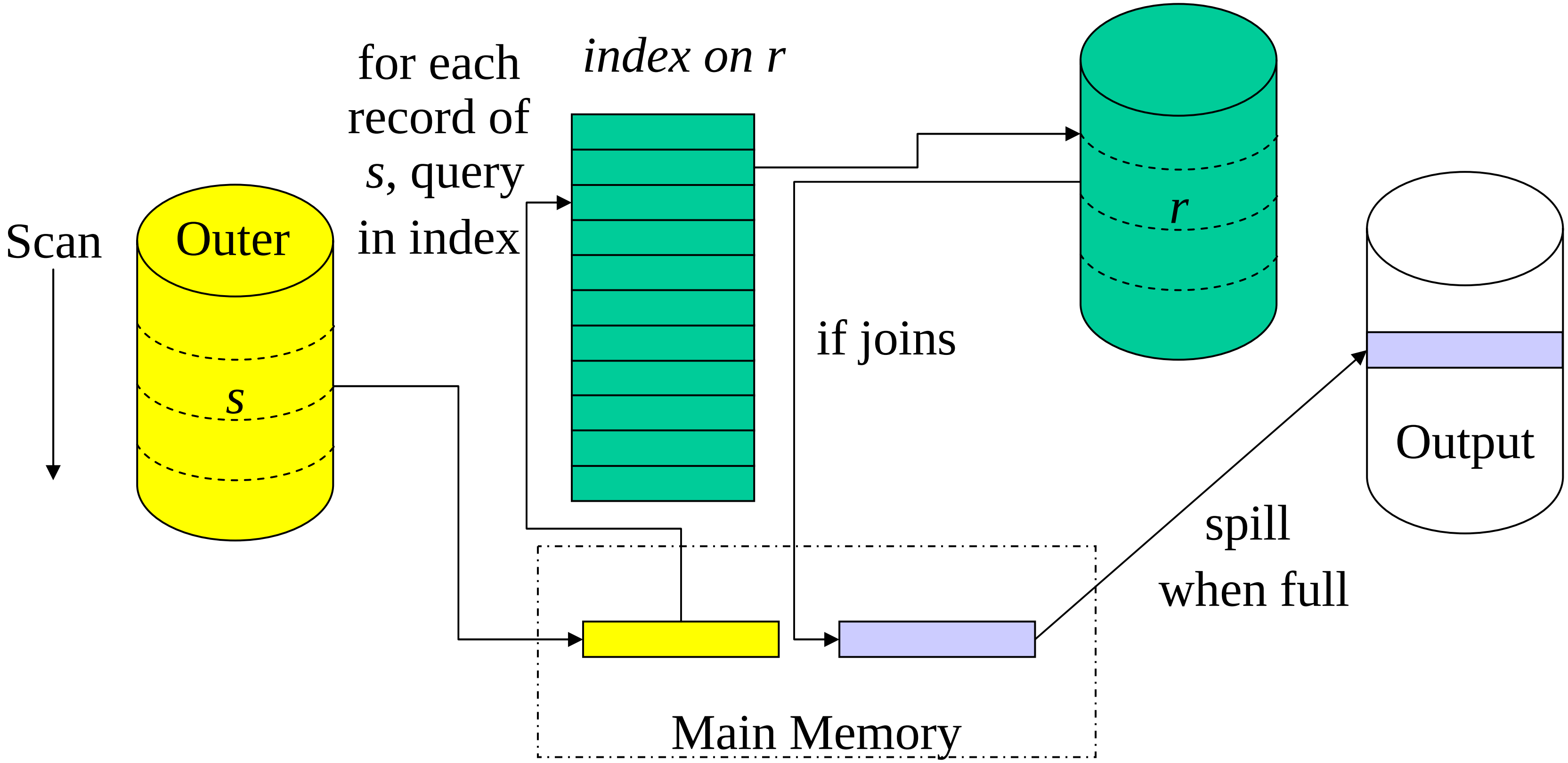
- Algorithm:

```
repeat: read ($n_B$ - 2) blocks from outer relation
    repeat: read 1 block from inner relation
        compare tuples
```

- Cost: $b_{outer} + (\lceil b_{outer} / (n_B - 2) \rceil) \times b_{inner}$
- *CheckedOut* as outer: $2.000 + \lceil 2.000/4 \rceil \times 10 = 7.000$
- *Customer* as outer: $10 + \lceil 10/4 \rceil \times 2.000 = 6.010$

Index-based Join

- Requires (at least) one index on a join attribute
  - A temporary index can be created



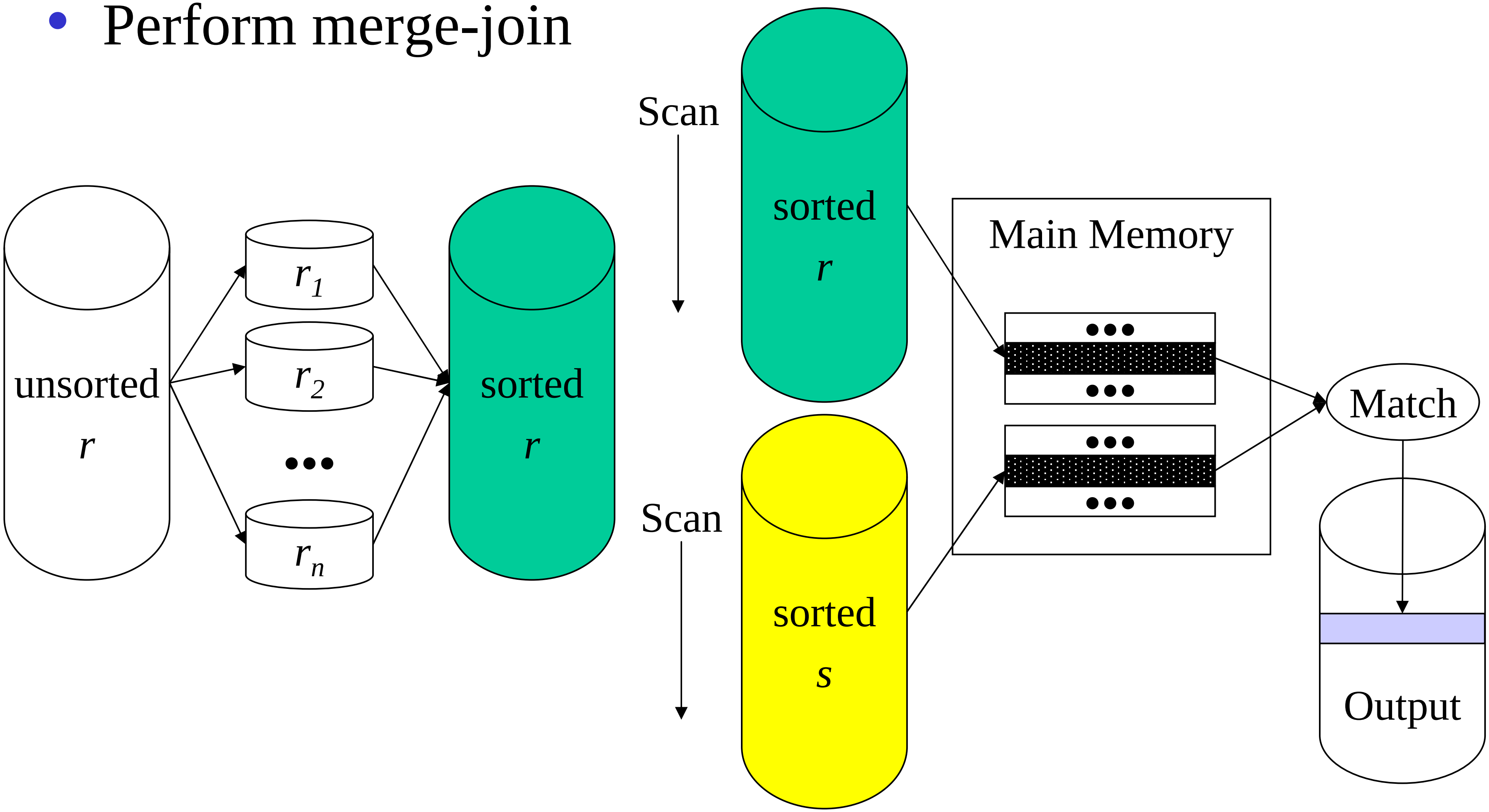
Example of Index-Based Join

$$\mathit{Customer} \bowtie_{C.CustomerID\, =\, CO.EmpId} \mathit{CheckedOut}$$

- Cost: $b_{outer} + r_{outer} \times$ cost use of index
- Assume that the video store has 10 employees.
  - There are 10 distinct *EmpID*s in *CheckedOut*.
- Assume 1-level index on *CustomerID* of *Customer*.
- Iterate through all 40.000 tuples in *CheckedOut* (outer rel.)
  - 2.000 disk reads ($b_{CheckedOut}$) to scan *CheckedOut*
  - For each *CheckedOut* tuple, search for matching *Customer* tuples using index.
    - 0 disk reads for index (in main memory) + 1 disk read for actual data block
- Cost: 2.000 + 40.000 × (0 + 1) = 42.000

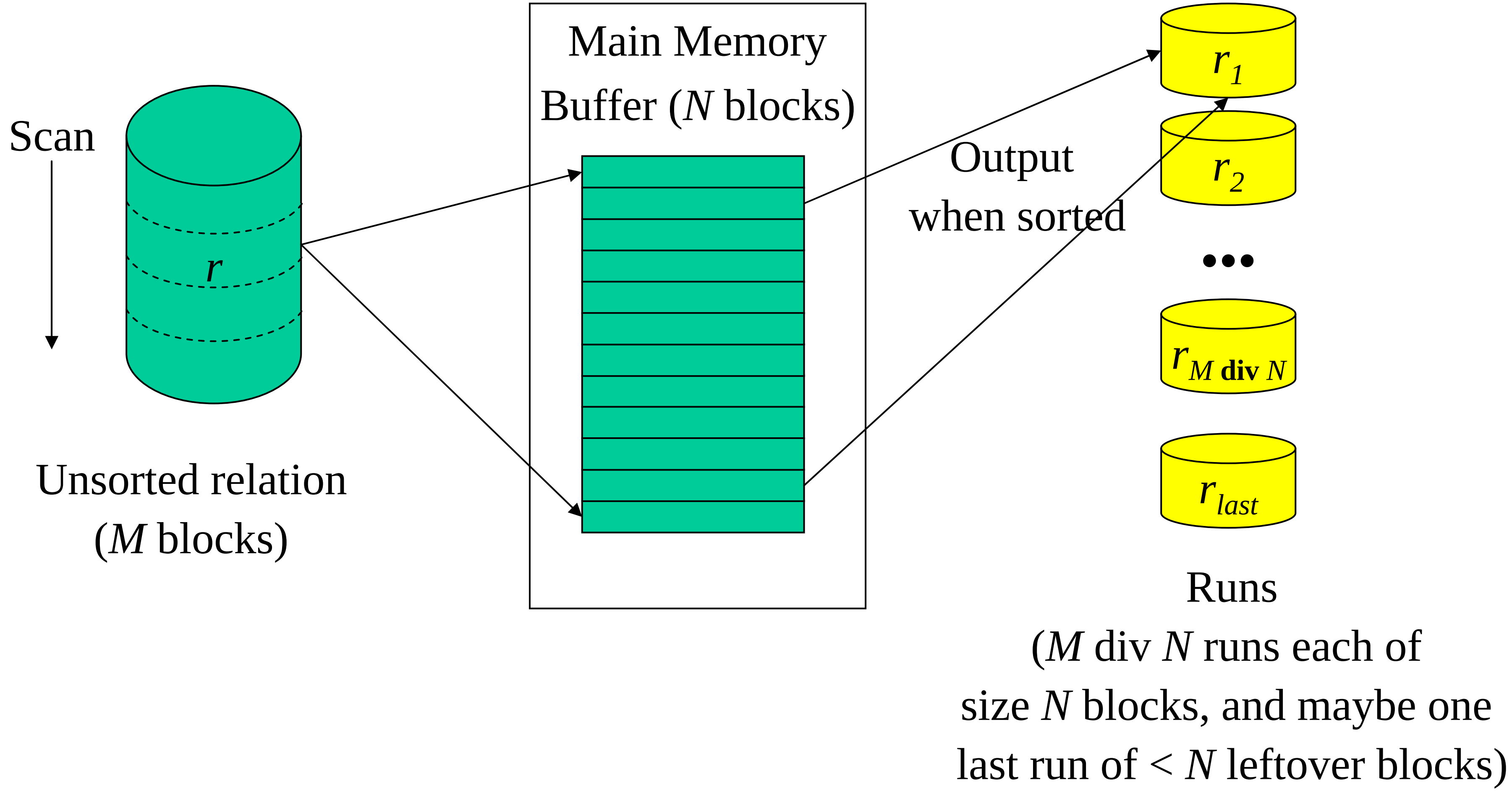
Sort-Merge Join

- Sort each relation using multiway merge-sort
- Perform merge-join



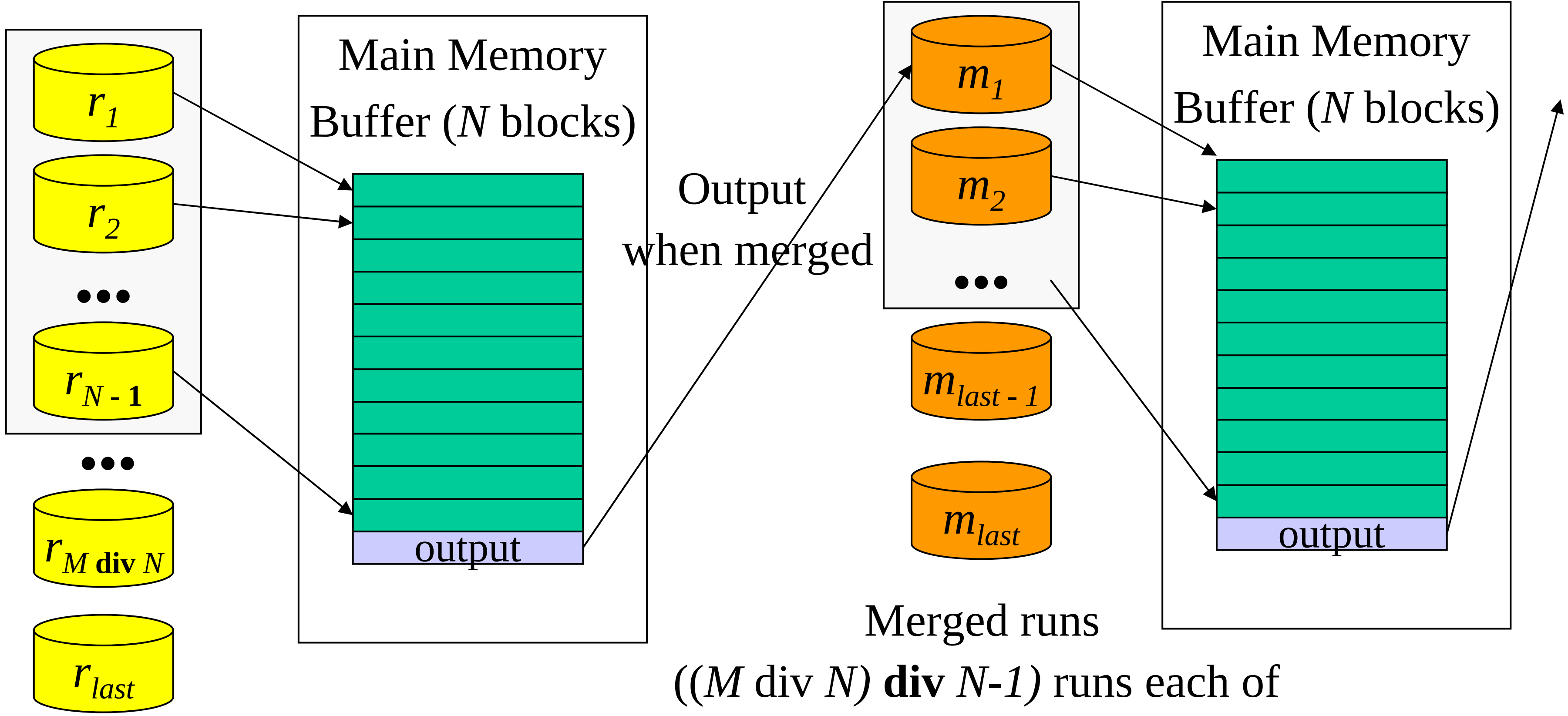
External or Disk-based Sorting

- Relation on disk often too large to fit into memory
- Sort in pieces, called *runs*



External or Disk-based Sorting, Cont.

- Runs are now repeatedly merged
- One memory buffer used to collect output



r_1

r_2

•••

$r_{N\text{ - }\mathbf{1}}$

•••

$r_{M\ \mathbf{div}\ N}$

r_{last}

sorted runs
(*N* blocks each)

Main Memory
Buffer (*N* blocks)

output

Output
when merged

m_1

m_2

•••

$m_{last\text{ - }1}$

m_{last}

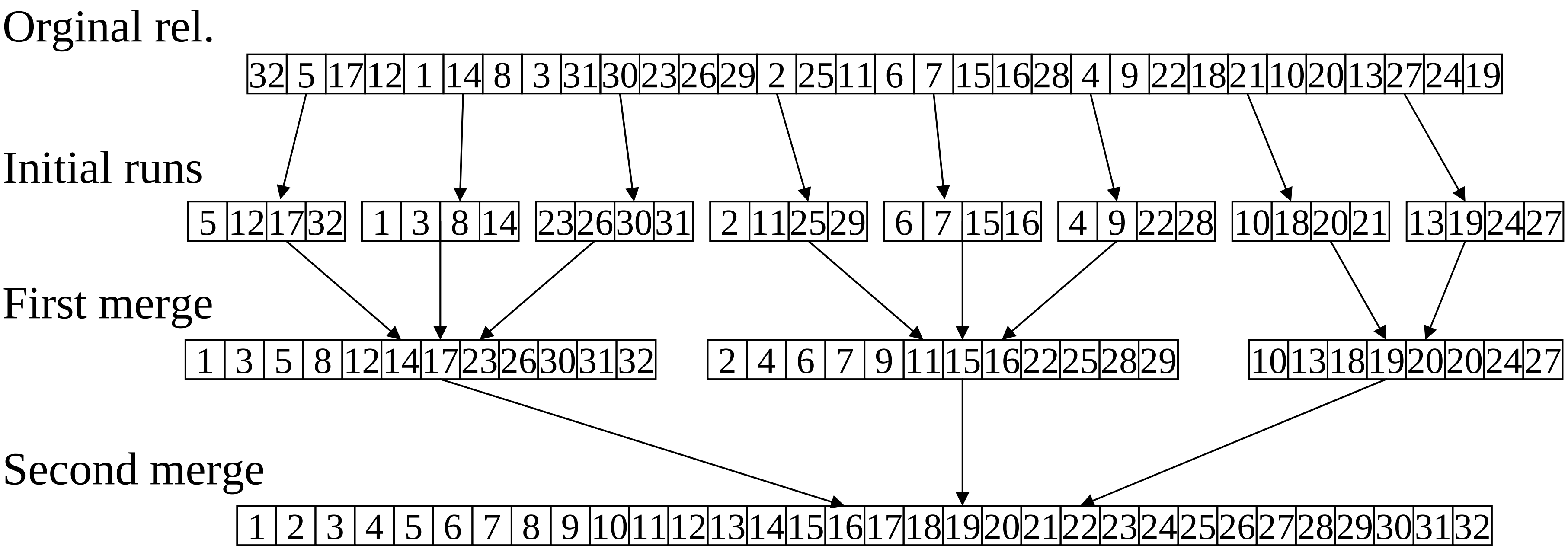
Main Memory
Buffer (*N* blocks)

output

Merged runs
((*M* div *N*) **div** *N-1*) runs each of
size *N*N-1* blocks, and maybe one
last run of < *N*N-1* leftover blocks)

External Sorting (Multiway Merge Sort)

- Buffer size is $n_B = 4$ (N)



Orginal rel.

32 5 17 12 1 14 8 3 31 30 23 26 29 2 25 11 6 7 15 16 28 4 9 22 18 21 10 20 13 27 24 19

Initial runs

5 12 17 32 | 1 3 8 14 | 23 26 30 31 | 2 11 25 29 | 6 7 15 16 | 4 9 22 28 | 10 18 20 21 | 13 19 24 27

First merge

1 3 5 8 12 14 17 23 26 30 31 32 | 2 4 6 7 9 11 15 16 22 25 28 29 | 10 13 18 19 20 20 24 27

Second merge

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32

- Cost: $2 \times b_{relation} + 2 \times b_{relation} \times \lceil \log_{n_B - 1} (b_{relation}/n_B) \rceil$
- $2 \times 32 + 2 \times 32 \times \log_3(32/4) = 192$

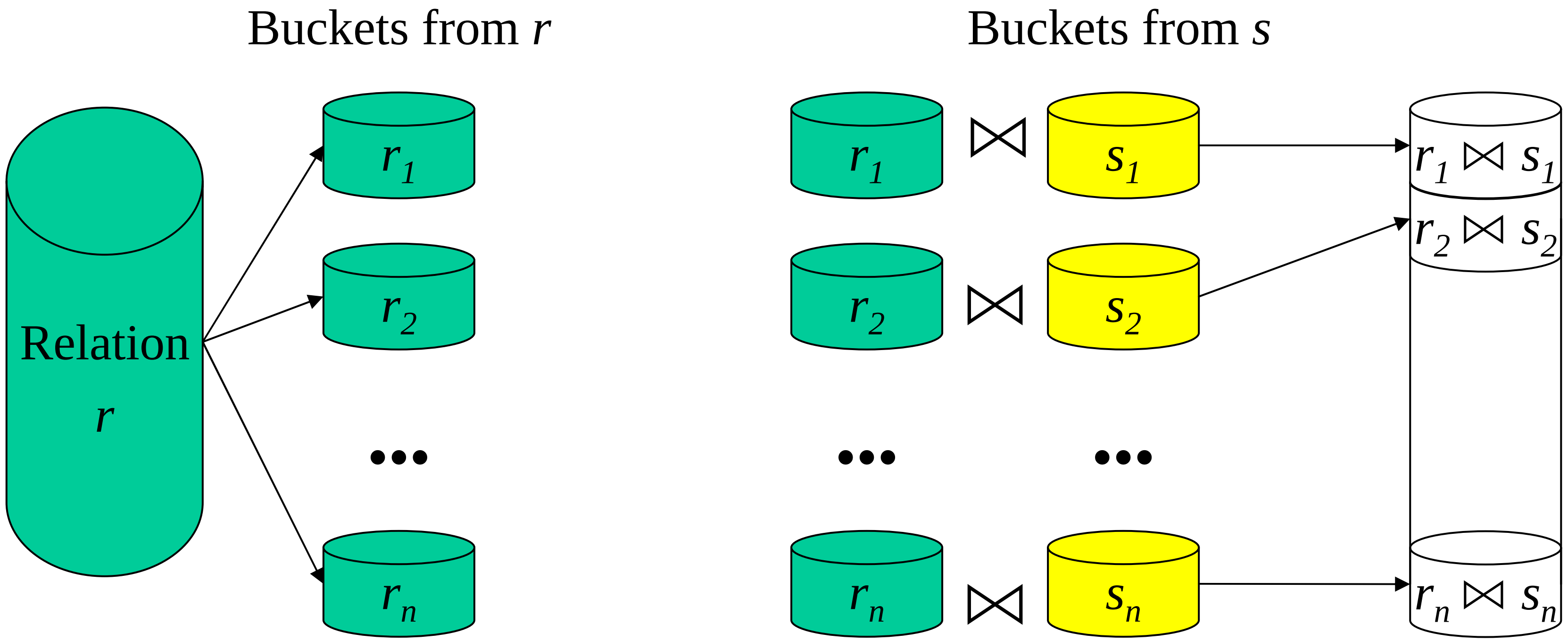
Example of Sort-Merge Join

- Cost to sort *CheckedOut* ($b_{CheckedOut}$= 10)
  - $\text{Cost}_{\text{Sort } CheckedOut} = 2 \times 2.000 + 2 \times 2.000 \times \lceil \log_5(2.000/6) \rceil = 20.000$
- Cost to sort *Customer* relation ($b_{Customer}$= 10)
  - $\text{Cost}_{\text{Sort } Customer} = 2 \times 10 + 2 \times 10 \times \lceil \log_5(10/6) \rceil = 40$
- Cost for merge join
  - Cost to scan sorted *Customer* + cost to scan sorted *CheckedOut*
  - $\text{Cost}_{\text{merge join}} = 10 + 2.000 = 2.010$

- $\text{Cost}_{\text{sort-merge join}} = \text{Cost}_{\text{Sort } Customer} + \text{Cost}_{\text{Sort } CheckedOut} + \text{Cost}_{\text{merge join}}$
- $\text{Cost}_{\text{sort-merge join}} = 20.000 + 40 + 2.010 = 22.050$

Hash Join

- Hash each relation on the join attributes
- Join corresponding buckets from each relation

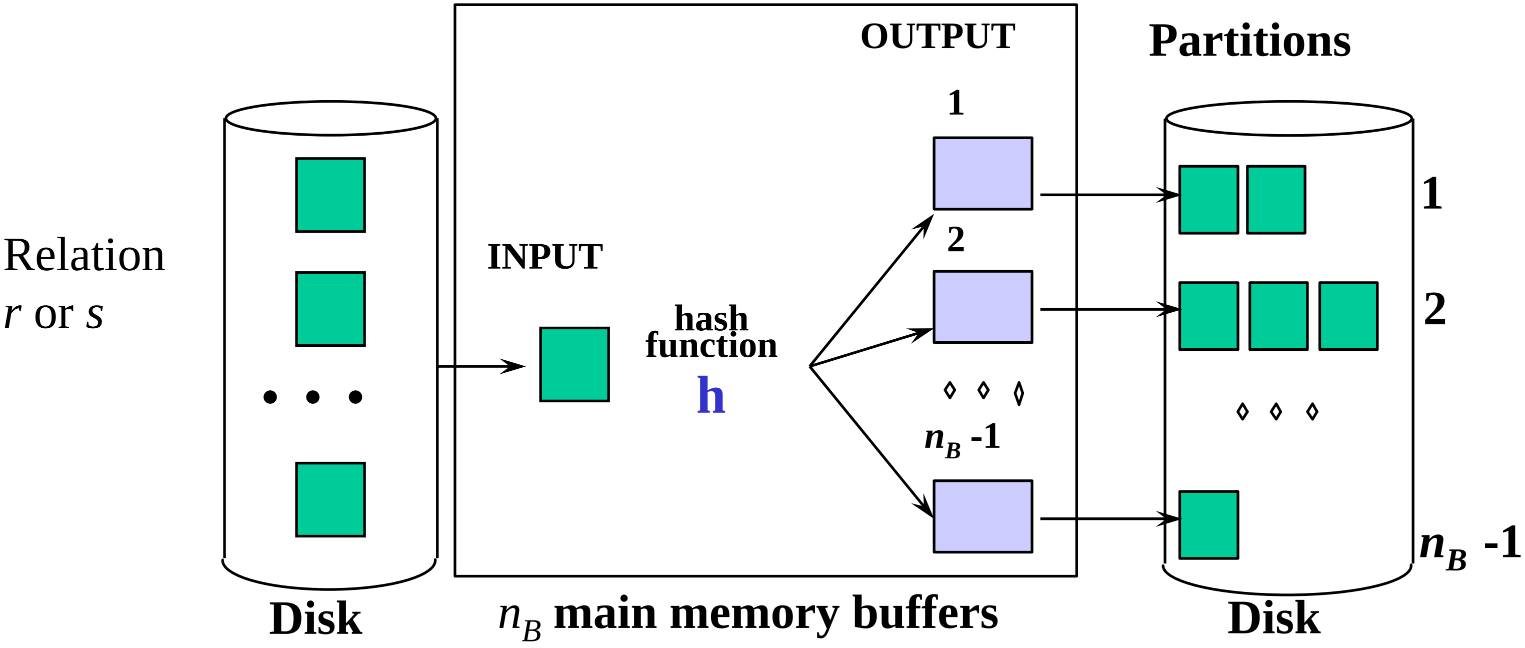


Hash *r* (same for s)

Join corresponding *r* and s buckets

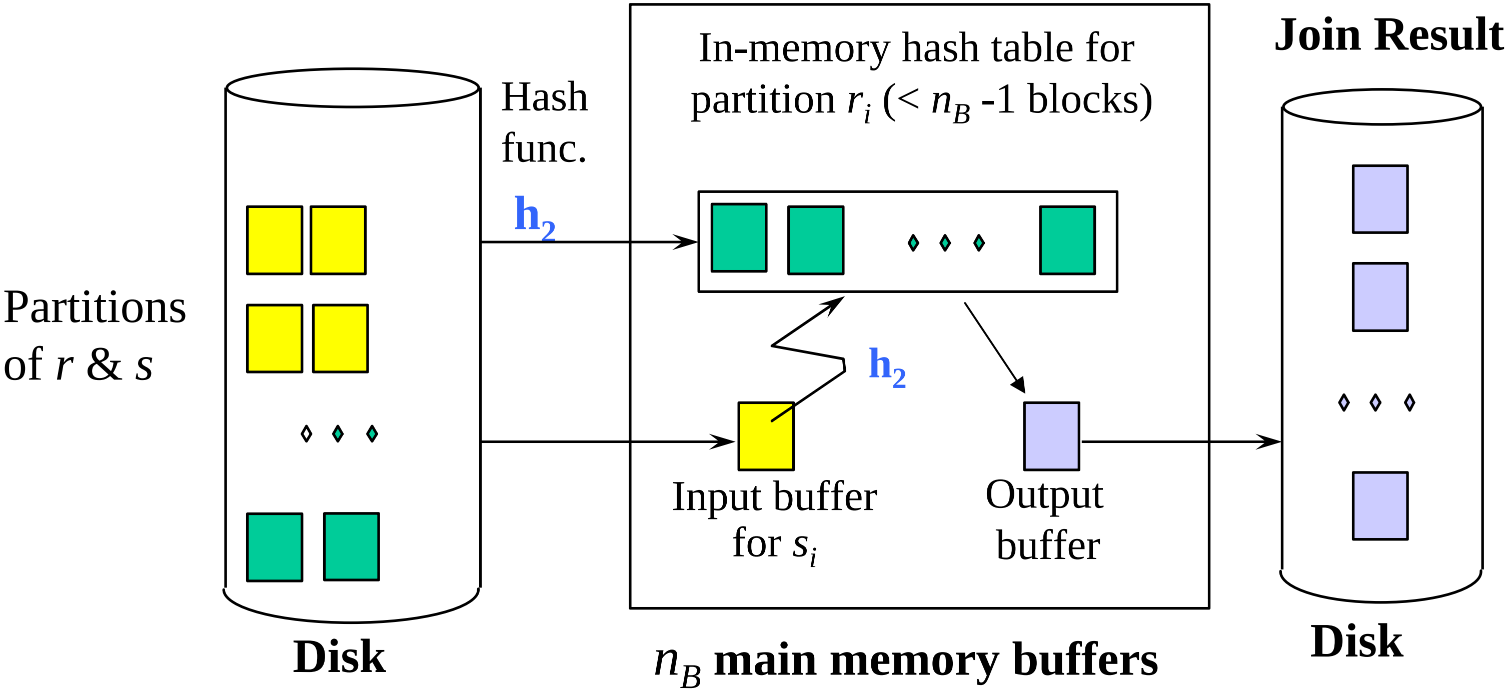
Partitioning Phase

- *Partitioning phase*: r divided into n_h partitions. The number of buffer blocks is n_B. One block used for reading r. ($n_h = n_B - 1$)
  - Similar with relation s
  - I/O cost: $2 \times (b_r + b_s)$



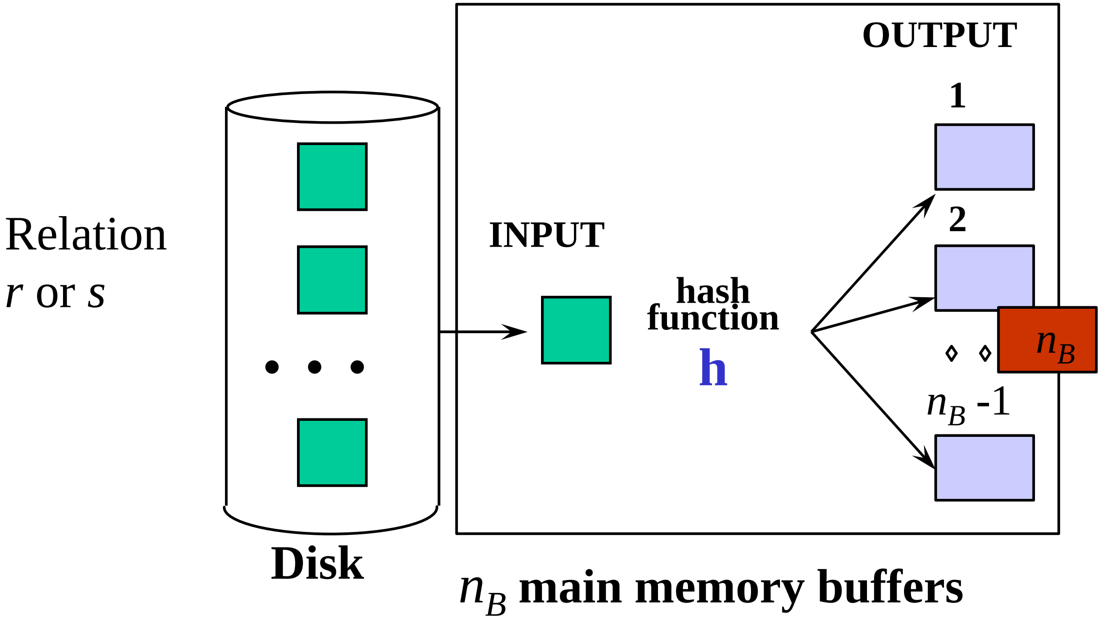
Joining Phase

- *Joining (or probing) phase*: n_h iterations where $r_i \bowtie s_i$.
  - Load r_i into memory and build an in-memory hash index on it using the join attribute. (h_2 needed, r_i called *build input*)
  - Load s_i, for each tuple in it, join it with r_i using h_2. (s_i called *probe input*)
  - I/O cost: $b_r + b_s + 4 \times n_h$ (each partition may have a partially filled block)
    - One write and one read for each partially filled block



Hash Join Cost

- $\text{Cost}_{\text{Total}} = \text{Cost}_{\text{Partitioning}} + \text{Cost}_{\text{Joining}}$
  $= 3 \times (b_r + b_s) + 2 \times n_h$
- Cost = 3 × (2000 + 10) + 2 × 5 = 6040
- Any problem not considered?
  - What if $n_h > n_B - 1$? I.e., more partitions than available buffer blocks!
  - How to solve it?



Recursive Partitioning

- Required if number of partitions n_h is greater than number of available buffer blocks n_B -1.
  - instead of partitioning n_h ways, use $n_B – 1$ partitions for s
  - Further partition the $n_B – 1$ partitions using a different hash function
  - Use same partitioning method on r
  - Rarely required: e.g., recursive partitioning not needed for relations of 1GB or less with memory size of 2MB, with block size of 4KB.
- $\text{Cost}_{\text{hjrp}} = 2 \times (b_r + b_s) \times \lceil \log_{n_B - 1} (b_s) - 1 \rceil + b_r + b_s$
- $\text{Cost}_{\text{hjrp}} = 2 \times (2000 + 10) \times \lceil \log_5 (10) - 1 \rceil + 2000 + 10$
  $= 6030$

Cost and Applicability of Join Strategies

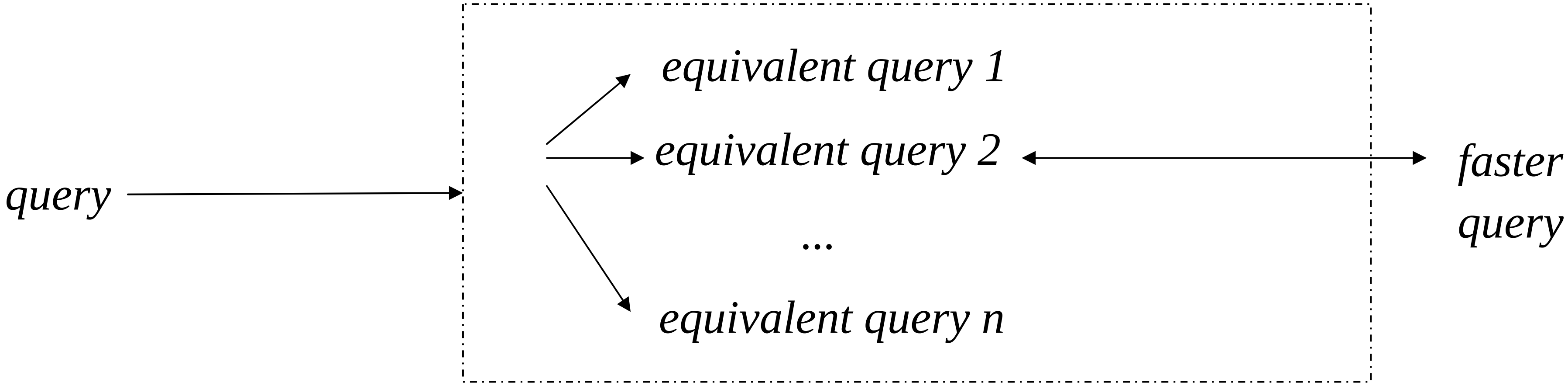
- Nested-loop join
  - Brute-force
  - Can handle all types of joins (=, <, >)
- Index-based join
  - Requires minimum one index on join attributes
- Sort-merge join
  - Requires that the files are sorted on the join attributes.
  - Sorting can be done for the purpose of the join.
  - A variation is also applicable when secondary indices are available instead.
- Hash join
  - Requires good hashing functions to be available.
  - Performance best if smallest relation fits in memory. ?

Query Processing + Optimization

- Operator Evaluation Strategies
- Query Optimization
  - Heuristic Query Optimization
  - Cost-based Query Optimization
- Query Tuning

Query Optimization

- Aim: Transform query into faster, equivalent query



query

equivalent query 1

equivalent query 2

…

equivalent query n

faster query

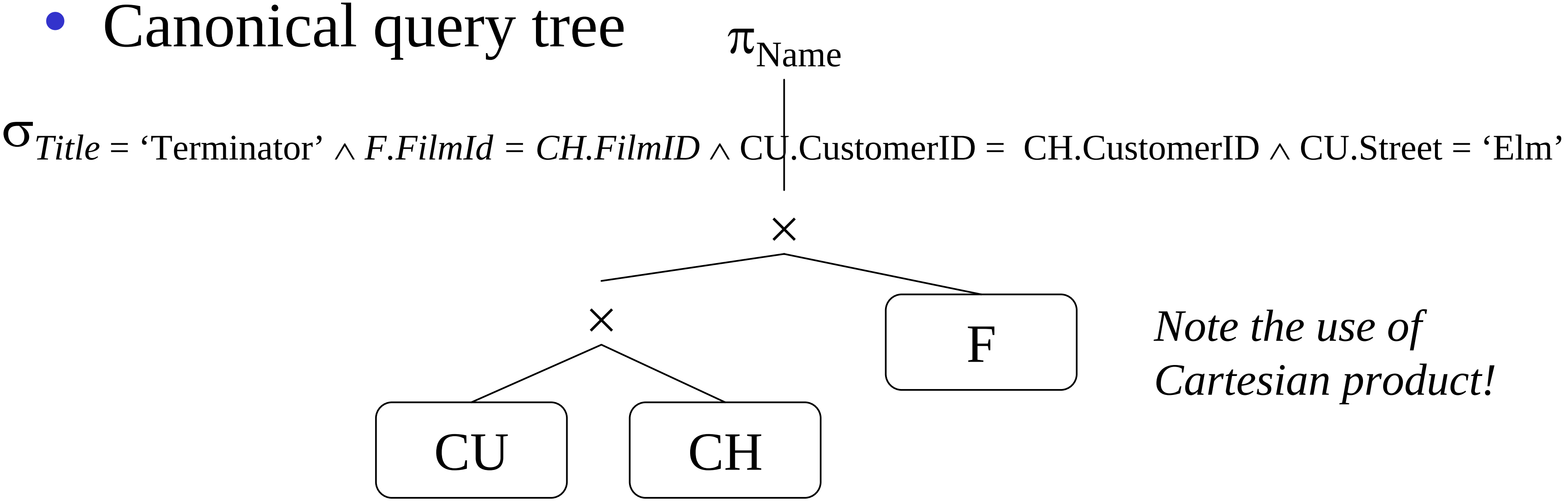
- Heuristic (logical) optimization
  - Query tree (relational algebra) optimization
  - Query graph optimization
- Cost-based (physical) optimization

Query Tree Optimization Example

- What are the names of customers living on Elm Street who have checked out “Terminator”?
- SQL query:

```
SELECT Name
FROM   Customer CU, CheckedOut CH, Film F
WHERE  Title = ’Terminator’ AND F.FilmId = CH.FilmID
AND CU.CustomerID = CH.CustomerID AND CU.Street = ‘Elm’
```

- Canonical query tree



π_{Name}

$\sigma_{Title = \text{‘Terminator’} \wedge F.FilmId = CH.FilmID \wedge \text{CU.CustomerID} = \text{CH.CustomerID} \wedge \text{CU.Street} = \text{‘Elm’}}$

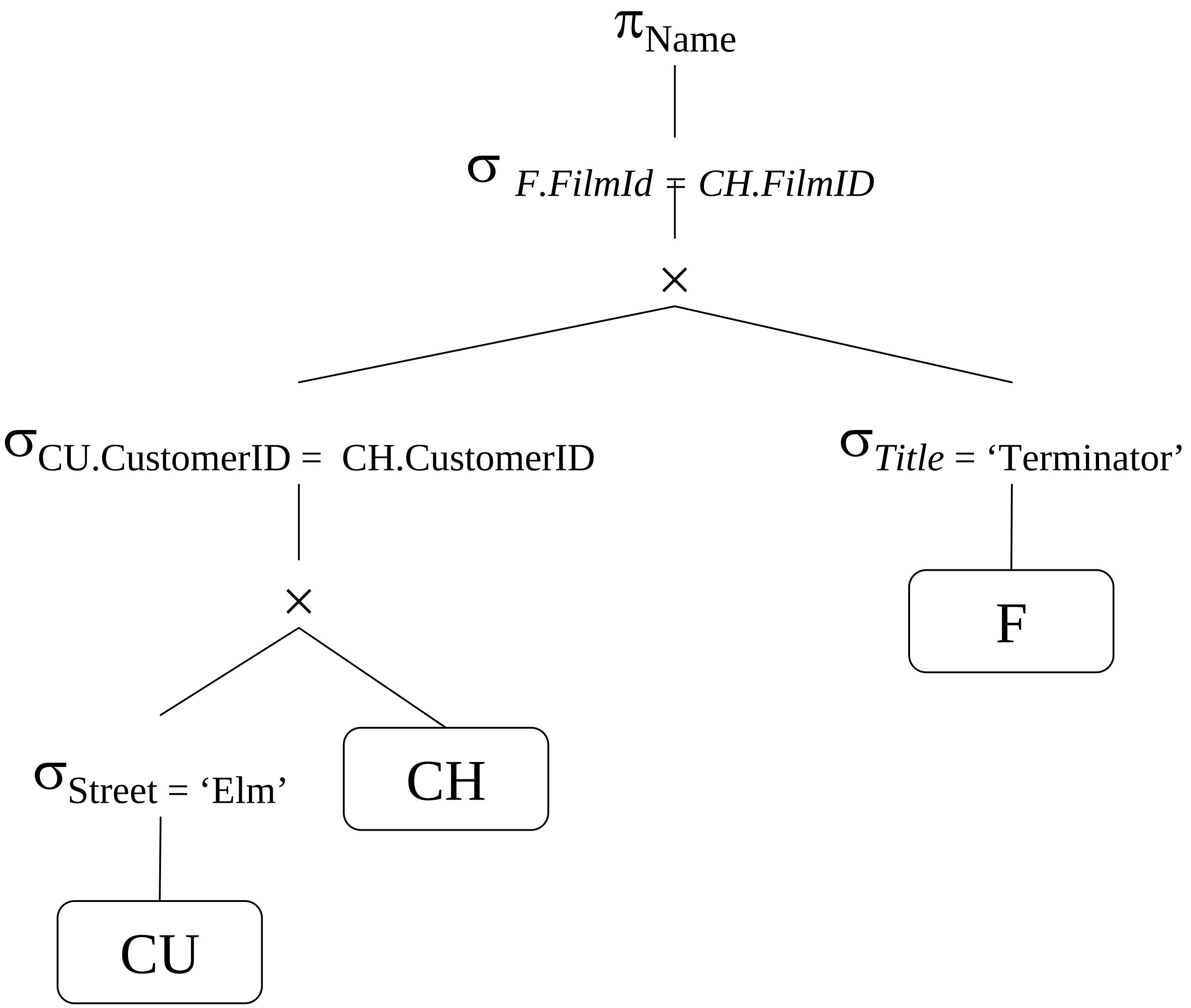
×

× F

CU CH

Note the use of Cartesian product!

Apply Selections Early



π_{Name}

$\sigma_{F.FilmId\ =\ CH.FilmID}$

×

$\sigma_{\text{CU.CustomerID = CH.CustomerID}}$

$\sigma_{Title\ =\ \text{‘Terminator’}}$

×

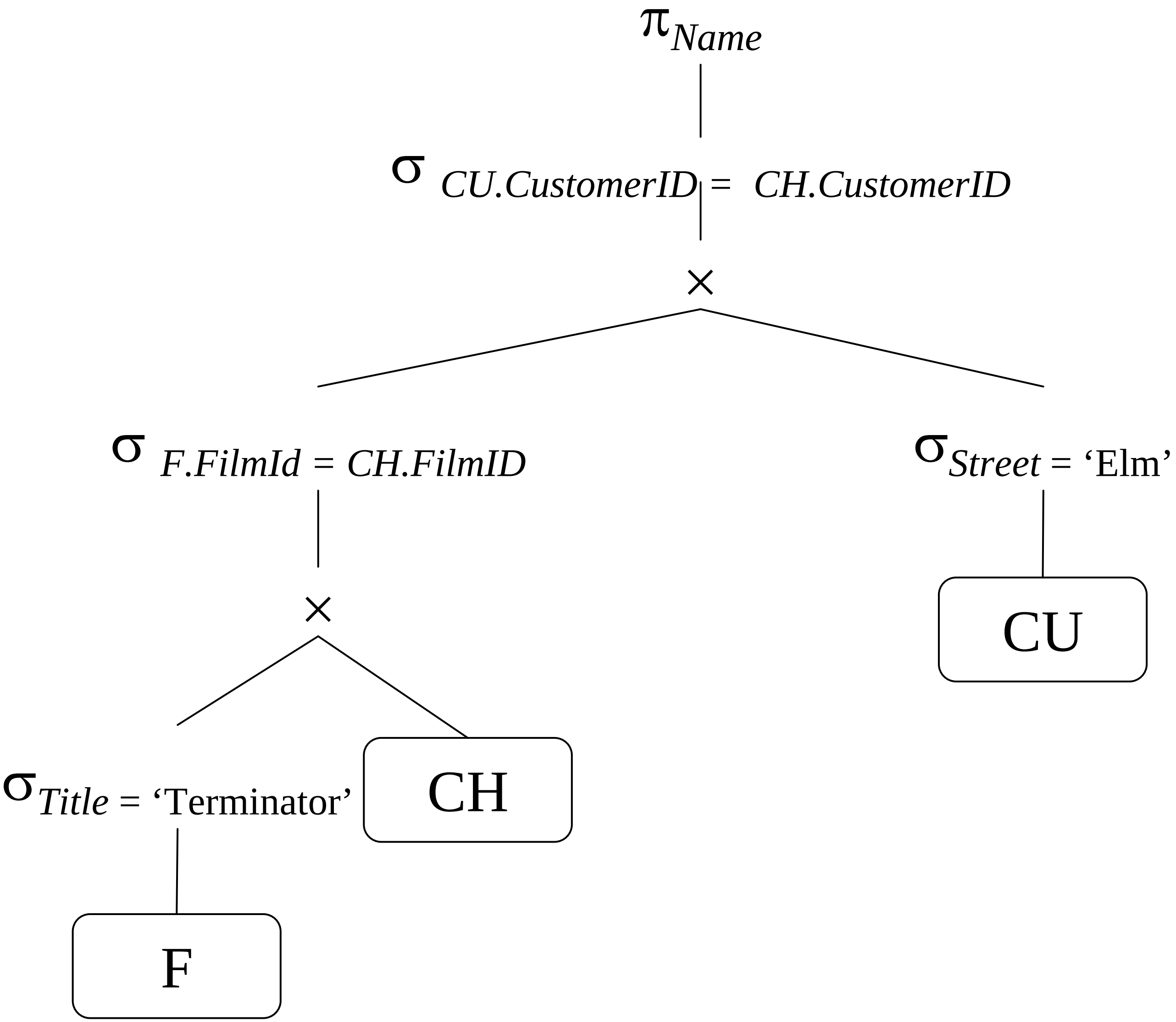
F

$\sigma_{\text{Street = ‘Elm’}}$

CH

CU

Apply More Restrictive Selections Early



π_{Name}

$\sigma_{CU.CustomerID = CH.CustomerID}$

$\times$

$\sigma_{F.FilmId = CH.FilmID}$

$\sigma_{Street = \text{'Elm'}}$

$\times$

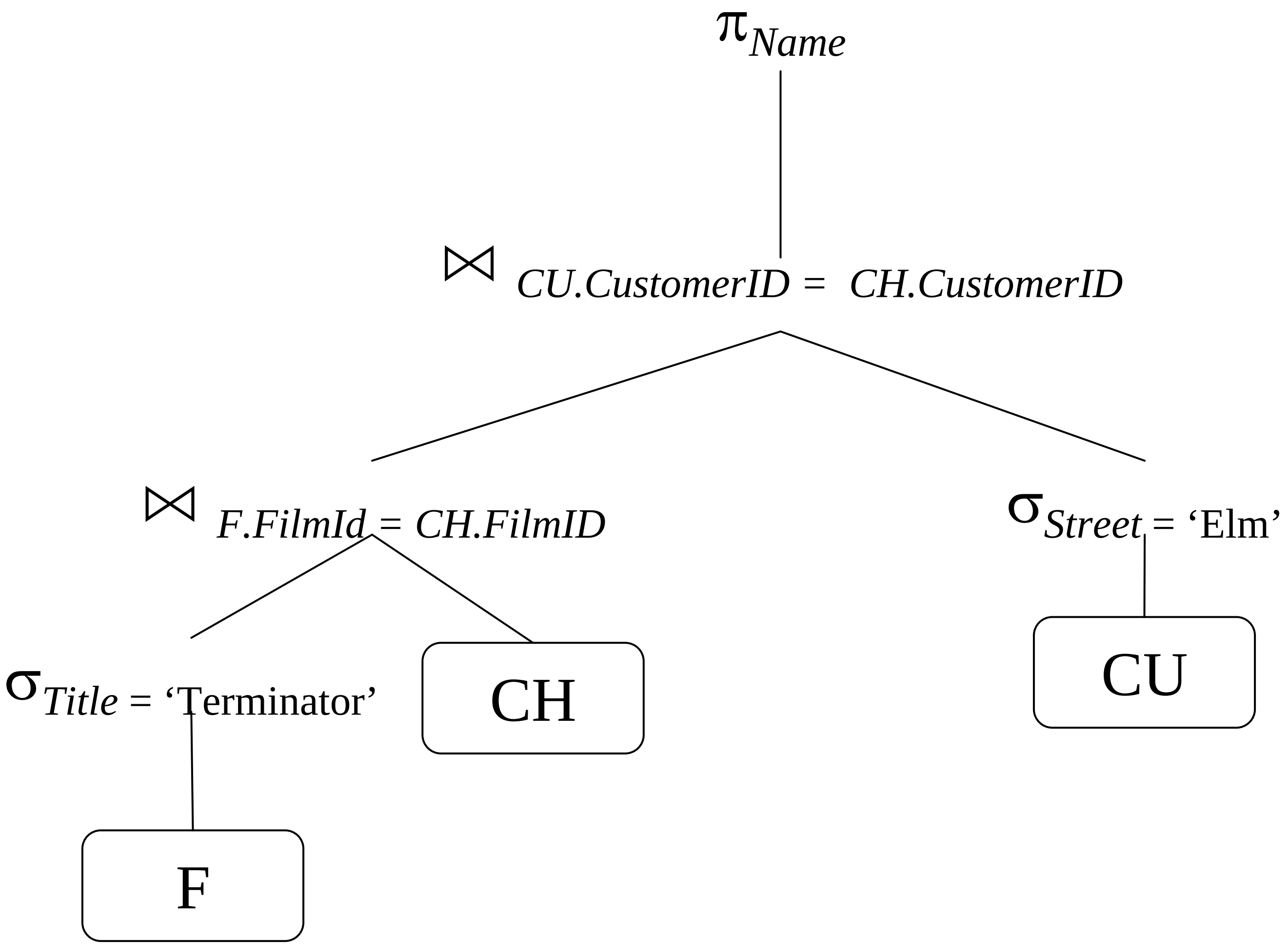
CU

$\sigma_{Title = \text{'Terminator'}}$

CH

F

Form Joins



π_{Name}

$\bowtie_{CU.CustomerID\ =\ CH.CustomerID}$

$\bowtie_{F.FilmId\ =\ CH.FilmID}$

$\sigma_{Street\ =\ \text{‘Elm’}}$

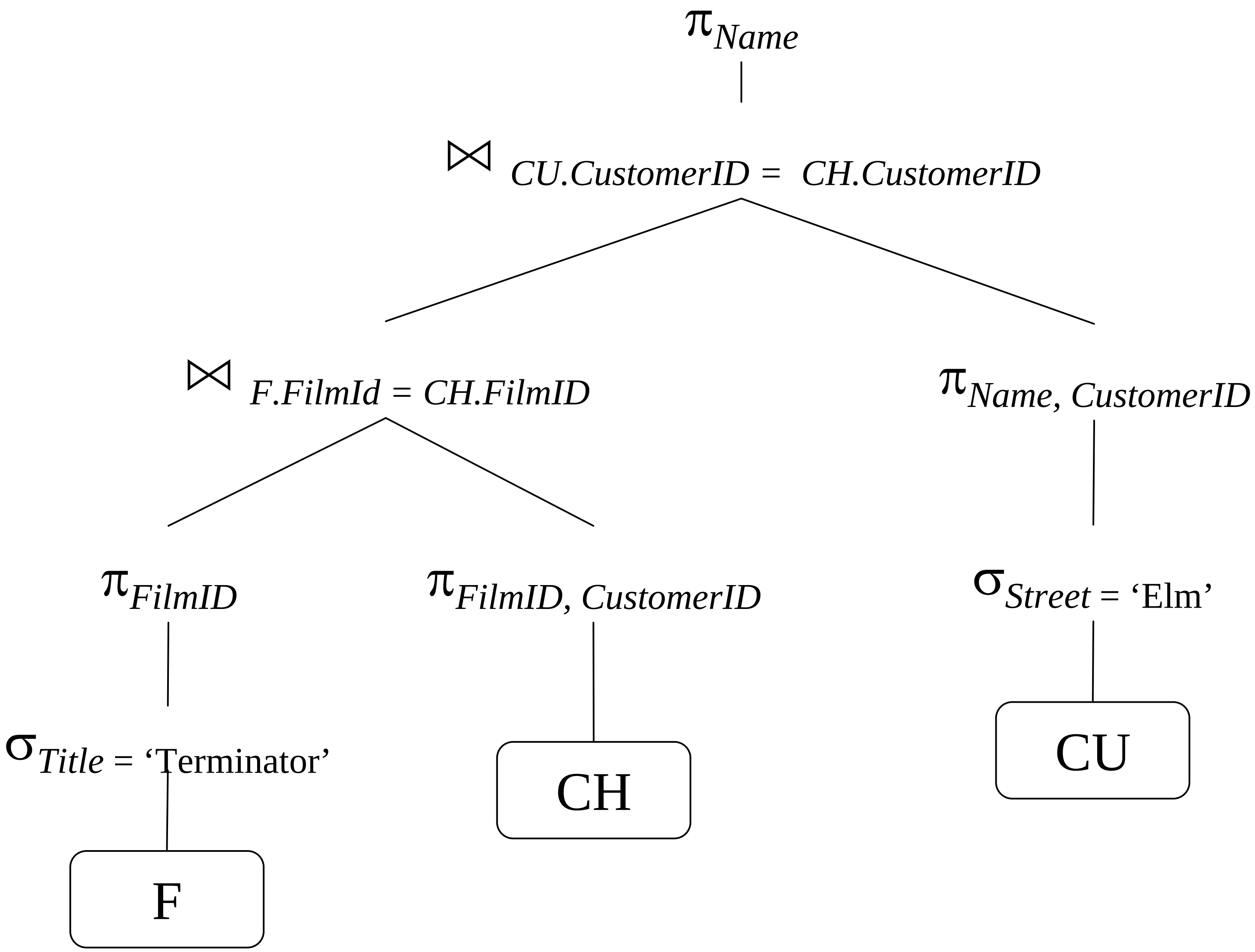
$\sigma_{Title\ =\ \text{‘Terminator’}}$

CH

CU

F

Apply Projections Early



π_{Name}

$\bowtie_{CU.CustomerID = CH.CustomerID}$

$\bowtie_{F.FilmId = CH.FilmID}$

$\pi_{Name, CustomerID}$

π_{FilmID}

$\pi_{FilmID, CustomerID}$

$\sigma_{Street = \text{'Elm'}}$

$\sigma_{Title = \text{'Terminator'}}$

CH

CU

F

Some Transformation Rules

- Cascade of σ: $\sigma_{c_1 \wedge c_2 \wedge \ldots \wedge c_n}(R) = \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(R))\ldots))$
- Commutativity of σ: $\sigma_{c_1}(\sigma_{c_2}(R)) = \sigma_{c_2}(\sigma_{c_1}(R))$
- Commuting σ with π: $\pi_L(\sigma_c(R)) = \sigma_c(\pi_L(R))$
    - Only if *c* involves solely attributes in *L*.
- Commuting σ with ⋈ : $\sigma_c(R \bowtie S) = \sigma_c(R) \bowtie S$
    - Only if *c* involves solely attributes in *R*.
- Commuting σ with *set operations*: $\sigma_c(R\ \theta\ S) = \sigma_c(R)\ \theta\ \sigma_c(S)$
    - Where θ is one of ∪, ∩, or -.
- Commutativity of ∪, ∩, and ⋈: $R\ \theta\ S = S\ \theta\ R$
    - Where θ is one of ∪, ∩ and ⋈.
- Associativity of ⋈, ∪, ∩: $(R\ \theta\ S)\ \theta\ T = R\ \theta\ (S\ \theta\ T)$

Transformation Algorithm Outline

- Transform a query represented in relational algebra to an equivalent one (generates the same result.)

- Step 1: Decompose σ operations.
- Step 2: Move σ as far down the query tree as possible.
- Step 3: Rearrange leaf nodes to apply the most restrictive σ operations first.
- Step 4: Form joins from × and subsequent σ operations.
- Step 5: Decompose π and move down the query tree as far as possible.
- Step 6: Identify candidates for combined operations.

Heuristic Query Optimization Summary

- Heuristic optimization transforms the query-tree by using a set of rules (Heuristics) that typically (but not in all cases) improve execution performance.
  - Perform selection early (reduces the number of tuples)
  - Perform projection early (reduces the number of attributes)
  - Perform most restrictive selection and join operations (i.e. with smallest result size) before other similar operations.

- Generate initial query tree from SQL statement.
- Transform query tree into more efficient query tree, via a series of tree modifications, each of which hopefully reduces the execution time.
- A single query tree is involved.

Cost-Based Optimization

- Use transformations to generate multiple candidate query trees from the canonical query tree.
- Statistics on the inputs to each operator are needed.
  - Statistics on leaf relations are stored in the system catalog.
  - Statistics on intermediate relations must be estimated; most important is the relations' cardinalities.
- Cost formulas estimate the cost of executing each operation in each candidate query tree.
  - Parameterized by statistics of the input relations.
  - Also dependent on the specific algorithm used by the operator.
  - Cost can be CPU time, I/O time, communication time, main memory usage, or a combination.
- The candidate query tree with the least total cost is selected for execution.

Relevant Statistics

- Per relation
  - Tuple size
  - Number of tuples (records): *r*
  - Load factor (fill factor), percentage of space used in each block
  - Blocking factor (number of records per block): *bfr*
  - Relation size in blocks: *b*
  - Relation organization
  - Number of overflow blocks

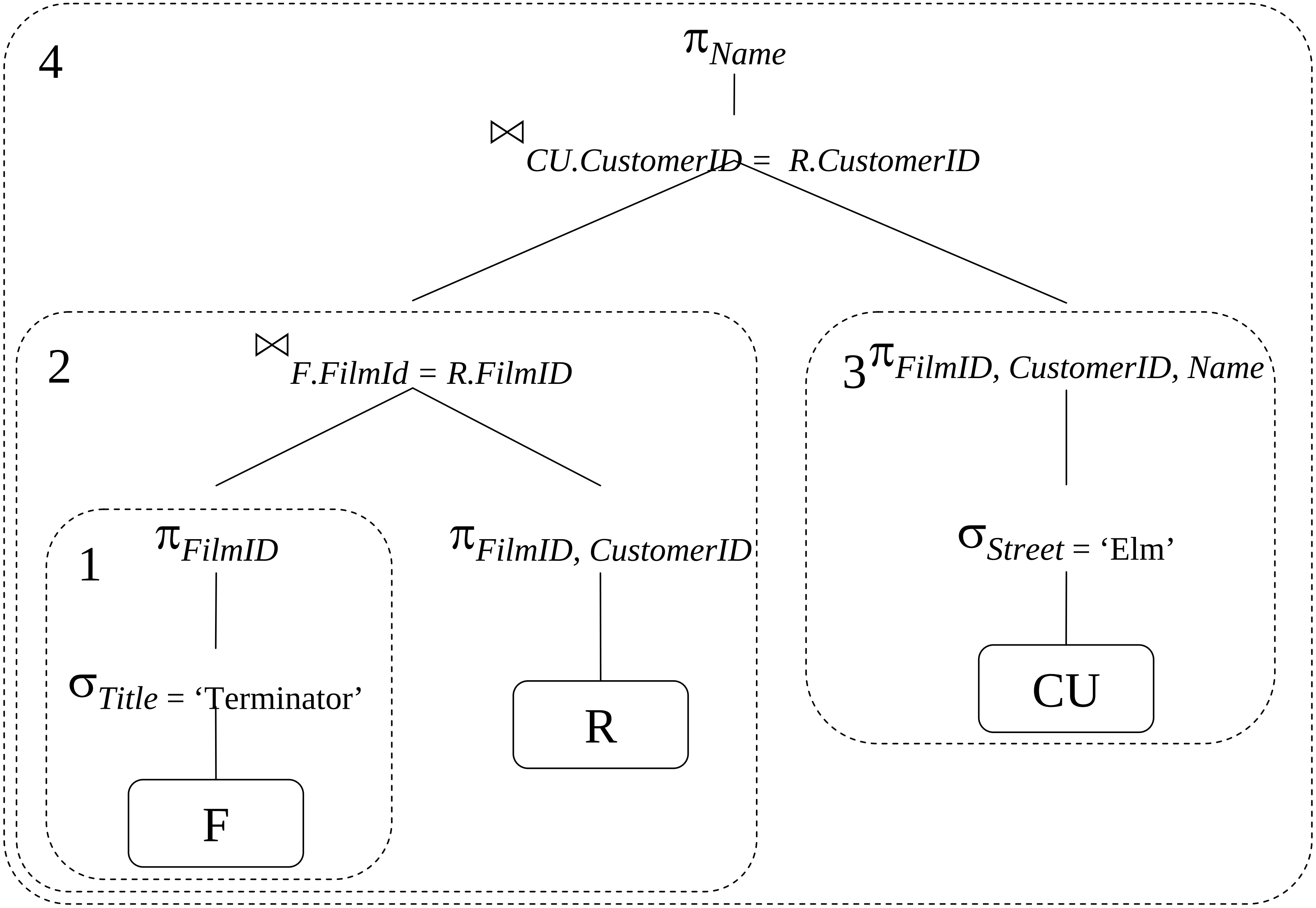
Relevant Statistics, cont.

- Per attribute
  - Attribute size and type
  - Number of distinct values for attribute A: d_A
  - Probability distribution over the values
  - Representation, e.g., compressed
  - Selection cardinality specifies the average size of $\sigma_{A = a}(R)$ for an arbitrary value a. (s_A)
    - Could be maintained for the "average" attribute value, or on a per-value basis, as a histogram.

Relevant Statistics, cont.

- Per Index
  - Base relation
  - Indexed attribute(s)
  - Organization, e.g., B^+-Tree, Hash, ISAM
  - Clustering index?
  - On key attribute(s)?
  - Sparse or dense?
  - Number of levels (if appropriate)
  - Number of first-level index blocks: b_1
- General
  - Available main memory blocks: N

Cost Estimation Example



4

π_{Name}

$\bowtie_{CU.CustomerID\ =\ R.CustomerID}$

2

$\bowtie_{F.FilmId\ =\ R.FilmID}$

1

π_{FilmID}

$\sigma_{Title\ =\ \text{‘Terminator’}}$

F

$\pi_{FilmID,\ CustomerID}$

R

3

$\pi_{FilmID,\ CustomerID,\ Name}$

$\sigma_{Street\ =\ \text{‘Elm’}}$

CU

Operation 1: σ followed by a π

- Statistics
  - Relation statistics: $r_{Film} = 5{,}000$ $b_{Film} = 50$
  - Attribute statistics: $s_{Title} = 1$
  - Index statistics: Secondary Hash Index on *Title*.
- Result relation size: 1 tuple.
- Operation: Use index with ‘Terminator’, then project on *FilmID*. Leave result in main memory (1 block).
- Cost (in disk accesses): $C_1 = 1 + 1 = 2$

Operation 2: $\bowtie$ followed by a π

- Statistics
  - Relation statistics: $r_{CheckedOut} = 40{,}000$ $b_{CheckedOut} = 2{,}000$
  - Attribute statistics: $s_{FilmID} = 8$
  - Index statistics: Secondary B$^+$-Tree Index for *CheckedOut* on *FilmID* with 2 levels.

- Result relation size: 8 tuples.

- Operation: Index join using B$^+$-Tree, then project on *CustomerID*. Leave result in main memory (one block).

- Cost: $C_2 = 1 + 1 + 8 = 10$

Operation 3: σ followed by a π

- Statistics
  - Relation statistics: $r_{Customer} = 200$ $b_{Customer} = 10$
  - Attribute statistics: $s_{Street} = 10$

- Result relation size: 10 tuples.

- Operation: Linear search of *Customer*. Leave result in main memory (one block).

- Cost: $C_3 = 10$

Operation 4: ⋈ followed by a π

- Operation: Main memory join on relations in main memory.

- Cost: $C_4 = 0$

- Total cost: $$C = \sum_{i=1}^{4} C_i = 2 + 10 + 10 + 0 = 22$$

Comparison

- Heuristic query optimization
  - Sequence of single query plans
  - Each plan is (presumably) more efficient than the previous.
  - Search is linear.
- Cost-based query optimization
  - Many query plans generated.
  - The cost of each is estimated, with the most efficient chosen.
  - Search is multi-dimensional, usually using dynamic programming. Still can be very expensive.
- Hybrid way
  - Systems may use *heuristics* to reduce the number of choices that must be made in a cost-based fashion.

Query Processing + Optimization

- Operator Evaluation Strategies
- Query Optimization
- Query Tuning

Query Tuning

- Query optimization is a very complex task.
  - Combinatorial explosion.
  - The task is to find one good query evaluation plan, not the best one.
- No optimizer optimizes all queries adequately.
- There is a need for query tuning.
  - All optimizers differ in their ability to optimize queries, making it difficult to prescribe principles.
- Having to tune queries is a fact of life.
  - Query tuning has a localized effect and is thus relatively attractive.
  - It is a time-consuming and specialized task.
  - It makes the queries harder to understand.
  - However, it is often a necessity.
  - This is not likely to change any time soon.

Query Tuning Issues

- Need too many disk accesses (eg. Scan for a point query)?
- Need unnecessary computation?
  - Redundant DISTINT

    SELECT DISTINCT cpr#

    FROM Employee

    WHERE dept = ‘computer’
- Relevant indexes are not used? (Next slide)
- Unnecessary nested subqueries?
- ……

Join on Clustering Index, and Integer

```
SELECT Employee.cpr#
FROM Employee, Student
WHERE Employee.name = Student.name
```

-->

```
SELECT Employee.cpr#
FROM Employee, Student
WHERE Employee.cpr# = Student.cpr#
```

Nested Queries

- Nested block is optimized independently, with the outer tuple considered as providing a selection condition.
- Outer block is optimized with the cost of ‘calling’ nested block computation taken into account.
- Implicit ordering of these blocks means that some good strategies are not considered. *The non-nested version of the query is typically optimized better.*

```
SELECT  S.sname
FROM  Sailors S
WHERE EXISTS
   (SELECT  *
    FROM  Reserves R
    WHERE  R.bid=103
    AND  R.sid=S.sid)
```

Nested block to optimize:

```
SELECT  *
FROM  Reserves R
WHERE  R.bid=103
   AND  S.sid= outer value
```

Equivalent non-nested query:

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

Unnesting Nested Queries

- Uncorrelated sub-queries with aggregates.
  - Most systems would compute the average only once.

```
SELECT ssn
FROM   emp
WHERE  salary > (SELECT AVG(salary) FROM emp)
```

- Uncorrelated sub-queries without aggregates.

```
SELECT ssn
FROM   emp
WHERE  dept IN (SELECT dept FROM techdept)
```

When is this acceptable?

  - Some systems may not use `emp`'s index on `dept`, so a transformation is desirable.

```
SELECT ssn
FROM   emp, techdept
WHERE  emp.dept = techdept.dept
```

Unnesting Nested Queries, cont.

- Watch out for duplicates! Consider a query and its rewritten counterpart.

```
SELECT AVG(salary)
FROM   emp
WHERE  manager IN (SELECT manager FROM techdept)
```

  - Unnested version, with problems: (what’s the problem?)

```
SELECT AVG(salary)
FROM   emp, techdept
WHERE  emp.manager = techdept.manager
```

  - This query may yield wrong results! A solution:

```
SELECT DISTINCT (manager) INTO temp
FROM   techdept

SELECT AVG(salary)
FROM   emp, temp
WHERE  emp. manager = temp. manager
```

Summary

- Query processing & optimization is the heart of a relational DBMS.
- Heuristic optimization is more efficient to generate, but may not yield the optimal query evaluation plan.
- Cost-based optimization relies on statistics gathered on the relations (the default in most DBMSs).
- Until query optimization is perfected, query tuning will be a fact of life.