It was brought to my attention that Friday 25/4 is Kapsejlads – are any of you planning to be here at the lecture?
Or should I provide a video of the lecture instead?

Query Evaluation

Databases, Aarhus University

Ira Assent

Intended learning outcomes

- Be able to
  - describe and compare different query processing strategies
  - apply and characterize join algorithms

Review: relational calculus

- Relational calculus more *declarative*, specifying result itself
- Expressive power of languages RA, RC identical
- Tuple Relational Calculus (TRC)
  - Tuple variable ranges over tuples of relation
    - Result tuples satisfy conditions
  - $\{t \mid Product(t) \wedge t.Price > 1000\}$

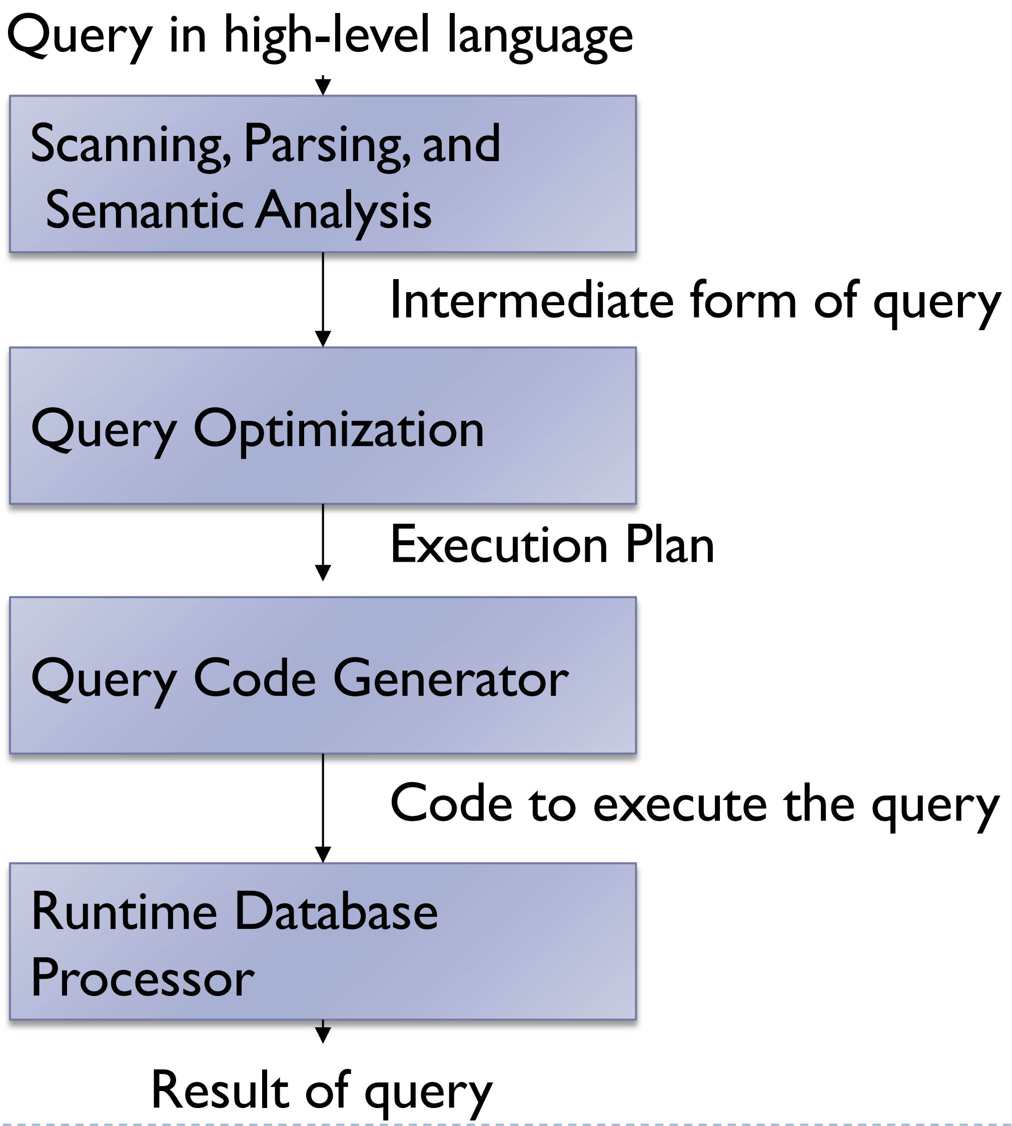
    expensive products
- Domain Relational Calculus (DRC)
  - ranges over domains of attributes

  $\{ I, N, P, C \mid Product(I, N, P, C) \wedge P > 1000\}$
- TRC and DRC otherwise same construction
- An expression is **safe** if all values in its result are from the domain of the expression
  - Means finite result set

List names of Students attending at least one course

1. $\{t.name \mid Student(t) \wedge \neg((\forall s\ \forall u(Course(s) \wedge Attends(u) \wedge\ u.cid=s.id \wedge t.id=u.id))\}$
2. $\{t.name \mid Student(t) \wedge \neg((\exists\ s\ \exists\ u(Course(s) \wedge Attends(u) \wedge u.cid=s.id \wedge t.id=u.id))\}$
3. $\{t.name \mid Student(t) \wedge ((\exists s\ \exists u(Course(s) \wedge Attends(u) \wedge u.cid=s.id \wedge t.id=u.id))\}$
4. $\{t.name \mid Student(t) \wedge ((\forall s\ \forall u(Course(s) \wedge Attends(u) \wedge\ u.cid=s.id \wedge t.id=u.id))\}$

Overview of Query Processing



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

Translating SQL Queries into Relational Algebra

- Query in SQL translated into relational algebra which is then optimized
  - queries are decomposed into **query blocks**
    - basic unit to be translated into algebraic operators and optimized
- A query block contains a single SELECT-FROM-WHERE expression, as well as GROUP BY and HAVING clause if part of the block
  - Nested queries identified as separate query blocks
  - Aggregate operators in SQL must be included in the extended algebra
  - Each query block is optimized

Example translation

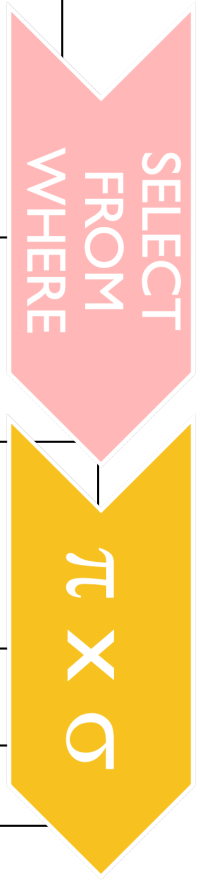
```
SELECT   LNAME, FNAME
FROM     EMPLOYEE
WHERE    SALARY > (   SELECT   MAX (SALARY)
                      FROM     EMPLOYEE
                      WHERE    DNO = 5         );
```

```
SELECT   LNAME, FNAME
FROM     EMPLOYEE
WHERE    SALARY > C;
```

```
SELECT   MAX (SALARY)
FROM     EMPLOYEE
WHERE    DNO = 5;
```

$\pi_{\text{LNAME, FNAME}} (\sigma_{\text{SALARY>C}}(\text{EMPLOYEE}))$

$\mathcal{F}_{\text{MAX SALARY}} (\sigma_{\text{DNO=5}} (\text{EMPLOYEE}))$



Query evaluation

- An SQL query is translated into a relational algebra expression
- The system needs to analyze this expression
- Goal: find out the fastest (or least costly in terms of resources) way of retrieving the result
  - So, query evaluation is about analyzing the expression and finding ways to retrieve the relevant information
  - We study in particular
    - Which data is needed
    - Where is the data, how can we retrieve it (e.g. index)
    - How large is the expected result
    - Are there different ways of determining the result?
      - We will study this in more detail when we get to query optimization

$\mathcal{F}_{\text{MAX SALARY}} (\sigma_{\text{DNO=5}} (\text{EMPLOYEE}))$

Types of selection queries

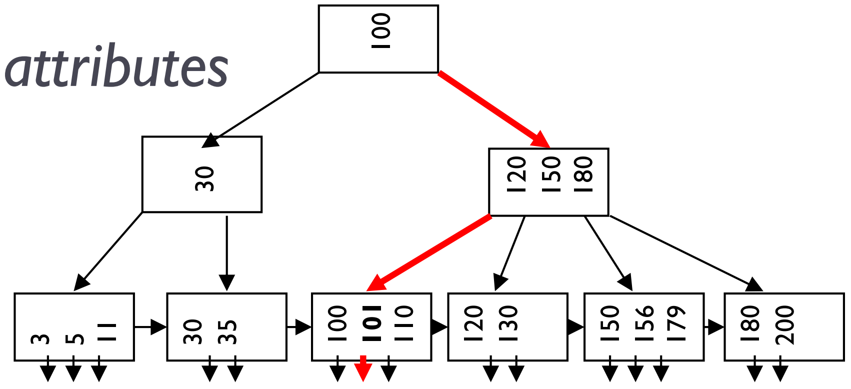
σ_C (R)

- Example selection types, depending on type of condition
  1. Point query
     - Condition is on a single value
     - E.g. $\sigma_{\text{ProductID} = 42}$ (*Product*)
     - Very selective, small result set
  2. Range range
     - Condition is on a range of values
     - E.g. $\sigma_{100 < \text{Price} < 117}$ (*Product*)
     - Selectivity depends on range and distribution of values in attribute
  - Conjunction
     - Combines logically two conditions with AND
     - E.g. $\sigma_{\text{Category} = \text{'Dairy'} \wedge \text{Manufacturer} = \text{'Arla'}}$ (*Product*)
     - Selectivity often high as more than one condition must be met
  - Disjunction
     - Combines logically two conditions with OR
     - E.g. $\sigma_{\text{CreationDate} < 1990 \vee \text{Manufacturer} = \text{'FDM'}}$ (*Product*)
     - Selectivity often low as any of the conditions suffices

Product (<u>ProductID</u>, Name, CreationDate, Price, Manufacturer, Category)

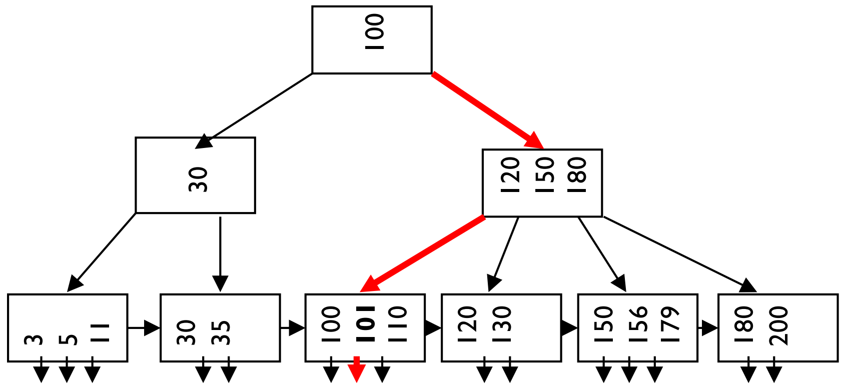
Recap: indexes

- When deciding what the most efficient data access is for executing an incoming SQL query consider indexes
  - Here often called access paths
- Index on a table: data structure that helps find rows quickly
  - index file consists of entries of the form (search key, pointer)
  - Index files are typically much smaller than the original file
  - A table may have several indexes
  - Virtual **sorting** of table *on chosen attributes*



Why B+-trees?

A. They fill nodes by at least 75%
B. They are balanced, so that there is at most one level height difference between leaves
C. They are I/O efficient
D. They use compact nodes with few entries
E. The fridge is far away



Executing selection for single condition

- **Linear search** (brute force)
  - Retrieve every record in the file, test whether its attribute values satisfy selection condition
- **Binary search**
  - If equality comparison on key attribute on which file is ordered
    - Compare value at half file range, then go to earlier half or later half of that, depending on comparison with current value
- **Index-based search** used if
  - equality comparison on key attribute with primary index, use index
  - Or if comparison condition is >, ≥, <, or ≤ on key with primary index, use index to find record satisfying corresponding equality condition, then retrieve all subsequent records in (ordered) file
  - Or if equality comparison on non-key attribute with clustering index

Executing conjunctions and disjunctions

- Conjunction
  - If attributes involved in equality conditions in conjunction match attributes in composite index (or hash structure), use index directly
    - E.g. composite index on (Category, Product) for query

      $\sigma_{\text{Category = 'Dairy'} \wedge \text{Manufacturer = 'Arla'}}$ (*Product*)
  - If attribute involved in conjunctive condition has index, use it to retrieve matching records and then check each for remaining simple conditions in the conjunctive condition
    - If more than one option, always pick most selective one first, i.e. the one that is expected to retrieve fewest records most efficiently
    - E.g. use index on *Product* for query $\sigma_{\text{Category = 'Dairy'} \wedge \text{Manufacturer = 'Arla'}}$ (*Product*), probably preferable to index on *Category*
      - Expect fewer products by Arla than dairy products overall
    - May use intersection of pointers in secondary index (like B-tree)
      1. Find list of pointers involving Dairy
      2. Find list of pointers involving Arla
      3. Retrieve data corresponding to intersection of pointers from both lists

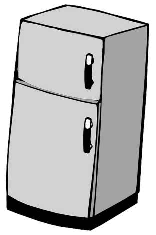
Join

- Most time-consuming operator
  - Often natural join or equijoin
- E.g. Customer * Purchased

Customer (<u>CustomerID</u>, Name, Street, City)

Purchased (<u>CustomerID</u>, <u>ProductID</u>)

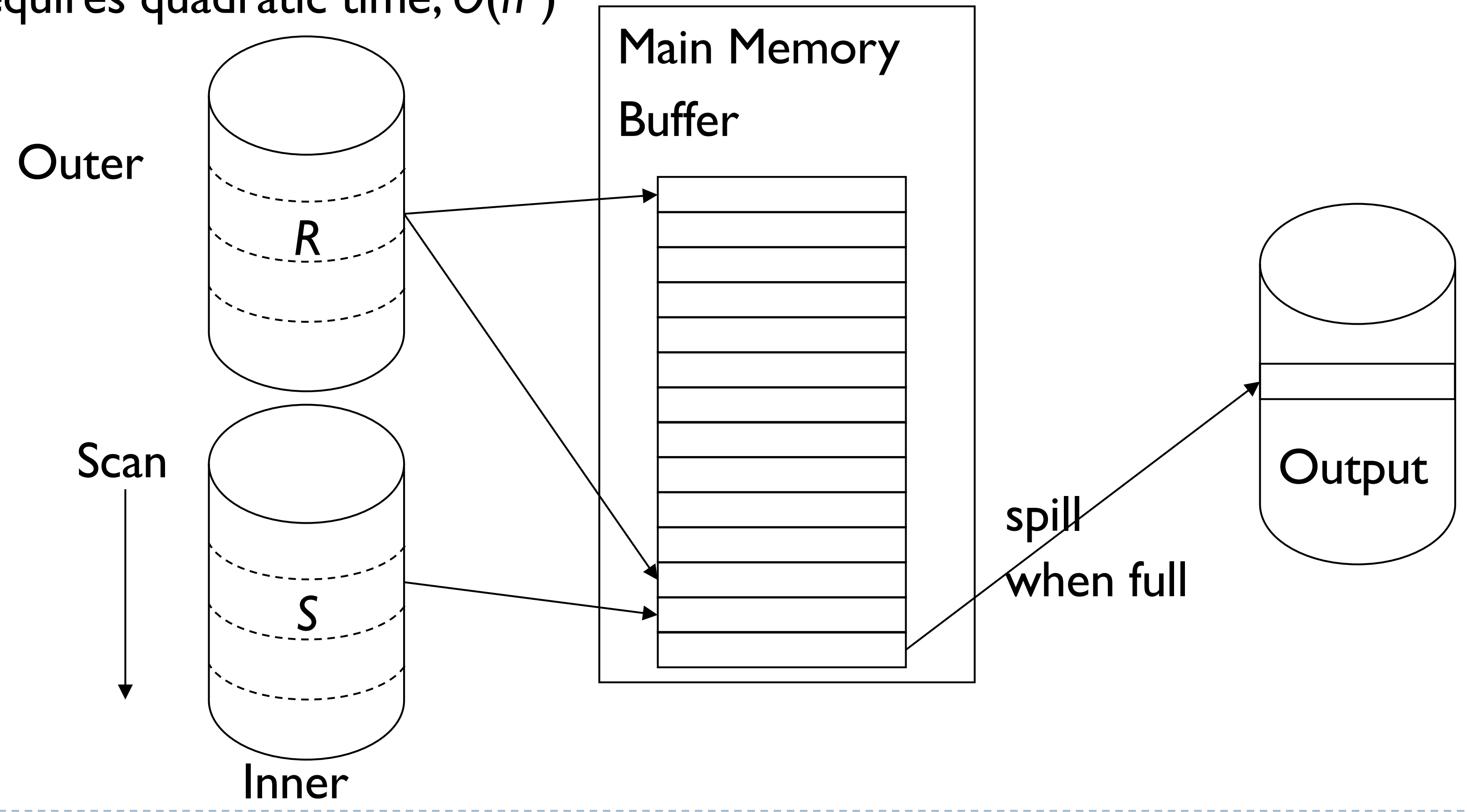
- Join Strategies
  - Nested loop join (brute force)
  - Index-based join
  - Sort-merge join
  - Hash join



- Strategies work on a **per block** (not per record) basis
- Relation sizes and join selectivities impact join cost

Nested Loop

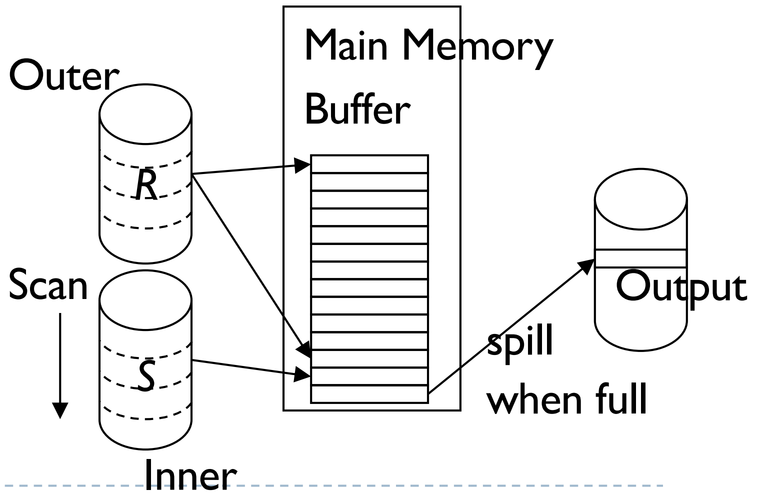
- For each block of the outer table R (outer loop), scan the entire inner table S (inner loop) and test whether the records satisfy the join condition (e.g. equality on join attribute)
- Requires quadratic time, $O(n^2)$



Nested Loop Join: runtime

VOTE

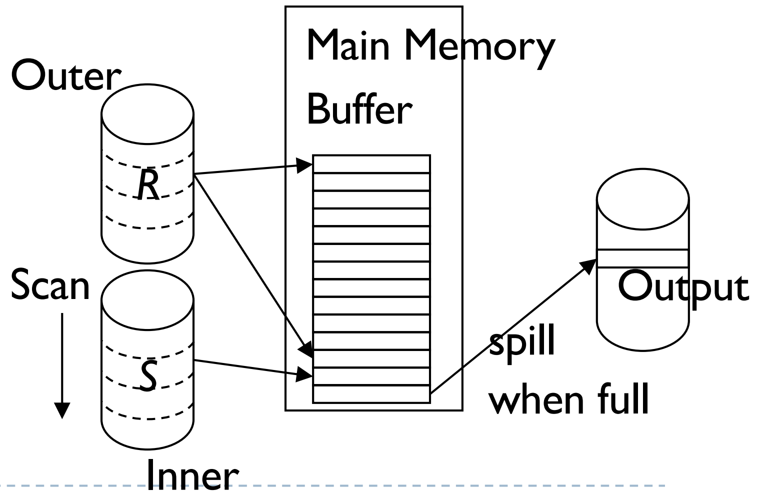
A. The size of the inner relation should be the largest
B. The size of the outer relation should be the largest
C. The size does not matter
D. The total size of the relations matters only if main memory is limited



Nested Loop Join: runtime

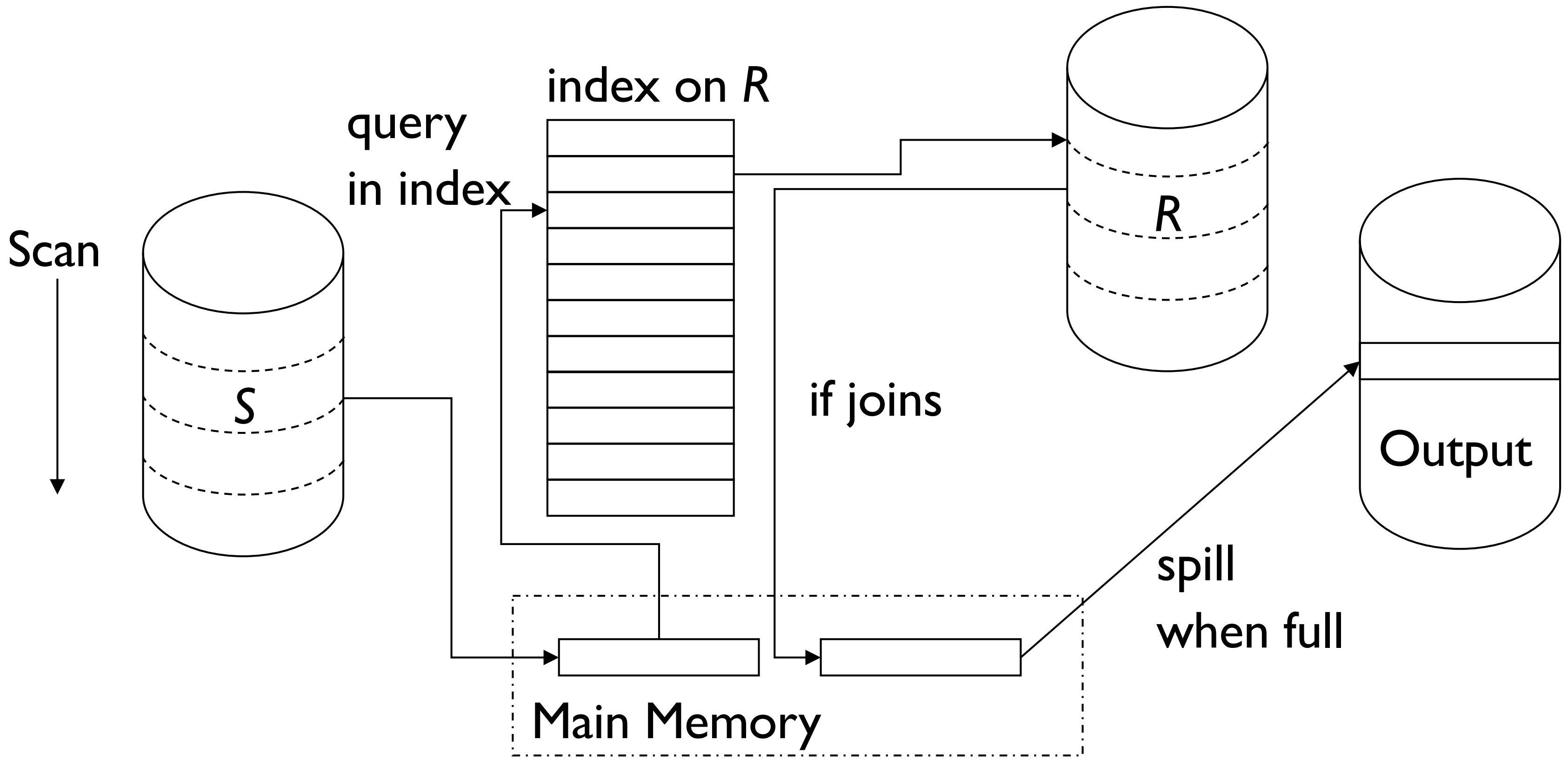
A. The size of the inner relation should be the largest

- Assuming more data in both relations than main memory, less I/O if outer relation smaller, so fewer scans of inner relation
- Let size of inner relation b_I, size of outer relation b_O
- Let number of buffers in main memory b_M (equivalent in size to disk blocks)
  - one block for reading from inner file and one block for writing to output file:
  - b_M-2 buffers available for "work"
- Outer: read one block at a time, only once, i.e. read b_O
- Inner: for each outer block read as much of inner as possible: $b_I/(b_M-2)$
  - i.e., inner is read $b_O*b_I/(b_M-2)$
- Total: $b_O+b_O*b_I/(b_M-2)$
- Example:
  - Outer 100.000, inner 10, main memory 12 blocks
  - 100.000+100.000*1= <u>200.000 blocks</u>
  - Swap roles: Outer 10, inner 100.000 blocks
  - 10+10*10.000= <u>100.010 blocks</u>



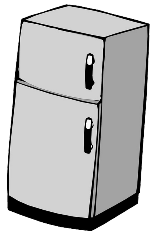
Index-based Join

- Requires (at least) one index on a join attribute e.g. for relation R
  - Single loop on the other relation S (outer) to find corresponding values on the indexed attribute in R (inner)
    - For each join value in S, use index to retrieve matching records from R



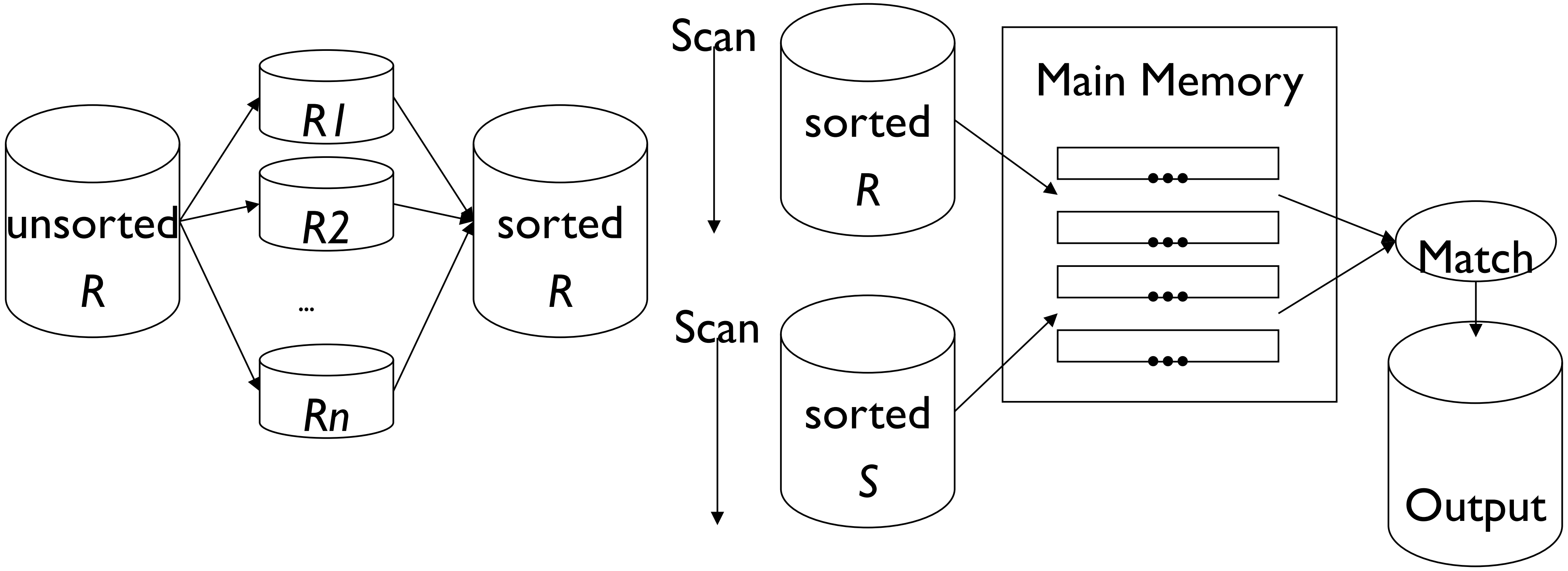
Algorithms for External Sorting

- Sorting is a primary algorithm in query processing
  - E.g. `ORDER BY`
  - But also in join algorithms as we will see
- **External sorting**
  - Sorting algorithms suitable for large files on disk that do not fit entirely in main memory, such as most database files
- Sort-Merge strategy:
  - Starts by sorting small subfiles (runs) of the main file and then merges the sorted runs, creating larger sorted subfiles that are merged in turn
  - Sorting phase: $n_R = \lceil (b/n_B) \rceil$
  - Merging phase: $d_M = \text{Min}\ (n_B - 1, n_R)$; $n_P = \lceil (\log_{dM}(n_R)) \rceil$
  - n_R: number of initial runs; b: number of file blocks;
  - n_B: available buffer space; d_M: degree of merging;
  - n_P: number of passes



Sort-Merge Join

- Sort each relation (e.g. using multiway merge-sort)
- Perform most efficient join: merge-join
  - Copy pairs of blocks into memory in order, then scan both to find matches
    - In this method, the records of each file are scanned only once each for matching with the other file —unless both A and B are non-key attributes, in which case the method needs to be modified slightly



Sort-Merge

```
set     $i \leftarrow 1$;
        $j \leftarrow b$;            {size of the file in blocks}
        $k \leftarrow n_B$;          {size of buffer in blocks}
        $m \leftarrow \lceil (j/k) \rceil$;
```

```
{**Sorting Phase**}
while ($i \leq m$)
do {
        read next $k$ blocks of the file into the buffer or if there are less than $k$ blocks
            remaining, then read in the remaining blocks;
        sort the records in the buffer and write as a temporary subfile;
        $i \leftarrow i + 1$;
}

{**Merging Phase:** merge subfiles until only 1 remains}
set     $i \leftarrow 1$;
        $p \leftarrow \lceil \log_{k-1} m \rceil$    {$p$ is the number of passes for the merging phase}
        $j \leftarrow m$;
while ($i \leq p$)
do {
        $n \leftarrow 1$;
        $q \leftarrow (j/(k-1)) \rceil$;  {number of subfiles to write in this pass}
        while ($n \leq q$)
        do {
            read next $k-1$ subfiles or remaining subfiles (from previous pass)
                one block at a time;
            merge and write as new subfile one block at a time;
            $n \leftarrow n + 1$;
        }
        $j \leftarrow q$;
        $i \leftarrow i + 1$;
}
```

Use Nested Loop join to implement left outer join?

```
SELECT FNAME, DNAME
FROM EMPLOYEE LEFT OUTER JOIN DEPARTMENT ON
DNO = DNUMBER;
```

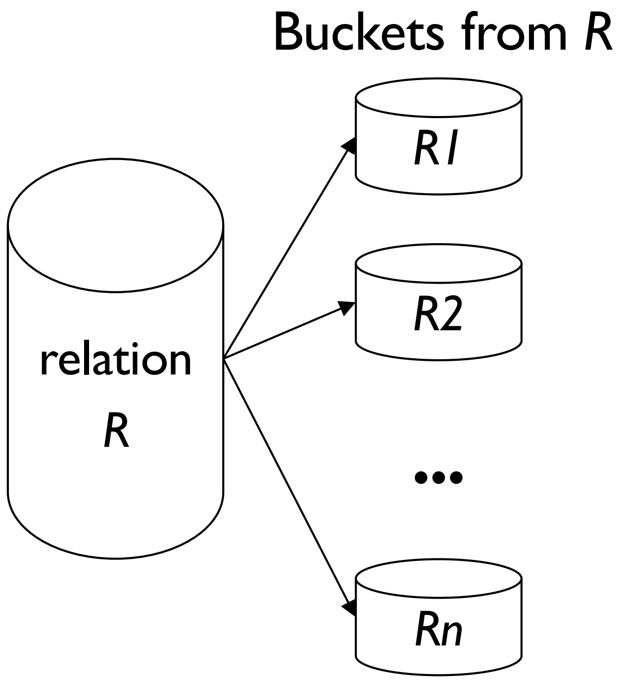
1. Run Nested Loop join then union with left relation
2. Run Nested Loop join then union with right relation
3. Use the left relation as outer relation and pad with nulls where necessary
4. Use the right relation as outer relation and pad with nulls where necessary
5. Not possible, operators are different

Outer Joins

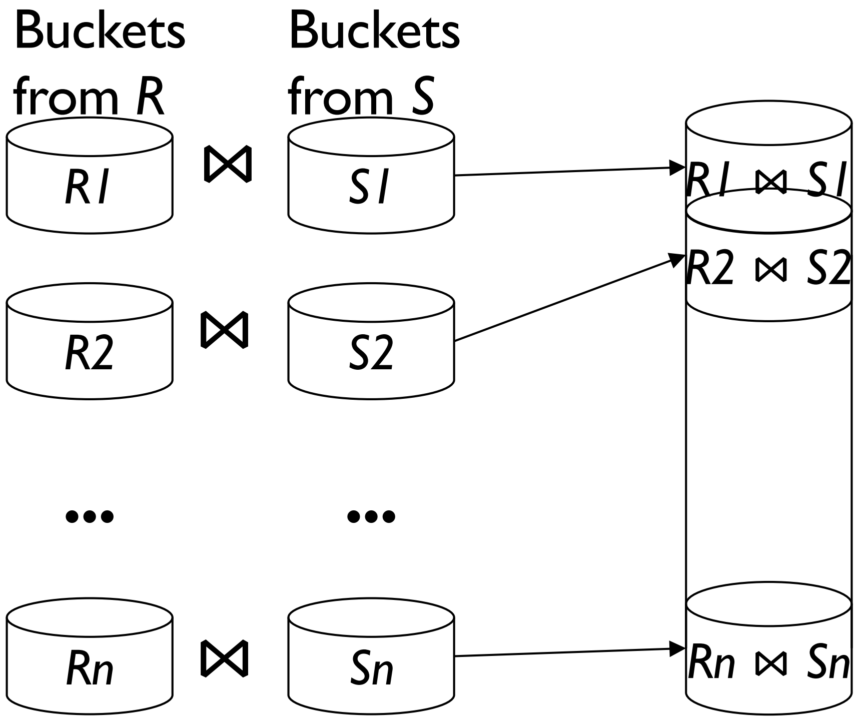
- `LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN`
- Example: `SELECT FNAME, DNAME FROM EMPLOYEE LEFT OUTER JOIN DEPARTMENT ON DNO = DNUMBER;`
  - Nested Loop or Sort-Merge joins can be modified to implement outer join
  - For example
    - For left outer join, use the left relation as outer relation and construct result from every tuple in the left relation
    - If there is a match, the concatenated tuple is saved in the result
    - If an outer tuple does not match, it is still included in the result but is padded with null value(s)

Hash Join

- Hash each relation on the join attributes
- Each bucket must be small enough to fit in memory
- Join corresponding buckets from each relation



Hash *R* (same for S)



Join corresponding *R* and S buckets

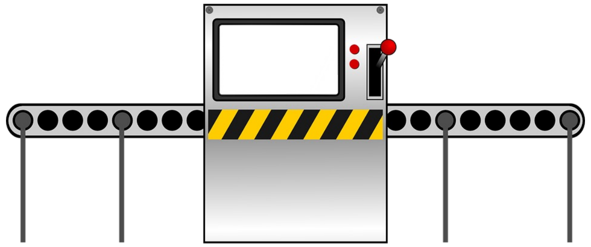
Other Operations

- Projection: generally straightforward $\boxed{\pi_{\text{Name}}\ (\textit{Customer})}$
  - if duplicate elimination necessary then usually done by sorting/hashing then scanning
- Cartesian product $\boxed{\textit{Customer} \times \textit{Product}}$
  - Inherently expensive: need to generate all combinations of tuples
  - Should be avoided if possible
- Union, intersection, and difference
  - Sort and scan
  - Hash into buckets, check each bucket

$$\boxed{\pi_{\text{CustomerID}}\ (\textit{Customer}) \begin{cases} \cup \\ \cap \\ \setminus \end{cases} \pi_{\text{CustomerID}}\ (\textit{Purchased})}$$

Combining Operations using Pipelining

- Motivation
  - A query is mapped into a sequence of operations
  - Each execution of an operation produces a temporary result
  - Generating and saving temporary files on disk is time consuming and expensive
- Alternative
  - Avoid constructing temporary results as much as possible
  - Pipeline the data through multiple operations - pass the result of a previous operator to the next without waiting to complete the previous operation



Pipelining example

- Example
  - $\pi_{LNAME, FNAME}(\sigma_{SALARY>C}(EMPLOYEE))$
  - Instead of selection results entirely, then doing projection, pass any tuples of selection directly to projection
    - Less space needed, faster
      - No need to store temporary relation to disk
- For pipelining to be effective, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation
- Also known as stream-based processing
- Pipelining may not always be possible – e.g., sort, hash-join

Aggregate Operations

- `MIN, MAX, SUM, COUNT, AVG`
  - Do table scan and maintain aggregate
    - E.g. replace minimum if smaller value is found, else continue
    - For average, need to maintain count and sum
  - Use index
    - Example `SELECT MAX(SALARY) FROM EMPLOYEE;`
      - If index on SALARY, then traverse it for largest value, i.e., follow rightmost pointer from root to leaf
- `SUM, COUNT, AVG`
  - For a dense index (each record has one index entry)
    - Apply the associated computation to the values in the index
  - For a non-dense index:
    - Actual number of records associated with each index entry must be accounted for
- With `GROUP BY`
  - aggregate operator must be applied separately to each group of tuples
  - Use sorting or hashing on group attributes to partition into appropriate groups
  - Compute aggregate for tuples in each group

Use clustering index for GROUP BY?

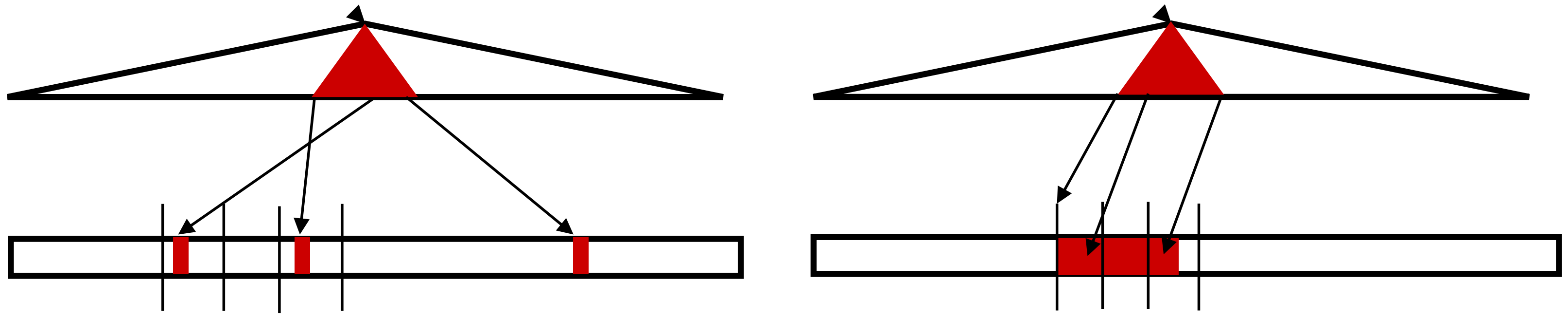
AUVOTE

1. Use index to hash group attribute into partitions
2. If index is on group attributes apply computation to attributes directly
3. Use index to get list of pointers to form partitions
4. Clustering index is not suitable for group aggregate computation

Use clustering index for GROUP BY?

If index is on group attributes apply computation to attributes directly

- Recall, that unlike general indexes where rows are scattered, a **clustering index** has consecutive rows



- Equivalent to sorting the table
- So, a clustering index means that records are already partitioned into groups

  so only need to compute aggregate

Summary

- Intended learning outcomes
  - Be able to
    - describe and compare different query processing strategies
    - apply and characterize join algorithms

- Acknowledgements
  - Richard T. Snodgrass (University of Arizona), Christian S. Jensen (Aalborg University), Kristian Torp (Aalborg University), Curtis Dyreson (Washington State University)

What was this all about?

Guidelines for your own review of today's session

- The selection operator is implemented as…
  - For conjunctions and disjunctions…
- Joins are implemented as…
  - Nested loop join is favoured when…
  - Index-join, sort-merge-join, and hash-join are respectively best when…
    - They work as follows…
- Other operators are handled by…
- Pipelining uses the idea that…
  - It benefits query processing in that…