

CS525: Advanced Database Organization

Notes 6: Query Processing Part V: Query Execution

Yousef M. Elmehdwi

Department of Computer Science

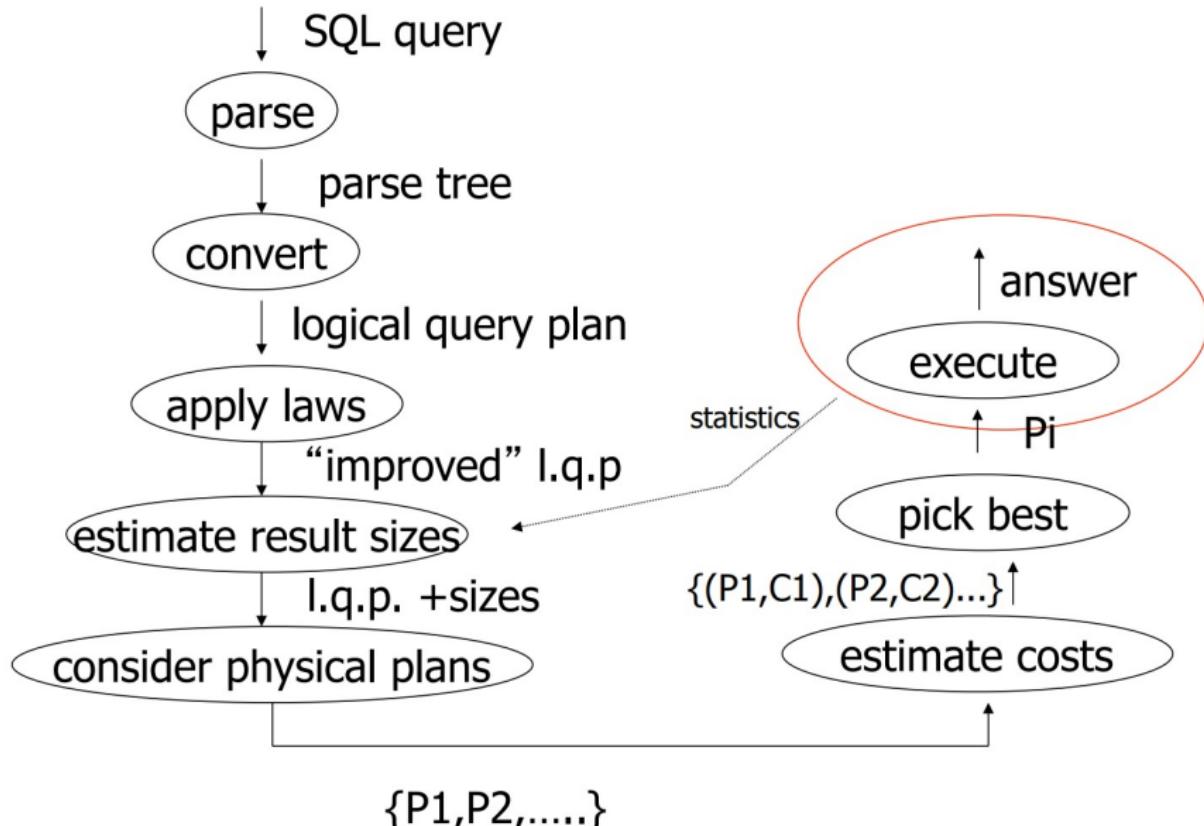
Illinois Institute of Technology

yelmehdwi@iit.edu

November 22th 2023

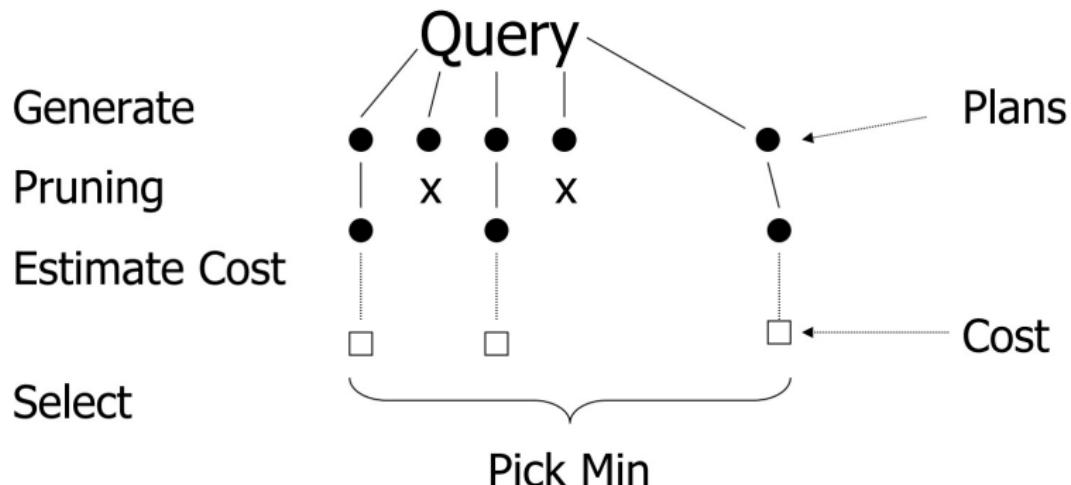
Slides: adapted from courses taught by [Shun Yan Cheung, Emory University](#), [Hector Garcia-Molina, Stanford](#), [Andy Pavlo, Carnegie Mellon University](#), & [Jennifer Welch, Texas A&M](#)

Query Processing



Query Optimization

- Generating and comparing plans

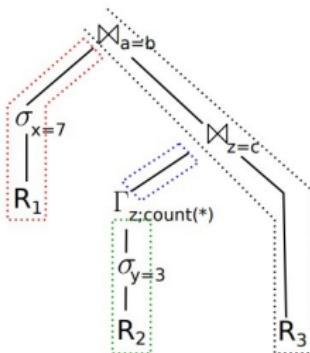


Query Execution

- Query plan is not directly executable.
- Need to translate into executable code
- From SQL to executable code¹

```
select      *
from        R1,R3,
           (select    R2.z,
                     count(*)
              from      R2
             where      R2.y=3
            group by   R2.z) R2
           where     R1.x=7
           and       R1.a=R3.b
           and       R2.z=R3.c
```

(a) Example SQL Query



(b) Execution Plan

```
initialize memory of σa=b, σc=z, and Γz
for each tuple t in R1
  if t.x = 7
    materialize t in hash table of σa=b
for each tuple t in R2
  if t.y = 3
    aggregate t in hash table of Γz
for each tuple t in Γz
  materialize t in hash table of σz=c
for each tuple t3 in R3
  for each match t2 in σz=c [t3.c]
    for each match t1 in σa=b [t3.b]
      output t1 ∘ t2 ∘ t3
```

(c) Compiled query (pseudo-code)

¹ Compiling Database Queries into Machine Code

Execution Strategies

- Compiled
 - Translate into C/C++/Assembler code
 - Compile, link, and execute code
- Interpreted
 - Generic operator implementations
 - Generic executor
 - Interprets query plan

Reading: Query Compiler

- How to Architect a Query Compiler, Revisited
- Compiling Database Queries into Machine Code

Query Plan

- The DBMS converts a SQL statement into a query plan.
- Operators are arranged in a tree.
- Data flows from the leaves towards the root.
- The output of the root node in the tree is the result of the query.
- Typically operators are binary (1- 2 children).
- The same query plan can be executed in multiple ways.
- Most DBMSs will want to use an index scan as much as possible.

Query Execution

- Here only
 - how to implement operators
 - what are the costs of implementations
 - how to implement queries
 - Data flow between operators
- Next part: How to choose good plan

Physical Operators used in Physical Query Plans

- **Physical query operators:** the programs (algorithms) used to execute a query
- **Physical query plan:** a sequence of **physical query operators** that accomplishes the execution of a query
- Many of the **physical plan operators** have multiple implementations
 - Some implementations of the operator are very efficient but require a large amount of memory
 - While other implementations are less efficient but require a smaller amount of memory
- Amount of buffers will determine which implementation that we can use to process the query

How to estimate costs

- If everything fits into memory
 - Standard computational complexity
- If not
 - Assume fixed memory available for buffering pages
 - Count I/O operations
 - Real systems combine this with CPU estimations

Cost and constraint of a Physical Query Plan

- Cost of a physical query plan: the number of disk blocks that are accessed by the execution of the physical query plan
- Constraint on a physical query plan
 - Memory limitation
 - The operators used in the physical query plan will require a certain minimum buffer allocation requirement
 - The amount of memory available can prohibit the choice of certain (more efficient) algorithms for an operator
 - Query optimization: find the least cost (physical) query plan such that total memory used by the operators \leq total available memory buffers

Assumption in calculating the cost of an operator

- **Cost of an operator:** the number of disk I/O operations (# disk blocks) performed by the operator
- Assumption in computing the cost of an operator
 - The output of the operator is left in memory
 - i.e., the cost of an operation does not include the disk I/O's to write result (to disk).
 - Unless the execution plan needs to write the result to disk (to save memory space)
- Reason:
 - Query plans often use pipelining to execute the operations in the query plans
 - Pipelining passes the output tuples using memory buffers

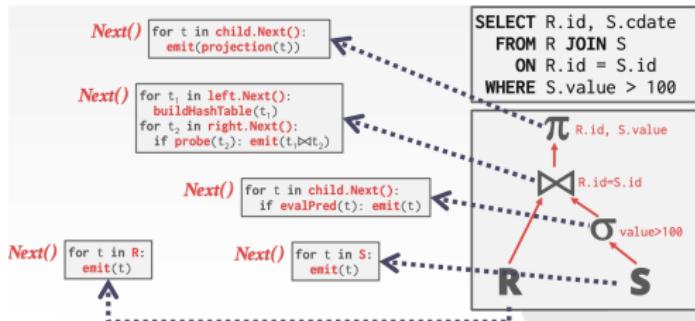
Processing Model

- A DBMS's processing model defines how the system executes a query plan.
 - It specifies things like the direction in which the query plan is read in either from **top-to-bottom** or from **bottom-to-top** as well as what kind of data is passed between operators along the way.
- There are different models of processing models that have various trade-offs for different workloads:
 1. Iterator Model (Also called **Volcano** or Pipeline Model)
 2. Materialization Model
 3. Vectorized / Batch Model
- These models can also be implemented to invoke the operators either from **top-to-bottom** (most common) or from **bottom-to-top**.

Iterator Model

- This is the most common processing model and is used by almost every (row-based) DBMS
- Allows for **pipelining** where the DBMS can process a tuple through as many operators as possible before having to retrieve the next tuple.
 - The series of tasks performed for a given tuple in the query plan is called a **pipeline**.
- Every query plan operator implements a **next** function:
 - On each call to **next**, the operator returns either a single tuple or a null marker if there are no more tuples.
 - The operator implements a loop that calls **next** on its children to retrieve their tuples and then process them (i.e., calling **next** on a parent calls **next** on their children).
- Also called **Volcano** or **Pipeline** Model.

Iterator Model Example



- Pseudo code of the different `next` functions for each of the operators
- The `next` functions are essentially *for* loops iterating over the output of their child operator.
 - For example, the root node calls `next` on its child, the join operator, which is an access method that loops over the relation R and emits a tuple up that is then operated on.
 - Each tuple is then processed up the plan as far as possible before the next tuple is retrieved.
 - After all tuples have been processed, a null pointer is sent that lets the parent nodes know to move on.

Iterator Model

1

```
for t in child.Next():
    emit(projection(t))
```

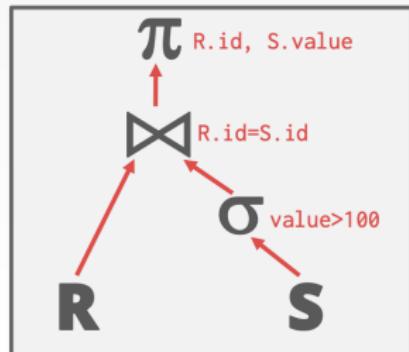
```
for t1 in left.Next():
    buildHashTable(t1)
for t2 in right.Next():
    if probe(t2): emit(t1 ▷ t2)
```

```
for t in child.Next():
    if evalPred(t): emit(t)
```

```
for t in R:
    emit(t)
```

```
for t in S:
    emit(t)
```

```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



Iterator Model

1

```
for t in child.Next():
    emit(projection(t))
```

2

```
for t1 in left.Next():
    buildHashTable(t1)
for t2 in right.Next():
    if probe(t2): emit(t1▷◁ t2)
```

```
for t in child.Next():
    if evalPred(t): emit(t)
```

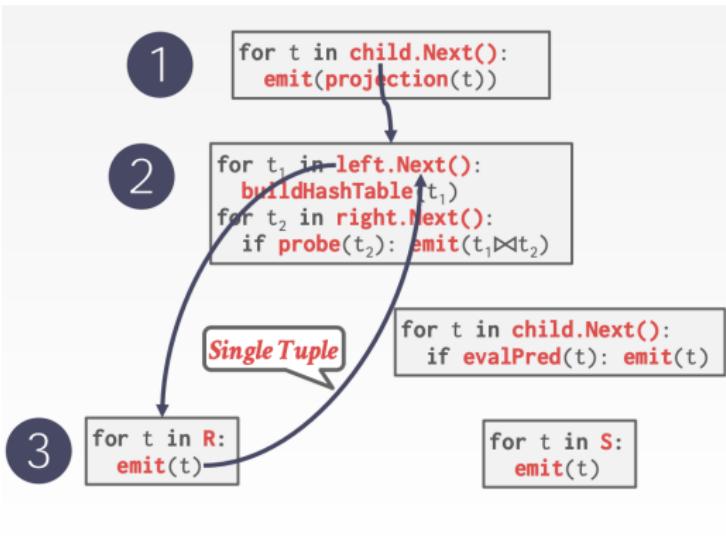
```
for t in R:
    emit(t)
```

```
for t in S:
    emit(t)
```

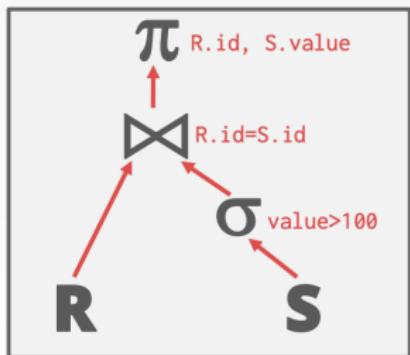
```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```

 $\pi_{R.id, S.value}$  $\sigma_{value > 100}$ R S

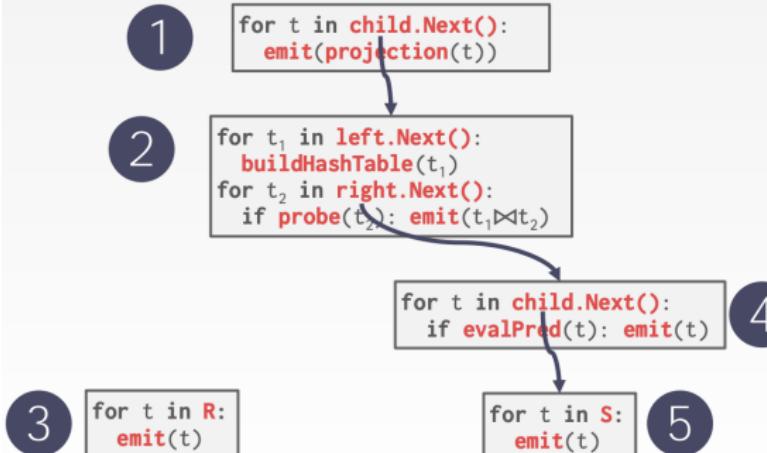
Iterator Model



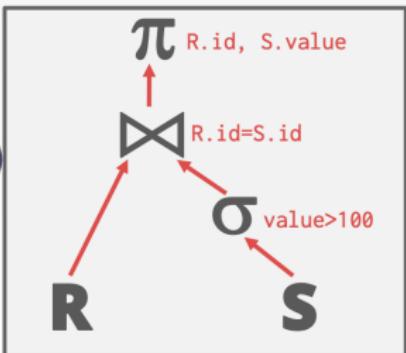
```
SELECT R.id, S.cdate  
FROM R JOIN S  
ON R.id = S.id  
WHERE S.value > 100
```



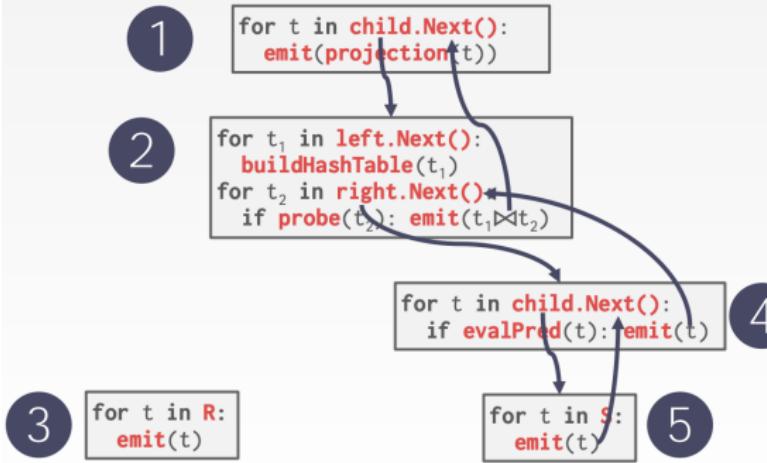
Iterator Model



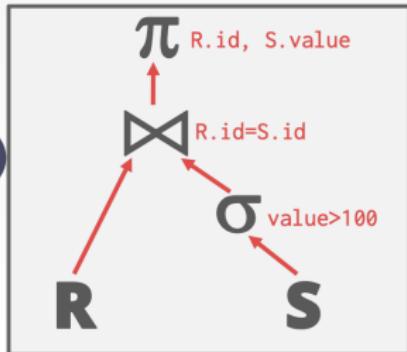
```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



Iterator Model



```
SELECT R.id, S.cdate  
FROM R JOIN S  
ON R.id = S.id  
WHERE S.value > 100
```



Iterator Model

- This is used in almost every DBMS.
- Allows for tuple **pipelining**
 - This is useful in disk-based systems because it allows us to fully use each tuple in memory before the next tuple is retrieved.
- Some operators will block until children emit all of their tuples (joins, subqueries, order by). These are known as **pipeline breakers**.
- Output control works easily with this approach (**LIMIT**) because an operator can stop invoking **next** on its children operators once it has all the tuples that it requires.

Materialization Model

- The **materialization model** is a specialization of the **iterator model** where each operator processes its input all at once and then emits its output all at once.
 - Instead of having a **next** function that returns a single tuple, each operator returns all of its tuples every time it is reached.
- The operator **materializes** its output as a single result.
- The output can be either whole tuples or subsets of columns.
- Every query plan operator implements an **output** function:
 - The operator processes all the tuples from its children at once.
 - The return result of this function is all the tuples that operator will ever emit. When the operator finishes executing, the DBMS never needs to return to it to retrieve more data.
- The DBMS can push down hints into to avoid scanning too many tuples.

Materialization Model Example

1

```
out = []
for t in child.Output():
    out.add(projection(t))
return out
```

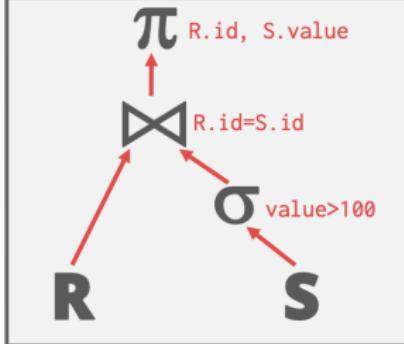
```
out = []
for t1 in left.Output():
    buildHashTable(t1)
for t2 in right.Output():
    if probe(t2): out.add(t1▷◁ t2)
return out
```

```
out = []
for t in child.Output():
    if evalPred(t): out.add(t)
return out
```

```
out = []
for t in R:
    out.add(t)
return out
```

```
out = []
for t in S:
    out.add(t)
return out
```

```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



- Starting at the root, the `child.Output()` function is called, which invokes the operators below, which returns all tuples back up.

Materialization Model

1

```
out = []
for t in child.Output():
    out.add(projection(t))
return out
```

2

```
out = []
for t1 in left.Output():
    buildHashTable(t1)
for t2 in right.Output():
    if probe(t2): out.add(t1 ▷ t2)
return out
```

3

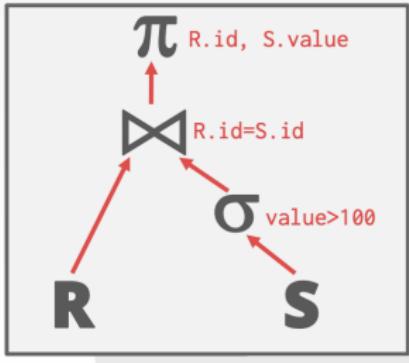
```
out = []
for t in R:
    out.add(t)
return out
```

All Tuples

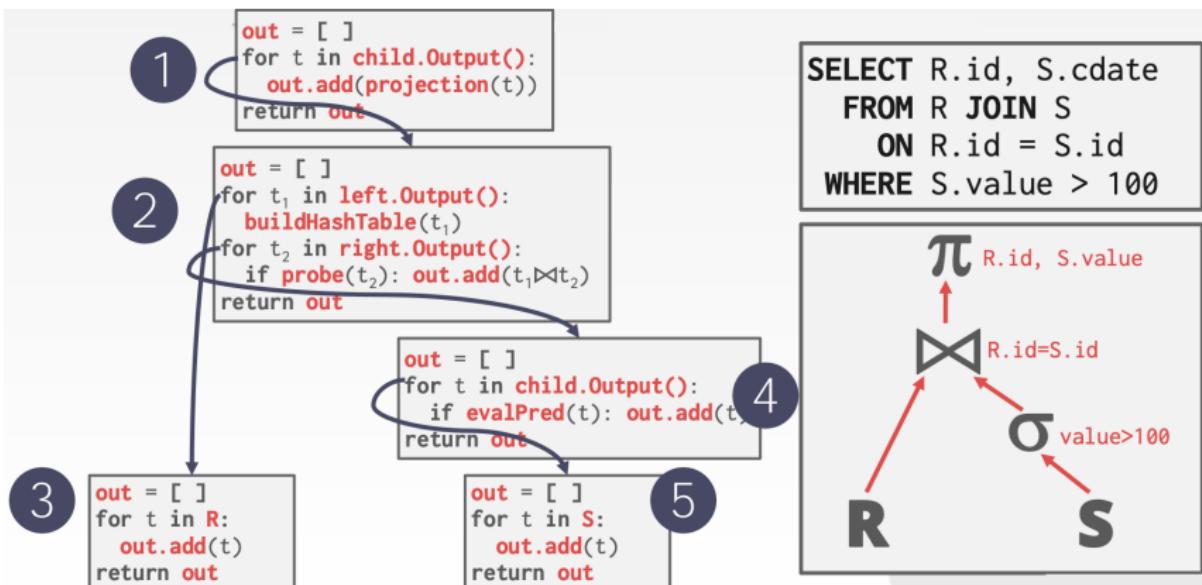
```
out = []
for t in child.Output():
    if evalPred(t): out.add(t)
return out
```

```
out = []
for t in S:
    out.add(t)
return out
```

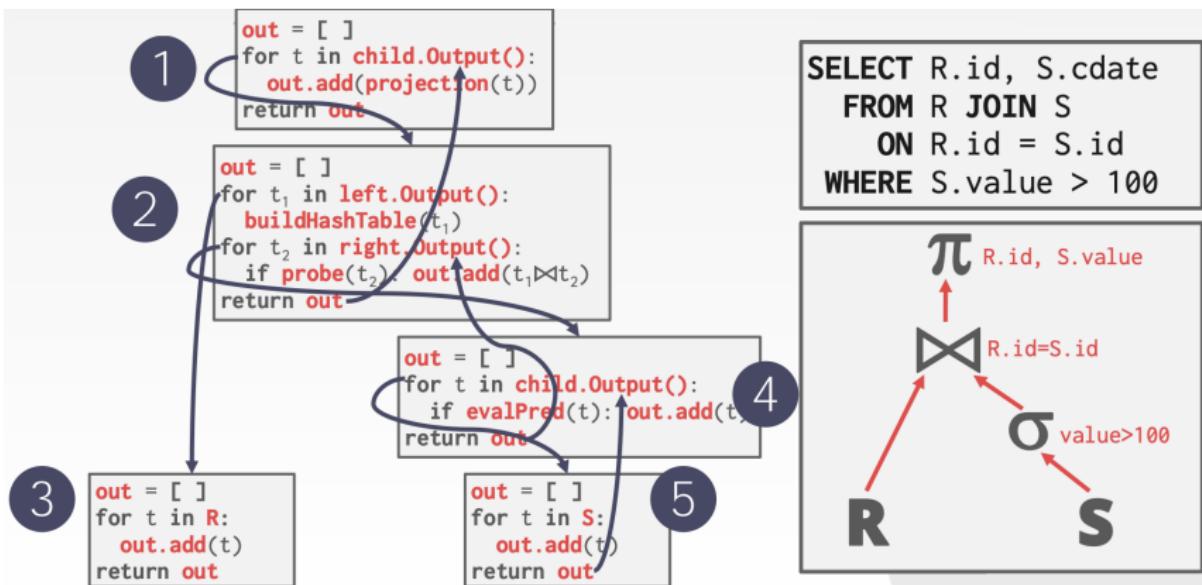
```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



Materialization Model



Materialization Model



Materialization Model

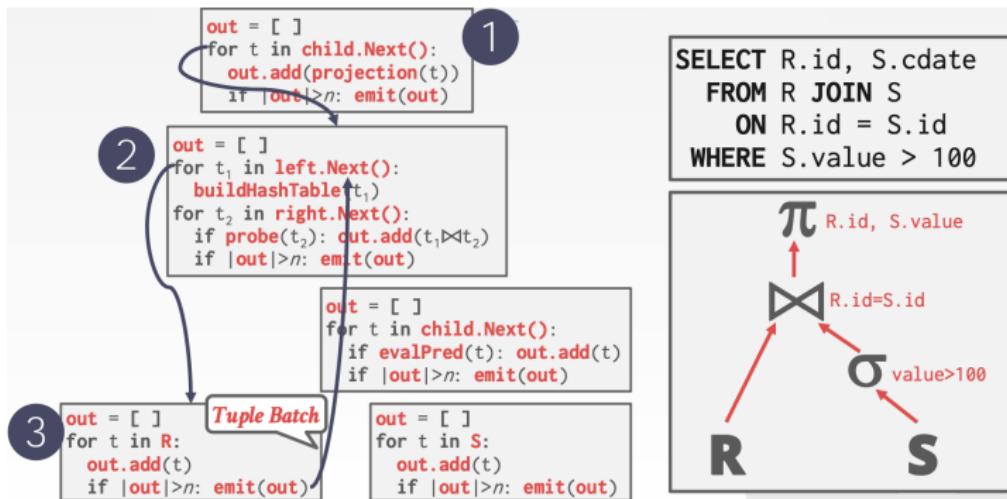
- This approach is better for OLTP workloads² because queries typically only access a small number of tuples at a time.
 - Thus, there are fewer function calls to retrieve tuples.
- Not good for OLAP queries with large intermediate results because the DBMS may have to spill those results to disk between operators.

²Online transaction processing (OLTP) captures, stores, and processes data from transactions in real time. Online analytical processing (OLAP) uses complex queries to analyze aggregated historical data from OLTP systems.

Vectorization Model

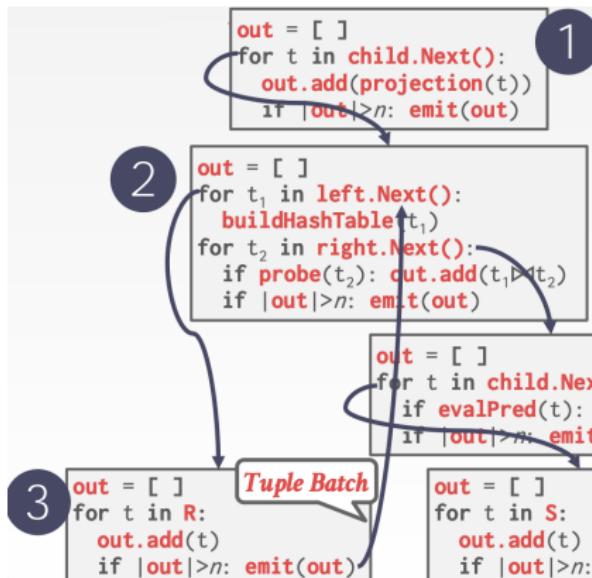
- Like the **iterator model** where each operator implements a **next** function.
- But each operator emits a batch (i.e., vector) of data instead of a single tuple:
 - The operator's internal loop processes multiple tuples at a time.
 - *The operator's internal loop implementation is optimized for processing batches of data instead of a single item at a time.*
 - The size of the batch can vary based on hardware or query properties. at a time.

Vectorization Model Example

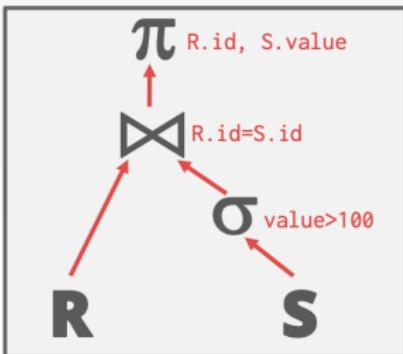


- The **vectorization model** is very similar to the **iterator model** except at every operator, an output buffer is compared to the desired emission size.
- If the buffer is larger, then a tuple batch is sent up.

Vectorization Model



```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



Vectorization Model

- This approach is ideal for OLAP queries that have to scan a large number of tuples because there are fewer invocations of the **next** function.
- Allows for operators to use vectorized (SIMD³) instructions to process batches of tuples.
 - Vectorization is a key tool for dramatically improving the performance of code running on modern CPUs.
 - Vectorization is the process of converting an algorithm from operating on a single value at a time to operating on a set of values at one time.
 - Modern CPUs provide direct support for vector operations where a single instruction is applied to multiple data (SIMD)

³ SIMD (Single Instruction Multiple Data) is an instruction set available mostly on all current processors. SIMD instructions give data-level parallelism on a unit (vector of data). A single instruction is executed in parallel on multiple data points as opposed to executing multiple instructions. This improves the performance of application.

Plan Processing Direction

- Approach #1: Top-to-Bottom

- Start with the root and **pull** data up from its children.
- Tuples are always passed with function calls.
- Much more common.

- Approach #2: Bottom-to-Top

- Start with leaf nodes and **push** data to their parents.
- Allows for tighter control of caches/registers in pipelines.

Estimating disk IOs

- Count # of disk blocks that must be read (or written) to execute query plan
- To estimate costs, we may have additional parameters
 - $B(R)$: # of blocks containing R tuples
 - $f(R)$: max # of tuples of R per block
 - M : # of memory buffers in main memory that is available to an operator
 - $HT(i)$: # levels in index i
 - $LB(i)$: # of leaf blocks in index i

The basic (relation) access operators and their cost

- An **access method** is the how the DBMS accesses the data stored in a table.
 - There is no corresponding operator in relational algebra.
- These will be the bottom operators in a query plan that **feed** data into the operators above it in the tree.
- In general, there are two approaches to access models:
 - **Sequential-Scan(R)**
 - Read tuples from the relation R by reading data blocks - one block at a time from disk to memory
 - **Index-Scan(R)**
 - The relation R must have an index
 - The index is scanned to find blocks that contain the desired tuples
 - All the blocks containing desired tuples are then read into the memory - one block at a time.
 - The **Multi-index** is an extension of the **index scan** that allows for multiple indexes to be accessed at once.

Sequential Scan

- For each page in the table:
 - Retrieve it from the buffer pool.
 - Iterate over each tuple and check whether to include it.
- The DBMS maintains an internal cursor that tracks the last page / slot it examined.
- This is almost always the worst thing that the DBMS can do to execute a query.

```
for page in table.pages:
    for t in page.tuples:
        if evalPred(t):
            // Do Something!
```

Sequential Scan Optimizations

- Prefetching
 - Fetches next few pages in advance so that the DBMS does not have to block when accessing each page.
- Parallelization
 - Execute the scan using multiple threads/processes in parallel.
- Buffer Pool Bypass
 - The scan operator stores pages that it fetches from disk in its local memory instead of the buffer pool. This avoids the **sequential flooding problem**.
- Zone Map
 - Pre-compute aggregations for each tuple attribute in a page.
 - The DBMS can then check whether it needs to access a page by checking its **Zone Map** first.
 - The **Zone Maps** for each page are stored in separate pages and there are typically multiple entries in each **Zone Map** page.
 - Thus, it is possible to reduce the total number of pages examined in a sequential scan.

Sequential Scan Optimizations

- Late Materialization

- DSM DBMS⁴ can delay stitching together tuples until the upper parts of the query plan.
- Each operator passes the minimal amount of information needed to the next operator (e.g., record id, offset to record in column).
- This is only useful in column-store systems (i.e., DSM).

- Heap Clustering

- Tuples are stored in the heap pages using an order specified by a clustering index.
- If the query accesses tuples using the clustering index's attributes, then the DBMS can jump directly to the pages that it needs.

⁴ Decomposition storage model/column store (DSM): The DBMS stores the values of a single attribute for all tuples contiguously in a page.

Index Scan

- The goal of an **index scan** is to identify an index in the table that will quickly allow the user to find the data he or she needs by avoiding useless operations.
- The DBMS picks an index (or indexes) to find the tuples that the query needs.
- Which index to use depends on:
 - What attributes the index contains
 - What attributes the query references
 - The attribute's value domains
 - Predicate composition
 - Whether the index has unique or non-unique keys

Index Scan

- Suppose that we have a single table with 100 tuples and two indexes:
 - Index #1: `age`
 - Index #2: `dept`

```
SELECT * FROM students
WHERE age < 30 AND dept = 'CS' AND country = 'US';
```

- Scenario #1

There are 99 people under the age of 30 but only 2 people in the CS department.

- Scenario #2

There are 99 people in the CS department but only 2 people under the age of 30.

Multi-Index Scan

- More advanced DBMSs can support **multi-index** scans.
- If there are multiple indexes that the DBMS can use for a query:
 - Compute sets of record ids using each matching index.
 - Combine these sets based on the query's predicates (union vs. intersect).
 - Retrieve the records and apply any predicates that may remain.
- Postgres calls this **Bitmap Scan**

Multi-Index Scan

```
SELECT * FROM students
WHERE age < 30 AND dept = 'CS' AND country = 'US';
```

- With an index on `age` and an index on `dept`,
 - We can retrieve the record ids satisfying `age<30` using the first,
 - Then retrieve the record ids satisfying `dept='CS'` using the second,
 - Take their intersection
 - Retrieve records and check `country='US'`.
- Set intersection can be done with bitmaps, hash tables, or Bloom filters.

Clustered and unclustered files/relations/indexes

- **Clustered file:** A file that stores records of one relation
- **Unclustered file:** A file that stores records of multiple relations
- **Clustered index:** Index that allows tuples to be read in an order that corresponds to physical order

The cost of the basic scan operators

- Relation R is **clustered**

Operator	I/O cost	Explanation
Sequential-Scan	$B(R)$	Read all blocks, and there are $B(R)$ blocks.
Index-Scan	$\leq B(R)$	Depends on number of values in the index is scanned

- Relation R is **unclustered**

Operator	I/O cost	Explanation
Sequential-Scan	$\sim T(R)$	Assuming next tuple is not found in the current block. We will read 1 block per tuple
Index-Scan	$\leq T(R)$	Depends on number of values in the index is scanned

Index Scan Page Sorting

- Retrieving tuples in the order that appear in an **unclustered index** is inefficient.
- The DBMS can first figure out all the tuples that it needs and then sort them based on their page id.

Categories of the query processing algorithms

- The algorithms in query processing can be broadly categorized as follows
 - One-pass algorithms
 - 2-pass algorithms
 - Multi-pass algorithms

One-pass algorithms

- Read input relations just once
- Use the **lowest** disk I/O operations
- But, also have the highest memory space requirements.
 - Store one relation in memory

2-pass algorithms

- Read input relations 2 times
 - The first time, the operation performs a preparation step (e.g., sort) and write the result to disk
 - The second time, the prepared result is read and the actual operation is performed
- Use a **large** number disk I/O operations
- But, also have a lower memory space requirements
 - They do not store entire relation in memory.
 - Only a portion of the relation is read

Multi-pass algorithms

- Read input relations more than 2 times
- They use the highest disk I/O operations
- But, they also have the lowest memory space requirements

General guideline for picking an algorithm

- If you have sufficient memory space to run a **one-pass algorithm**, do that.
- Otherwise, check if you have sufficient memory space to run a **2-pass algorithm**. If so, do that.
- Otherwise, run a **multi-pass algorithm**
- Note
 - These general guidelines may need to be modified if you can use an index to speed up the access to the tuples/records

Disk-Oriented DBMS

- Just like it cannot assume that a table fits entirely in memory, a disk-oriented DBMS cannot assume that the results of a query fits in memory.
- We are going use on the buffer pool to implement algorithms that need to spill to disk.
- We are also going to prefer algorithms that maximize the amount of sequential access.

Operators Overview

- (External) Sorting
- Joins (Nested Loop, Merge, Hash, ...)
- Aggregation (Sorting, Hash)
- Selection, Projection (Index, Scan)
- Union, Set Difference
- Intersection
- Duplicate Elimination

Sorting

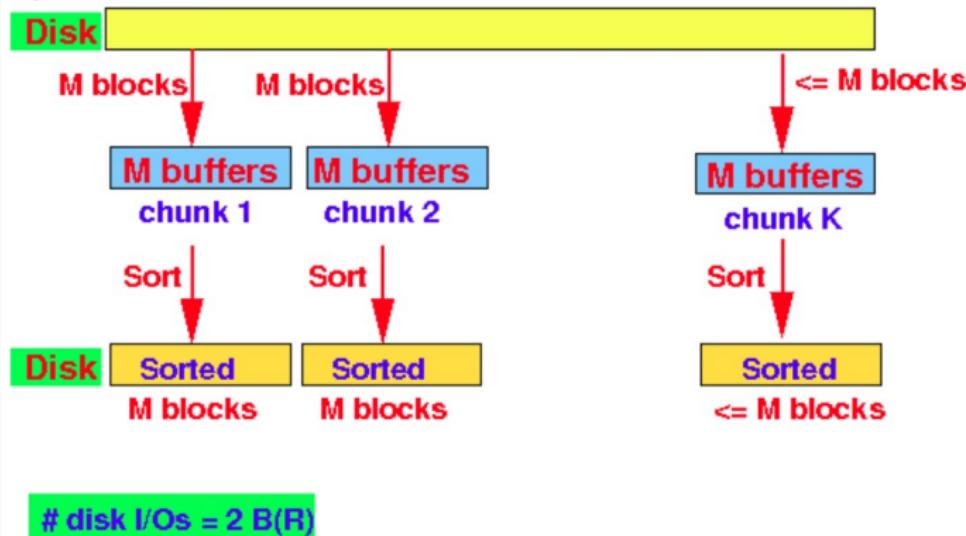
- Why do we want/need to sort
 - Query requires sorting (ORDER BY)
 - Operators require sorted input
 - Merge-join
 - Aggregation by sorting
 - Duplicate removal using sorting
- Traditional sort algorithms (e.g., quick sort) requires that the file must fit in the main memory to be sorted
- The [Two-Pass Multiway Merge Sort](#) (TPMMS) algorithm can be used to sort files that are larger than the main memory

The TPMMS algorithm

- Suppose
 - There are M buffers available for storing the file data
 - 1 buffer can hold 1 data block
- **Pass 1**
 - Divide the input file into chunks of M blocks each
 - Sort each chunk individually using the M buffers
 - Write the sorted chunks to disk

The TPMMS algorithm: Pass 1

input file R:

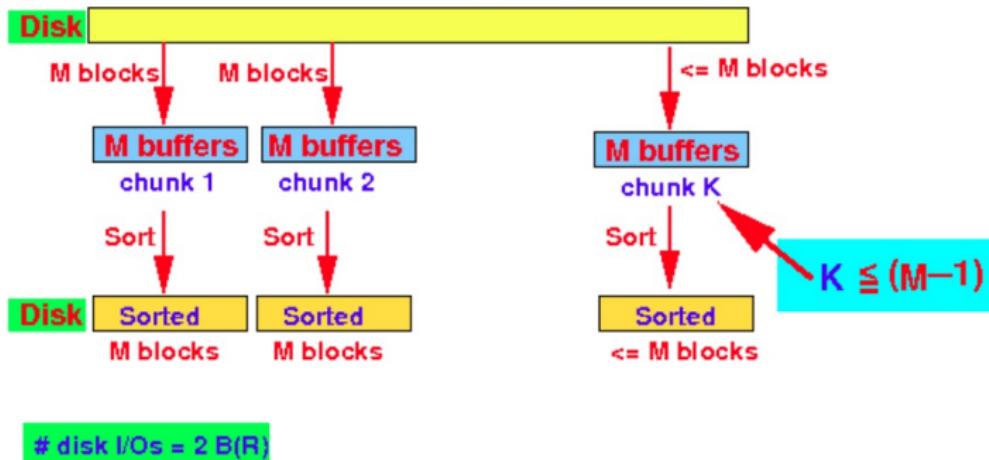


- Number of disk IOs performed in **Pass 1**:
 - We read the input file one time and
 - We write fragments that total equal to the input file
- Number of disk IOs performed in **Pass 1**: **2 B(R)**

The TPMMS algorithm: Constraint

- The number of chunks ($K \leq M-1$)
- This constraint is imposed by **Pass 2** of the TPMMS algorithm

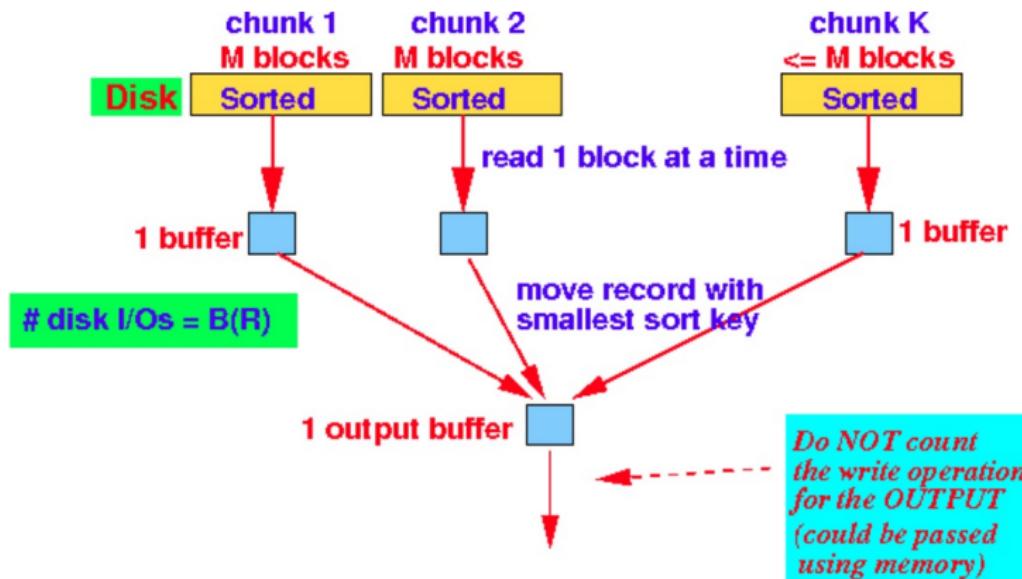
input file R:



The TPMMS algorithm: Pass 2

- Divide the M buffers into
 - $M - 1$ input buffers (use 1 buffer to read a sorted chunk)
 - 1 output buffer (to write the total sorted output)
- Because we only have $M-1$ input buffers to read the K sorted chunks, we have the constraint that:
 - $\# \text{ sorted chunks } K \leq M - 1$
- How to use 1 output buffer to merge sort the K sorted chunks into one sorted output file:
 - Find the record with the smallest sort key among the K buffers
 - Move the record with the smallest sort key to the output buffer
 - If the output buffer is full, then write (empty) the output buffer to disk
 - If some input buffer is empty
 - Read the next (sorted) block from the sorted chunk if there is more data
 - If there is no more data in the chunk, then ignore the chunk in the merge operation
 - Repeat until data in all chunks are processed

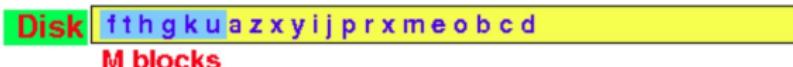
The TPMMS algorithm: Pass 2



The TPMMS algorithm: Example

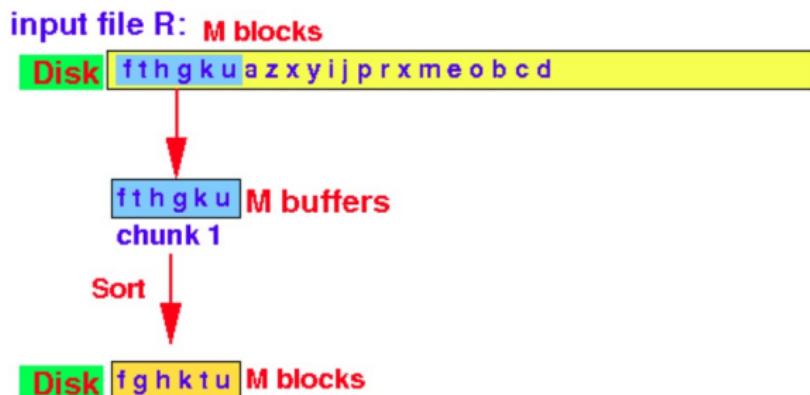
- Sort the following input file

input file R:



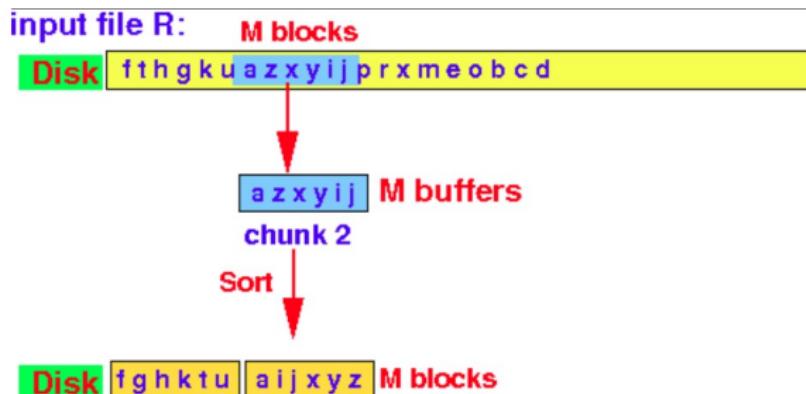
The TPMMS algorithm: Example: Pass 1

- Step 1: sort first chunk of M blocks



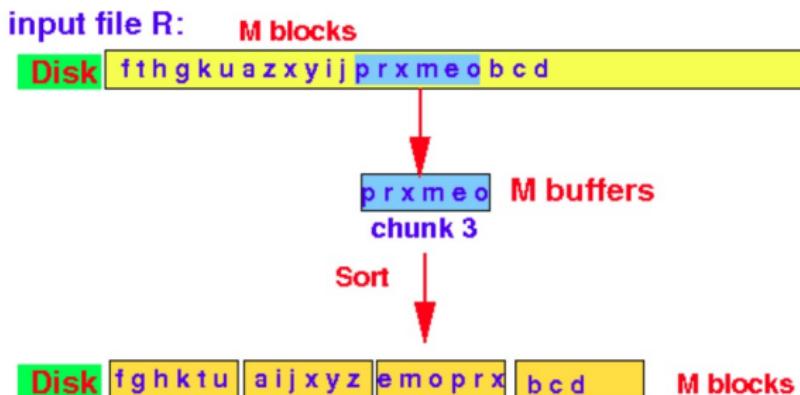
The TPMMS algorithm: Example: Pass 1

- Step 2: sort second chunk of M blocks



The TPMMS algorithm: Example: Pass 1

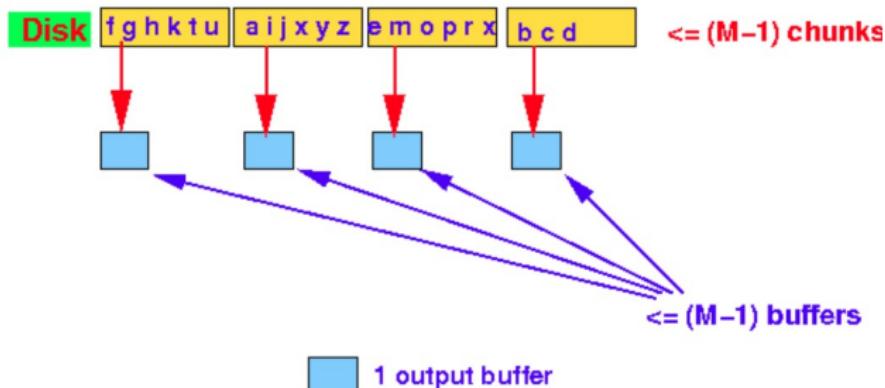
- And so on



The TPMMS algorithm: Example: Pass 2

- Use 1 buffer to read each chunk and use 1 buffer for output

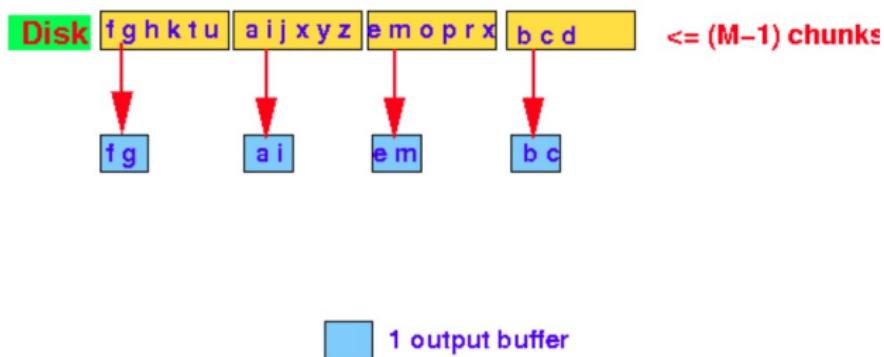
Using M buffers:



The TPMMS algorithm: Example: Pass 2

- Read each chunk 1 block at a time

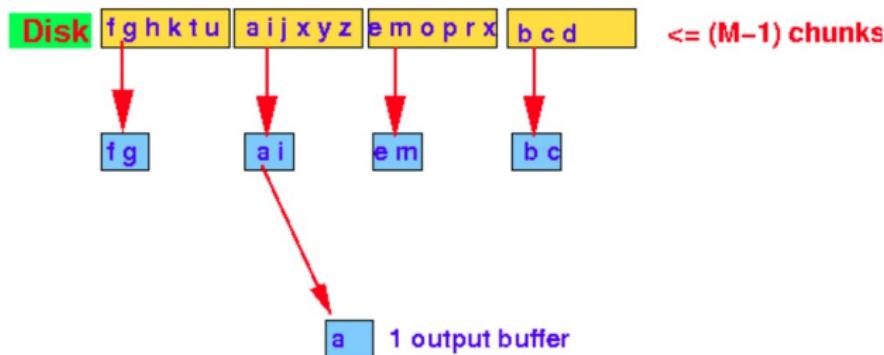
Using M buffers:



The TPMMS algorithm: Example: Pass 2

- Move the smallest element to the output buffer

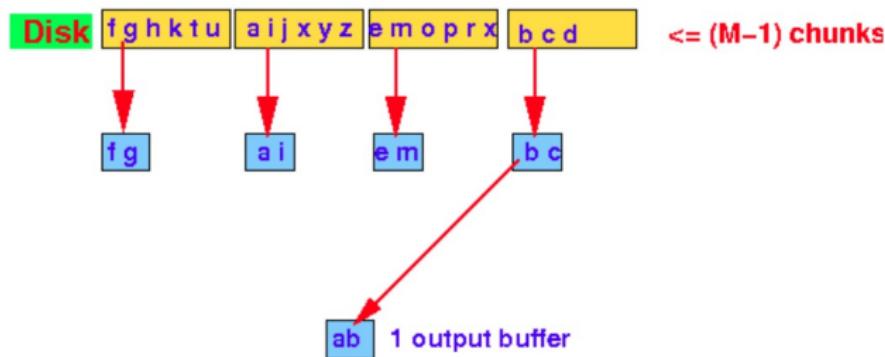
Using M buffers:



The TPMMS algorithm: Example: Pass 2

- And so on

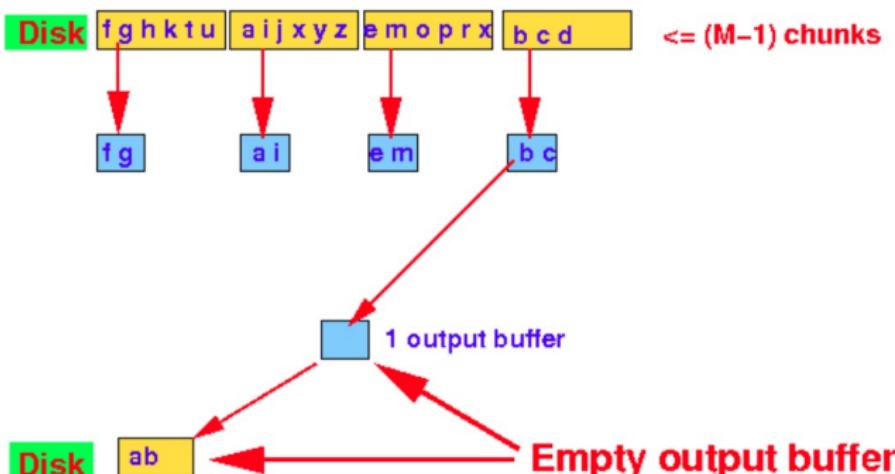
Using M buffers:



The TPMMS algorithm: Example: Pass 2

- When output buffer is full, empty it for re-use

Using M buffers:



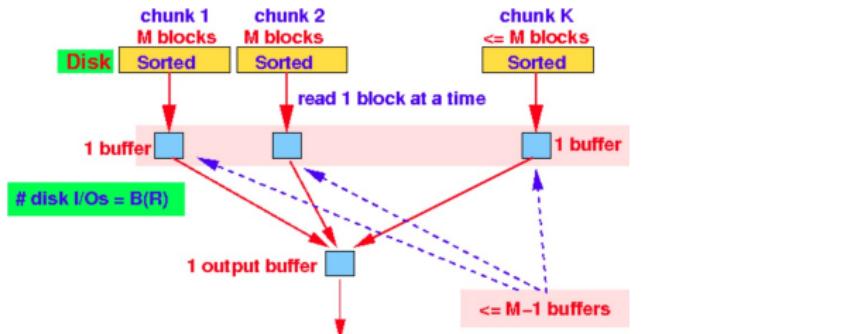
- And so on

File size constraint on the TPMMS algorithm

- File size constraint:
 - The maximum size of a file that can be sorted by the TPMMS algorithm is $B(R) \leq M(M - 1)$

File size constraint on the TPMMS algorithm: Reason

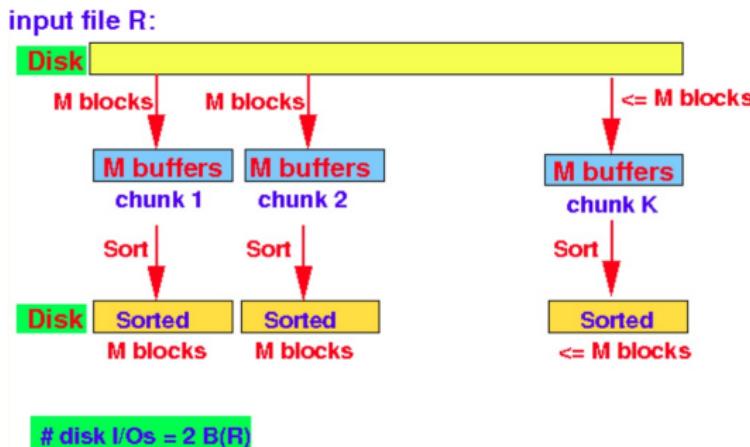
- In **Pass 2**, we can allocate at most $M-1$ buffers to read the sorted chunks



- Therefore, the number of chunks that can be generated by **Pass 1** must be $\leq M - 1$

File size constraint on the TPMMS algorithm: Reason

- Each chunk has the size of M blocks



- Therefore, the maximum file size is

- number of chunks in **Pass 1** $\leq M - 1$
- 1 chunk = M blocks
- $\therefore \text{File size} \leq (M - 1) \times M \text{ blocks}$
 $B(R) \leq M^2$.

⇒ The memory constraint required to run the TPMMS algorithm is
 $M \geq \sqrt{B(R)}$

Cost of running TPMMS of a relation R

- Cost of TPMMS on relation $R = 3 \times B(R)$
- But, we do not include the output (write) cost in the total because we could use pipelining to pass the tuples to the next operator
- Note
 - If the result is to be written to disk (i.e., materialize), then cost of TPMMS on relation $R = 4 \times B(R)$

TPMMS can sort very large files

- Suppose
 - 1 block = 64 K bytes
 - Memory size = 1 G bytes
- $M = \frac{1G}{64K} \text{ buffers} = \frac{1,024 \times 1,024 \times 1,024}{64 \times 1,024} = 16,384 \text{ buffers}$
- \Rightarrow Max file size that can be sorted

$$\begin{aligned}B(R) &\leq M(M - 1) \text{ blocks} \\&\leq 16,384 (16,384 - 1) \\&\leq 268,419,072 \text{ blocks } (1 \text{ block} = 64 \text{ K}) \\&\leq 255984000 \times 64 \text{ K} \\&\leq 17,178,820,608 \text{ K} \\&\cong 17 \text{ TeraBytes}\end{aligned}$$

When to use 2-Pass algorithms in relational algebra operations

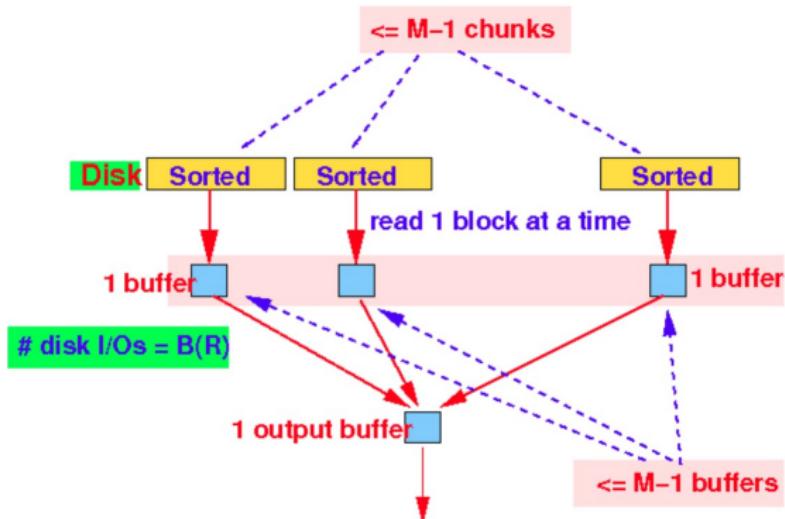
- Unary operator
 - If $B(R) \geq \text{available \# buffers } (M)$
then use a 2-Pass algorithm (if $B(R) \leq M^2$)
- Binary operator
 - If $B(R) \geq \text{available \# buffers } (M)$ and $B(S) \geq \text{available \# buffers } (M)$
then use a 2-Pass algorithm (if $B(R)+B(S) \leq M^2$)
- Multi-Pass MMS
 - The 2-Pass multiway merge sort can be generalized to multi-Pass (3 Passes or more)
 - These algorithms can sort a file of arbitrary size

The multi-pass multiway merge sort algorithm

- Recall the 2-pass multiway sort (TPMMS) algorithm
 - Pass 1
 - Divide the input file into chunks of M blocks each
 - Sort each chunk individually using the M buffers
 - Write the sorted chunks to disk
 - Requirement: The number of chunks (K) $\leq M - 1$
 - Pass 2
 - Divide the M buffers into: $M - 1$ input buffers and 1 output buffer
 - Use the $M - 1$ input buffers to read the K sorted chunks (1 block at a time)
 - Merge sort the K sorted chunks together into a sorted file using 1 output buffer as follows:
 - Find the record with the smallest sort key among the K buffers
 - Move the record with the smallest sort key to the output buffer
 - When the output buffer is full, then write the output buffer to disk
 - When some input buffer is empty, then read another block from the sorted chunk if there is more data

An important observation of the TPMMS algorithm

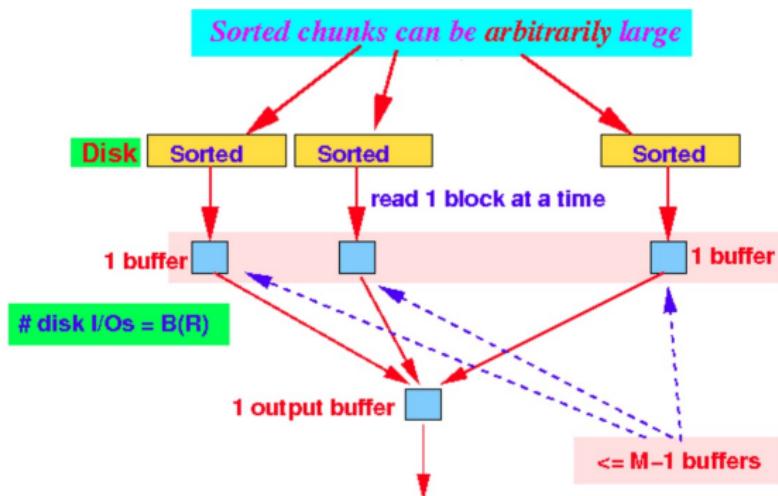
- Consider the Pass 2 in the TPMMS algorithm



- The restriction is # (sorted) chunks $\leq M - 1$ because # buffers used must be $\leq M$

An important observation of the TPMMS algorithm

- There is no restriction on the size of a (sorted) chunk

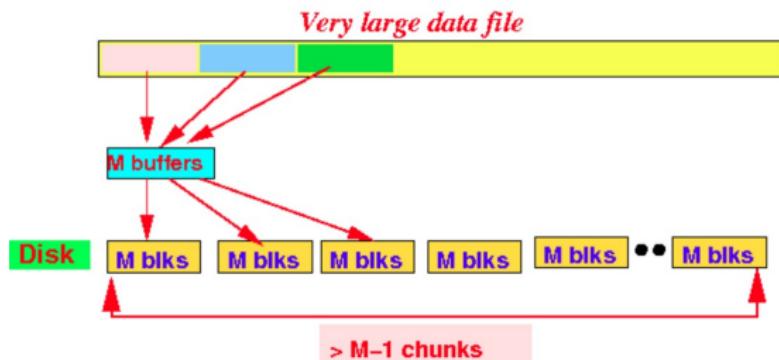


Increasing the size of sorted chunks

- Fact: We can use the M buffers to merge sort (any number) $\leq M - 1$ sorted chunks into one larger (sorted) chunk
- Suppose we have a very large file:



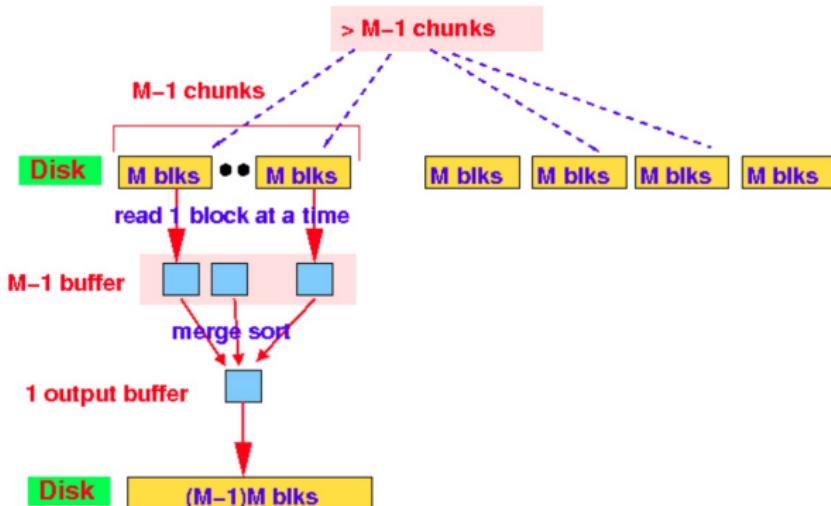
- We first use M buffers to sort the file into chunks of M blocks



- Suppose we get $> (M - 1)$ chunks

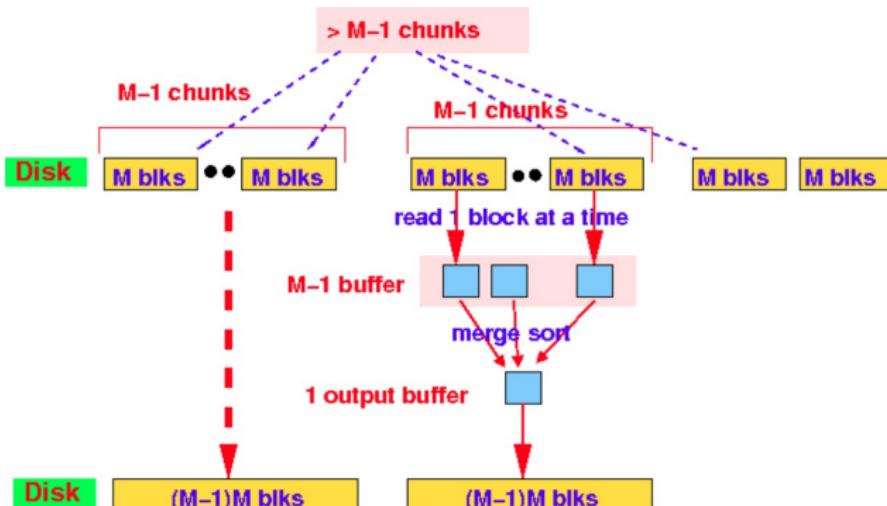
Increasing the size of sorted chunks

- We (re)-use the M buffers to merge the first $(M - 1)$ chunks into a chunk of size $M(M - 1)$ blocks



Increasing the size of sorted chunks

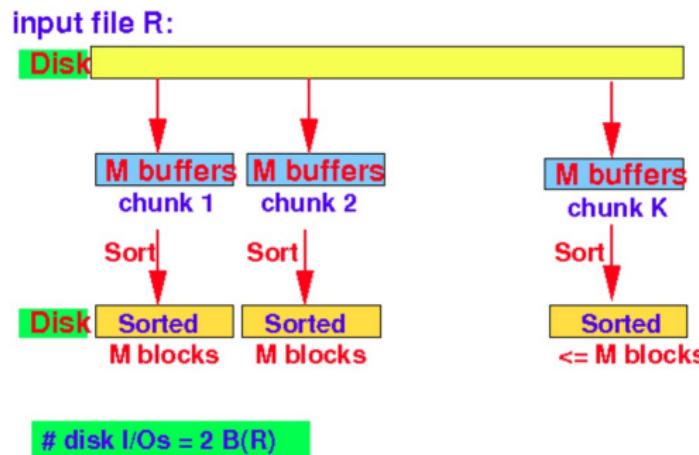
- Then, we (re)-use the M buffers to merge the 2^{nd} ($M - 1$) chunks into a chunk of size $M(M - 1)$ blocks



- And so on. We will have large sorted chunks

A 3-Pass Multiway Sort Algorithm

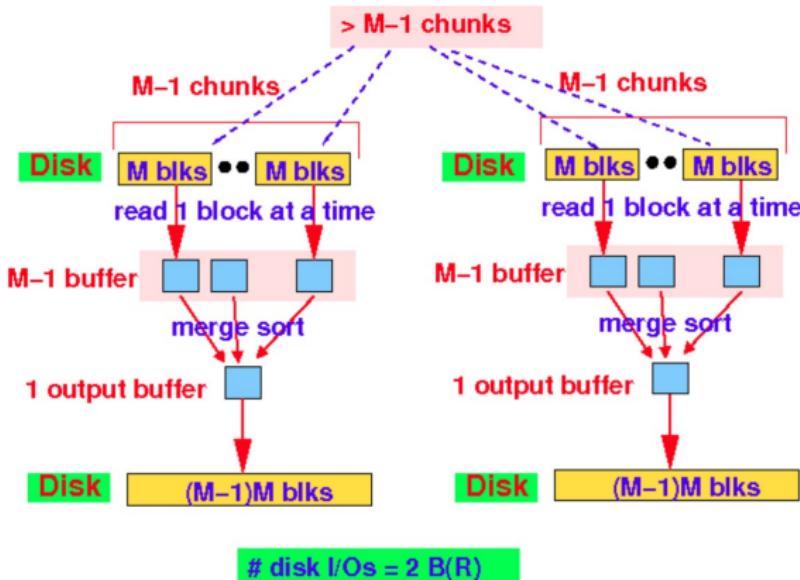
- Pass 1: (same as Phase 1 in TPMMS)
 - Divide the input file into chunks of M blocks each
 - Sort each chunk individually using the M buffers
 - Write the sorted chunks to disk



- Number of disk I/Os used in step:
 - $B(R)$ disk I/Os to read the entire input file plus
 - $B(R)$ disk I/Os to write the entire file (= all the sorted chunks)

A 3-Pass Multiway Sort Algorithm

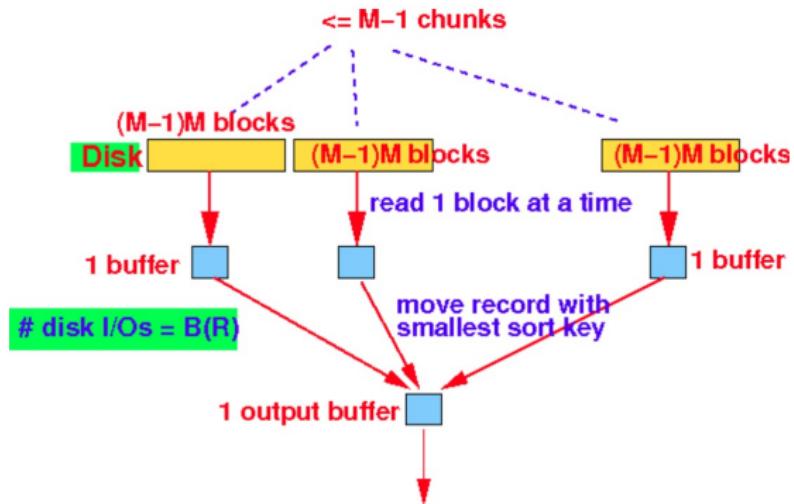
- Pass 2: merge groups of $(M - 1)$ sorted chunks into a $M(M - 1)$ block sorted “super” chunk and write sorted “super” chunk to disk



- Number of disk I/Os used in step:
 - $B(R)$ disk I/Os to read the entire file (= all the chunks in the file) plus
 - $B(R)$ disk I/Os to write the entire file (= all the “super” chunks in file)

A 3-Pass Multiway Sort Algorithm

- Pass 3: merge upto $(M - 1)$ (much larger) sorted chunks



A 3-Pass Multiway Sort Algorithm

- Maximum file that can be handled by the 3-Pass Multiway Sort:
 - There are no memory restrictions on **Pass 1** and **Pass 2**
 - **Pass 3** must merge: $\leq M - 1$ chunks
 - Each chunk in **Pass 3** has size $\leq M \times (M - 1)$ blocks
 - Maximum file size $\leq M \times (M - 1)^2$ blocks
- Number of disk IOs used
 - **Pass 1:** read R + write sorted chunks = $2 \times B(R)$
 - **Pass 2:** read sorted chunks + write bigger chunks = $2 \times B(R)$
 - **Pass 3:** read bigger chunks + sort = $1 \times B(R)$
 - **Total = $5 \times B(R)$** , (let's not count final output write to disk)

k-Pass Multiway Merge Sort

- **Pass 1**
 - Same as Pass 1 of TPMMS
 - Cost: $2 \times B(R)$ disk IOs
 - Size of each chunk at end: M blocks
- **Pass 2, 3, ..., k-1 ((k-2) passes):**
 - Use the “increase chunk size” algorithm
 - Cost per pass: $2 \times B(R)$ disk IOs
 - Cost for $k-2$ passes: $2(k-2) \times B(R)$ disk IOs
 - Size increase per pass: $(M - 1)$ times
 - Size of chunks at end: $M \times (M - 1)^{k-2}$ blocks
- **Pass k**
 - Same as Pass 2 of TPMMS
 - Cost: $B(R)$ disk IOs (do not count output)
 - Size of sorted file $\leq M \times (M - 1)^{k-1}$ blocks

k-pass multiway merge sort

- Total # disk IOs performed by the k-Pass multiway merge algorithm
 - $\# \text{ disk IOs} = (2k - 1) \times B(R)$ (if we don't count final output IO)
Or:
 - $\# \text{ disk IOs} = 2 \times k \times B(R)$ (if we include final output IO)
 - $\text{Max file size} \leq M \times (M - 1)^{k-1}$ blocks

Operators Overview

- (External) Sorting
- Joins (Nested Loop, Merge, Hash, ...)
- Aggregation (Sorting, Hash)
- Selection, Projection (Index, Scan)
- Union, Set Difference
- Intersection
- Duplicate Elimination

Joins

- Example $R \bowtie S$ over common attribute C
- $T(R) = 10,000$
- $T(S) = 5,000$
- $S(R) = S(S) = \frac{1}{10}$ blocks (each block 10 tuples)
- Memory available = 101 blocks
- Metric: # of IOs (ignoring writing of result)

Caution

- This may not be the best way to compare
 - ignoring CPU costs
 - ignoring timing
 - ignoring double buffering requirements

Options

- Transformations: $R \bowtie_C S, S \bowtie_C R$
- Join algorithms
 - Iteration (nested loops)
 - Merge join
 - Join with index
 - Hash join
- Factors that affect performance
 1. Tuples of relation stored physically together?
 2. Relations sorted by join attribute
 3. Indexes exist?

Nested-Loop Joins

- We now consider algorithms for the join operator
- The simplest one is the **nested-loop join**, a one-and-a-half pass algorithm.
- One table is read once, the other one multiple times.
- It is not necessary that one relation fits in main memory
- Perform the join through two nested loops over the two input relations.

Tuple-based Nested-loop Join

Algorithm 1: Tuple-based Nested-loop Join

```
1 for each tuple  $r \in R$  do
2   for each tuple  $s \in S$  do
3     if  $(r.C = s.C)$  then
4       output  $(r, s)$ 
5     end
6   end
7 end
```

- Advantage:
 - Outer relation R , inner relation S .
 - Requires only 2 input buffers. 1 buffer to read tuples for relation R and 1 buffer to read tuples for relation S
- Applicable to
 - Any join condition C
 - Cross-product

Example 1(a): Tuple-based Nested Loop Join $R \bowtie S$

- Relations not clustered
- Recall:
 - $T(R) = 10,000$
 - $T(S) = 5,000$
 - Memory available = 101 blocks
- R as the outer relation
- Cost for each R tuple r : [Read tuple r + Read relation S] = 1+5,000
- Total IO cost = $10,000 \times (1+5,000) = 50,010,000$ IOs
- Can we do better?

Block-based nested loop join algorithm

- Can do much better by organizing access to both relations by blocks.
 - Use as much buffer space as possible ($M - 1$) to store tuples of the outer relation.
- Use $(M - 1)$ buffers to read and index data blocks from the smaller relation S
- Use 1 buffer to read data blocks from the larger relation R and compute the Join result

Algorithm 2: Block-based Nested-loop Join

```
1 for each  $M-1$  blocks of  $S$  do
2   Organize these tuples into a search structure (e.g., hash table)
3   for each block  $b$  of  $R$  do
4     Read  $b$  into main memory
5     for each tuple  $t \in$  block  $b$  do
6       - Find the tuples  $s_1, s_2, \dots$  of  $S$  (in the search structure) that join
        with  $t$ 
7       - Output  $(t, s_1), (t, s_2), \dots$ 
8     end
9   end
10 end
```

Example 1(a): Block-based Nested Loop Join $R \bowtie S$

- Relations not contiguous
- Recall:
 - $T(R) = 10,000$
 - $T(S) = 5,000$
 - $S(R) = S(S) = \frac{1}{10}$ blocks
 - Memory available = 101 blocks
- R as the outer relation
- 100 buffers for R , 1 buffer for S
- cost for each R chunk:
 - read chunk: 1,000 IOs
 - read S : 5,000 IOs
- 10 R chunks
- Total I/O cost is $10 \times 6,000 = 60,000$ IOs

Can we do better?

- Reverse join order $S \bowtie R$
- Cost when using smaller relation S in the outer loop
- 100 buffers for S , 1 buffer for R
- cost for each S chunk:
 - read chunk: 1,000 IOs
 - read R : 10,000 IOs
- 5 S chunks
- Total I/O cost is $5 \times 11,000 = 55,000$ IOs
- In general, there is a slight advantage to using the smaller relation in the outer loop.

Example 1(b): Block-based Nested Loop Join $R \bowtie S$

- Performance is dramatically improved when input relations are clustered (read by block).
- With clustered relations, for each S chunk:
 - read chunk: 100 IOs
 - read R : 1,000 IOs
- 5 S chunks
- Total I/O cost is $5 \times 1,100 = 5,500$ IOs

Two-Pass Algorithms Based on Sorting: Sort-merge Join

- If the input relations are sorted, the efficiency of duplicate elimination, set-theoretic operations and join can be greatly improved.
- In the following, we present a simple **sort-merge join** algorithm.
- It is called **merge-join**, if step/phase (1) can be skipped, since the input relations R and S are already sorted.

Two-Pass Algorithms Based on Sorting: Sort-merge Join

- Phase 1: perform a complete TPMMS on both relations and materialize the result on disk
 - Relation R : Sort relation R using the TPMMS algorithm
 - Relation S : Sort relation S using the TPMMS algorithm
- Phase 2: join the sorted relations:
 - Use 1 buffer to read relation R . The smallest join value may occupy more than 1 buffer
 - Use 1 buffer to read relation S . The smallest join value may occupy more than 1 buffer
 - If necessary (= when smallest join value may occupy more than 1 buffer):
 - Use the remaining $M - 2$ buffers to store tuples in R and/or S that contain all smallest joining attribute values

Two-Pass Algorithms Based on Sorting: Sort-merge Join

Algorithm 3: Sort-merge join

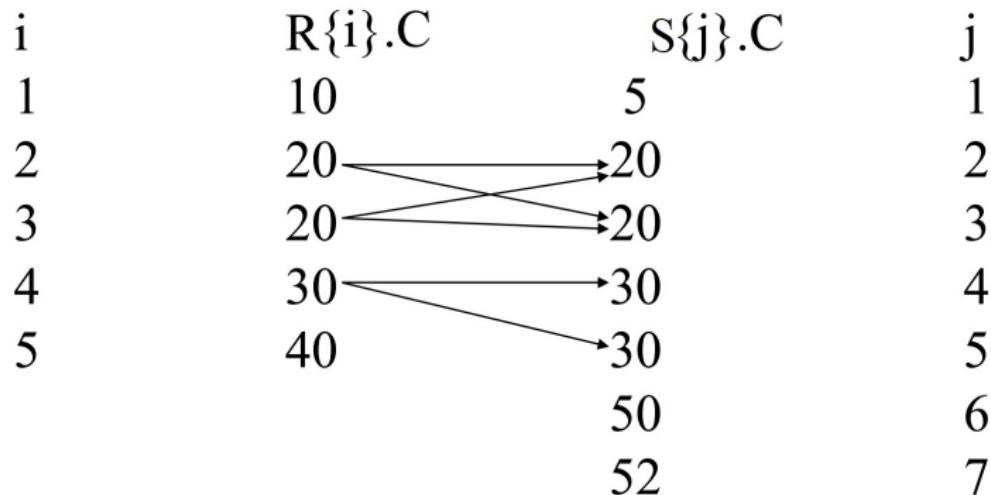
```
1 if R and S not sorted then
2   | sort them
3 end
4 i ← 1; j ← 1;
5 while ((i ≤ T(R) ∧ j ≤ T(S)) do
6   | if (R{i}.C = S{j}.C) then
7     |   | outputTuples
8   | else if (R{i}.C > S{j}.C) then
9     |   |   j ← j+1
10  | else
11    |   |   i ← i+1
12  | end
13 end
```

Algorithm 4: Output-Tuples

```
1 while (R{i}.C = S{j}.C) ∧ i ≤ T(R) do
2   | k ← j;
3   | while (R{i}.C = S{k}.C) ∧ k ≤ T(S) do
4     |   | output pair R{i}, S{k}
5     |   |   k ← k + 1
6   | end
7   |   i ← i+1
8 end
```

- Procedure `outputTuples` produces all pairs of tuples from R and S with $R\{i\}.C = S\{j\}.C$
- In the worst case, need to match each pairs of tuples from R and S (nested-loop join).

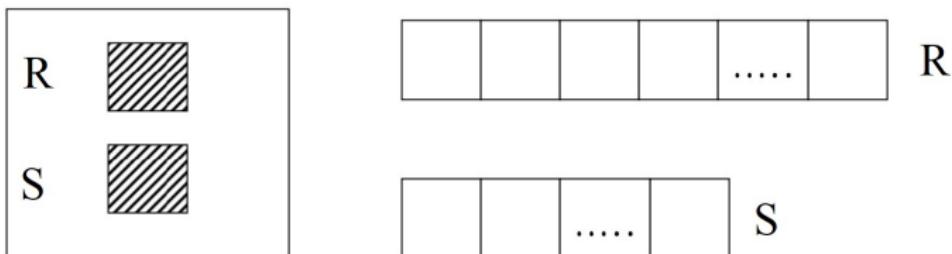
Sort-merge Join: Example



Example 1(c) Merge Join

- Both R, S ordered by C; relations contiguous

Memory



- Total cost: Read R cost + read S cost = $1,000 + 500 = 1,500$ IOs

Example 1(d) Merge Join

- R, S not ordered but contiguous
- Do Two-Phase Multiway Merge-Sort (TPMMS).
 - IO cost is $4 \times B(R)$, if sorting is used as a first step of `sort-join` and the results must be written to the disk.
 - If relation R is too big, apply the idea recursively.
 - Divide R into chunks of size $M(M-1)$, use TPMMS to sort each one, and take resulting sorted lists as input for a third (merge) phase.
 - This leads to Multi-Phase Multiway Merge Sort.

Example 1(d) Merge Join

- R, S not ordered but contiguous
- Sort cost: each tuple is read, written, read, written
- Join cost: each tuple is read
- Sort cost R: $4 \times 1,000 = 4,000$
- Sort cost S: $4 \times 500 = 2,000$
- Total cost = sort cost + join cost = $6,000 + 1,500 = 7,500$ IOs
- Total cost = $5(B(R) + B(S))$

Note

- Nested loop join (best version discussed above) needs only 5,500 IOs, i.e. outperforms sort-join.
- However, the situation changes for the following scenario:
 - $R = 10,000$ blocks
 - $S = 5,000$ blocks
 - Both R, S clustered, not ordered
 - Nested-loops join: $\frac{5,000}{100} \times (100 + 10,000) = 50 \times 10,100 = 505,000$ IOs
 - Merge join: $5 \times (10,000 + 5,000) = 75,000$ IOs
 - Sort-join clearly outperforms nested-loop join

Two-Pass Algorithms Based on Sorting: Sort-merge Join

- Simple sort-join costs $5(B(R) + B(S))$ IOs.
- It requires $M \geq \sqrt{B(R)}$ and $M \geq \sqrt{B(S)}$
- It assumes that tuples with the same join attribute value fit in M blocks.

Can we improve on merge join?

- Hint: do we really need the fully sorted files?
- If we do not have to worry about large numbers of tuples with the same join attribute value, then we can combine the second phase of the sort with the actual join (merge).
- We can save the writing to disk in the sort step and the reading in the merge step.

Advanced Sort-merge Join

- This algorithm is an **advanced sort-merge join**.
- Repeatedly find the least C-value c among the tuples in all input buffers.
- Instead of writing a sorted output buffer to disk, and reading it again later, identify all the tuples of both relations that have $C=c$.
- Cost is only $3(B(R)+B(S))$ IOs.
- Since we have to simultaneously sort both input tables and keep them in memory, the memory requirements are getting larger:

$$M \geq \sqrt{B(R)+B(S)}$$

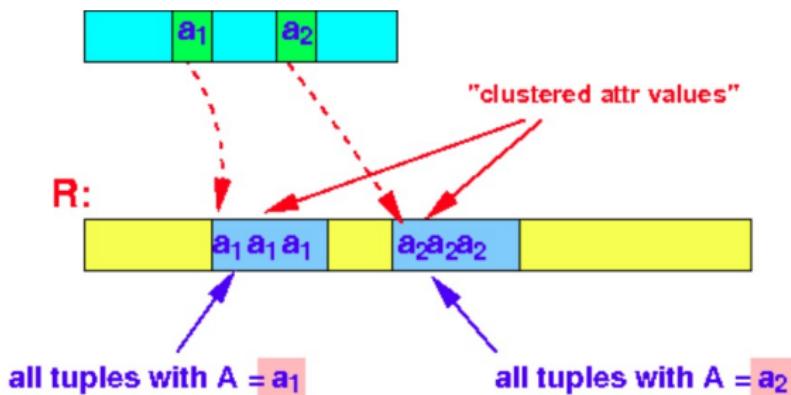
Index-based algorithms

- Index-based algorithms are especially useful for the selection operator, but also for the join operator.
- We distinguish clustering and non-clustering indexes.
- A clustering index is an index where all tuples with a given search key value appear on (roughly) as few blocks as possible.
- One relation can have only one clustering index, but multiple non-clustering indexes.

Index-based algorithms

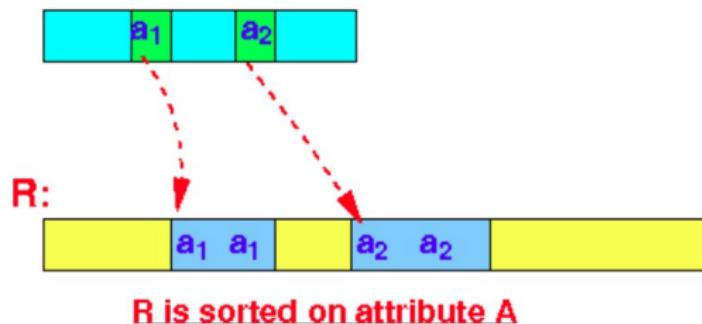
- Clustering index

- An index on attribute(s) A on a file is a clustering index when all tuples with attribute value $A = a$ are stored sequentially (= consecutively) in the data file



Index-based algorithms

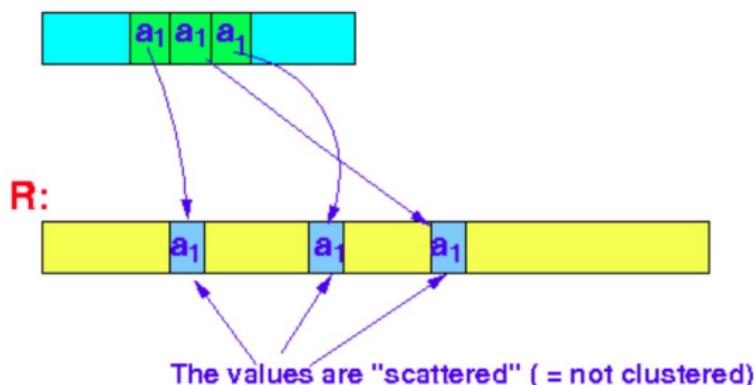
- Common example of a clustering index when the relation R is sorted on the attribute(s) A



Index-based algorithms

- Non-clustering index
 - an index that is not clustering

Non-clustering index on A



Join algorithm for a clustering index

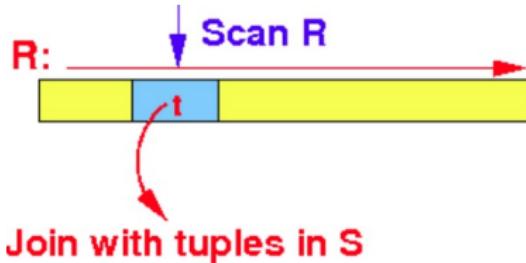
- Assume S.C index

Algorithm 5: Join using index S.C

```
1 Read a block of R in b
2 for each tuple  $t_r \in b$  do
3   Use  $t_r.C$  to lookup in index S.C
4   // You get a list of record addresses
5   for each record address  $s$  do
6     read tuple at  $s$ 
7     output  $t_r, t_s$  pair // Join result
8   end
9 end
```

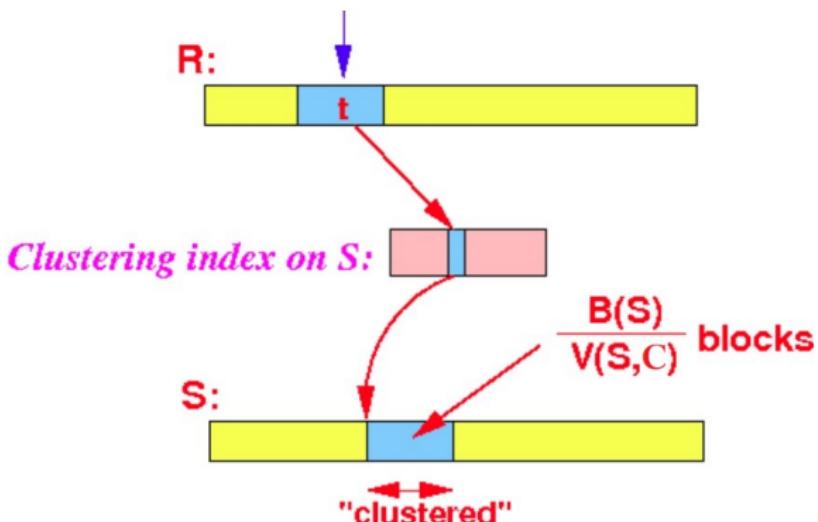
Join algorithm for a clustering index: Performance

- The join algorithm will scan the relation R once



Join algorithm for a clustering index: Performance

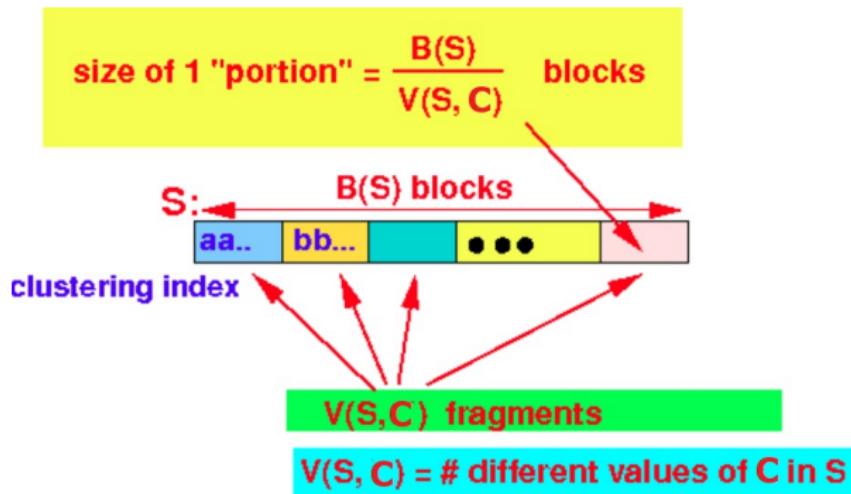
- For each tuple $t \in R$, we read the following portion in relation S



- For each tuple $t \in R$, we read (= access) this portion in S
 - portion of S read per tuple in R = $\frac{B(S)}{V(S,C)}$ blocks

Join algorithm for a clustering index: Performance

- because

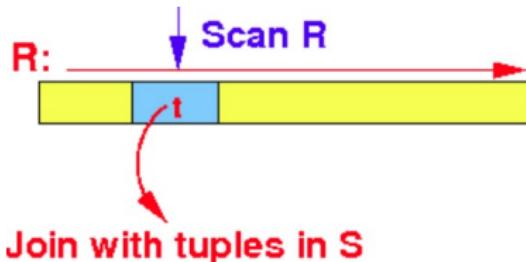


Join algorithm for a clustering index: Performance

- Therefore: # disk IO used in join algorithm with an clustering index
 - # disk IOs = Scan R once + # tuples in R × # blocks of S read per tuple of R
 - if relation R is clustered:
 - # disk IOs = $B(R) + T(R) \times \frac{B(S)}{V(S,C)}$
 - Approximate: # disk IOs $\cong T(R) \times \frac{B(S)}{V(S,C)}$
 - if relation R is not-clustered:
 - # disk IOs = $T(R) + T(R) \times \frac{B(S)}{V(S,C)}$
 - Approximate: # disk IOs $\cong T(R) \times \frac{B(S)}{V(S,C)}$

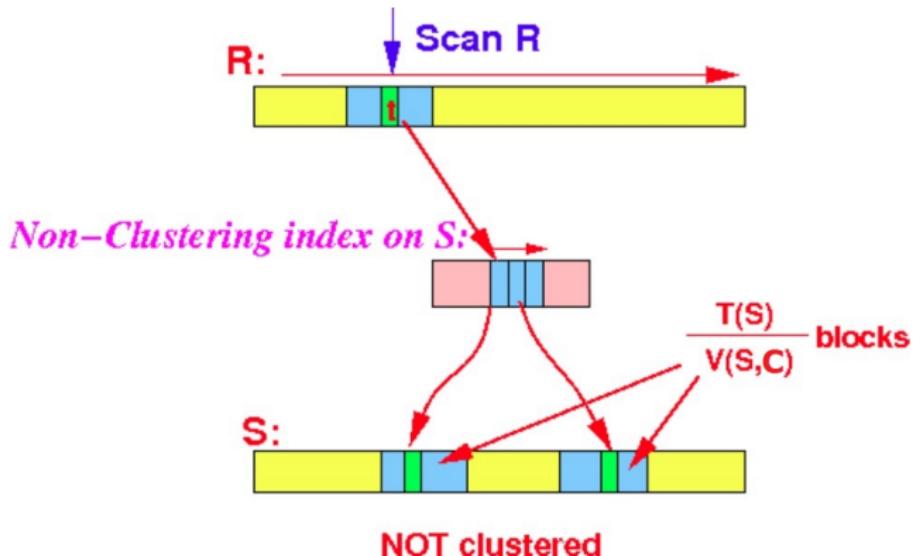
Join algorithm for non-clustering index: Performance

- The join algorithm will scan the relation R once



Join algorithm for non-clustering index: Performance

- For each tuple $t \in R$, we read the following portion in relation S



- For each tuple $t \in R$, we read (= access) this portion in S
 - portion of S read per tuple in $R = \frac{T(S)}{V(S,C)}$ blocks
 - We assumed 1 tuple of S in each block read

Join algorithm for non-clustering index: Performance

- # disk IO used in join algorithm with non-clustering index
 - # disk IOs = Scan R once + # tuples in R × # blocks of S read per tuple of R
 - if relation R is clustered:
 - # disk IOs = $B(R) + T(R) \times \frac{T(S)}{V(S,C)}$
 - Approximate: # disk IOs $\cong T(R) \times \frac{T(S)}{V(S,C)}$
 - if relation R is not-clustered:
 - # disk IOs = $T(R) + T(R) \times \frac{T(S)}{V(S,C)}$
 - Approximate: # disk IOs $\cong T(R) \times \frac{T(S)}{V(S,C)}$

When to use the Index-Join algorithm

- One of the relations in the join is very small and
- The other (large) relation has an index on the join attribute(s)

Example 1(e) Index Join

- Assume R.C non-clustering index exists, 2 levels
- Assume S contiguous, unordered
- Assume R.C index fits in memory
- Cost: # disk IOs = $B(S) + T(S) \times \frac{T(R)}{V(R,C)}$

Example 1(e) Index Join

- Cost
 - reads of S : 500 IOs
 - for each S tuple:
 - probe index: no IO
 - if match, read R tuple: 1 IO

Example 1(e) Index Join

- What is expected number of matching tuples?
 - a. say $R.C$ is key, $S.C$ is foreign key then expect 1 match
 - b. say $V(R,C) = 5000$, $T(R) = 10,000$ with uniform distribution assumption expect $\frac{T(R)}{V(R,C)} = \frac{10,000}{5,000} = 2$ matching tuples
- Total cost of index join
 - a. Total cost = $B(S) + T(S) \times 1 = 500 + 5000 \times 1 = 5,500$ IO
 - b. Total cost = $B(S) + T(S) \times \frac{T(R)}{V(R,C)} = 500 + 5000 \times 2 = 10,500$ IO
- Will any of these change if we have a clustering index?
- What if index does not fit in memory?

Hash Join: One-pass Algorithm for $R \bowtie S$

- Assumption
 - The relation S is the smaller relation
 - Building a search structure using S will minimize the memory requirement
- Phase 1
 - Use 1 buffer and scan the SMALLER relation first.
 - Build a search structure H on the SMALLER relation to help speed up finding common elements.
- Phase 2
 - Output only those tuples in R that have join attributes equal to some tuple in S
 - We use the search structure H to implement the test $t(\text{join attrs}) \in H$ efficiently
 - For H , we can use hash table or some binary search tree

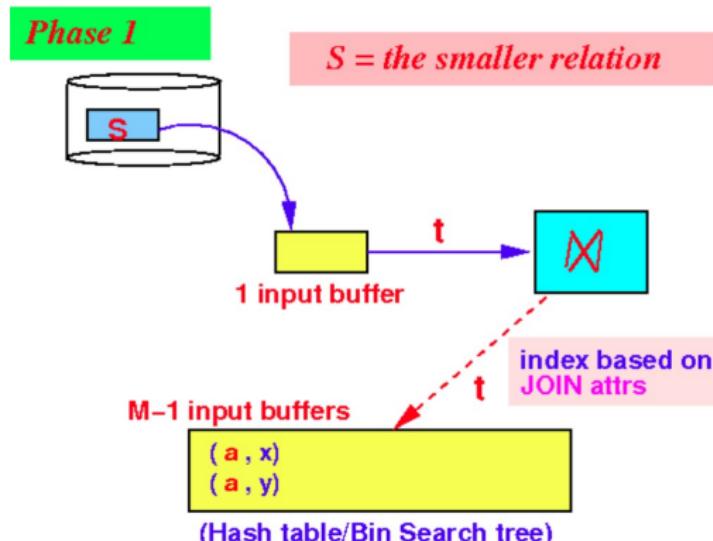
Hash Join: One-pass Algorithm for $R \bowtie S$

Algorithm 6: One-pass Hash \bowtie

```
1 while ( $S$  has more data blocks) do
2     | Read 1 data block in buffer b
3     | for (each tuple  $t \in b$ ) do
4         |   | Insert  $t$  in  $H$  // Build search structure (hash table or search tree)
5     | end
6 end
7 while ( $R$  has more data blocks) do
8     | Read 1 data block in buffer b
9     | for (each tuple  $t \in b$ ) do
10        |   | if ( $t(\text{join attrs}) \in H$ ) then
11            |   |   | Let  $s$  = the tuple in  $H$  with  $t(\text{join attrs})$ 
12            |   |   | Output  $t, s$  // successful join
13        |   | end
14    | end
15 end
```

Hash Join: One-pass Algorithm for $R \bowtie S$

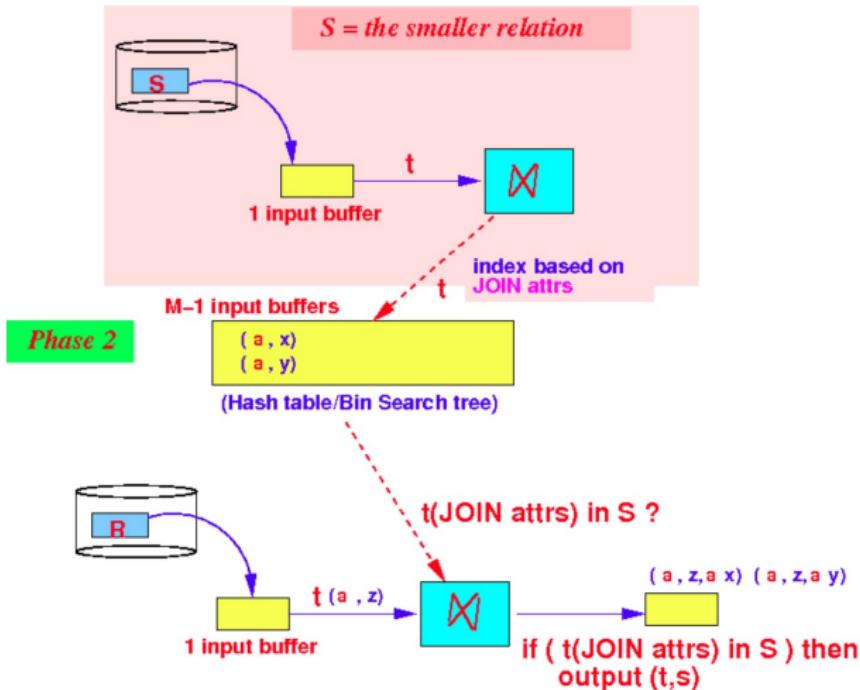
- Buffer utilization when there are M buffers available
 - Phase 1: partition the M buffers as follows



- Use 1 buffer for input from S
- Use $M-1$ buffers for the search structure

Buffer utilization

- Phase 2: partition the M buffers as follows



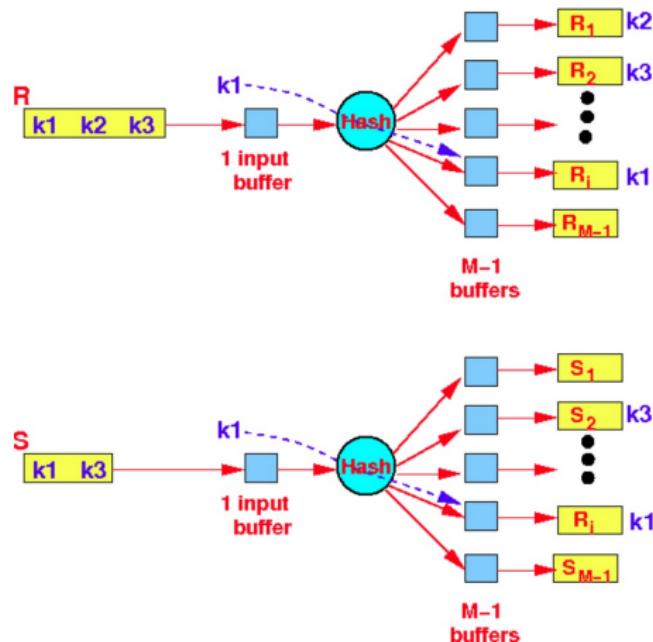
- Use 1 buffer for input from R
- Still using $M-1$ buffers for the search structure in Phase 2

Hash Join: One-pass Algorithm for $R \bowtie S$: Cost

- # disk I/O used
 - $B(R) + B(S)$, if the relations R and S are clustered
- Memory requirement
 - $M \geq B(S) + 1$ buffer
 - Relation S is the smaller relation

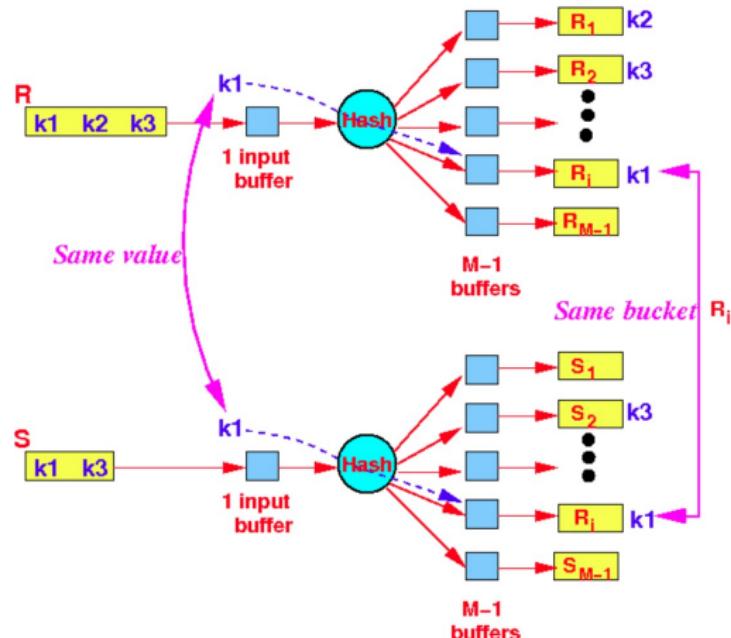
Two-pass algorithm for Join based on hashing

- Phase 1: hash both relations R and S using the same hash function to the join attribute(s).



Two-pass algorithm for Join based on hashing

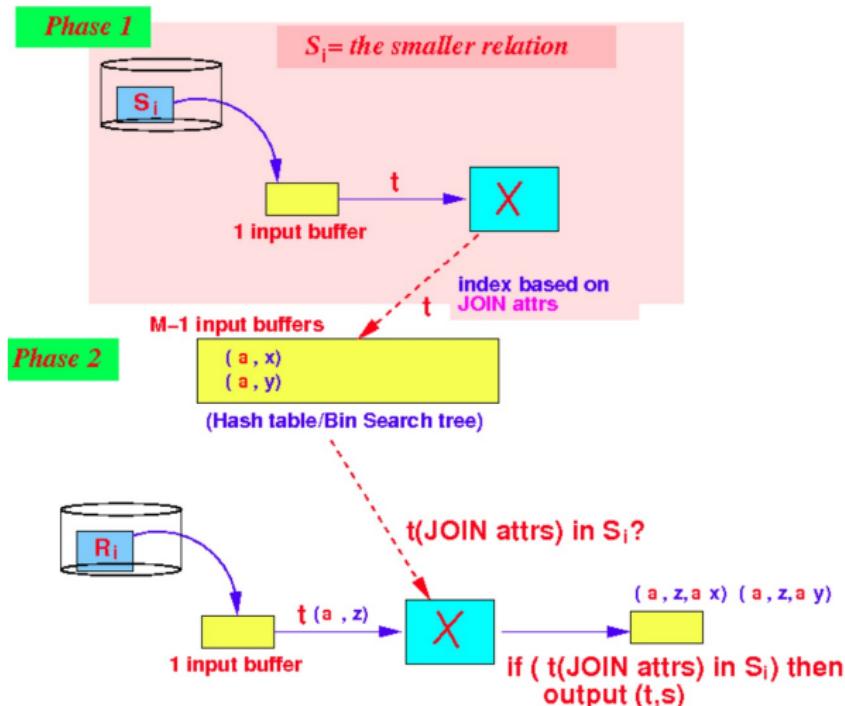
- Notice that: Same attribute value in R and S will be hashed into the same bucket



- Therefore, we can process the (equality) joining tuples in the \bowtie operation by examining the corresponding pair of buckets alone

Two-pass algorithm for Join based on hashing

- Phase 2: process each pair of sub-relation R_i and S_i using the one-pass \bowtie algorithm



Two-pass algorithm for Join based on hashing: Cost

- Cost incurred during Phase 1: (Hashing relations R and S)
 - Read R: $B(R)$
 - Write $M-1$ buckets (total): $B(R)$
 - Read S: $B(S)$
 - Write $M-1$ buckets (total): $B(S)$
- Cost incurred during Phase 2: (run one-pass algorithm)
 - $B(R) + B(S)$
- Total cost of the $R \bowtie S$ operation

$$\text{Cost of } (R \bowtie S) = 3 \times B(R) + 3 \times B(S)$$

Two-pass algorithm for Join based on hashing: Memory constraint

- The one-pass algorithm used in Phase 2 has this memory constraint
 - All tuples of the smaller sub-relation S_i must fit in the search structure using $M - 1$ buffers
 - $\therefore \min(B(R_i), B(S_i)) \leq M-1$ blocks
 - The best case scenario of the hashing is

- Every sub-relation has the same size

$$B(R_i) = \frac{1}{M-1} \times B(R)$$

$$B(S_i) = \frac{1}{M-1} \times B(S)$$

$$\Rightarrow (M - 1)^2 \geq \min(B(R), B(S))$$

$$\Leftrightarrow M - 1 \geq \sqrt{\min(B(R), B(S))}$$

- Memory constraint:

$$M \geq \sqrt{\min(B(R), B(S))} + 1$$

Example 1(f) Hash Join

- R, S contiguous (un-ordered)
- Total cost = $3 \times (1000+500) = 4500$

EXPLAIN Statements

- How to know which plan/access paths the database choose for evaluating a query?
 - Use the so called EXPLAIN-statement
 - EXPLAIN analyzes the execution of a given SQL query and stores it's results into explain tables within the DB
 - Shows operation execution order
 - Collects metrics for each basic operation

Summary

- Nested-loop ok for “small” relations (relative to memory size)
- Hash-join usually is best for `equi-join` (join condition is equal), where relations not sorted and no indexes exist
- Sort-merge join is good for `non-equijoin` e.g., $R.C > S.C$
- If relations already sorted, use `merge join`
- If index exists, `index-join` can be efficient (depends on expected result size)
- Watch: **27:21: Postgres Open 2016 - Identifying Slow Queries and Fixing Them!**

Next

- Query Optimization Physical
 - How to efficiently choose an efficient plan