University of Glasgow | School of Computing Science

Introduction to Data Science and Systems

Zaiqiao Meng
zaiqiao.meng@glasgow.ac.uk

Lecture 10 - Query Processing

Recap of previous lectures

- Physical Design: given a specific file type provide a primary access path based on a specific field; e.g., search via SSN
  - Heap (random order) files, Sequential Files, Hash Files
- Index Design: given any file type provide secondary access paths using more than one fields; e.g., search via SSN, Salary, Name, etc.
  - Primary Index, Clustering Index, Secondary index, Multi-level Index, and Search Tree (B+ tree)

Typical workflow

[Query] →

Parser → [AST] →

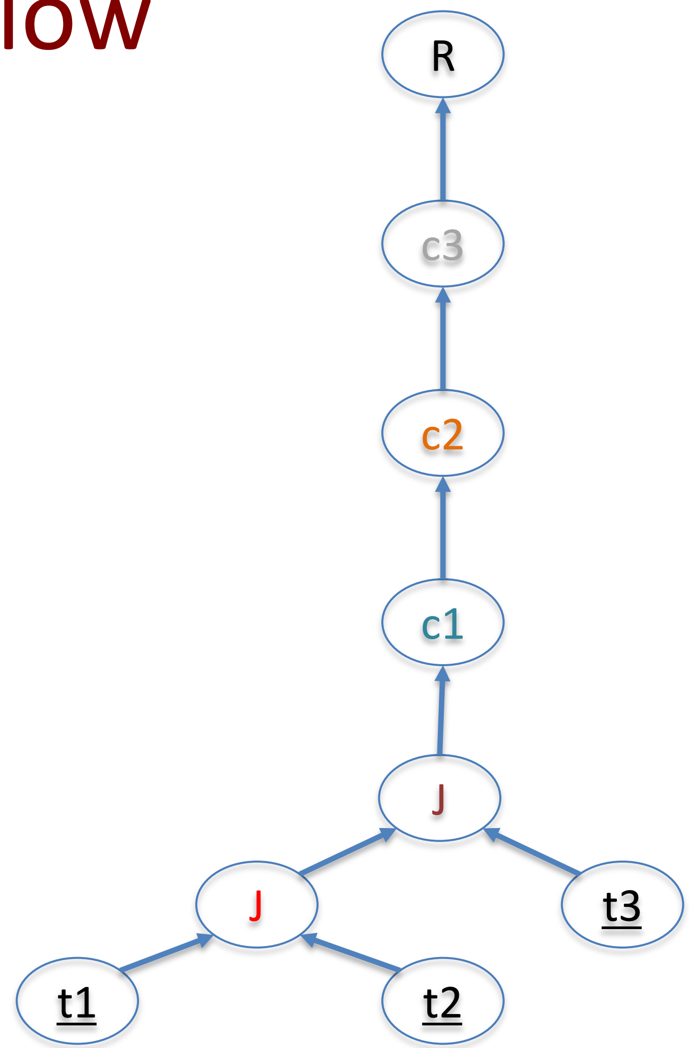
Planner/Rewriter/Algebriser → [Logical Plan] →

Optimiser → [Physical Plan] →

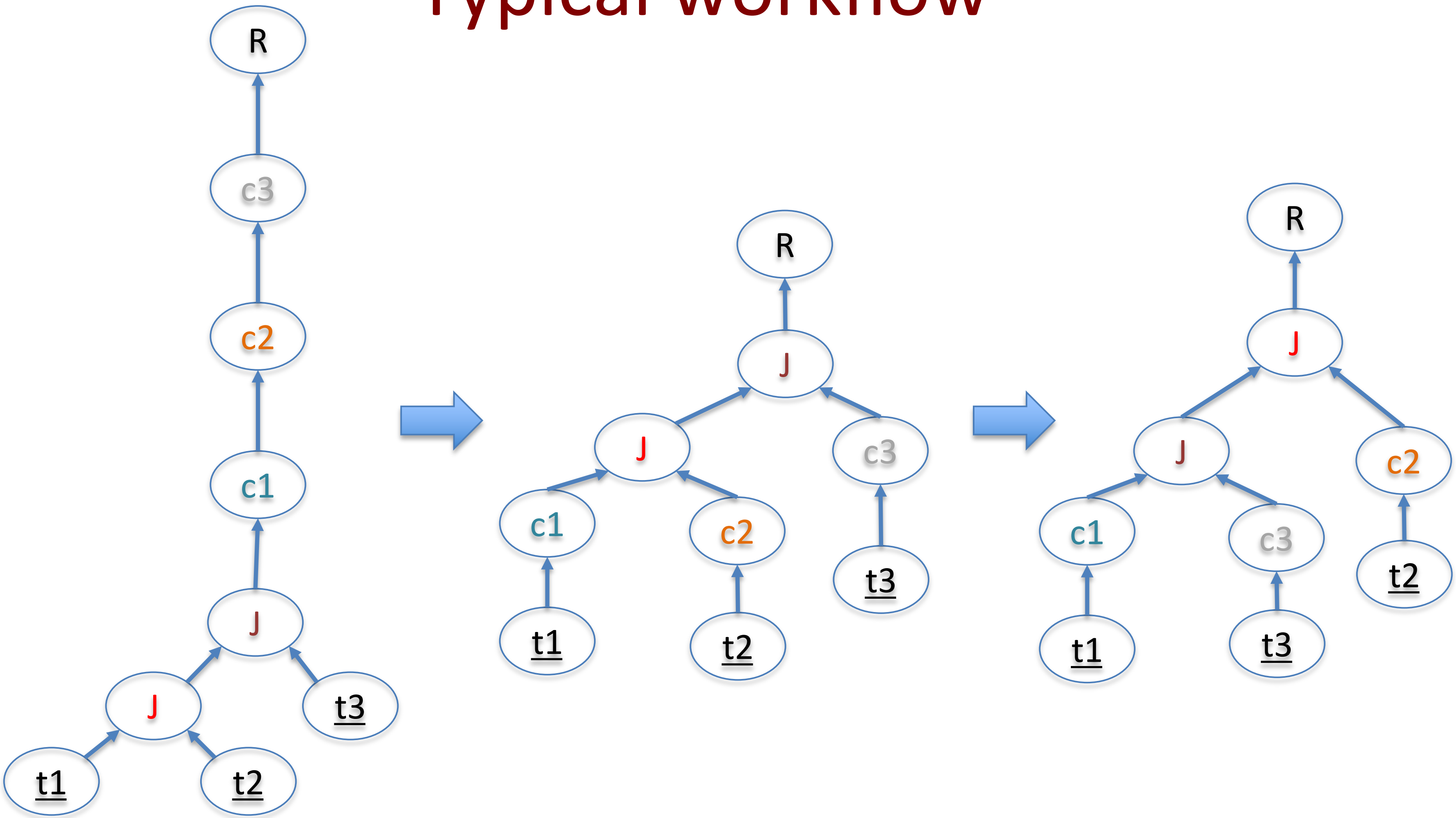
Code generation/execution → [Result set]

Typical workflow

SELECT *
FROM t1, t2, t3
WHERE
t1.a1=t2.a1 AND t1.a2=t3.a1
AND t1.a1 BETWEEN m AND n
AND t2.a1 < p
AND t3.a2 LIKE '%q%';



Typical workflow



Basic Tool: Sorting

Fundamental Algorithm: **sorting**

- *Almost* all SQL queries involve *sorting* of tuples w.r.t. *ad-hoc* sorting requests defined by the user, e.g.,
  - `CREATE INDEX ON EMPLOYEE(SSN)` **means** *sort* by SSN
  - `ORDER BY Name` **means** *sort* by Name
  - `SELECT DISTINCT Salary` **means** *eliminate* duplicates of Salary
  - `EMPLOYEE INNER JOIN DEPARTMENT ON DNO=DNUMBER` is *expedited* by sorting of EMPLOYEE and DEPARTMENT
  - `SELECT DNO, COUNT (*) FROM EMPLOYEE GROUP BY DNO`
  - …

- *In reality*, we cannot store the *entire* relation file into memory for sorting the tuples (fundamental limitation) ☹
- **External Sorting:** sorting algorithm that is *suitable* for large files stored on *disk* that do not fit *entirely* in main memory

External Sorting: Overview

Principle: Divide & Conquer (Sort)

- **Divide:** a file into many *smaller* sub-files
- **Sort-Merge**: start by *sorting small* sub-files and then *merge* the sorted sub-files, creating *larger sorted* sub-files, that are merged in turn
- **Requirement**: *buffer* space in main memory for the *actual sorting* of the sub-files and the *actual merging* of two (or *more*) sub-files at every step

Sorting Phase: *blocks* of sub-files are sorted in memory with an internal sorting algorithm, e.g., *quick-sort* or *bubble-sort*, and are *written* back to disk

Merging Phase: *sorted blocks* from sub-files are loaded from disk and *merged* in memory (merge-list/parallel-merge algorithm) and the *sorted* results are written back to disk

Note: There are many passes: already sorted blocks should be merged into bigger ones

Aside: External Sorting Cost

Lemma: The cost of the sort-merge strategy in *block accesses* is:

$$(2 \cdot b) + (2 \cdot b \cdot \log_M(L))$$

- ***b*** is the number of file *blocks*
- **M** is the *degree of merging*, i.e., the number of *sorted blocks merged* in each pass,
- **L** is the number of the initial *sorted sub-files* (before entering the merging phase)

Proof: is left for exercise ☺

Note:

- M = 2 gives the worst-case performance of the algorithm; **why?**
  - Because: *merge* a *pair* of blocks at each step
- M > 2: merge more than *two* blocks at each step; (M-*way merging*)[*]

[*] Knuth, Donald (1998). "Chapter 5.4.1. *Multiway Merging and Replacement Selection*". Sorting and Searching. The Art of Computer Programming. 3 (2nd ed.). Addison-Wesley. pp. 158–168.

Strategies for SELECT

SELECT * FROM *relation* WHERE *selection-condition*

- **S1. Linear Search** (*serial scan*): Retrieve *every* record from the file, and test whether it *satisfies* the selection condition.

```
SELECT * FROM EMPLOYEE WHERE SSN = '12345678'
```

Precondition: none

Expected Cost: *b*/2

- **S2. Binary Search**: The selection condition involves an ***ordering key***, where the file is sorted.

```
SELECT * FROM EMPLOYEE WHERE SSN = '12345678'
```

Precondition: file sorted by SSN

Expected Cost (sorted file): $\log_2(b)$

Expected Cost (unsorted file): $\log_2(b) + 2 \cdot b + 2 \cdot b \cdot \log_M(L))$

Strategies for SELECT

- **S3. Use of Primary Index** *or* **Hash Function over a key:** The selection condition involves an *equality on a* ***key attribute*** *with a Primary Index* (ISAM) or a hash function

```
SELECT * FROM EMPLOYEE WHERE SSN = '12345678'
```

Precondition: Primary Index of level *t* over the key, i.e., file is ordered by key.

Expected Cost (sorted file): $t + 1$

- **S4. Use of Primary Index in a Range:** The *selection condition is range*: >, ≥, <, ≤ *on a* ***key attribute*** *with a Primary Index* (ISAM)
- Use the Index to find the record satisfying the *equality* (e.g., DNUMBER = 5) and then *retrieve all subsequent* blocks in the *ordered file*

```
SELECT * FROM DEPARTMENT WHERE DNUMBER ≥ 5;
```

Precondition: Primary Index of level *t* over the key, i.e., file is ordered by key

Expected Cost (sorted file): $(t+1) + O(b)$

Note: Do *not* use Hashing for range queries!

Strategies for SELECT

- **S5. Use of Clustering Index to retrieve Multiple Records**: The selection condition involves an *equality* on a ***non-key attribute*** with a Clustering Index.
- Use the Index to retrieve *all* the *contiguous* blocks in the cluster corresponding to the *non-key* condition.

```
SELECT * FROM EMPLOYEE WHERE DNO = 5;
```

Precondition: Clustering Index of level *t* over the non-key; file is ordered by non-key

Expected cost (sorted file): (t+1) + O(b/c)

Note: c := #distinct values of the non-key attribute

Strategies for SELECT

- **S6. Use of Secondary Index (B+ Tree) over Equality**:

Key: Retrieve a *single* record if the indexing field is *unique.*

```
SELECT * FROM DEPARTMENT WHERE MGR_SSN = '1234567';
```

Precondition: File is not ordered by key

Expected Cost: $t + 1$

Note: B+ Leaf Node points at the *unique* block with MGR_SSN.

Non-key: Retrieve *multiple* records if the indexing field is *not* a key.

```
SELECT * FROM EMPLOYEE WHERE SALARY = 40K;
```

Precondition: File is not ordered by non-key

Expected Cost: $t + 1 + O(b)$

Note: B+ Leaf Node points to a *group* of pointers to blocks with Salary = 40K

Strategies for SELECT

- **S7. Use a Secondary Index (B+ Tree) over a Range:** Retrieve *multiple records* if the *indexing field* is involved in a range

```
SELECT * FROM EMPLOYEE
WHERE SALARY <= 40K AND SALARY >= 10K;
```

Precondition: File is not ordered by non-key

Note: B+ Leaf Nodes contain the indexing field values sorted, thus, a *serial* scan provides pointers to the records satisfying the query

Methodology:

- Find the *first* Leaf Node, e.g., Salary = 10K
- Load the cluster of pointers to blocks with Salary = 10K (load the blocks)
- Follow the Leaf Node-*next-tree-pointers* and repeat the same
- Stop at the last Leaf Node with Salary > 40K.

Expected Cost: $t + 1 + O(b*n)$, ***n = #values in the range***

Strategies for Disjunctive SELECT

- **Disjunctive Selections** involve conditions connected with OR

```
SELECT * FROM EMPLOYEE
WHERE SALARY > 10000 OR NAME LIKE '%Chris%'
```

→ The *final* result *must* contain tuples satisfying the *union* of the conditions

Methodology:

- **IF**: an *access path* exists, e.g., B+/hash/primary-index for *all* of the attributes, then:
  - use *each* to retrieve the *set* of records satisfying *one* condition
  - *union* all sets to get the final result
- **ELSE**: if *none* of the attributes have an access path, linear search is unavoidable!

Strategies for Conjunctive SELECT

- **Conjunctive Selections** involve conditions connected with AND

```
SELECT * FROM EMPLOYEE
WHERE SALARY > 40000 AND NAME LIKE '%Chris%'
```

Methodology:

- **IF**: an *access path exists* (index) for *each* of the attributes, use it to retrieve sets of tuples *individually* satisfying the corresponding condition, e.g., `Salary > 40000`, for Salary
- Go *through* this *set* of records to check which record satisfies *also* the other condition(s), e.g., `Name LIKE '%Chris%'`

- **Which is the *best* order of using the indexes?**
- **Answer**: This is *optimization; u*se the order which **minimizes** the cost!
- **Answer**: Selectivity estimation…

Strategies for JOIN

Observation: the *most resource-consuming operator*!

Focus: *two-way equijoin*, i.e., join *two* relations with equality '='

```
SELECT *
FROM   EMPLOYEE E, DEPARTMENT D
WHERE  E.DNO = D.DNUMBER
```

Five fundamental strategies for join processing:

- Naïve join (*no access path*)
- Nested-loop join (*no access path*)
- Index-based nested-loop join (*index*; B+ Trees)
- Merge-join (*sorted relations*)
- Hash-join (*hashed relations*)

Naïve Join

Idea: A *naïve natural* strategy, which *does not require* any access path

Join query: **R**.A = **S**.B

- **Step 1:** Compute the Cartesian product of **R** and **S**, i.e., *all* tuples from **R** are concatenated (*combined*) with *all* tuples from **S**
- **Step 2:** Store the result in a file **T** and for concatenated tuples $t = (r, s)$ with $r \in$ **R** and $s \in$ **S** check *iff* r.A = s.B

Algorithm Naïve Join

T = Cartesian R x S

Scan T, a tuple $t \in$ T at a time: $t = (r, s)$

If r.A = s.B **then** add (r, s) to the result file

Else go to next tuple $t \in$ T

Outcome: very *inefficient* -- typically the result is a very small *subset* of the Cartesian product!

- **What-If:** *no* tuples are actually combined; predict the matching tuples in advance!

Nested-Loop Join

Idea: A *non-naïve natural* strategy, which *does not require* any access path;

Algorithm Nested-Loop Join

For each tuple *r* ∈ **R**

For each tuple *s* ∈ **S**

If *r*.A = *s*.B **then** add (*r*, *s*) to the result file

Note: the outer & inner loops are *over* blocks and *not* over tuples!

Note: Re-form the pseudocode in a *block-centric* programming mode ☺

Challenge: Which relation should be in the *outer* loop and which in the *inner* loop to *minimize* the join processing cost? **Optimization…**

Nested-Loop Join: Algorithm

Step 1:

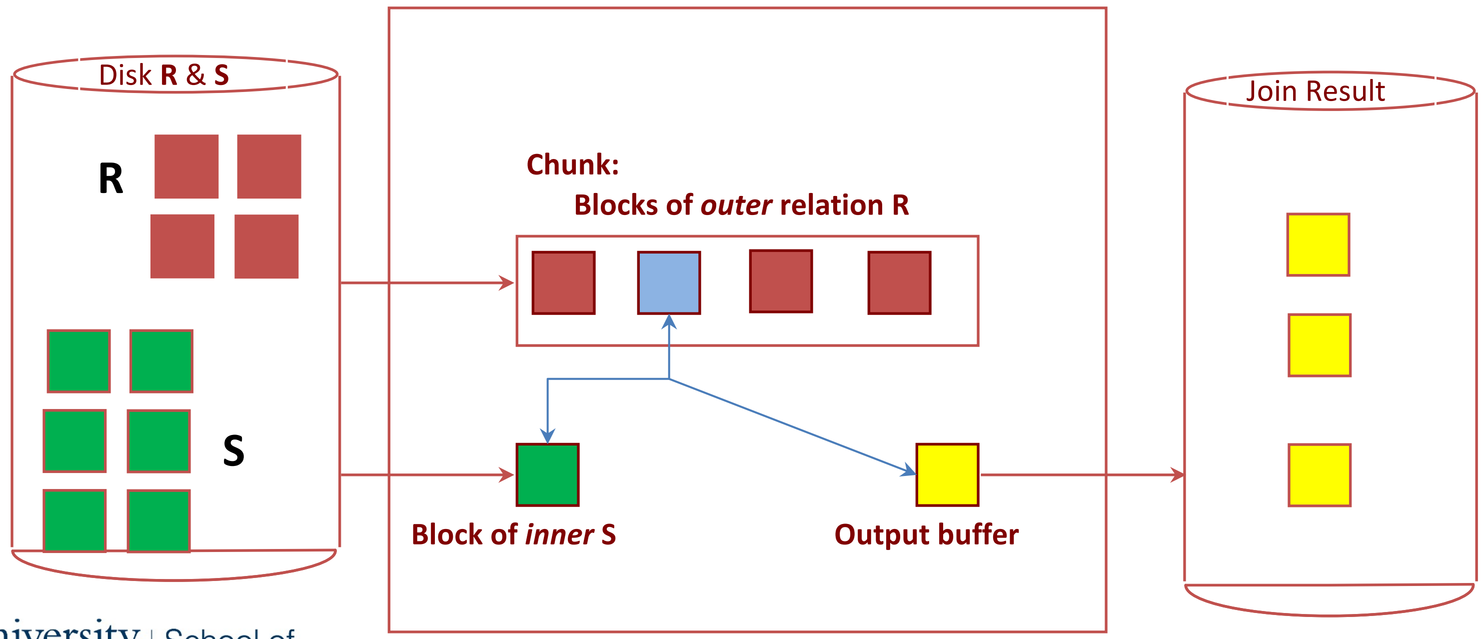
- Load a set (*chunk*) of blocks from the *outer* relation **R.**
- Load first block from *inner* relation **S**
- Maintain an *output* buffer for the matching (*resulting*) tuples (*r*, *s*): *r*.A = *s*.B

Step 2:

- Join the **S** block with *each* **R** block from the chunk
- For each *matching* tuple *r* ∈ **R**-block and *s* ∈ **S**-block **add** (*r*, *s*) to Output buffer (if *full*, write to disk)

Step 3: If more **S**-blocks, read *next* **S**-block and GOTO **Step 2**

Step 4: If more **R**-chunks, GOTO **Step 1**



Index-based Nested-Loop Join

Idea: Use of an *index* on either A or B joining attributes: **R**.A = **S**.B

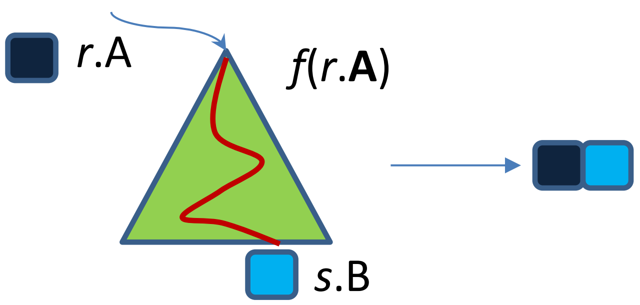
Focus: Assume an *index* *f*(B) on joining attribute B of relation **S**

Algorithm **Index-Based Nested-Loop Join**

For each tuple $r \in$ **R**

  Use *index* of B from **S** by *f*(*r*.A), to *retrieve all* tuples $s \in$ **S** having *s*.B = *r*.A

  For each *such* tuple $s \in$ **S**, add matching tuple (*r*, *s*) to the result file;



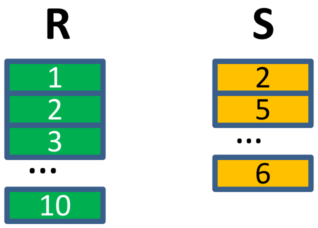
Claim: Much faster compared to the nested-loop join, **why?**

Because: We get *immediate* access on $s \in$ **S** with *s*.B = *r*.A by *searching* for *r*.A using the index *f* on B, avoiding linear search on **S**

Sort-Merge Join

Idea: Use of the *merge-sort algorithm over* two *ordered* relations w.r.t. their joining attributes.

Pre-condition: Relations **R** and **S** are *physically ordered* on their joining attributes A and B;



Methodology:

- **Step 1:** Load a pair {**R**.**block**, **S**.**block**} of *sorted* blocks into the memory;
- **Step 2:** Both blocks are *linearly* scanned *concurrently* over the joining attributes (list-merge mode);
- **Step 3:** If matching tuples *found* then store them in a buffer.

Gain: The blocks of each file are scanned *only* once!

But: If **R** and **S** are *not* a-priori *physically* ordered on attributes A and B then *sort* them first!

Hash-Join

Pre-condition:

- File **R** is partitioned into M *buckets* w.r.t. **hash function** *h* over joining attribute A
- File **S** is *also* partitioned into M *buckets* w.r.t. the *same hash* function *h* over attribute B

Assumption*:* **R** is the *smallest file* and fits into main memory: M buckets of **R** are in memory

Algorithm Hash-Join

/*Partitioning phase */

For each tuple *r* ∈ **R**,

  Compute *y* = *h*(*r*.A) /* address of bucket*/

  Place tuple *r* into *bucket y = h*(*r*.A) in memory

/*Probing phase*/

For each tuple *s* ∈ **S**,

  Compute *y* = *h*(*s*.B) /*use the same hash function *h**/

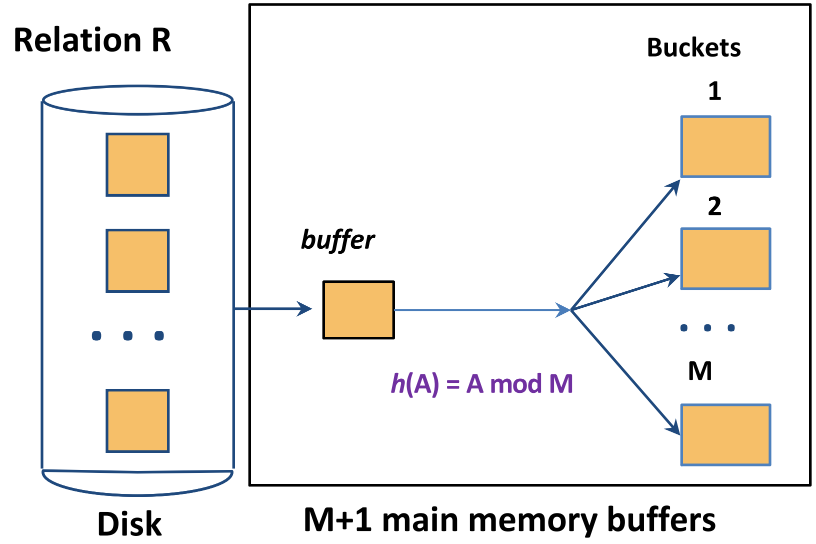
  Find the *bucket y* = *h*(*s*.B) in memory (of the **R** partition)

  For *each* tuple *r* ∈ **R** in the bucket *y* = *h*(*s*.B)

    If *s*.B = *r*.A **add** (*r*, *s*) to the result file; /*join*/

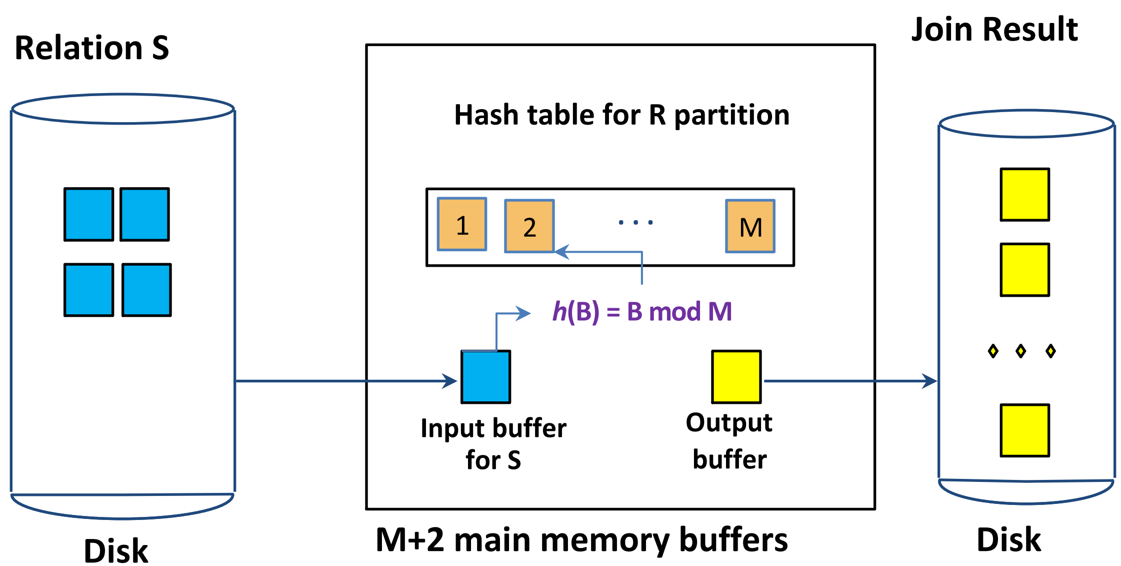
Partitioning Phase

Partition of **R** over attribute A using *hash* ***h*(A) = A mod M** into M buckets.



Probing Phase

Hashing each tuple *s* from **S**, using *hash* ***h*(*s*.B) = *s*.B mod M** to identify the *y* = *h*(*s*.B) bucket in memory.



So Far…

- **Naïve Join:** Exploit *nothing*. Cartesian product and then check…
- **Nested-Loop:** Exploit *nothing*. Computing-oriented join
  - *Which* relation should be in the *outer* loop? Influences the join cost
  - Can you *predict* the cost then? **Optimization…**

- **Index-based Nested-Loop:** Exploit at *least* one *index*. Use *index* to find the matched tuples as quick as possible ☺️
  - If we have two indexes (over R.A and over S.B), which one to use? Influences the join cost! **Optimization…**

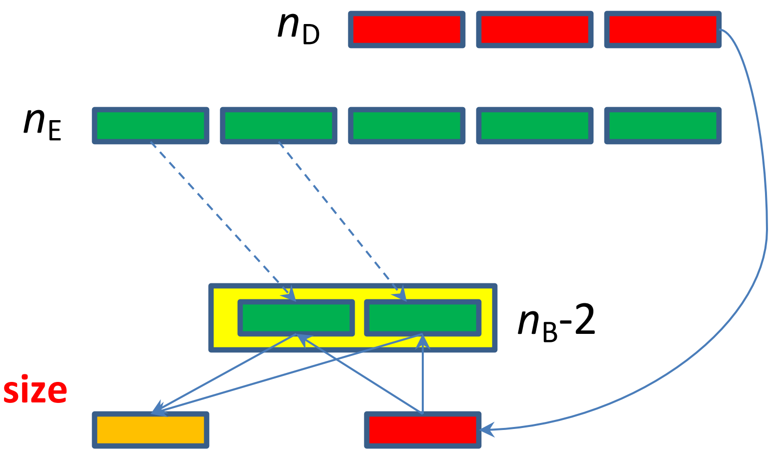
- **Merge-Join**: Exploit *both ordered* relations; otherwise; sort them ☹️

- **Hash-Join:** Exploit *hashing*. Hash *one* relation first. Which one? **Optimization…**
- Use the same hashing function to find the matched tuples in the same bucket ☺️

- **Challenge: Predict the join cost & choose the best strategy!**

Nested-Loop Join Cost Prediction

```
SELECT * FROM EMPLOYEE E, DEPARTMENT D
WHERE E.DNO = D.DNUMBER
```



Employee (E): n_E blocks used at the *outer* loop

Department (D): n_D blocks used at the *inner* loop

Memory: n_B blocks available:

- 1 block for *reading* the **inner** file D,
- 1 block for *writing* the join **result**,
- n_B-2 blocks for *reading* the **outer** file E: **chunk size**

Observation 1: Each block of *outer* relation E is read *once*

Observation 2: The *inner* relation D is read *once each time* we read (n_B-2) blocks of E

Nested-Loop Join Cost Prediction

- Total number of blocks read for *outer* relation E: $\mathbf{n_E}$
- **Outer Loops:** Number of *chunks* of (n_B-2) blocks of *outer* relation read: $\mathbf{ceil(n_E/(n_B-2))}$
- For *each* chunk of (n_B-2) blocks read *all* the blocks of *inner* relation D:
- Total number of block read in all outer loops: $\mathbf{n_D * ceil(n_E/(n_B-2))}$

Total Expected Cost: $\mathbf{n_E + n_D * ceil(n_E/(n_B-2))}$ **block accesses**

Example: $\mathbf{n_E}$ = 2,000 blocks; $\mathbf{n_D}$ = 10 blocks; $\mathbf{n_B}$ = 7 blocks

Strategy Cost 1: (E *outer*; D *inner*) $\mathbf{n_E + n_D * ceil(n_E/(n_B-2)) = 6{,}000}$ **block accesses**

Strategy Cost 2: (D *outer*; E *inner*) $\mathbf{n_D + n_E * ceil(n_D/(n_B-2)) = 4{,}010}$ **block accesses**

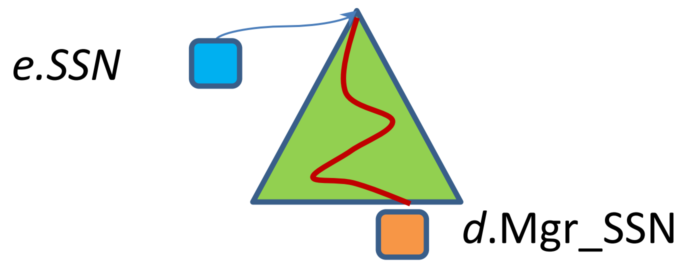
Lesson Learnt: The file with *fewer* blocks goes at the outer loop, since we *minimize* the outer loops for reading the *inner* file

Index-based Nested-Loop Cost Prediction

```
SELECT * FROM Employee E, Department D
WHERE    D.Mgr_Ssn = E.Ssn
```

- B+ Tree on **Mgr_Ssn** with level $\boldsymbol{x_D = 2}$
- B+ Tree on **SSN** with level $\boldsymbol{x_E = 4}$
- Relation E: $\boldsymbol{r_E}$ **=** 6000 tuples; $\mathbf{n_E}$ = 2,000 blocks; Relation D: $\boldsymbol{r_D}$ = 50 tuples; $\mathbf{n_D}$ = 10 blocks

Strategy 1: Tuple *e* ∈ E and *use* the B+ Tree on Mgr_Ssn to find tuple *d* ∈ D: *e*.Ssn = *d*.Mgr_Ssn.

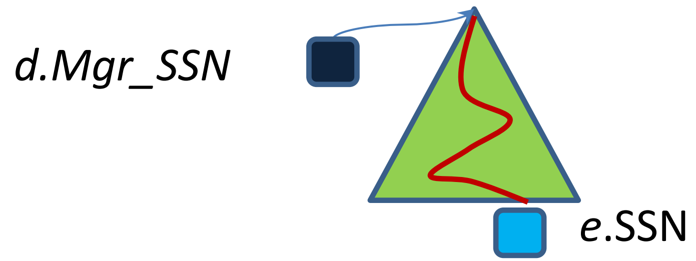


Observation: *not all employees are managers;*

Strategy Cost 1: $\boldsymbol{n_E + r_E*(x_D + 1)}$ = **20,000 block accesses;**

Index-based Nested-Loop Cost Prediction

Strategy 2: Tuple *d* ∈ D and *use* the B+ Tree on SSN to find tuple *e* ∈ E:
e.Ssn = *d*.Mgr_Ssn



Observation: *every department* has *one* manager – *search is fruitful…*

Strategy Cost 2: $\boldsymbol{n_D + r_D*(x_E + 1)}$ = **260 block accesses;**

- **Huge difference (20,000 vs 260 block accesses):**
- *every* record in Department is joined with *exactly* one record in Employee (*unique* Manager)
- *only some employees* from Employee are managers of departments…

Lesson Learnt:

- Use the PK index of the *referenced* relation (E) pointed by the FK of the *referencing* relation (D)
- **Note:** *not* for recursive FK-PK relationships, e.g., employee-supervisor

Sort-Merge Cost Prediction

Requirement: Efficient if *both* Employee E and Department D are *already* sorted by their joining attributes: SSN and Mgr_Ssn

Observation: only a *single* pass is made for *each* file.

Strategy Cost: $n_E + n_D$ = **2,010 block accesses**

IF *both* files are *required* to be sorted by the joining attributes

THEN use external sorting!

Strategy Cost 1: External sorting (2-way merge): $\mathbf{2 \cdot n_E + 2 \cdot n_E \cdot \log_2(\mathrm{ceil}(n_E / n_B))}$

- $\mathrm{ceil}(n_E / n_B))$: number of *initial* unsorted blocks
- n_B: number of available memory blocks.

Strategy Cost 2: External sorting (2-way merge): $\mathbf{2 \cdot n_D + 2 \cdot n_D \cdot \log_2(\mathrm{ceil}(n_D / n_B))}$

Sort-Merge Cost Prediction

Total Strategy Cost:

$$n_E + n_D + 2 \cdot n_E + 2 \cdot n_E \cdot \log_2(\text{ceil}(n_E / n_B)) + 2 \cdot n_D + 2 \cdot n_D \cdot \log_2(\text{ceil}(n_D / n_B))$$

Example: n_E = 2,000 blocks; n_D = 10 blocks; n_B = 7 blocks, we get:

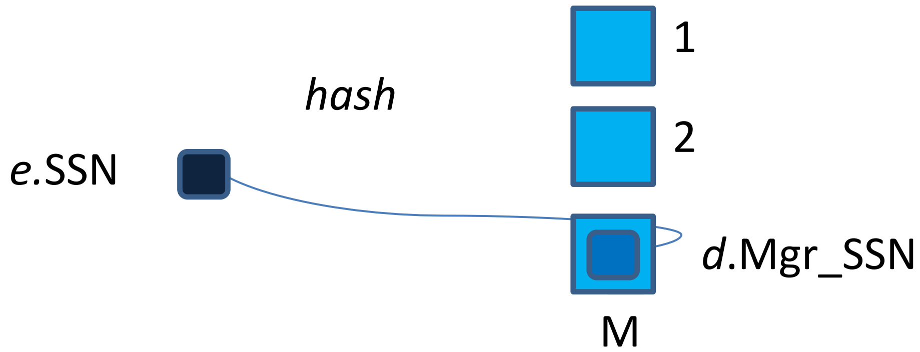
2010 + 4000 + 4000 log(286) + 20 + 20 log(2) = **38,690 block accesses;** only **5.1%** is devoted to join!

Lesson Learnt: Think before sort *only* for joining purposes!

Hash-Join Cost Prediction

Best Case: Memory $\boldsymbol{n_B > n_D + 2}$

n_D: blocks for the *smallest* of the two relations (e.g., Department)



- *Whole* relation Department *fits* in memory and is *hashed* into M buckets
- *Each* Employee tuple is *loaded* and *hashed* on joining attribute SSN
- The corresponding *bucket* is found and searched for a matching tuple
- The *result* is stored in another buffer (that's why $n_B > n_D + 2$)

Best-Case Strategy Cost: $\boldsymbol{n_E + n_D}$ block accesses

Hash-Join Cost Prediction

Normal Case: The *smallest* relation **cannot** fit in memory

Partitioning Phase

- **Read** both relations E & D first (one *block* at a time);

  Partial Cost: $n_E + n_D$
- **Partition** into M *buckets* using the *same* hashing function
  - The M *main* buckets *fit* in memory; *overflown* buckets in disk!
- **Store** the *main* buckets of each relation to the disk

  Partial Cost: $n_E + n_D$

Probing Phase

For each *m* =1…M bucket **Do**

- **Read** the *m*-th bucket from E and the *m*-th bucket from D

  Partial Cost: $n_E + n_D$
- **Perform** *join* focusing *only* on the tuples from the *same* bucket *m*

Expected Cost: $3(n_E + n_D)$ block accesses

Putting it all together: Join Cost Prediction

- Naïve Join Cost: $\boldsymbol{n_E * n_D}$: **20,000 block accesses**
- Nested-Loop Cost (*best*): $\mathbf{n_D + n_E * ceil(n_D/(n_B-2))}$: **4,010 block accesses**
- Index-based Nested-Loop Cost (*best*): $\boldsymbol{n_D + r_D*(x_E + 1)}$: **260 block accesses**
- Sort-Merge Cost (*already* sorted): $\boldsymbol{n_E + n_D}$: **2,010 block accesses**
- Hash-Join Normal-Case Cost: $\boldsymbol{3(n_E + n_D)}$: **6,030 block accesses**

Special Thanks

Special Thanks to Dr Nikos Ntarmos who is the original author of the slides.