L12: Query Processing and Optimization

DSAN 6300/PPOL 6810: Relational Databases and SQL Programming

Irina Vayndiner

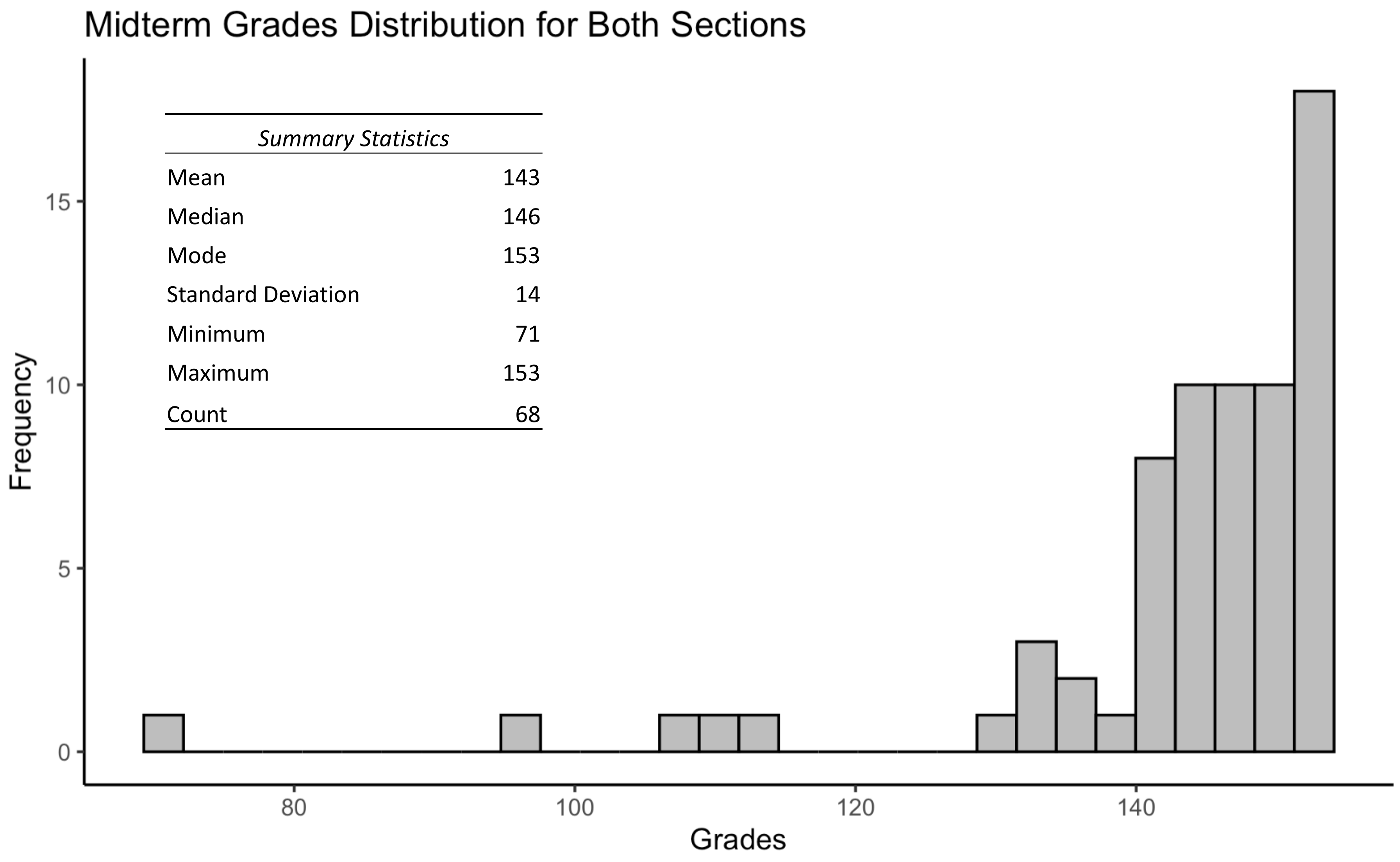
November 16 & 20, 2023

GEORGETOWN UNIVERSITY

Logistics

- HW04 (mini-project)
  - Clarification: Build a **<u>report</u>** means a create a document that you could submit to your boss
    - Not just a csv file
  - Due Wed, 12/6 (no extensions!)
  - If you submit on/before 12/4, you will get 3 extra (bonus) points ☺
- Q04 (based on today's lecture/lab) will be published on 11/20
  - Due on **Tue, 11/28**
- Test: **Tue, 12/5** (All sections) on zoom
  - No early or late dates!
  - 250 points
  - Covers full course materials:
    - Quiz part, Programming part
    - No cheating (we will check!)

Midterm Results



Summary Statistics	
Mean	143
Median	146
Mode	153
Standard Deviation	14
Minimum	71
Maximum	153
Count	68

Agenda for today's class

- Today:
  - Lecture: Query Processing and Optimization
  - Lab: Using explain plan
    - To submit: Make sure you include ALL sql queries and cvs files and the SCREENSHOT of the Task 5 Execution Plan in your submission.

Outline: Query Processing and Optimization

- Overview
- Measures of Query Cost
- Selection Operation
- Sorting
- Join Operation
- Other Operations
- Evaluation of Execution Plan
- Transformation of Relational Expressions
- Statistical Information for Cost Estimation
- Cost-based and rules-based optimization
- Tips for users

Why We Need Query Optimization

- SQL is a powerful language, it allows to express the same query in many different ways.
  - I know you noticed by now ☺
- Ideally, all SQL statements that are logically equivalent should not only produce the same results, but to run the same, optimal, time. That is what we *want* to expect from a query ***optimizer***
- In reality, performance of two logically equivalent queries can differ wildly – literally seconds vs. days!
- You can influence it by writing efficient SQL queries vs. poor ones, there are tools and techniques that can help you in that.
- In order to understand how you can improve your query you need to understand what is going on "under the hood" of a query optimizer - the heart of any database engine.

Purpose of Query Optimizer

- The optimizer attempts to generate the *most optimal execution* plan for a SQL statement.
- The optimizer attempts to choose the plan *with the lowest cost* among all considered candidate plans.
  - The optimizer uses available statistics to calculate cost.
  - For a specific query in a given environment, the cost computation accounts for factors of query execution such as **I/O**, and also CPU and network cost.
- Because the database has many internal statistics and tools at its disposal, the optimizer is usually in a better position than the user to determine the optimal method of statement execution. For this reason, SQL statements use the optimizer.
  - Will give you tips on what you can do to tune the queries in the end of class today.

Our plan for today

- We will first look at what optimizer is doing internally
- Then develop some strategies:
    - What to do
    - What to avoid in our queries

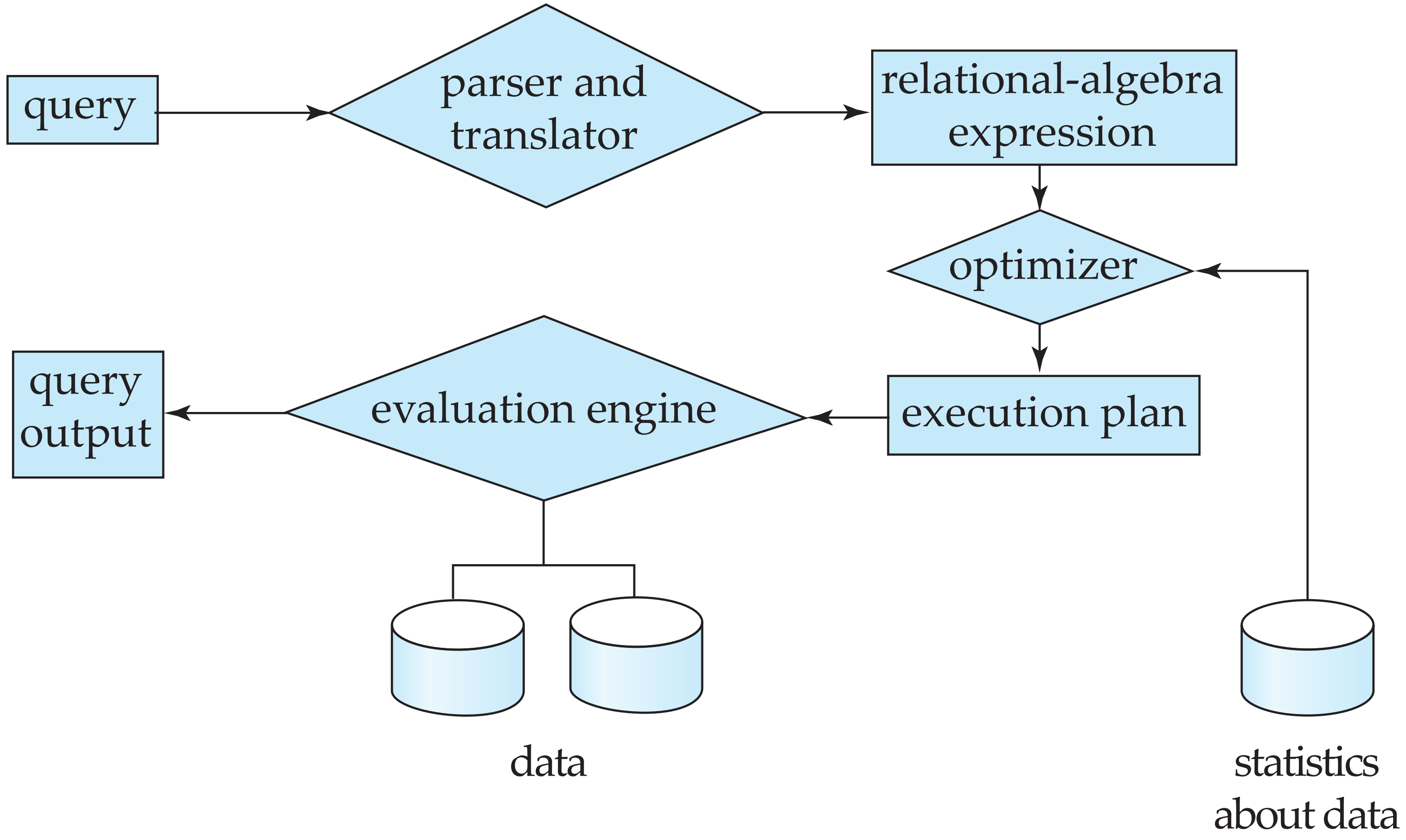
SQL engines use Query Optimization

- Some examples
  - MySQL
  - Oracle
  - MS SQL Server
  - Hive
  - SparkSQL (not a database)
  - Presto (FB) (not a database)
  - Etc etc

Query Optimization is an Advanced Topic

Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



Purpose of Query Optimizer

- The optimizer attempts to generate the **best execution** plan for a SQL statement.
- The cost-based optimizer attempts to choose the plan with **the lowest cost** among all considered candidate plans.
  - Use less resources
- Because the database has many internal statistics and tools at its disposal, the optimizer is usually in a better position than the user to determine the optimal method of statement execution.
  - Though you doing the right things helps!

Basic Steps in Query Processing: Optimization

- Consider the query:

  select salary

  from instructor

  where salary < 75000;

- This query may have many equivalent relational algebra expressions
  - E.g., $\sigma_{salary<75000}(\Pi_{salary}(instructor))$

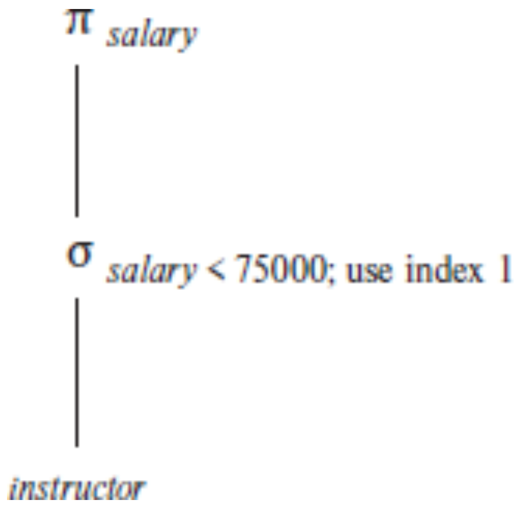
  is equivalent to

  $\Pi_{salary}(\sigma_{salary<75000}(instructor))$

- Then each relational algebra operation can be evaluated using one of several different algorithms.
  - For $\sigma_{salary<75000}(instructor)$, for example:
    - Perform complete relation scan and discard instructors with salary $\geq$ 75000
    - Or, use an index on *salary* to find instructors with salary < 75000

Basic Steps: Optimization

- Again, a relational-algebra expression can be evaluated in many ways.
- Expression specifying detailed evaluation strategy is called an **Evaluation plan**.
- Example of evaluation plan per picture below:
  - For relation *instructor*
  - Use an index on *salary* to find instructors with salary < 75000
  - Then projection: select *salary* attribute



Basic Steps: Optimization (continued)

- **Query Optimization**: Amongst all equivalent evaluation plans choose the one with **lowest** cost.
  - Cost is estimated using information from the *database catalog*
    - e.g. number of tuples in each relation, size of tuples, etc.
- In this lecture we will look into:
  - How to measure query costs
  - Algorithms for evaluating relational algebra operations
  - How to combine algorithms for individual operations in order to evaluate a complete expression
  - How to optimize queries
    - again, how to find an evaluation plan with lowest estimated cost (expressed in terms of performance or resource consumption)

How does Optimizer makes a choice: Measures of Query Cost

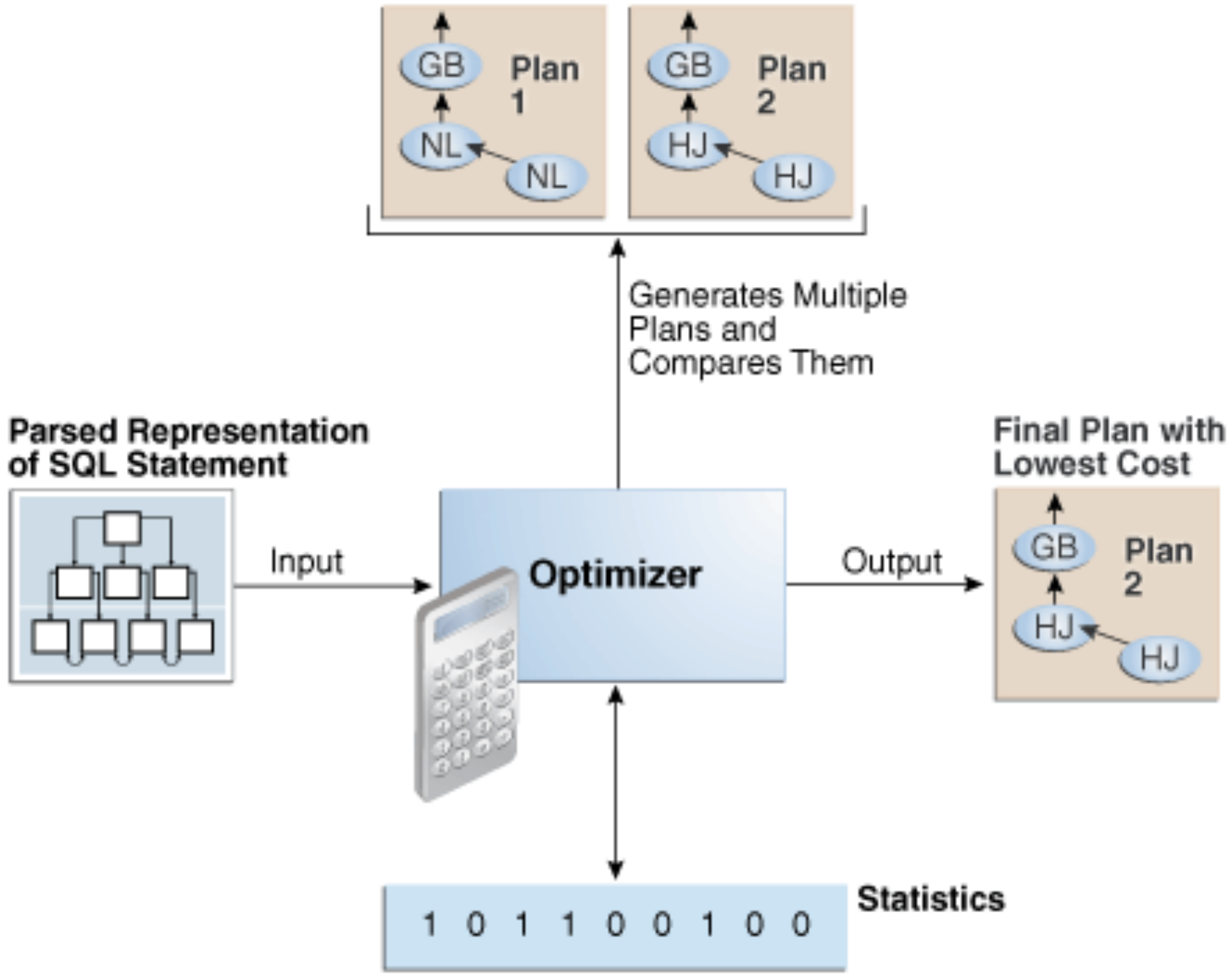
- Many factors contribute to time cost
  - disk access (read/write/*seek*), CPU, and network communication
- Cost can be measured based on
  - **response time**, i.e. total elapsed time for answering query

  or
  - total **resource consumption**
- Resource consumption is used more often
  - Response time harder to estimate *ahead of time*
  - Minimizing resource consumption is a good idea in a shared database
- We will focus on disk resource since it is often the most expensive. However:
  - Real systems do take CPU cost into account
  - Network costs must be considered for parallel systems

Cost-Based Optimization (CBO)

- Query optimization is the overall process of choosing the most efficient means of executing a SQL statement.
    - SQL is a nonprocedural language, so the optimizer can merge, reorganize, and process in any order.
- For a given query and environment, the optimizer assigns a relative numerical cost to each step of a possible plan, and then factors these values together to generate an overall cost estimate for the plan.
- After calculating the costs of alternative plans, the optimizer chooses the plan with the lowest cost estimate.
- For this reason, the optimizer is called the **cost-based optimizer (CBO)** to contrast it with the legacy rule-based optimizer (RBO) that was based on heuristics.

Cost-Based Optimization (CBO) Flow


GB
Plan 1
NL
NL
GB
Plan 2
HJ
HJ
Generates Multiple Plans and Compares Them
Parsed Representation of SQL Statement
Input
Optimizer
Output
Final Plan with Lowest Cost
GB
Plan 2
HJ
HJ
Statistics
1 0 1 1 0 0 1 0 0

https://docs.oracle.com/database/121/TGSQL/tgsql_optcncpt.htm#TGSQL196

Evaluation Plan Operations

- In order to select the best evaluation plan the Optimizer needs to be able to estimate their costs
- Cost of a plan depends on the costs of **all** operations of a plan
- Cost of the following operations need to be evaluated first
  - Selection
  - Sorting
  - Joins
  - Projection (duplicate elimination)
  - Set operations
  - Outer Joins
  - Aggregations
- Each of these operations can be done in several different ways, and cost will differ

Measures of Query Cost – the next level of details

- Disk performance I/O metrics:
  - **Seek time** - the time from when a read or write request is issued to when data transfer begins (defined by internal mechanics of the disk)
  - **Block transfer time** – time required for a logical unit of data (a block) to retrieved from (read) or stored on (written) the disk
- **Definition: Block** is a sequence of bytes, usually containing some whole number of records, having a maximum length [on that specific hardware].

- Disk cost metrics:
  - (Number of seeks) * (average-seek-cost)
  - (Number of blocks read) * (average-block-read-cost)
  - (Number of blocks written) * (average-block-write-cost)

Measures of Query Cost (continued)

- For simplicity, we just use the **number of block transfers** from disk and the **number of seeks** as the cost measures
  - t_T – one block *transfer* time (assuming, for simplicity, that write and read costs are the same)
  - t_S – Time for one *seek*
  - Cost of *number of block transfers* b and *number of seeks* S
    $b * t_T + S * t_S$, t_S and t_T depend on where data is stored
- Example, for 4 KB blocks:
  - High end magnetic disk (HDD): t_S = 4,000 microsec and t_T =100 microsec
  - Solid State Drive (SSD) : t_S = 20-90 microsec and t_T = 2-10 microsec

Selection Operation

Selection Operation: Table Scan

- Find all records that satisfy a condition:
  - E.g. $\sigma_{ID=1450}(instructor)$
- **File scan (full table scan)**.
  - In the most general case we need to scan each file block and test all records to see whether they satisfy the selection condition.
  - Cost estimate = b_r block transfers + 1 seek= $b_r * t_T + t_S$
    - b_r denotes number of blocks containing records from relation r
    - t_T – time to read one block, t_S – time for one seek
  - If selection is on a key attribute, can stop on finding record
    - Cost (on average) = (b_r /2) block transfers + 1 seek
  - Full scan is universal and can be applied regardless of
    - Selection condition
    - Ordering of records in the file
    - Availability of indices
  - Sometimes it is the best option

Selections Using Indices

- **Index scan** – search algorithms that use an index
  - selection condition must be on search-key of index.
  - h_i - index height (number of levels in the index tree)
- Lets' consider two examples
- **Clustering index, equality on key**.
  - Retrieve a single record that satisfies the corresponding equality condition
  - Cost = $h_i * (t_T + t_S) + 1 * (t_S + t_T)$
  - Index lookup traverses the height of the tree plus 1 to fetch the record itself
- **Clustering index, equality on non-key**
  - Retrieve multiple records. Records will be on consecutive blocks
    - Let n be a number of blocks containing matching records
  - $Cost = h_i * (t_T + t_S) + t_S + t_T * n$
  - Blocks assumed to be stored sequentially, since it is a clustering index, and do not require additional seeks

Selections Using Indices

- **Secondary (non-clustering) index, equality on key/non-key**.
  - Retrieve a single record if the search-key is a candidate key
    - *Cost* $= h_i * (t_T + t_S) + 1 * (t_T + t_S)$
  - Retrieve multiple records if search-key is not a candidate key
    - Each of *n* matching records may be on a different block
      - Cost $= h_i * (t_T + t_S) + \mathrm{n} * (t_T + t_S)$

- Many other use cases can be considered (inequality, etc.)

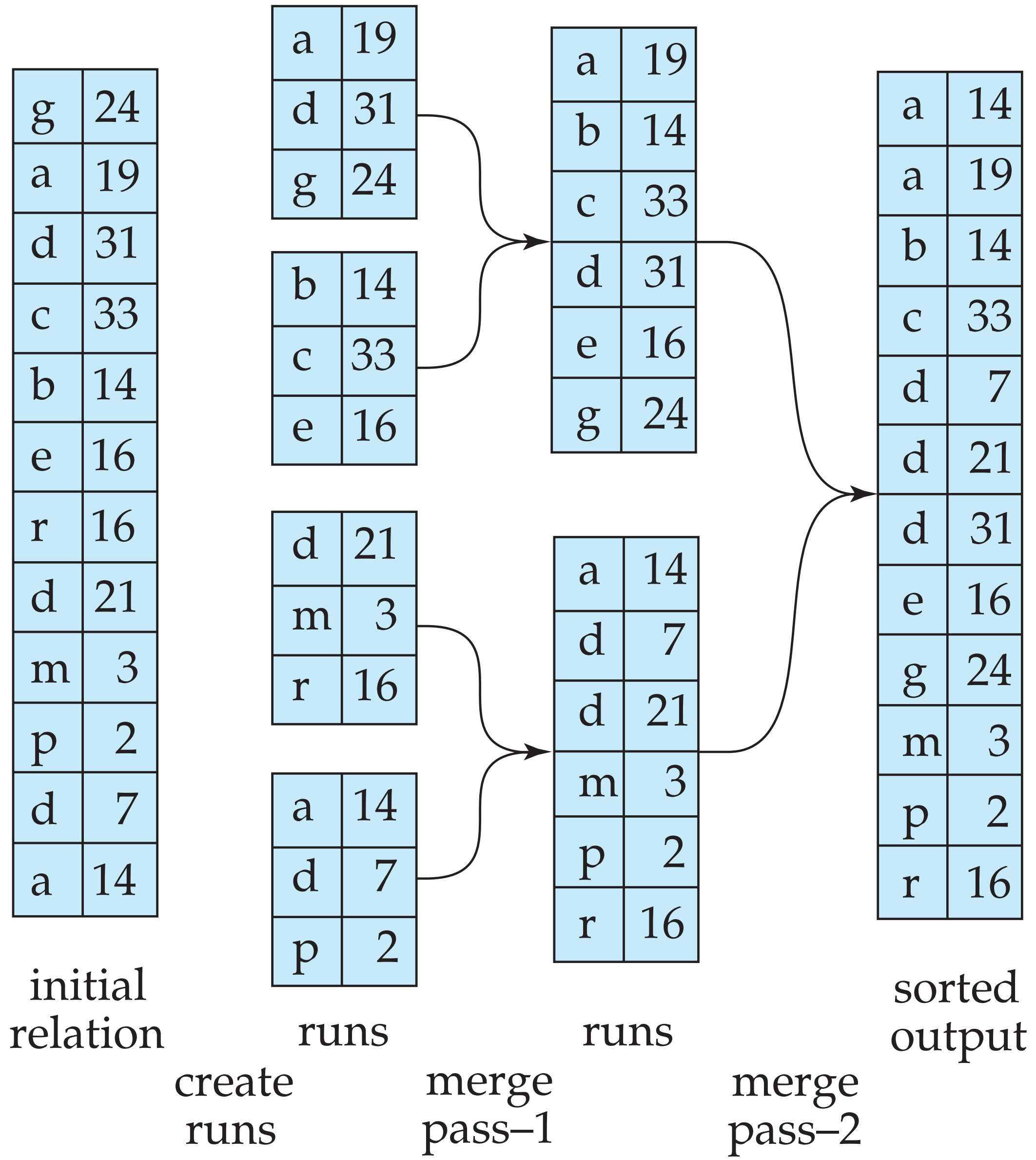
Sorting Operation

Sorting

- Sorting is a frequently used operation.
- Sometimes optimizer can consider sorting a relation just to execute one query since other plan operations (like joins) will be more efficient
- For this reason, it may be desirable to order the records physically.
- For relations that fit in memory, there are efficient algorithms (like *quicksort)*. We will neglect the cost of in-memory sorting.
- However for relations that don't fit in memory, **external sort-merge** is a good choice.
    - Let's look at the cost of sorting in that case

External Sorting Using Sort-Merge

1. Create sorted runs (run = small temp file on disk)
   - Repeatedly do the following till the end of the relation:
     - Read M blocks of relation into memory
     - Sort the in-memory blocks
     - Write sorted data to disk
2. Merge the runs
   - Repeat until all input is empty:
     - Select the first record (in sort order) among **all** runs
     - Write the record to the output.
     - Delete the record from its input



g 24 a 19 d 31 c 33 b 14 e 16 r 16 d 21 m 3 p 2 d 7 a 14
a 19 d 31 g 24
b 14 c 33 e 16
d 21 m 3 r 16
a 14 d 7 p 2
a 19 b 14 c 33 d 31 e 16 g 24
a 14 d 7 d 21 m 3 p 2 r 16
a 14 a 19 b 14 c 33 d 7 d 21 d 31 e 16 g 24 m 3 p 2 r 16
initial relation
runs
runs
sorted output
create runs
merge pass–1
merge pass–2

Conclusion: More memory is better for performance!

Join Operation

Join Operation

- Probably the *most costly operation* is Join
- Several algorithms to implement joins, most often used are:
  - Nested-loop (NL) join
  - Indexed nested-loop (INL) join
  - Merge-join (MJ)
  - Hash-join (HJ)

- "Which one is better?" Choice depends on cost estimate

- In the examples to follow we will use this information:
  - Number of records of *student*: 5,000 *takes*: 10,000
  - Number of blocks of *student*: 100 *takes*: 400

Nested-Loop Join (NL)

- The simplest Join method
- To compute the join $r \bowtie_\theta s$:

```
for each tuple $t_r$ in $r$ do begin
  for each tuple $t_s$ in $s$ do begin
      test pair $(t_r, t_s)$ to see if they satisfy join condition $\theta$
      if they do, add a row to the result set.
  end
end
```

- *r* is called the **outer relation** and *s* the **inner relation** of the join.
- In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is
  - $b_r + n_r * b_s$ (block transfers) **plus**
  - $b_r + n_r$ (seeks)

  where n_r is is the number of records in *r (and the number of iterations!)*
- If the smaller relation, say S, fits entirely in memory, use that as the inner relation.
  - Reduces cost to $b_r + b_s$ block transfers, since we only need to read relation S once

Nested-Loop Join Example

- Assuming worst case memory availability, the cost estimate is
  - with *student* as outer relation:
    - $5000 * 400 + 100 =$ 2,000,100 block transfers
  - with *takes* as the outer relation
    - $10000 * 100 + 400 =$ 1,000,400 block transfers (here bigger as outer)
- If smaller relation (*student*) fits entirely in memory, the cost estimate will be
    - 400+100 = 500 block transfers
- **Main take-aways**
  - The most expensive join method since it examines every pair of tuples in the two relations.
  - Order of join is important! Use *bigger* relation as outer
    - Optimizer usually makes that choice
  - But, it does not require any indices and can be used with any kind of join condition (A good thing!)
  - Overall, should be avoided, if possible, but sometimes is the only option

Indexed Nested-Loop (INL)Join

- Index lookups can replace full table scans if
  - join is an equijoin or natural join **and**
  - an index is available on the inner relation's join attribute
    - Some Optimizers can construct an index just to compute a join.
- For each tuple t_r (in the outer relation *r)* use the index to look up tuples in inner relation *s* that satisfy the join condition with tuple t_r.
- Worst case: buffer has space for only one block of *r*, and, for each tuple in *r*, we perform an index lookup on *s.*
- Number of block transfers: $b_r + n_r * (h_i + 1)$
  - h_i - index height
- If indices are available on join attributes of both *r* and *s,* use the relation with **fewer** tuples as the outer relation.
  - Different than what we just saw with non-indexed NL Join

Example of Indexed Nested-Loop Join Costs

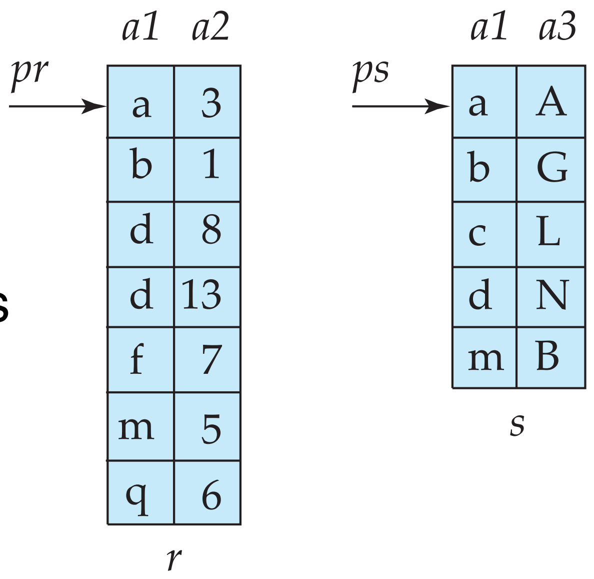
- Compute *student* ⋈ *takes,* with *student* as the outer relation.
- Let *takes* have a primary B$^+$-tree index on the attribute *ID*
- Since *takes* has 10,000 tuples, assume the height of the tree is 4, plus 1 more access is needed to find the actual data
- *student* has 5000 tuples
- Number of transfers for indexed nested loops join
  - 100 (number of blocks in student table) + 5000 * (4+1) = 25,100 block transfers.
- **Main take-aways**
  - Much more efficient than basic (without an index) nested join
    - 25K vs 1M block transfers in our previous example
  - But, it requires an index on a join attribute
  - Use *smaller* relation as the outer relation.

Merge-Join (MJ)

- Also sometimes called sort-merge-join
- Can be used only for equijoins and natural joins

1. Sorts both relations on their join attribute *a1* (if not already sorted on the join attributes).
2. Merges the sorted relations to join them
   - Join step is similar to the merge stage of the sort-merge algorithm.
   - Associates one pointer with each relation. As the algorithm proceeds, the pointers move through the relation.

- See your textbook for the detailed algorithm
- **Main take-away**
  - Much faster than basic nested loop, since no iterations required
  - Does not require an index!
  - But, it requires sorting of the relationships

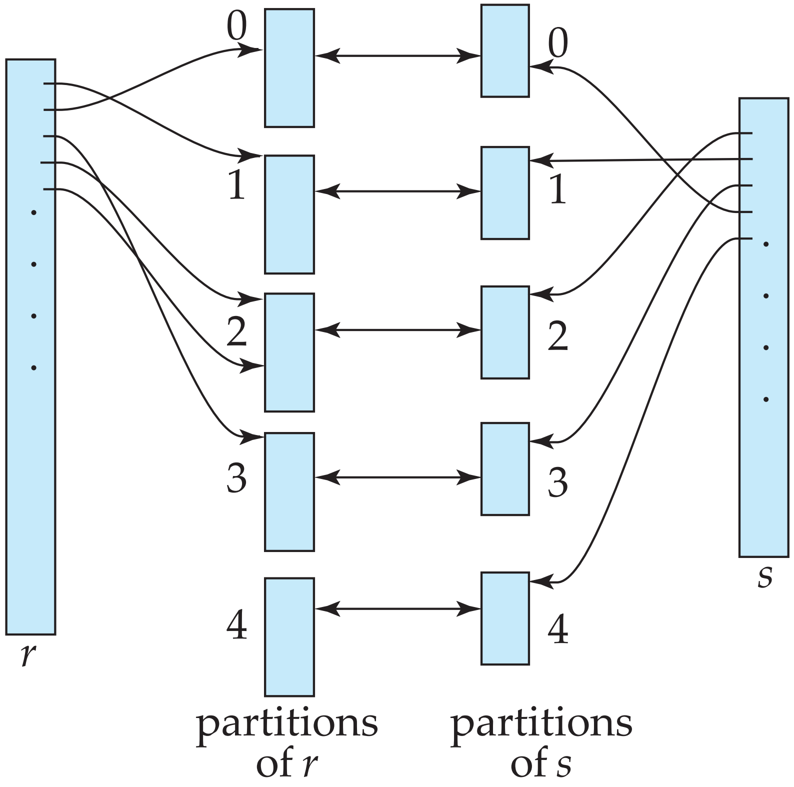


a1	*a2*
a	3
b	1
d	8
d	13
f	7
m	5
q	6

a1	*a3*
a	A
b	G
c	L
d	N
m	B

Hash-Join (HJ) Algorithm

- Another popular Join algorithm
- The same hash function *h* is used to partition or bucket tuples of both relations
  - Prefer that each bucket can fit into memory
- *h* maps *A* values to {0, 1, ..., *n*}, where *A* denotes the common attributes of *r* and *s* used in the join.
  - $r_0, r_1, . . ., r_n$ denote partitions of *r*
    - Each tuple $t_r \in r$ is put in partition r_i where $i = h(t_r[A])$.
  - Same for s



Hash-Join Algorithm (continued)

- Like merge-join, applicable for equijoins and natural joins.
- Tuples in r_i need to be compared ***only*** with tuples in **corresponding** bucket s_i since tuples that satisfy the join condition will have the same value for the join attributes.

Hash-Join Main Take-Aways

- In case when the number of tuples in the outer relation is large
    - Hash-Join can be more efficient than even the indexed nested loops join
- In case when the outer relation is small
    - Indexed nested loops join can have a much lower cost than hash –join
- If number of rows in both relations is large
    - Hash-Join may be better

- That's assuming the number of tuples in the outer relation is known at query optimization time
    - The best join algorithm can be chosen at that time.
    - Sometimes it is possible, sometimes not
        - Eg. Temp tables, or statistics is not updated

Projection Operation

Projection Operation

- **Projection**:
  - perform projection on each tuple
  - followed by duplicate elimination.
- **Duplicate elimination** can be implemented via sorting or hashing.
  - *Sorting*: duplicates will come adjacent to each other, and all but one set of duplicates can be deleted.
  - *Hashing* – duplicates will come into the same bucket and then will be eliminated

Calculation the Cost of Execution Plan

Calculating the Cost of Entire Evaluation Plan

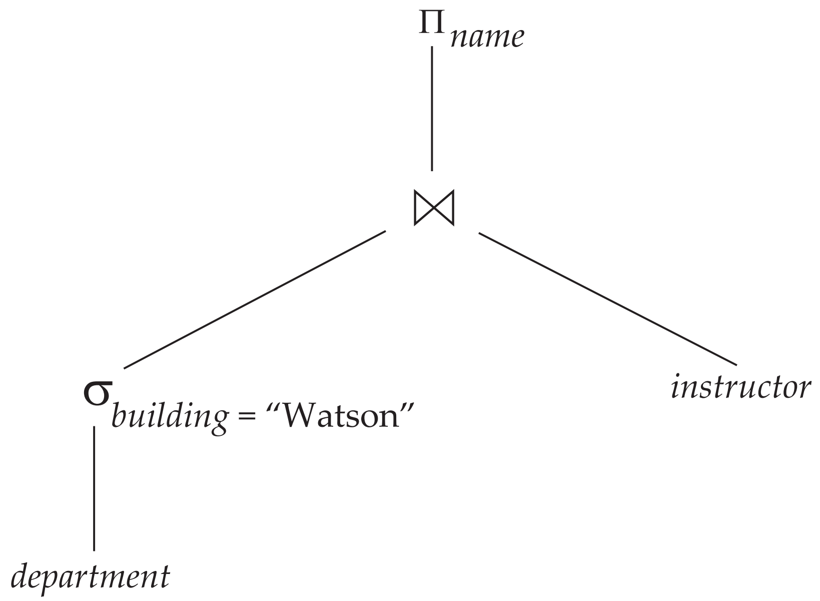
- So far: we have seen algorithms for individual operations
- Our real goal is to calculate the cost of an *entire* evaluation plan.
- There are two alternatives how the query is evaluated:
  - **Materialization**: generate results of a node whose inputs are relations or are already computed, **materialize** (store) it on disk. Repeat.
  - **Pipelining**: pass on tuples to parent nodes even as an operation is being executed

Materialization

- **Task**: Select names of all instructors who work in dept located in Watson building
- **Materialized evaluation**: evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations.
- E.g., in figure below, compute and store

$$\sigma_{building="Watson"}(department)$$

then compute and store its join with *instructor,* and finally compute the projection on *name.*



Materialization (continued)

- Materialized evaluation is always possible
- Cost of writing results to disk and reading it back can be quite high
  - Overall cost = Sum of costs of individual operations +
    cost of writing intermediate results to disk

Pipelining

- **Pipelined evaluation**: evaluate several operations simultaneously, passing the results of one operation on to the next.
- E.g., in previous expression tree, don't store result of

$$\sigma_{building="\text{Watson}"}(department)$$

  - instead, pass tuples directly to the join.
  - Similarly, don't store result of join, pass tuples directly to projection.
- Much cheaper than materialization: no need to store a temporary relation to disk.
- Pipelining may not always be possible, e.g. sort
  - **why?**
- Pipelines can be executed in two ways:
  - ***demand driven***
  - ***producer driven***

Pipelining (continued)

- In **demand driven (pull)**, or **lazy** evaluation
  - Request comes from top level operation
    - At that time next tuple is requested (not full result set)
    - Each operation requests next tuple from children operations as required, in order to output its next tuple
- In **producer-driven (push)**, or **eager** pipelining
  - Operators produce tuples “eagerly” and pass them up to their parents
    - Buffers with intermediate data are ready to go at all times
      - Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
      - If buffer is full, child waits till there is space in the buffer, and then generates more tuples
- **Main points:**
  - Due to pipelining, the cost of a plan is less than the sum of all operations
  - Optimizers takes that into account when comparing evaluation plans

Query Execution Plan Optimization

Query Optimization Example

- Consider a query Find the names of all instructors in the Music department together with the course title of all the courses that the instructors teach
- The Relational Algebra expression for that query

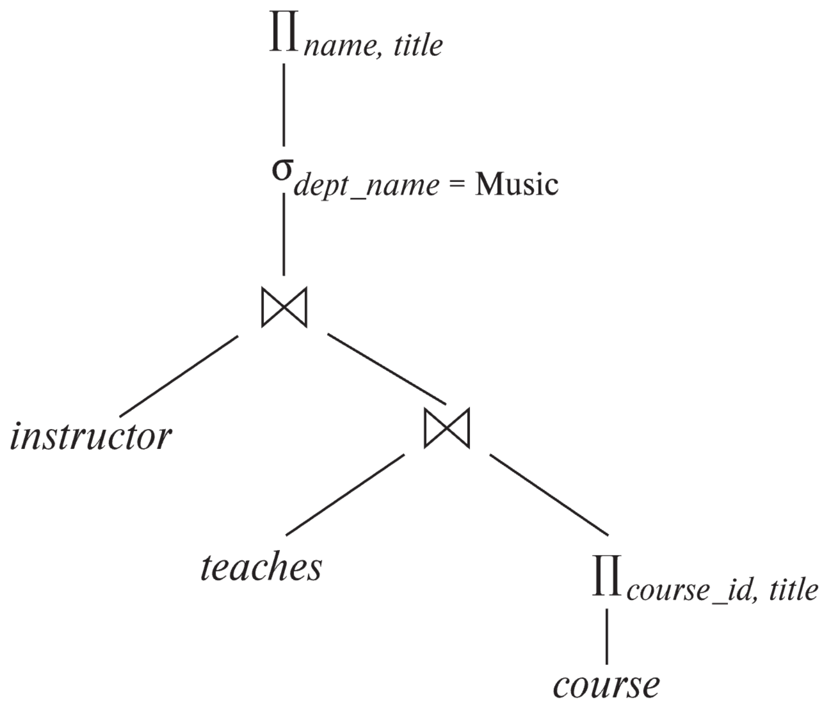
$$\Pi_{name,title}\,(\sigma_{dept\_name=\text{"Music"}}\,(instructor \bowtie (teaches \bowtie \Pi_{course\_id,title}(course))))$$

- The join *instructor* ⋈ *teaches* ⋈ $\Pi_{course\_id,title}$(*course*) can create a very large intermediate result, but we only need those tuple that have "Music" as the instructor's *dept_name*.
- We can write an equivalent expression by choosing only Music instructors early, and that reduces the number of tuples in the instructor that we need to join, and thus dramatically reduce the size of the intermediate set.

$$\Pi_{name,title}\,((\sigma_{dept\_name=\text{"Music"}}\,(instructor)) \bowtie (teaches \bowtie \Pi_{course\_id,title}(course)))$$

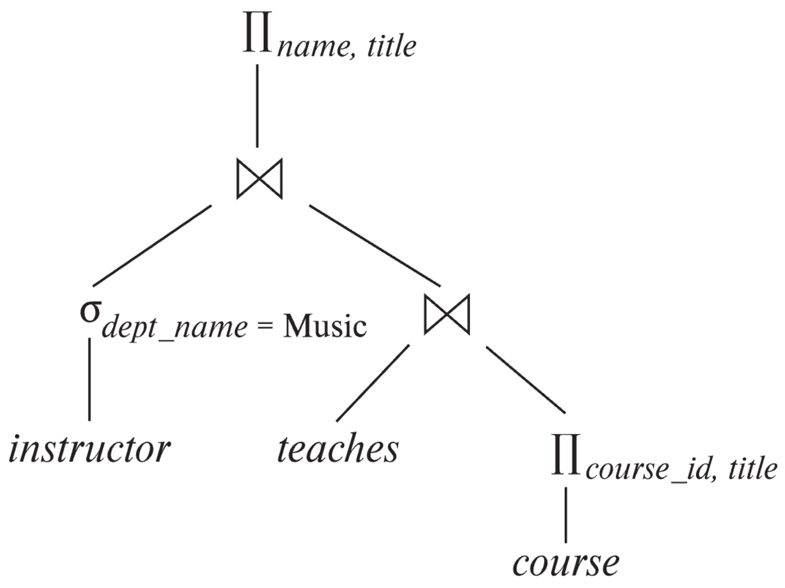
Query Optimization Problem

- These expression trees show the initial and transformed expressions.



$\Pi_{name,\ title}$
$\sigma_{dept\_name = \text{Music}}$
$\bowtie$
instructor
$\bowtie$
teaches
$\Pi_{course\_id,\ title}$
course

(a) Initial expression tree

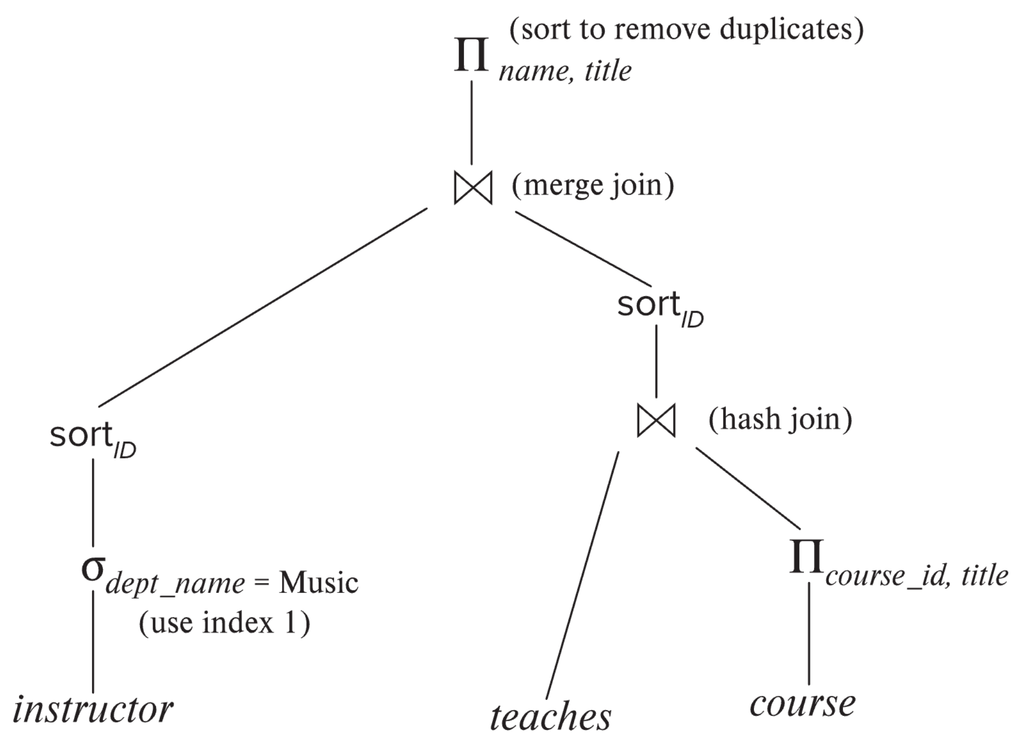


$\Pi_{name,\ title}$
$\bowtie$
$\sigma_{dept\_name = \text{Music}}$
$\bowtie$
instructor
teaches
$\Pi_{course\_id,\ title}$
course

(b) Transformed expression tree

Evaluation Plan

- An **evaluation plan** defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.



Viewing Query Evaluation Plans

- Most databases support **explain** <query>
  - Displays plan chosen by query optimizer, along with cost estimates
  - Some syntax variations between databases
    - Oracle: **explain plan for** <query> followed by **select** * **from** table (*dbms_xplan.display*)
    - SQL Server: **set showplan_text on**
- Some databases (e.g. PostgreSQL) support **explain analyse** <query>
  - Shows actual runtime statistics found by running the query, in addition to showing the plan
- Some databases (e.g. PostgreSQL) show cost as *f..l*
  - *f* is the cost of delivering first tuple and *l* is cost of delivering all results
- MySQL Workbench shows execution plan in text and visual forms, and also runtime statistics
  - From sql: **explain** your_query (text)
  - From the Menu: Query=>Explain current statement (visual)

Query Optimization Steps

- Cost difference between evaluation plans for a query can be **enormous**
  - E.g. seconds vs. days in some cases
- Steps in **cost-based query optimization**
  - Generate logically equivalent expressions using equivalence rules
  - Annotate resultant expressions to get alternative query plans
  - Choose the cheapest plan based on estimated cost
- Estimation of plan cost based on **statistical information about relations**.
  - Examples:
    - Number of tuples, number of distinct values for an attribute
  - Statistics estimation for intermediate results
    - To compute cost of complex expressions
  - Cost formulae for algorithms, computed using statistics

Improving Query with Logically Equivalent Query Plans

- Pushing Selection
- Pushing Projection
- Optimal Join Ordering
- Optimizing Nested Subqueries

Transformation Example 1: Pushing Selections

- Example of what we just saw is called "pushing selection"
  - Query: Find the names of all instructors in the Music department, along with the titles of the courses that they teach
    - $\Pi_{name,\ title}(\sigma_{dept\_name=\text{'Music'}}(instructor \bowtie (teaches \bowtie \Pi_{course\_id,\ title}(course))))$
  - Transformation using
    - $\Pi_{name,\ title}((\sigma_{dept\_name=\text{'Music'}}(instructor)) \bowtie (teaches \bowtie \Pi_{course\_id,\ title}(course)))$
- Performing the selection as early as possible reduces the size of the relation to be joined.

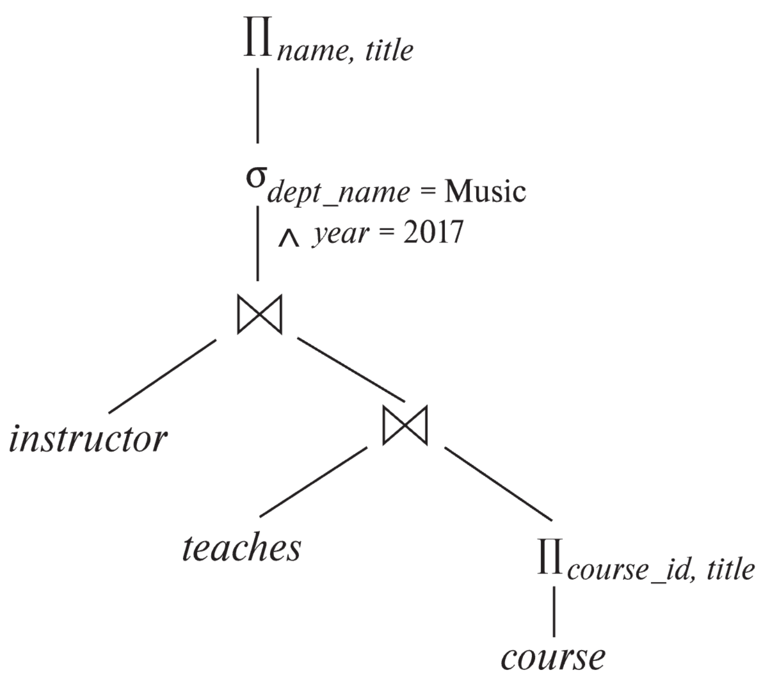
Transformation Example 2: Pushing Selections

- Recall that the cost of a join depends primarily on the size of a relation.
  - Goal: reduce the size of intermediate relations.
- Query: Find the names of all instructors in the Music department who have taught a course in 2017, along with the titles of the courses that they taught
  - $\Pi_{name,\ title}(\sigma_{dept\_name = \text{"Music"} \land year = 2017}$ $(instructor \bowtie (teaches \bowtie \Pi_{course\_id,\ title}(course))))$

- Instead of forming a full size join of three relations, we can reduce the size by applying the "perform selections early" rule, resulting in the expression

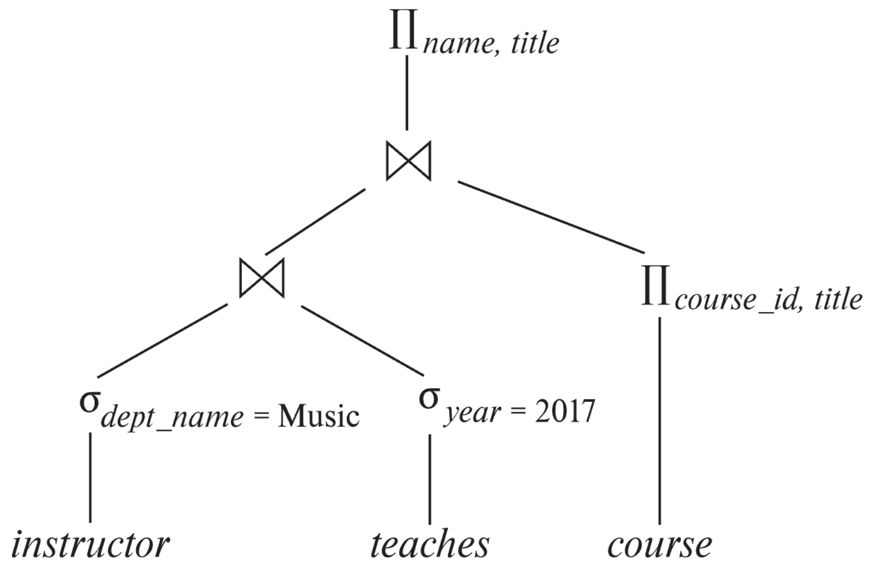
$$\Pi_{name,\ title}(\sigma_{dept\_name = \text{"Music"}}(instructor) \bowtie \sigma_{year = 2017}(teaches) \bowtie \Pi_{course\_id,\ title}(course))$$

Transformation Example: Pushing Selections



$\Pi_{name,\ title}$
$\sigma_{dept\_name = \text{Music} \wedge year = 2017}$
$\bowtie$
instructor
$\bowtie$
teaches
$\Pi_{course\_id,\ title}$
course

(a) Initial expression tree



(b) Tree after multiple transformations

Transformation Example: Pushing Projections

- Consider: $\Pi_{name,\ title}(\sigma_{dept\_name=\text{“Music”}}(instructor) \bowtie teaches) \bowtie \Pi_{course\_id,\ title}(course))))$
- When we compute

  $(\sigma_{dept\_name = \text{“Music”}}(instructor \bowtie teaches)$

  we obtain a relation whose schema is:
  (*ID, name, dept_name, salary, course_id, sec_id, semester, year)*
- *W*e want to keep name and course_id only and do it early
- Pushing projections eliminate unneeded attributes from intermediate results to get:

  $$\Pi_{name,\ title}(\Pi_{name,\ course\_id}(\sigma_{dept\_name=\text{“Music”}}(instructor) \bowtie teaches)) \bowtie \Pi_{course\_id,\ title}(course))))$$
- Performing the projection as early as possible reduces the size of the relation to be joined.

Optimal Join Ordering

- For all relations r_1, r_2, and r_3,

$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$

(Join Associativity) $\bowtie$

- If $r_2 \bowtie r_3$ is quite large and $r_1 \bowtie r_2$ is small, we choose

$$(r_1 \bowtie r_2) \bowtie r_3$$

so that we compute and store a smaller temporary relation.

Optimal Join Ordering Example

- Consider the expression

$$\Pi_{name,\ title}(\sigma_{dept\_name=\text{"Music"}}(instructor) \bowtie teaches) \bowtie \Pi_{course\_id,\ title}(course))))$$

- Could compute $teaches \bowtie \Pi_{course\_id,\ title}(course)$ first, and join result with temp table $\sigma_{dept\_name=\text{"Music"}}(instructor)$
but the result of the first join is likely to be a large relation.
- But: Only a small fraction of the university's instructors are likely to be from the Music department
  - It is better to compute join our temp table with teaches

    $$\sigma_{dept\_name=\text{"Music"}}(instructor) \bowtie teaches$$

    first.

Optimizing Nested Subqueries

- Nested query example:

```
select name
from instructor
where exists (select *
                    from teaches
                    where instructor.ID = teaches.ID and teaches.year = 2019)
```

- SQL conceptually treats nested subqueries in the where clause as functions that take parameters and return a single value or set of values
  - Parameters are variables from outer level query that are used in the nested subquery; such variables are called **correlation variables**
- Conceptually, nested subquery is executed once for each tuple in the cross-product generated by the outer level **from** clause
  - Such evaluation is called **correlated evaluation**

Optimizing Nested Sub-queries (continued)

- Correlated evaluation may be **quite inefficient** since
  - a large number of calls may be made to the nested query
  - there may be unnecessary I/O as a result
- SQL optimizers attempt to transform nested subqueries to joins where possible, enabling use of efficient join techniques
- E.g.: earlier nested query can be rewritten as

```
with t1 as
      (select distinct ID
      from teaches
      where year=2019)
select name
from instructor, t1
where instructor.ID = t1.ID
```

Using Materialized Views

- Recall that a **materialized view** is a view which content is pre-computed and stored.
- Consider the view

create view *department_total_salary*(*dept_name, total_salary*) **as**
select *dept_name*, **sum**(*salary*)
from *instructor*
group by *dept_name*

- Materializing the above view would be very useful if the total salary by department is requested frequently
  - Saves the effort of finding multiple tuples and adding up their amounts

Materialized View Maintenance

- The task of keeping a materialized view up-to-date with the underlying data is known as **materialized view maintenance**
- Materialized views can be maintained by re-computation on every update
- A better option is to use **incremental view maintenance**
  - Changes to database relations are used to compute changes to the materialized view, which is then updated
- View maintenance can be done by
  - Manually defining triggers on insert, delete, and update of each relation in the view definition
  - Manually written code to update the view whenever database relations are updated
  - Periodic re-computation (e.g. nightly)
  - Incremental maintenance supported by many database systems
    - Avoids manual effort/correctness issues

Statistics for Cost Estimation

Cost Estimation

- Cost of each operator as described earlier today
  - Need statistics of input relations
    - E.g. number of tuples, sizes of tuples
- Inputs can be results of sub-expressions
  - Need to estimate statistics of expression results
  - To do so, we require additional statistics
    - E.g. number of distinct values for an attribute
    - Many databases also store *n* **most-frequent values** and their counts

Statistical Information for Cost Estimation

- The database-system catalog stores the following statistical information about database relations:
  - n_r*:* number of tuples in a relation *r.*
  - b_r: number of blocks containing tuples of *r.*
  - l_r: size of a tuple of *r.*
  - f_r*:* blocking factor of *r* — i.e., the number of tuples of *r* that fit into one block.
  - *V(A, r):* number of distinct values that appear in *r* for attribute *A*

Statistical Information Example

Our example:

student $\bowtie$ *takes*

Catalog information for join examples:

- $n_{student} = 5000$.
- $f_{student} = 50$, which implies that $b_{student} = 5000/50 = 100$.
- $n_{takes} = 10000$.
- $f_{takes} = 25$, which implies that $b_{takes} = 10000/25 = 400$.
- *V(ID, takes)* = 2500, which implies that on average, each student who has taken a course has taken 4 courses.
  - Attribute *ID* in *takes* is a foreign key referencing *student.*
- *V*(*ID, student*) = 5000 (*primary key!*)

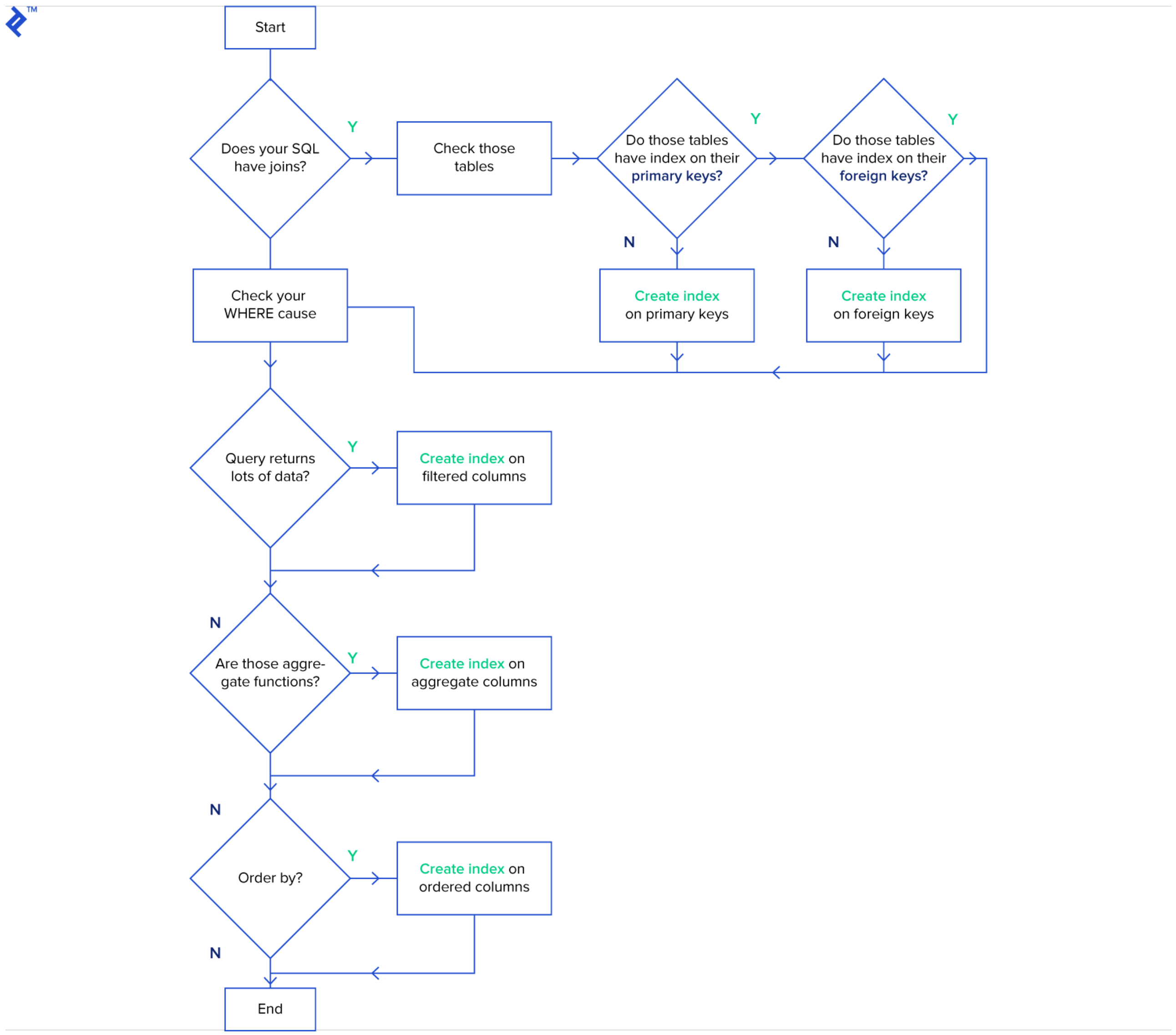
Use of Statistics/Histograms

- Histograms and other statistics usually computed based on a random sample
- Statistics may be out of date
    - Some database require an **analyze** command to be executed to update statistics
    - Others automatically recompute statistics
        - e.g. when number of tuples in a relation changes by some percentage

Heuristic: Rules-based Optimization (RBO)

- Cost-based optimization is expensive
- Systems may use *heuristics* to reduce the number of choices that must be made in a cost-based fashion.
- Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:
  - Perform selection early (reduces the number of tuples)
  - Perform projection early (reduces the number of attributes)
  - Perform most restrictive selection and join operations (i.e. with smallest result size) before other similar operations.
  - Some systems use only heuristics, others combine heuristics with partial cost-based optimization.

Example of Database Tuning Tips


Start
Does your SQL have joins?
Y
Check those tables
Do those tables have index on their primary keys?
Y
Do those tables have index on their foreign keys?
Y
N
Create index on primary keys
N
Create index on foreign keys
Check your WHERE cause
Query returns lots of data?
Y
Create index on filtered columns
N
Are those aggregate functions?
Y
Create index on aggregate columns
N
Order by?
Y
Create index on ordered columns
N
End

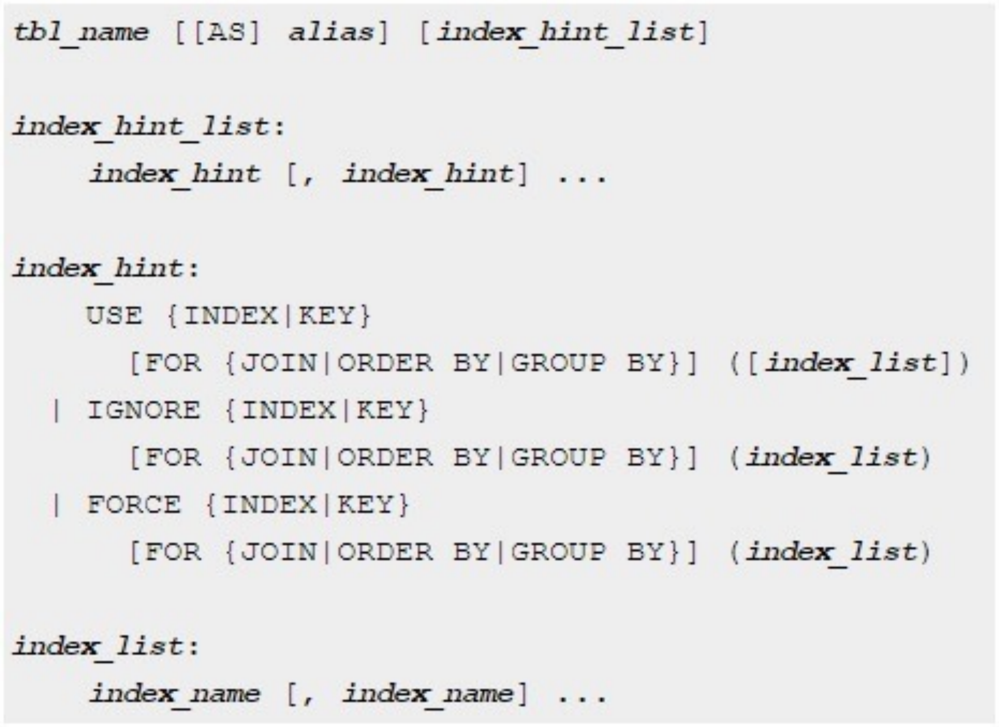
https://www.toptal.com/sql-server/sql-database-tuning-for-developers

Tips for MySQL Query Optimization

- Avoid using functions in predicates
  - Database doesn't use index if it has some function predefined in the column.
- Avoid using wildcard (%) in the beginning of a predicate
  - In most cases, this brings major performance limitation.
- Avoid unnecessary columns in SELECT clause
  - Unnecessary columns cause additional load on the database
- Use inner join, instead of outer join if possible
  - Outer join limits MySQL query optimization options
- The ORDER BY clause is mandatory in SQL if you expect to get a sorted result : Sort is slow!
- Use DISTINCT and UNION only if it is necessary
  - Again: Sort is slow!
  - UNION ALL has more efficiency, e.g. you know there are no duplicates

Tips for MySQL Query Optimization (continued)

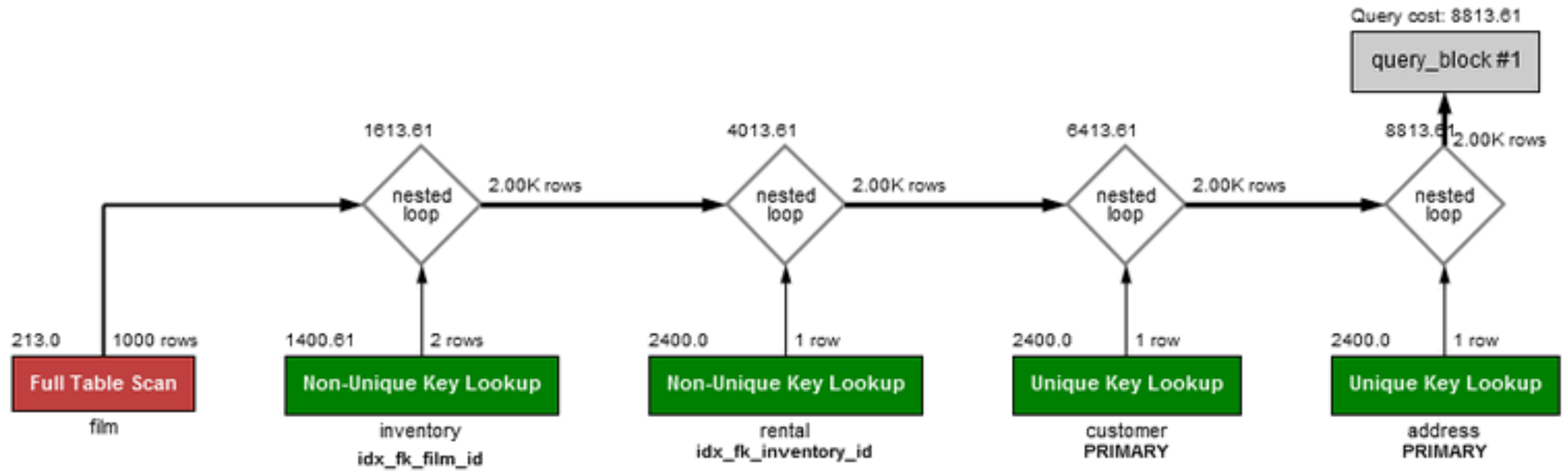
- Use Optimizer hints
  - MySQL has: USE INDEX, IGNORE INDEX, FORCE INDEX
    - More info: https://dev.mysql.com/doc/refman/8.0/en/index-hints.html



```
tbl_name [[AS] alias] [index_hint_list]

index_hint_list:
    index_hint [, index_hint] ...

index_hint:
    USE {INDEX|KEY}
      [FOR {JOIN|ORDER BY|GROUP BY}] ([index_list])
  | IGNORE {INDEX|KEY}
      [FOR {JOIN|ORDER BY|GROUP BY}] (index_list)
  | FORCE {INDEX|KEY}
      [FOR {JOIN|ORDER BY|GROUP BY}] (index_list)

index_list:
    index_name [, index_name] ...
```

- Use EXPLAIN plan
  - Most RDBMS have that functionality

MySQL Visual Explain Example



- Graphic Conventions
  - Standard Boxes: tables
  - Rounded boxes: operations such as GROUP and SORT
  - Framed boxes: sub-queries
  - Diamonds: joins
- Textual Conventions
  - Standard text below boxes: table (or alias) name
  - Bold text below boxes: key/index that was used
  - Number in top right of a box: number of rows used from the table after filtering
  - Number in top left of a box: relative cost of accessing that table
  - Number to the right of nested loop diamonds: number of rows produced by the JOIN
  - Number above the loop diamonds: relative cost of the JOIN

Lab

- Lab: Using explain plan
- To submit:
    - include **ALL sql queries**
    - include the **SCREENSHOT of the Task 5 Execution Plan** in your submission.