CSE462/562: Database Systems (Spring 23)

Lecture 15: Query processing - single-table query

3/30/2023 & 4/4/2023

University at Buffalo
Department of Computer Science and Engineering
School of Engineering and Applied Sciences

Single-table queries

- We'll start with the simplest single-table queries w/o or w/ aggregations
  - How to translate it into a query plan?
  - How to implement each operator?
  - How to measure the cost of each operator?

```
SELECT E
FROM R
WHERE P
ORDER BY S
```

```
SELECT G, SUM(E)
FROM R
WHERE P
GROUP BY G
HAVING P'
ORDER BY S
```

SQL -> logical plan

- We'll start with the simplest single-table queries w/o or w/ aggregations
  - How to translate it into a query plan?
  - How to implement each operator?
  - How to measure the cost of each operator?

```
SELECT E
FROM R
WHERE P
ORDER BY S
```

$\Rightarrow$

$$Sort_S(\pi_E \sigma_P R)$$

```
SELECT G, SUM(E)
FROM R
WHERE P
GROUP BY G
HAVING P'
ORDER BY S
```

$\Rightarrow$

$$Sort_S\left(\sigma_{P'}\ {}_G\gamma_{SUM(E)}\sigma_P R\right)$$

Logical plan -> physical plan

- We'll start with the simplest single-table queries w/o or w/ aggregations
  - How to translate it into a query plan?
  - How to implement each operator?
  - How to measure the cost of each operator?

- A few basic operators
  - Selection: σ
  - Projection: π (w/ and w/o deduplication)
  - Aggregation: γ w/o or w/ group by
  - Set operators: $\cup, -, \cap$
  - Hashing or Sorting (later lectures)
  - Cartesian product: $\times$ or Join: $\bowtie$ (later lectures)

- Question: what are the alternatives? How to evaluate their efficiency?

Measuring cost

- We'll start with the simplest single-table queries w/o or w/ aggregations
  - How to translate it into a query plan?
  - How to implement each operator?
  - How to measure the cost of each operator?

- For disk-based systems, we mainly measure the number of I/Os
  - Differences between random I/O and sequential I/O
  - Faster storage -> also need to measure the CPU cost

- A simple cost model
  - t_T: average time to transfer a page of data (data transfer time)
  - t_S: average time to randomly seek data (seek time + rotation delay)
    - For SSD, time overhead for initiating an I/O request

  - Cost = $B \times t_T + S \times t_S$
    - B: number of pages read/written; S: number of random I/O

Typical t_T and T_S

	HDD*	**SSD†**
t_T (ms)	0.1	0.01
t_S (ms)	4	0.09

Data from DB Concept book (Ch. 15.2).
Assuming 4KB pages.
* typical HDD with 40 MB/s transfer rate, 15000 rpm disk in 2018
† typical SATA SSD that supports 10K IOPS (QD-1), 400 MB/s sequential read rate

Measuring cost

- Other assumptions
  - Ignoring the buffer effect for random pages
    - Do consider the private workspace size M for the operators
  - Omitting the cost of transferring output to the user/disk
    - Common to any equivalent plan

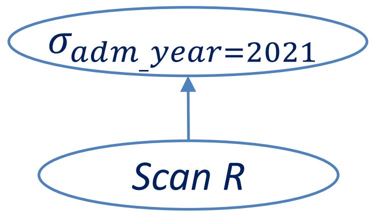
- Notations: for relation R
  - T_R: number of records, N_R: number of pages in its heap file, B_R: (average) number of tuples per page
  - h_I: height of a B-tree index I over the file
  - M: private workspace size in pages
- Running example
  - $t_S = 4\ ms$, $t_T = 0.1\ ms$, *4000-byte page*
  - Student: R(sid: int, name: varchar(19), login: varchar(19), major: char(2), adm_year: int)
    - 50 bytes/tuple, $B_R = 80$, $T_R = 40{,}000$, $N_R = 500$
  - Enrollment: E(sid: int, semester: char(3), cno: int, grade: double)
    - 20 bytes/tuple, $B_E = 200$, $T_E = 200{,}000$, $N_E = 1000$

Selection $\boldsymbol{\sigma}$

- Scan is usually the leaf-level of logical plans
  - Represents reading an entire relation -- not really a relational operator
- Selection $\sigma_P Q$
  - P is usually conjunctions or disjunctions $Q.attr\ op\ value$ but can also be User-Defined Functions (UDF)
  - selects records satisfying some predicate from the child
  - Child may be a scan or some other operators

  - Many possible implementation of selection depending on
    - the predicate P
    - the available file/index for the scan

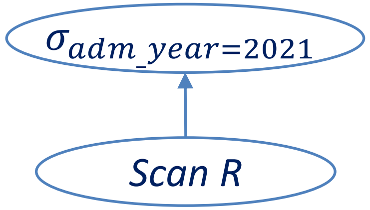
op is an operator: <, <=, =, <>, >, >=, …



Logical plan for $\sigma_{adm\_year=2021} R$

Simple selection: linear scan

- Consider a simple selection $\sigma_{R.attr\ op\ value} R$
  - Assume that the child is a relation stored in some disk file/index
- Most straight-forward implementation is linear scan
  - Scan each page and each record on the page
    - emits a record only if the predicate $R.attr\ op\ value$ evaluates to true
  - Applies to any predicate P or file
  - Also works for pipelining -- can do selection on the fly without writing temporary files

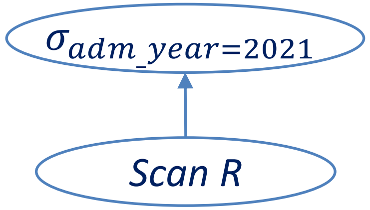


Logical plan for $\sigma_{adm\_year=2021} R$

- Cost: $t_S + N_R \times t_T$
  - 1 seek to the start of the file and N_R pages to read
  - the “last resort” -- usually the slowest implementation
  - cost for $\sigma_{adm\_year=2021} R$: $t_S + 500 \times t_T = 54\ ms$

Simple selection: binary search on sorted file

- If the file on R is sorted *on the search key*
  - use binary search to locate the first record, then scan the remaining tuples
  - Cost: $\lceil \log_2 N_R \rceil \times (t_S + t_T) + (N - 1) \times t_T$



Logical plan for $\sigma_{adm\_year=2021} R$

binary search cost, all random I/Os

scanning cost, -1 accounts for the first page read during binary search

- N: the number of pages with matching records, which can be approximated as
  - $\mathrm{N} = \lceil s N_R \rceil$
  - s: *selectivity, i.e., the percentage of records that satisfy the predicate (discussed later in QO)*

- Running example: suppose R is sorted on adm_year and selectivity is $s = 10\%$
  - cost = $\lceil \log_2 500 \rceil \times (t_S + t_T) + (\lceil 0.1 \times 500 \rceil - 1) * t_T = 41.8\ ms$

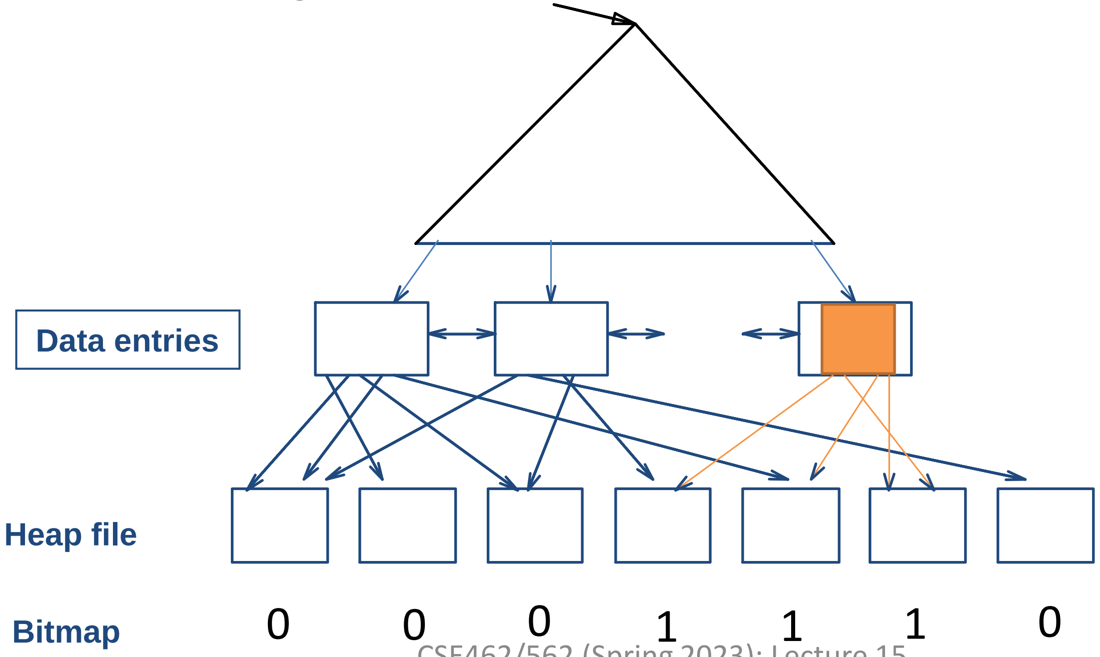
Simple selection: index scan

T: # of matching records
F: # of data entries per leaf page
N: # of pages with matching records

- If the file has a B-Tree index I over the search key, assuming alternative 2 for data entries
  - cost varies depending on whether it's clustered
- Assuming selectivity is $s = 0.1$, the number of matching records is T and the number of pages with matching records is N, assume $h = 3$
  cost =
  - $h_I \times (t_T + t_S)$ for finding qualifying data entries +
  - cost for retrieving the heap records
    - clustered: $t_S + N \times t_T \approx t_S + \lceil sN_R \rceil \times t_T$ (total $= 12.3 + 9 = 21.3\, ms$)
    - unclustered: $\left(\left\lceil \frac{T}{F} \right\rceil - 1\right) \times t_T + T \times (t_T + t_S)$
      $= \left(\left\lceil \frac{\lceil sT_R \rceil}{F} \right\rceil - 1\right) \times t_T + \lceil sT_R \rceil \times (t_T + t_S)$ (total = $12.3 + 16401.3 = 16413.3\, ms$)
      - can we do better?

Simple selection: index scan (cont'd)

- Refinement for unclustered index scan: bitmap index scan
  1. Initialize a bitmap with one bit for each page in the file (usually fits in mem even for a large file)
  2. Find the first qualifying data entry
  3. Scan all the data entries and mark all the unique pages with the matching records in the bitmap
  4. Scan all the pages with bit 1 (linear scan on page)
- Alternative: collect all RID in memory in step 3, sort and fetch tuples in RID order
  - more expensive unless RIDs fit in memory
  - might make sense for faster storage (thus CPU cost matters)



Simple selection: index scan (cont'd)

T: # of matching records
F: # of data entries per leaf page
N: # of pages with matching records

- Cost of bitmap index scan =
  - (tree search) $h \times (t_S + t_T) +$
  - (scan of data entries) $\left(\left\lceil \frac{T}{F} \right\rceil - 1\right) \times t_T +$ (assuming leaf level is consecutive from bulk loading)
  - (scan of data pages) $N \times (t_S + t_T)$ (when N is small and thus most involve random seeks) or
    $t_S + N \times t_T$ (when N is close to N_R and it's close to sequential scan)
- Example 1 (large selectivity): $s = 0.9, \mathrm{F} = 300, \mathrm{T} = \lceil s\mathrm{T_R} \rceil = 36000, \mathrm{N} = 500$ =>
  cost = $4.1 \times 3 + 0.1 \times \left(\lceil \frac{36000}{300} \rceil - 1\right) + 4 + 0.1 \times 500 = 78.2\ ms$ (unclustered)
  vs $4.1 \times 3 + 4 + 0.1 \times \lceil 0.9 \times 500 \rceil = 61.3\ ms$ (clustered)
- Example 2 (moderate selectivity): $s = 0.1, F = 300, T = \lceil sT_R \rceil = 4000, \mathrm{E}[\mathrm{N}] \approx 500$ (think: why?)
  cost = $4.1 \times 3 + 0.1 \times \left(\lceil \frac{4000}{300} \rceil - 1\right) + 4 + 0.1 \times 500 = 67.6\ ms$ (unclustered)
  vs $4.1 \times 3 + 4 + 0.1 \times \lceil 0.1 \times 500 \rceil = 21.3\ ms$ (clustered)
- Example 3 (small selectivity): $s = 0.0001, F = 300, T = \lceil sT_R \rceil = 4, N = 4$
  cost = $4.1 \times 3 + 0.1 \times \left(\lceil \frac{4}{300} \rceil - 1\right) + 4.1 \times 4 = 28.7\ ms$ (unclustered)
  vs $4.1 \times 3 + 4 + 0.1 \times \lceil 0.0001 \times 500 \rceil = 16.4\ ms$ (clustered)
- Trade-offs:
  - Only slightly more expensive than a linear scan when selectivity is close to 1
  - Only slightly more expensive than a regular secondary index scan when selectivity is close to 0 (<< linear scan)
  - Only works poorly when the selectivity is moderate -- better off with clustered index
    - To show that, let $I_i = 1$ if page i has any matching record (an indicator variable) and assume uniform distribution in search key
    - $E[N] = \sum_{1 \le i \le N_R} E[I_i] = \sum_{1 \le i \le N_R} \Pr\{I_i = 1\} = N_R(1 - (1 - s)^{B_R})$

General selection predicates

- Atom predicate: $attr\ op\ value$ or UDF
- General predicates:
  - Conjunction ∧ (and), disjunction ∨ (or), negation ¬ (not) of atoms or general predicates
  - e.g., $\sigma_{(adm\_year>=2019 \vee major='CS') \wedge sid >= 1000}R$
- Most general cases can always be handled by linear scans
  - Slow!
- Optimization for special cases:
  - Conjunction of simple selection predicates $\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_r$
    - where θ_i is an atom
  - Disjunction of selection predicates $\theta_1 \vee \theta_2 \vee \cdots \vee \theta_r$
  - Transforming a predicate P into *Conjunctive Normal Form (CNF)* or *Disjunction Normal Form (DNF)* for additional optimization opportunities
    - e.g., $(adm\_year >= 2019 \vee major =' CS') \wedge sid >= 1000$ *(CNF)*
      $\Leftrightarrow (adm\_year >= 2019 \wedge sid \geq 1000) \vee (major =' CS' \wedge sid \geq 1000)$ *(DNF)*

Conjunctive selection with one index

- $\theta_1 \land \theta_2 \land \cdots \land \theta_r$
  - Choosing one or a prefix of predicates that can be answered using one index
    - Apply the rest of the predicates over the result on the fly
  - For instance, a B-Tree over (f_1, f_2) can select for predicates over a prefix of its index keys
    - $f_1\ op\ value$ (where $op \in \{<, \leq, =, >, \geq\}$)
    - $f_1 = value \land f_2\ op\ value$ (where $op \in \{<, \leq, =, >, \geq\}$)
    - If allow using skip scan (jump scan), f_2 op value or $f_1\ op\ value \land f_2\ op\ value$
  - What if there're multiple choices?
    - Considerations: selectivity, type of indexes, actual cost (access path selection in QO)

  - Cost is the same as index scans/bitmap index scans

Conjunctive selection with multiple indexes

- $\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_r$
  - What if the atoms or several conjunctions of atoms can be answered by different indexes?
  - Example: $\sigma_{major='CS' \wedge adm\_year=2021} R$ when we have two indexes $I_1(major)$ and $I_2(adm\_year)$
- Algorithm:
  1. Collect all the RIDs using both indexes
  2. Compute the intersection of the RIDs
  3. Fetch the heap records of the RIDs in the result set
- Cost: index search + collecting data entries+ sort + intersection + fetching heap records

Partial matches for conjunctive selection

- $\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_r$
  - What if only part of the predicates can be optimized with indexes
    - Apply the remaining predicates over the result and discard those that do not satisfy
    - e.g.,$\sigma_{major='CS' \wedge adm\_year=2021}$ with a hash index $I(major)$
      - Index Scan for all CS majors using $I(major)$
      - Apply the predicate $adm\_year = 2021$ over the heap records on the fly

    - Note the remaining predicates do not need to be in conjunctive normal form!
      - Can be arbitrary predicates (e.g., UDF)

Disjunction selection with multiple indexes

- $\theta_1 \vee \theta_2 \vee \cdots \vee \theta_r$
  - Only optimizable if all clauses θ_i can be optimized using some index
  - Otherwise, fall back to linear scan
- Algorithm:
  1. Collect all the RIDs using both indexes
  2. Compute the union of the RIDs
  3. Fetch the heap records of the RIDs in the result set

- Cost: index search + collecting data entries+ sort + union + fetching heap records

An excursion: expression evaluation

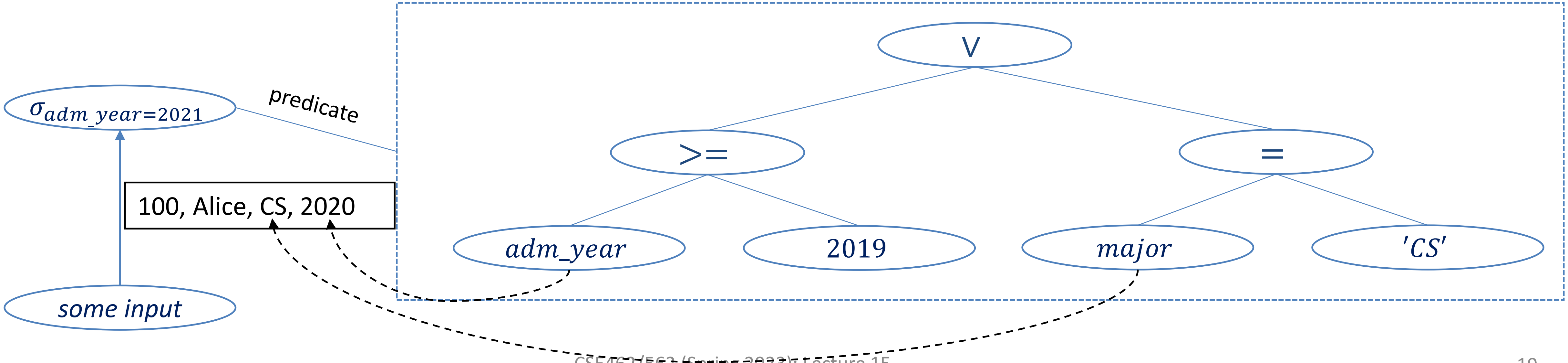
- So far, we assume expression evaluation is a black box
  - Does the predicate evaluate to true in selection?
  - Projection list evaluation?
  - …

- How does it work?
  - How costly are they?

Expression tree

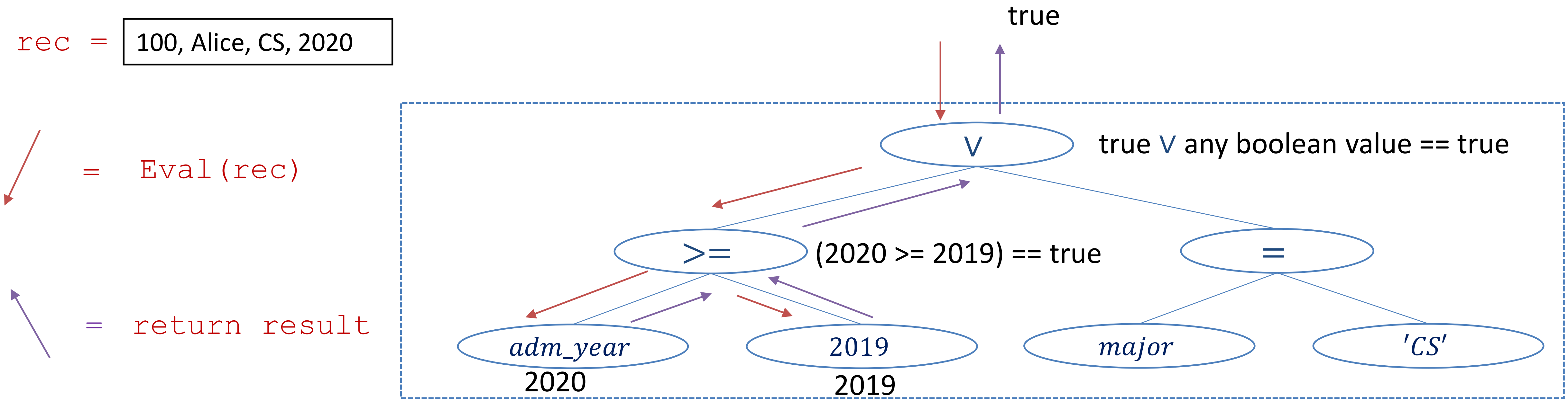
- A tree that represents an expression
  - Leaf nodes: literals, variables
  - Internal nodes: operators (+, -, *, /, …), function calls, …
- Expressions in QP are attached to a plan node
  - Variables refers to columns in the output of some plan node
    - usually output from child, but could be intermediate outputs within certain operators
- Example: predicate $adm\_year >= 2019 \lor major = 'CS'$

Q: what are the variables in query plan?
A: (short answer) columns in the output



Expression evaluation

- Interpretation vs Compilation
  - type checking?
- In the course project Taco-DB, we use interpretation (for ease of implementation)
  - recursive evaluation through `Eval()` calls



Projection π

- Without deduplication
  - evaluate projection list for the records on the fly
  - cost: no additional I/O
  - sometimes baked into other operators (i.e., all operators can be followed by an implicit projection)

- With deduplication
  - Requires materialization (blocking)
  - Hash or Sort
    - Hash -> build a hash table where duplicates are dropped
    - Sort -> emit a record only if it is the first record or it is different from the previous one
  - Result set fits in memory => easy to implement (does not add I/O cost)
  - When result sets exceed configured workspace size M,
    - Need to use external hashing and sorting algorithms (next lecture)
    - Optimization opportunities
    - Will come back to this later after we discuss external hashing and sorting

Projection over selection: Index only scan

- For $\pi_{E_1,E_2,\ldots,E_k}\sigma_P R$
  - Let Var(E) be the set of attributes in the expression E
    - e.g., $Var(R.sid > 100) = \{R.sid\}$
      $Var\big(length(R.name) + length(R.login)\big) = \{R.name, R.login\}$
  - Suppose there's an index I over R whose index key is K_I, such that
    - $\bigcup_{1\le i\le k} Var(E_i) \cup Var(P) \subseteq K_I$
    - we can perform an index scan without fetching the heap records (index-only scan)
    - Note: attributes that only appear in the projection list can be non-key columns in index
    - Might be useful even if search key does not match the index key
      - Cheaper than heap scan due to high fan-out

- Cost = tree search cost + cost for scanning all matching data entries
  = $h \times (t_S + t_T) + \left(\left\lceil \frac{T}{F} \right\rceil - 1\right) \times t_T$ (assuming leaf level is consecutive on disk due to bulk loading)
- Example: $\pi_{adm\_year,sid}\sigma_{adm\_year=2021}R$, B-Tree index on $R(adm\_year, sid)$
  $\mathrm{h} = 3, s = 0.1, T = \lceil sT_R \rceil = 4000, F = 300$
  - cost of index-only scan = $3 \times 4.1 + \left(\left\lceil \frac{4000}{300} \right\rceil - 1\right) \times 0.1 = 13.6\ ms$
    vs cost of index scan (clustered) = $3 \times 4.1 + 4 + 0.1 \times \lceil 0.1 \times 500 \rceil = 21.3\ ms$

Aggregation γ without grouping

> F *is an aggregation function, e.g.,*
> $SUM, COUNT, VAR, STDDEV, AVG, MIN, MAX$ *or UDA etc.*

- $\gamma_{F_1(E_1),F_2(E_2),\dots,F_k(E_k)}Q$
  - Blocking
  - Only produce one row of output

  - An aggregation can be expressed as three functions: $F = \left(F^{init}, F^{acc}, F^{final}\right)$
    - Initialization $F^{init}: void \ \rightarrow A$ (where A is some internal state of the aggregation)
    - Accumulation $F^{acc}: (A, T) \rightarrow A$ or $(A, T) \rightarrow void$
    - Finalization $F^{final}: A \ \rightarrow V$ (where V is the final type of the aggregation)
    - Some systems also have an optional combine function $F^{combine}: (A, A) \rightarrow A$
      - allows parallelizing the aggregation

  - Example: AVG of integers
    - AVG^{init} ( ): create a pair of (s, c) -- s: sum of values, c: number of values
    - $AVG^{acc}\left((s, c), x\right) = (s + x, c + 1)$
    - $AVG^{final}\left((s, c)\right) = 1.0 \ * s \ / \ c$
  - Cost: does not add additional I/O cost

Aggregation γ without grouping

F is an aggregation function, e.g.,
$SUM, COUNT, VAR, STDDEV, AVG, MIN, MAX$ or UDA etc.

- Example: AVG of integers
  - AVG^{init} ( ): create a pair of (s, c) -- s: sum of values, c: number of values
  - $AVG^{acc}\big((s, c), x\big) = (s + x, c)$
  - $AVG^{final}\big((s, c)\big) = 1.0 \; * s \; / \; c$

  - Consider a column in a table with the following values
    - $5, 4, 1, 3, 2$
    - Steps:
      - AVG^{init} ( ) = (0.0, 0)
      - $AVG^{acc}\big((0.0, 0), 5\big) = (5.0, 1)$
      - $AVG^{acc}\big((5.0, 1), 4\big) = (9.0, 2)$
      - $AVG^{acc}\big((9.0, 2), 1\big) = (10.0, 3)$
      - $AVG^{acc}\big((10.0, 3), 3\big) = (13.0,4)$
      - $AVG^{acc}\big((13.0, 4), 2\big) = (15.0, 5)$
      - $AVG^{final}\big((15.0, 5)\big) = 3.0 = \frac{5+4+1+3+2}{5}$

Aggregation γ with grouping

- ${}_{G_1,G_2,\ldots,G_n}\gamma_{F_1(E_1),F_2(E_2),\ldots,F_k(E_k)}Q$
  - Blocking
  - One record per group (distinct values in $G_1, G_2, \ldots, G_n$)
    - Let group by columns be $\mathcal{G} = (G_1, G_2, \ldots, G_n)$
  - Solution: sorting or hashing

Aggregation γ with grouping

- $_{G_1,G_2,\ldots,G_n}\gamma_{F_1(E_1),F_2(E_2),\ldots,F_k(E_k)}Q$
  - Blocking
  - One record per group (distinct values in $G_1, G_2, \ldots, G_n$)
    - Let group by columns be $\mathcal{G} = (G_1, G_2, \ldots, G_n)$
  - Sort-based solution: sort all tuples in Q on $\mathcal{G}$; for each result t
    1. If t is the first one, $g \leftarrow \pi_{\mathcal{G}} t$ and $a_1 \leftarrow F_1^{init}(\,), \ldots \mathrm{a_k} \leftarrow F_k^{init}(\,)$
    2. If t is not the first and $\pi_{\mathcal{G}} t \neq g$, emit $\mathrm{g} \circ \left(F_1^{final}(a_1), \ldots F_k^{final}(a_k)\right)$
       - Then, $\mathrm{g} \leftarrow \pi_{\mathcal{G}} t$ and $\mathrm{a_1} \leftarrow F_1^{init}(\,), \ldots \mathrm{a_k} \leftarrow F_k^{init}(\,)$
    3. In both cases, $\mathrm{a_1} \leftarrow F_1^{acc}(a_1, \pi_{E_1} t\,), \ldots \mathrm{a_k} \leftarrow F_k^{acc}(a_k, \pi_{E_k} t\,)$
    4. After the last record is read, emit the last group as $\mathrm{g} \circ \left(F_1^{final}(a_1), \ldots F_k^{final}(a_k)\right)$
  - If there are too many groups, use external sorting
    - Optimization opportunities (next lecture)

Aggregation γ with grouping

- Example for sort-based solution:
  - Consider two columns (x, y) with the following values
    - $(1, 1.0), (2, 2.0), (1, 4.0), (2, 6.0)$
    - ${}_{x}\gamma_{SUM(y)}$
    - Step 1: sort by x
      - $(1, 1.0), (1, 4.0), (2, 2.0), (2, 6.0)$
    - Step 2: scan and calculate the group aggregates
      - Scan (1, 1.0): $g \leftarrow x = 1$, $a_1 \leftarrow 0.0 + 1.0 = 1.0$
      - Scan (1, 4.0): $a_1 \leftarrow a_1 + 4.0 = 5$
      - Scan (2, 2.0):
        - Since $x = 2 \neq g = 1$, emit $(g, a_1) = (1, 5.0)$ as a result
        - $g \leftarrow x = 2$, $a_1 \leftarrow 0.0 + 2.0 = 2.0$
      - Scan (2, 6.0): $a_1 \leftarrow a_1 + 6.0 = 8.0$
    - Step 3: emit the final group: $(g, a_1) = (2, 8.0)$

Result

x	SUM(y)
1	5.0
2	8.0

Aggregation γ with grouping

- ${}_{G_1,G_2,\ldots,G_n}\gamma_{F_1(E_1),F_2(E_2),\ldots,F_k(E_k)} Q$
  - Blocking
  - One record per group (distinct values in $G_1, G_2, \ldots, G_n$)
    - Let group by columns be $\mathcal{G} = (G_1, G_2, \ldots, G_n)$ or $\bigcup_{1\le i\le n} Var(G_i)$
  - Hash-based solution: create a hash table from $\mathcal{G}$ to $(A_1, A_2, \ldots, A_k)$
    - Maintain the hash table using the aggregation functions while reading records from Q
    - After deplete the records in Q, scan the hash table, and
    - emit one row for each distinct value in $\mathcal{G}$ and compute its final value using the finalization functions

  - Again, if there are too many groups, use external hashing
    - Optimization opportunities (next lecture)

Aggregation γ with grouping

- Example for hash-based solution:
  - Consider two columns (x, y) with the following values
    - $(1, 1.0), (2, 2.0), (1, 4.0), (2, 6.0)$
      - assume $h(1) = 2, h(2) = 0$
    - ${}_{x}\gamma_{SUM(y)}$
    - Step 1: create an empty hash table
    - Step 2: scan records and maintain aggregates
      - scan (1, 1.0): $x[h(1)] \leftarrow x = 1$, $a_1[h(1)] \leftarrow 0.0 + y = 1.0$
      - scan (2, 2.0): $x[h(2)] \leftarrow x = 2$, $a_1[h(2)] \leftarrow 0.0 + y = 2.0$

hash table

h(x)	**0**	**1**	**2**	**3**
x	2		1	
a_1	2.0		1.0	

Aggregation γ with grouping

- Example for hash-based solution:
  - Consider two columns (x, y) with the following values
    - $(1, 1.0), (2, 2.0), (1, 4.0), (2, 6.0)$
      - assume $h(1) = 2, h(2) = 0$
    - ${}_x\gamma_{SUM(y)}$
    - Step 1: create an empty hash table
    - Step 2: scan records and maintain aggregates
      - scan (1, 1.0): $x[h(1)] \leftarrow x = 1$, $a_1[h(1)] \leftarrow 0.0 + y = 1.0$
      - scan (2, 2.0): $x[h(2)] \leftarrow x = 2$, $a_1[h(2)] \leftarrow 0.0 + y = 2.0$
      - scan (1, 4.0): $a_1[h(1)] \leftarrow a_1[h(1)] + y = 1.0 + 4.0 = 5.0$
      - scan (2, 6.0): $a_1[h(2)] \leftarrow a_1[h(2)] + y = 2.0 + 6.0 = 8.0$
    - Step 3: scan hash table and emit results

hash table

h(x)	0	1	2	3
x	2		1	
a_1	8.0		5.0	

Result

x	SUM(y)
1	5.0
2	8.0

Set operators ∪,∩ , —

- SQL performs deduplication before the set operators by default, unless one specifies ALL
  - e.g., A = {1, 1, 2}, B = {1, 2}
    - `SELECT * FROM A EXCEPT SELECT * FROM B;` -- result is empty
    - `SELECT * FROM A EXCEPT ALL SELECT * FROM B;` -- result is {1} (one row)
  - UNION ALL can be made pipelining: emit everything from LHS and then RHS

  - All the others are similar: using UNION as an example
    - Solution: sorting or hashing
    - sorting: sort A and B separately, merge them together by removing any duplicates
      - Similar to a sort-merge join we will discuss in later lectures
    - hashing: create a hash table over all the attributes, scan A and B
      - Only keep the first occurrence of each distinct value
  - Once again, optimization opportunities exist when the result set(s) of A and/or B do not fit in memory

Summary

- This lecture:
  - Operators for single-table queries
    - cost analysis
  - Expression evaluation
- Next lecture:
  - External sorting