Database Management Systems

Lecture 10

Evaluating Relational Operators

Query Optimization

Sabina S. CS

* queries – composed of <u>relational operators</u>:

- selection (σ)
  - selects a subset of records from a relation
- projection (π)
  - eliminates certain columns from a relation
- join ($\otimes$)
  - combines data from two relations
- cross-product ($R1 \times R2$)
  - returns every record in R1 concatenated with every record in R2
- set-difference ($R1 - R2$)
  - returns records that belong to R1 and don't belong to R2
- union ($R1 \cup R2$)
  - returns all records in relations R1 and R2

*Review lecture notes on *Relational Algebra* (*Databases* course)

* queries – composed of <u>relational operators</u>:

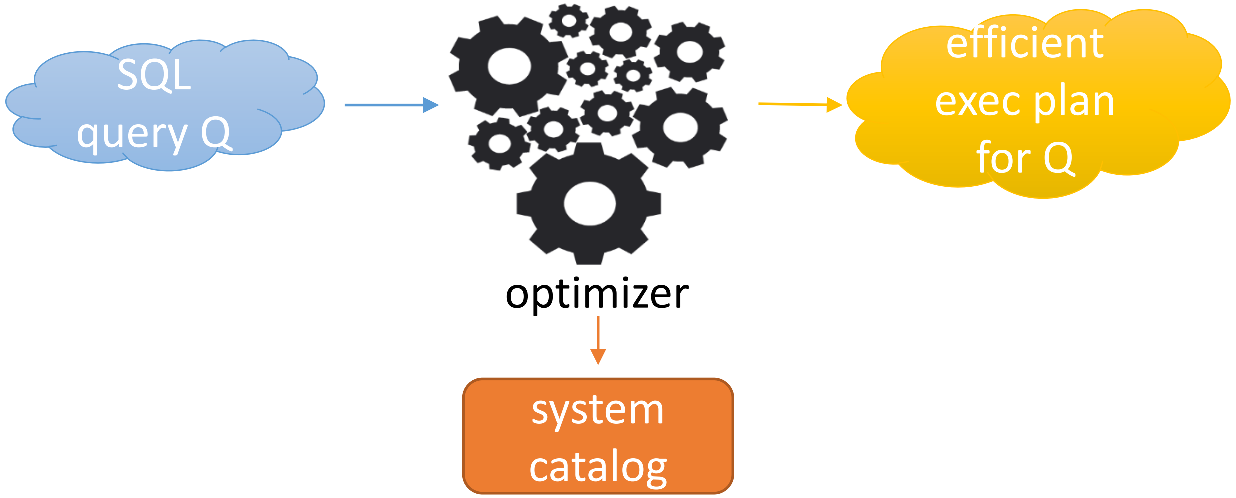
- intersection (R1 ∩ R2)
    - returns records that belong to both R1 and R2
- grouping and aggregate operators (algebra extensions)

- every operation returns a relation => operations can be composed

- an operator can have several implementation algorithms

* <u>optimizer</u>

- input: SQL query Q
- output: an efficient execution plan for evaluating Q



* algorithms for operators - based on 3 techniques:

- <u>iteration</u>:
    - examine iteratively:
        - all tuples in input relations

      or

        - data entries in indexes, provided they contain all the necessary fields (data entries are smaller than data records)
- <u>indexing</u>:
    - used when the query contains a selection condition or a join condition
    - examine only the tuples that meet the condition, using an index
- <u>partitioning</u>:
    - partition the tuples
    - decompose operation into collection of cheaper operations on partitions

* algorithms for operators - based on 3 techniques:

- <u>partitioning</u>:
    - partitioning techniques
        - sorting
        - hashing

* <u>access paths</u>

- *access path* = way of retrieving tuples from a relation
  - file scan

  or
  - an index *I* + a matching selection condition *C*
- condition *C* matches index *I* if *I* can be used to retrieve just the tuples satisfying *C*
- if relation *R* has an index *I* that matches selection condition *C*, then there are at least 2 access paths for *R* (file scan; index)

*Review lecture notes on *Indexes* (*Databases* course)

* access paths - example:

- relation *Students[SID, Name, City]*
- *I* - tree index on *Students* with search key *<Name>*
- query Q:

  SELECT *

  FROM Students

  WHERE Name = 'Ionescu'
- condition *C*: *Name = 'Ionescu'*
- *C* matches *I*, i.e., index *I* can be used to retrieve only the *Students* tuples satisfying *C*
- the following condition also matches the index: *Name > 'Ionescu'*

* access paths - example:

- relation *Students[SID, Name, City]*
- *I* - hash index on *Students* with search key <*Name*>
- query Q:

  SELECT *

  FROM Students

  WHERE Name = 'Ionescu'
- condition *C*: *Name = 'Ionescu'*
- *C* matches *I*, i.e., index *I* can be used to retrieve only the *Students* tuples satisfying *C*
- condition *Name > 'Ionescu'* doesn’t match *I* (since *I* is a hash index; it cannot be used to retrieve just the tuples satisfying *Name > 'Ionescu'*)

* <u>access paths</u>

- to sum up:
  - condition *C: attr op value*, *op* $\in \{<, <=, =, <>, >=, >\}$
  - condition *C* matches index *I* if:
  - the search key of *I* is *attr* and:
    - *I* is a tree index or
    - *I* is a hash index and *op* is *=*

* <u>access paths</u>

- index *I*, selection condition *C*

- *I* - hash index
- condition *C* of the form:
  - $\bigwedge_{i=1}^{n} T_i$
  - term T_i: $attr = value$
- *I* matches *C* if *C* has one term for each attribute in the search key of *I*

Condition	Hash index with search key <a, b, c>
a = 10 AND b = 5 AND c = 2	Yes
a = 10 AND b = 5	No
b = 5	No
b = 5 AND c = 2	No

* <u>access paths</u>

- index *I*, selection condition *C*

- *I* - tree index
- condition *C* of the form:
  - $\bigwedge_{i=1}^{n} T_i$
  - term T_i: $attr\ op\ value$; *op* ϵ *{<, <=, =, <>, >=, >}*
- *I* matches *C* if *C* has one term for each attribute in a prefix of the search key of *I*
- examples of prefixes for search key <a, b, c, d>: <a>, <a, b, c>; <a, c> and <b, c>, on the other hand, are not prefixes for this search key

Condition	B+ tree index with search key <a, b, c>
a = 10 AND b = 5 AND c = 2	Yes
a = 10 AND b = 5	Yes
b = 5	No
b = 5 AND c = 2	No

* access paths

- selectivity of an access path
  - the number of retrieved pages when using the access path to obtain the desired tuples
  - both data pages and index pages are counted
- example:

SELECT *
FROM Students
WHERE Name = 'Ionescu'

  - access paths:
    - file scan – selectivity could be 1000
    - matching index I with search key <Name> – selectivity could be 3
- most selective access path
  - retrieves the fewest pages, i.e., data retrieval cost is minimized

* <u>general selection conditions</u>

- in general, a selection condition can contain one or several terms of the form:
    - *attr op constant*
    - *attr1 op attr2*,

  combined with $\wedge$ and $\vee$

```
SELECT *
FROM Exams
WHERE SID = 7 AND EDate = '04-01-2021'
```

$$\sigma_{SID=7 \wedge EDate='04-01-2021'}(Exams)$$

* <u>general selection conditions</u>

- process a selection operation with a general selection condition C -> express C in CNF (conjunctive normal form)
- condition in CNF:
  - collection of conjuncts connected with the ∧ operator
  - a *conjunct* has one or more terms connected with the ∨ operator
  - *term*:
    - *attr op constant*
    - *attr1 op attr2*

- example:

  condition $(EDate < \text{'4-1-2021'} \wedge Grade = 10) \vee CID = 5 \vee SID = 3$

  is rewritten in CNF:

  $(EDate < \text{'4-1-2021'} \vee CID = 5 \vee SID = 3) \wedge (Grade = 10 \vee CID = 5 \vee SID = 3)$

* <u>general selection conditions</u> matching an index

- relation R[a, b, c, d, e], index I with search key <a, b, c>

Condition	B+ tree index	Hash index
a = 10 AND b = 5 AND c = 2	Yes	Yes
a = 10 AND b = 5	Yes	No
b = 5	No	No
b = 5 AND c = 2	No	No
d = 2	No	No
a = 20 AND b = 10 AND c = 5 AND d = 11	Partly	Partly

Condition – CNF selection condition

B+ tree index / Hash index – B+ tree / hash index I matches (*Yes*) / doesn't match (*No*) / matches a part of (*Partly*) the selection condition

- for the condition in the last row (a = 20 AND b = 10 AND c = 5 AND d = 11):
  - use index I to retrieve tuples satisfying *a = 20 AND b = 10 AND c = 5*, then apply *d = 11* to each retrieved tuple

* <u>general selection conditions</u> matching an index

- relation R[a, b, c, d]
- index I1 with search key <a, b>
- B+ tree index I2 with search key <c>

Condition	Indexes
c < 100 AND a = 3 AND b = 5	- use I1 or I2 to retrieve tuples - then check terms in the selection condition that do not match the index for each retrieved tuple - e.g., use the B+ tree index to retrieve tuples where c < 100; then apply *a = 3 AND b = 5* to each retrieved tuple

* running example - schema

- Students (<u>SID: integer</u>, SName: string, Age: integer)
- Courses (<u>CID: integer</u>, CName: string, Description: string)
- Exams (<u>SID: integer, CID: integer</u>, EDate: date, Grade: integer, FacultyMember: string)

- Students
  - every record has 50 bytes
  - there are 80 records / page
  - 500 pages of Students tuples
- Courses
  - every record has 50 bytes
  - there are 80 records / page
  - 100 pages of Courses tuples

* running example - schema

- Students (SID: integer, SName: string, Age: integer)
- Courses (CID: integer, CName: string, Description: string)
- Exams (SID: integer, CID: integer, EDate: date, Grade: integer, FacultyMember: string)

- Exams
  - every record has 40 bytes
  - there are 100 records / page
  - 1000 pages of Exams tuples

* <u>joins</u>

```
SELECT *
FROM Exams E, Students S
WHERE E.SID = S.SID
```

- algebra: $E \otimes_{E.SID=S.SID} S$
  - to be carefully optimized
  - size of $E \times S$ is large, so computing $E \times S$ followed by selection is inefficient
- E
  - M pages
  - p_E records / page
- S
  - N pages
  - p_S records / page
- evaluation: number of I/O operations

* joins – implementation techniques
    - iteration
        - Simple/Page-Oriented Nested Loops Join
        - Block Nested Loops Join
    - indexing
        - Index Nested Loops Join
    - partitioning
        - Sort-Merge Join
        - Hash Join

- equality join, one join column
    - join condition: $E_i = S_j$

Simple Nested Loops Join

```
foreach tuple e ∈ E do
    foreach tuple s ∈ S do
        if e_i == s_j then add <e, s> to the result
```

- for each record in the outer relation E, scan the entire inner relation S
- cost
  - **$M + p_E * M * N$** = 1000 + 100*1000*500 I/Os = 1000 + ($5 * 10^7$) I/Os
    - M I/Os – cost of scanning E
    - N I/Os – cost of scanning S
    - S is scanned p_E* M times (there are p_E* M records in the outer relation E)

* E - M pages, p_E records / page *   * 1000 pages *   * 100 records / page*

* S - N pages, p_S records / page *   * 500 pages *   * 80 records / page *

Page-Oriented Nested Loops Join

```
foreach page pe ∈ E do
    foreach page ps ∈ S do
        if e_i == s_j then add <e, s> to the result
```

- for each page in E read each page in S
- pairs of records <e, s> that meet the join condition are added to the result (where record *e* is on page *pe*, and record *s* – on page *ps*)
- refinement of Simple Nested Loops Join

Page-Oriented Nested Loops Join

```
foreach page pe ∈ E do
    foreach page ps ∈ S do
        if e_i == s_j then add <e, s> to the result
```

- cost
  - **M + M*N** = 1000 + 1000*500 I/Os = 501.000 I/Os
    - M I/Os – cost of scanning E; N I/Os – cost of scanning S
    - S is scanned M times
    - significantly lower than the cost of Simple Nested Loops Join (improvement - factor of p_E)
  - if the smaller table (*S*) is chosen as outer table:

  => cost = 500 + 500 * 1000 I/Os = 500.500 I/Os

* E - M pages, p_E records / page * * 1000 pages * * 100 records / page*

* S - N pages, p_S records / page * * 500 pages * * 80 records / page *

Block Nested Loops Join

- previously presented join algorithms do not use buffer pages effectively
- join relations R1 and R2; R1 – the smaller relation
- assumption – the smaller relation fits in main memory
- improvement:
    - store smaller relation R1 in memory
    - keep at least 2 extra buffer pages B1 and B2
    - use B1 to read the larger relation R2 (one page at a time)
    - use B2 as the output buffer (i.e., for tuples in the result of the join)
    - for each tuple in R2, search R1 for matching tuples

  => optimal cost: *number of pages in R1 + number of pages in R2*, since R1 is scanned only once, R2 is also scanned only once

Block Nested Loops Join

- refinement
  - don’t store the smaller relation in main memory as is, build an in-memory hash table for it instead
  - the I/O cost remains unchanged, but the CPU cost is usually much lower (since for each tuple in the larger relation, the smaller relation is examined to find matching tuples)

References

- [Ra02] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (3rd Edition), McGraw-Hill, 2002
- [Da03] DATE, C.J., An Introduction to Database Systems (8th Edition), Addison-Wesley, 2003
- [Ga09] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book (2nd Edition), Pearson Education, 2009
- [Ra02S] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, Slides for the 3rd Edition, http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html
- [Si11] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts (6th Edition), McGraw-Hill, 2011
- [Si19S] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, Slides for the 7th Edition, http://codex.cs.yale.edu/avi/db-book/
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems, http://infolab.stanford.edu/~ullman/fcdb.html