Query Processing

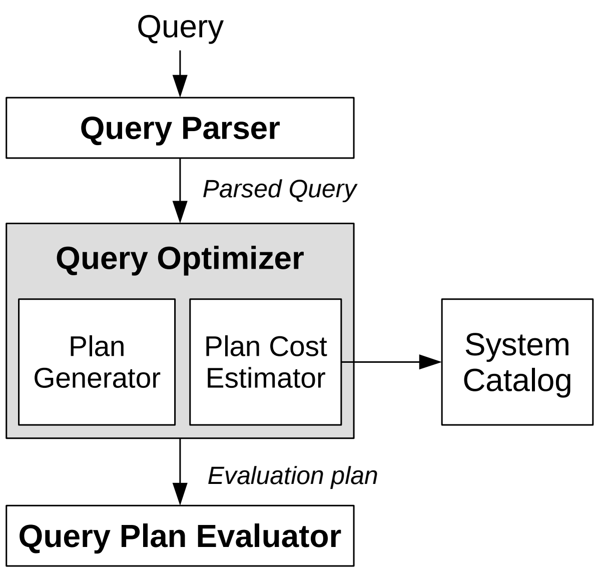
Dr Paolo Guagliardo

THE UNIVERSITY of EDINBURGH
informatics

Fall 2020 (v20.1.0)

Declarative queries (SQL, relational calculus) must be translated into a procedural language (relational algebra) to be executed

- Several ways (**evaluation plans**) to obtain the same answers
- Several algorithms available for each operator
- How do we find a **good** procedural query to execute?



The system catalog (1)

Contains metadata and statistics about the database
which are used to find the best way to evaluate a query

System-wide information, such as the page size

For each table

- table name, file name and file structure
- attribute names and types
- name of indexes on the table
- integrity constraints

For each index

- index name and structure (B-tree or hash)
- attributes of the search key

The system catalog (2)

Commonly stored statistics about tables and indexes

Cardinality	: number of tuples in each table
Size	: number of pages for each table
Index cardinality	: number of **distinct search key values** of each index
Index size	: number of pages for each index
Index height	: number of non-leaf levels of each tree index
Index range	: min & max values of search key in each index

Access paths

Access path: a way in which the rows of a table can be retrieved

- A file scan
- An index plus a **matching** selection condition

For a condition θ in CNF

- A hash index matches θ if there is a conjunct $A =$ *value* for each attribute A in the search key of the index
- A tree index matches θ if there is a conjunct A **op** *value* for each attribute A in a **prefix** of the search key of the index

where **op** $\in \{<, \leq, =, \neq, \geq, >\}$

Examples of access paths (1)

Suppose we have a relation R over attributes A, B, C, D
and the following selection conditions:

$$\begin{aligned}
\theta_1 &: A = 1 \wedge B = 2 \wedge C = 0 \\
\theta_2 &: A = 1 \wedge B < 2 \wedge C = 0 \\
\theta_3 &: A = 1 \wedge C = 0
\end{aligned}$$

A hash index for R on the search key (A, B, C)

- Matches θ_1, but does **not** match θ_2 and θ_3

A tree index for R on the search key (A, B, C)

- Matches θ_1 and θ_2, but not θ_3

Examples of access paths (2)

Suppose we have a relation R over attributes A, B, C, D

Consider the condition $A = 1 \wedge B = 2 \wedge C = 0$

An index (hash or tree) on the search key (B, C)

- can be used to retrieve tuples matching $B = 2 \wedge C = 0$
- retrieved tuples must be additionally filtered by $A = 1$

Consider the condition $B = 2 \wedge C = 0 \wedge D > 3$

If we have an index on (B, C) and a tree index on D

- both indexes match (different parts of) the condition
- we can choose one of the indexes to retrieve tuples
- the conjuncts that are not matched must be checked

Selectivity of access paths

Total number of pages retrieved when an access path is used

Most selective access path: retrieves the fewest pages

Selectivity depends on the conjuncts an index matches

- each conjunct acts as a filter on the table
- Reduction factor:
  the fraction of tuples satisfying a given conjunct
- can be estimated using information in the system catalog

Evaluation of selection

Given a selection $\sigma_\theta(R)$

- If no index on R matches θ we have to **scan** R
- If one or more indexes on R match θ
  1. use the **most selective** index to retrieve matching rows
  2. apply remaining conjuncts in θ to the retrieved rows

Evaluation of projection

Scan table or index (with an appropriate search key)
and output required subset of fields for each tuple

Duplicate elimination

Sort the table first, then do one pass to eliminate duplicates

Projection with duplicate elimination

1. Scan R and produce tuples with desired attributes
2. Sort the tuples using all attributes as sorting key
3. Scan the sorted result to discard duplicates

If R has M pages, this costs $O(M \log M)$ I/Os

Improvement:

- Scan in (1) can be combined with first pass of sorting
- Scan in (3) can be combined with last pass of sorting

Join processing

Join is the most common and **expensive** operation

Several available join algorithms

- Nested Loops Join
- Block Nested Loops Join
- Index Nested Loop Join
- Sort-Merge Join
- Hash Join

Nested Loops Join

Simplest algorithm to compute $R \bowtie_\theta S$

```
1. for each page $P_R$ of $R$ do
2.    for each page $P_S$ of S do
3.       for each tuple $r \in P_R$ do
4.          for each tuple $s \in P_S$ do
5.             if $rs$ satisfies $\theta$ then
6.                add $rs$ to result
```

R is the outer relation (scanned once)
S is the inner relation (scanned multiple times)

If R has M pages and S has N pages, the cost is $M + M \cdot N$ I/Os
If R has m tuples and S has n tuples, the CPU cost is $O(m \cdot n)$

Block Nested Loops Join

If we have B buffer pages available we can:

- read R in blocks of $B-2$ pages
- use one buffer page for reading the pages of S
- use one buffer page for output

```
1. for each block $B_R$ of $B-2$ pages of $R$ do
2.    for each page $P_S$ of S do
3.       for each tuple $r \in B_R$ do
4.          for each tuple $s \in P_S$ do
5.             if $rs$ satisfies $\theta$ then
6.                add $rs$ to result
```

If R has M pages and S has N pages, cost is $M + \left\lceil \frac{M}{B-2} \right\rceil \cdot N$ I/Os

Index Nested Loops Join

If there is an index matching the join condition,
make the indexed relation be the inner one

```
1. for each $P_R$ of $R$ do
2.    for each matching page $P_S$ of S do
3.       for each tuple $r \in P_R$ do
4.          for each tuple $s \in P_S$ do
5.             if $rs$ satisfies condition then
6.                add $rs$ to result
```

Cost depends on the index and the number of matching tuples
Better than simple nested loops: it does not enumerate $R \times S$

Sort-merge join (1)

Consider $R \bowtie_\theta S$ where θ is $R.A_1 = S.B_1 \wedge \cdots \wedge R.A_n = S.B_n$

```
1.  Sort $R$ on $X = A_1, \ldots, A_n$ and $S$ on $Y = B_1, \ldots, B_n$
2.  Set $r :=$ first tuple of $R$ and $s :=$ first tuple of $S$
3.  while $r \neq$ EOF and $s \neq$ EOF do
4.      while $r[X] < s[Y]$ do { $r :=$ next($R$) }
5.      while $r[X] > s[Y]$ do { $s :=$ next($S$) }
6.      Set $p := s$
7.      while $r[X] = s[Y]$ do
8.          $p := s$
9.          while $r[X] = p[Y]$ do
10.             Add $rp$ to result
11.             $p :=$ next($S$)
12.         $r :=$ next($R$)
13.     Set $s := p$
```

Sort-merge join (2)

Works only for equijoins (the condition is a conjunction of equalities)

Cost

- Sorting R costs $O(M \log M)$ if R has M pages
- Sorting S costs $O(N \log N)$ if S has N pages
- **Merging phase** costs $M + N$ I/Os
  if no partition of S is scanned multiple times
  otherwise $O(M \cdot N)$ in the worst case

Typically the merging phase is just a single scan of each relation

- if at least one relation has **no duplicates** in the join attributes
- this is the case for key-foreign key joins (very common)

Hash join (1)

Consider $R \bowtie_\theta S$ where θ is $R.A_1 = S.B_1 \wedge \cdots \wedge R.A_n = S.B_n$

Partitioning phase: split R and S into partitions
using a hash function on the values of $\underbrace{A_1, \ldots, A_n}_{X}$ and $\underbrace{B_1, \ldots, B_n}_{Y}$

1. Choose number of buckets k and appropriate hash function h
2.

```
for each $r \in R$ do
    $i := h(r[X])$
    $H_i^R := H_i^R \cup \{r\}$
```

```
for each $s \in S$ do
    $i := h(s[Y])$
    $H_i^S := H_i^S \cup \{s\}$
```

Probing phase: compare tuples in each partition of R
only with tuples in the **corresponding** partition of S

```
for $i = 1, \ldots, k$ do
    read $H_i^R$ ; read $H_i^S$
    add $H_i^R \bowtie_\theta H_i^S$ to result
```

Hash join (2)

Works only for equijoins (the condition is a conjunction of equalities)

Cost

- Partioning phase: scan R and S once and write them out once
  Cost is $2(M + N)$ I/Os if R has M pages and S has N pages
- Probing phase: scan each partition once ($M + N$ I/Os)
  if there are no **overflows**

  Total cost is $3(M + N)$

If there are overflows, recursive partitioning is used
The cost becomes $O\big((M + N)\log(M + N)\big)$

Other operations

Set operations

- Expensive aspect is given by duplicate elimination
- Same technique as for projection (using sorting)

Group by

- Typically implemented through sorting
- If there is a **tree index** matching the grouping attributes tuples can be retrieved in appropriate order without sorting

Aggregation

- Carried out using temporary counters in main memory

Query optimization

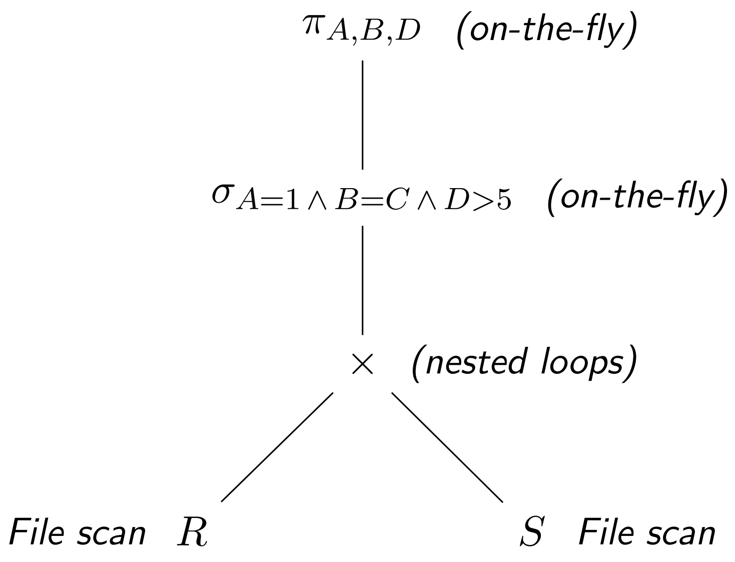
Query plan: relational algebra tree extended with annotations

- which **access path** to use for each table
- which **implementation method** to use for each operator

Optimization involves the following steps:

1. Enumerating alternative plans to evaluate the query
2. Estimating the cost of each enumerated plan
3. Choosing the plan with the lowest estimated cost

A possible query plan for
`SELECT A,B,D FROM R,S WHERE A=1 AND B=C AND D>5`


$\pi_{A,B,D}$ (on-the-fly)
$\sigma_{A=1 \wedge B=C \wedge D>5}$ (on-the-fly)
$\times$ (nested loops)
File scan R
S File scan

Pipelined evaluation

Pipelining: the result of an operator is passed directly to the next

Materialization: intermediate result is written to a temporary table

A unary operator is applied **on-the-fly** if its input is pipelined

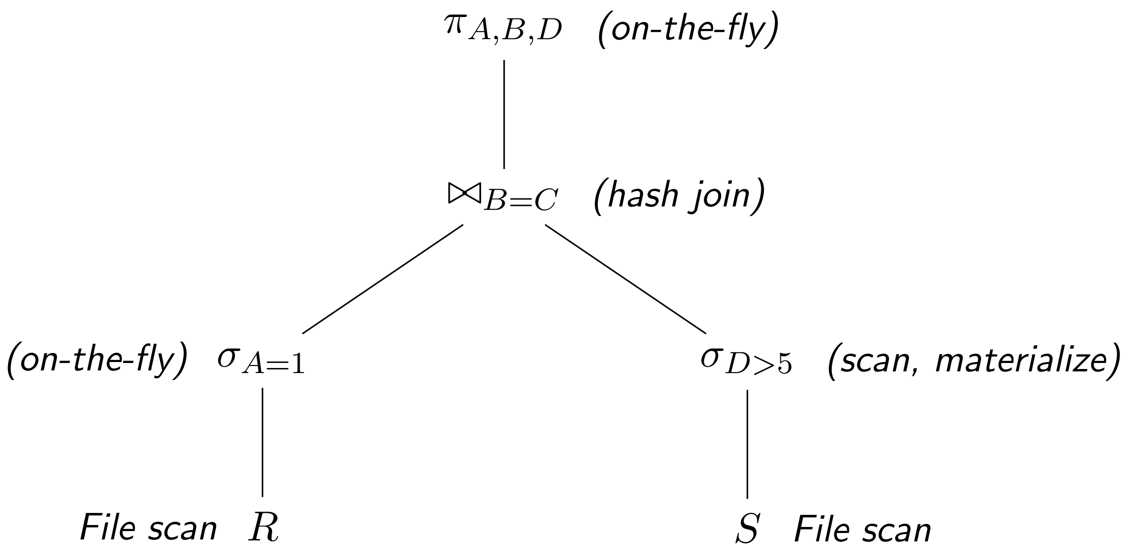
Iterator interface

- Hides the internal implementation details of each operator
- Supports functions:

open	initialize, allocate buffers, pass arguments in
get_next	retrieve and process tuples from input nodes
close	deallocate buffers and state information

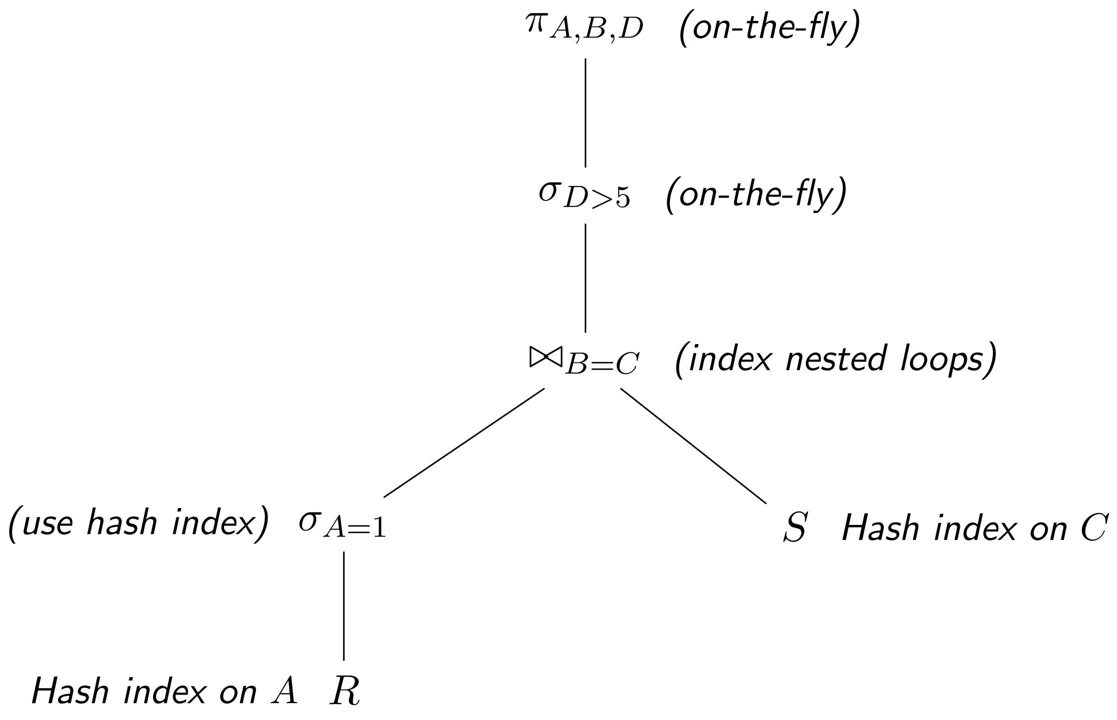
Alternative plans

- Selections and cross-products can be combined into joins
- Joins can be extensively reordered
- Selections and projections can be pushed ahead of joins


$\pi_{A,B,D}$ (on-the-fly)
$\bowtie_{B=C}$ (hash join)
(on-the-fly) $\sigma_{A=1}$
$\sigma_{D>5}$ (scan, materialize)
File scan R
S File scan

Using indexes

If there are indexes, other plans may be available


$\pi_{A,B,D}$ (on-the-fly)
$\sigma_{D>5}$ (on-the-fly)
$\bowtie_{B=C}$ (index nested loops)
(use hash index) $\sigma_{A=1}$
S Hash index on C
Hash index on A R

Join order

Join is associate and commutative
$\Longrightarrow$ many combinations of binary joins to get same result

Linear trees: at least one child of each join node is a base table

Left-deep trees: the **right** child of each join node is a base table

Bushy trees: non-linear trees

Advantages of left-deep trees:

- if too many alternatives we need to **prune** the search space
- allow us to generate **fully pipelined** plans

Estimating plan cost

I/O cost given by:

1. Reading the input tables (possibly more than once)
2. Materializing intermediate results (if needed)
3. Sorting final result (for duplicate elimination and ordering)

Estimating result size

Selection: input size multiplied by reduction factor of condition

Join: max result size (= product of input tables sizes)
multiplied by reduction factor of the join condition

Reduction factors are estimated using statistics
periodically collected about (a **sample** of) the data

Reduction factor

$RF(A = c) \simeq \frac{1}{m}$
where m is the number of distinct values in A

$RF(A = B) \simeq \frac{1}{\max(m,n)}$
where m and n are the number of distinct values in A and B

$RF(A > c) \simeq \max\bigl(0, \frac{H-c}{H-L}\bigr)$
where H and L are the highest and lowest values in A

$RF(\theta_1 \wedge \theta_2) \simeq RF(\theta_1) \cdot RF(\theta_2)$

$RF(\theta_1 \vee \theta_2) \simeq \min\bigl(1, RF(\theta_1) + RF(\theta_2) - RF(\theta_1) \cdot RF(\theta_2)\bigr)$

$RF(\neg\theta) \simeq 1 - RF(\theta)$