Query processing

Lecture 5
2ID35, Spring 2015

George Fletcher

Faculteit Wiskunde & Informatica
Technische Universiteit Eindhoven

6 May 2015

Where we've been

Last time

- hash-based indexing
- join indexes
- models of indexing

Where we're headed

Today's agenda

- Evaluation of relational operators

The life of a query

- Employee(EID, EName, ECity)
- Company(CID, CName, CCity)
- WorksFor(EID, CID, Salary)

The life of a query

- Employee(EID, EName, ECity)
- Company(CID, CName, CCity)
- WorksFor(EID, CID, Salary)

```
SELECT E.EName
FROM Employee E, WorksFor W
WHERE E.EID = W.EID AND W.Salary > 5000
```

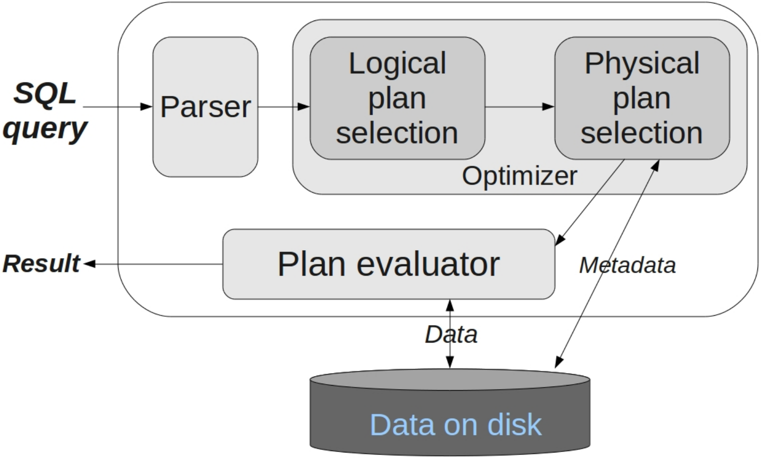
The life of a query

- Employee(EID, EName, ECity)
- Company(CID, CName, CCity)
- WorksFor(EID, CID, Salary)

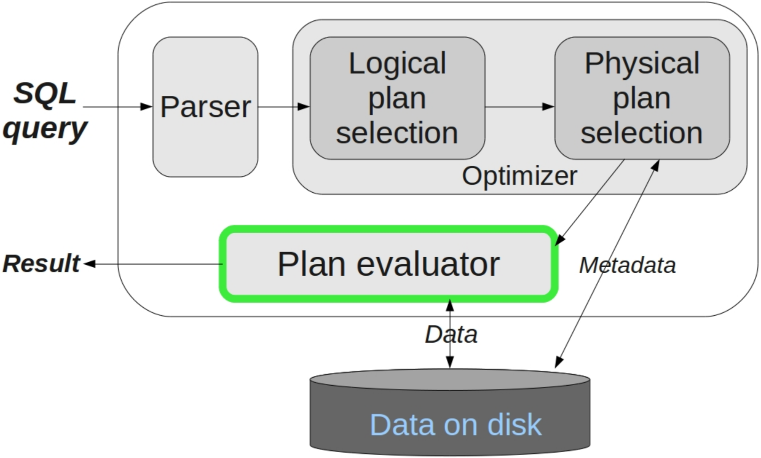
```
SELECT E.EName
FROM Employee E, WorksFor W
WHERE E.EID = W.EID AND W.Salary > 5000
```

$$\pi_{E.ename}(\sigma_{W.salary>5000}(E \bowtie_{E.eid=W.eid} W))$$

The life of a query



The life of a query: evaluation



The life of a query: plans & physical operators

- A physical query plan is what the evaluation engine executes (i.e., interprets)

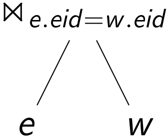
The life of a query: plans & physical operators

- A physical query plan is what the evaluation engine executes (i.e., interprets)
- A plan is a tree of physical operators
  - i.e., operators which access and manipulate physical data
- Each physical operator consumes a relation and outputs a relation

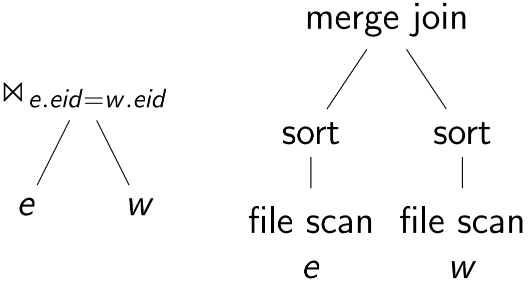
The life of a query: plans & physical operators

- A physical query plan is what the evaluation engine executes (i.e., interprets)
- A plan is a tree of physical operators
  - i.e., operators which access and manipulate physical data
- Each physical operator consumes a relation and outputs a relation
- (logical) RA operations may be mapped to multiple (physical) operators
  - and, there are often multiple mappings to choose from

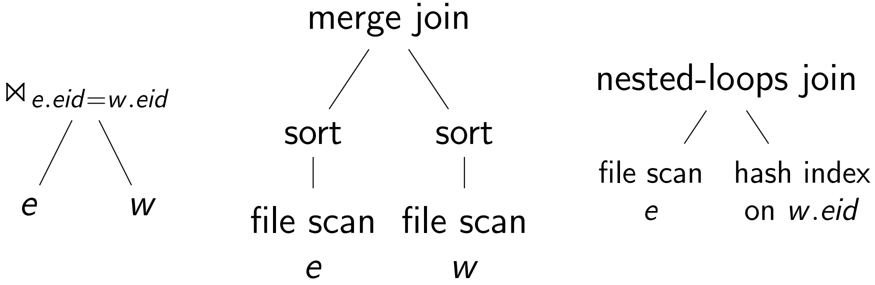
The life of a query: plans & physical operators



The life of a query: plans & physical operators



The life of a query: plans & physical operators



The life of a query: pipelining

- pipelining: read, process, propagate
  - the opposite is to materialize intermediate results
  - in practice, materialization becomes necessary (due to “blocking” operators, such as sorting)

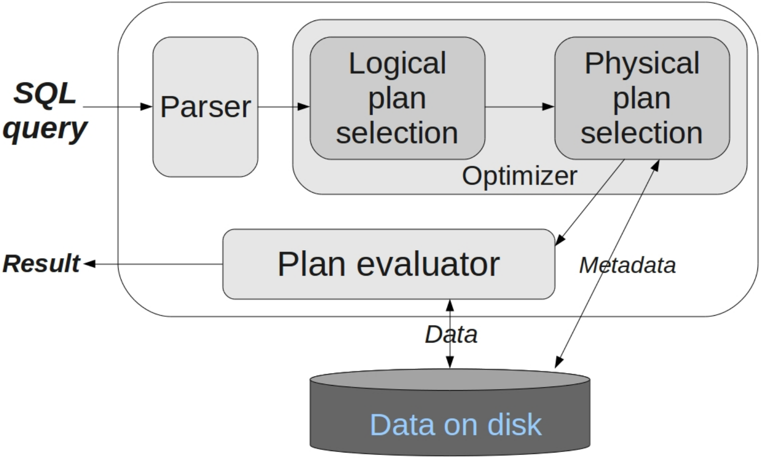
The life of a query: pipelining

- pipelining: read, process, propagate
  - the opposite is to materialize intermediate results
  - in practice, materialization becomes necessary (due to “blocking” operators, such as sorting)
- benefits of pipelining
  - no buffering
  - faster evaluation (since no I/O)
  - better resource utilization (more in-memory ops)

The life of a query: pipelining

- pipelining: read, process, propagate
  - the opposite is to materialize intermediate results
  - in practice, materialization becomes necessary (due to “blocking” operators, such as sorting)
- benefits of pipelining
  - no buffering
  - faster evaluation (since no I/O)
  - better resource utilization (more in-memory ops)
- hence, pipeline is simulated through the operator interface: open(), getNext(), close()
  - push model (buffering in calling operator)
  - pull model (buffering in called operator)
  - streams model (buffering in the connections)
- pull (demand-driven) model is common

The life of a query



- Optimizer is responsible for coming up with good physical plan

The life of a query: evaluation

Today: For each relational operation, how can we efficiently implement it?

- i.e., for each RA operation, what are some possible physical operators?

$$
\begin{array}{ll}
\sigma_\theta(R) & \pi_{A_1,\ldots,A_n}(R) \\
R \cup S & R - S \\
R \cap S & R \times S \\
R \bowtie S &
\end{array}
$$

Query evaluation

Three common techniques

- iteration (scan): examine all tuples in a table or index

Query evaluation

Three common techniques

- iteration (scan): examine all tuples in a table or index
- partitioning: of tuples on a sort key, and applying operations on buckets.
  - Sorting and hashing are commonly used partitioning techniques

Query evaluation

Three common techniques

- iteration (scan): examine all tuples in a table or index
- partitioning: of tuples on a sort key, and applying operations on buckets.
  - Sorting and hashing are commonly used partitioning techniques
- indexing: if a selection or join condition is specified, use available index to inspect just those tuples satisfying the condition

Query evaluation: access paths

A method of retrieving tuples.

Query evaluation: access paths

A method of retrieving tuples. Either:

- a file scan, or

Query evaluation: access paths

A method of retrieving tuples. Either:

- a file scan, or
- an index plus a matching selection condition C
  - a hash index matches C if there is a term "att = val" in C for each attribute in the index's search key
    - e.g., hash index on (e.ecity, e.eid) can be used for $\sigma_{E.ecity=Delft \wedge E.eid=1234}(E)$ but not for $\sigma_{E.ecity=Delft}(E)$

Query evaluation: access paths

A method of retrieving tuples. Either:

- a file scan, or
- an index plus a matching selection condition C
  - a hash index matches C if there is a term "att $=$ val" in C for each attribute in the index's search key
    - e.g., hash index on (e.ecity, e.eid) can be used for $\sigma_{E.ecity=Delft \wedge E.eid=1234}(E)$ but not for $\sigma_{E.ecity=Delft}(E)$
  - a tree index matches C if there is a term "att θ val" for each attribute in a prefix of the index's search key ($\theta \in \{<, \leq, =, \geq, >, \neq\}$)
    - e.g., B+tree on (e.ecity, e.eid) can be used for $\sigma_{E.ecity=Delft}(E)$ and $\sigma_{E.ecity=Delft \wedge E.eid=1234}(E)$, but not for $\sigma_{E.eid=1234}(E)$

Query evaluation: access paths

Selectivity of an access path

- number of (index and data) pages retrieved
- "most selective" means retrieves fewest pages
  - cf. the notion of *access overhead* from last lecture

The selection operation σ

Suppose the employee relation E occupies $B_E = 1000$ pages and has $T_E = 10,000$ tuples.

The selection operation σ

Suppose the employee relation E occupies $B_E = 1000$ pages and has $T_E = 10,000$ tuples.

To evaluate $\sigma_{ecity=Eindhoven}(E)$, we can

- scan E (i.e., 1000 I/Os), or

The selection operation σ

Suppose the employee relation E occupies $B_E = 1000$ pages and has $T_E = 10,000$ tuples.

To evaluate $\sigma_{ecity=Eindhoven}(E)$, we can

- scan E (i.e., 1000 I/Os), or
- use index if available
  - if clustering, then great

The selection operation σ

Suppose the employee relation E occupies $B_E = 1000$ pages and has $T_E = 10,000$ tuples.

To evaluate $\sigma_{ecity=Eindhoven}(E)$, we can

- scan E (i.e., 1000 I/Os), or
- use index if available
  - if clustering, then great
  - if non-clustering, then potentially worse than scan (i.e., worst case is 10,000 I/Os)

The selection operation σ

$\sigma_{att\ \theta\ val}(R)$

The selection operation σ

$\sigma_{att\ \theta\ val}(R)$

- no index, unsorted data
  - iteration (i.e., full file scan)
  - $\mathcal{O}(B_R)$

The selection operation σ

$\sigma_{att\ \theta\ val}(R)$

- no index, unsorted data
  - iteration (i.e., full file scan)
  - $\mathcal{O}(B_R)$
- no index, sorted data
  - binary search
  - $\mathcal{O}(\log B_R)$

The selection operation σ

$\sigma_{att\ \theta\ val}(R)$

- no index, unsorted data
  - iteration (i.e., full file scan)
  - $\mathcal{O}(B_R)$
- no index, sorted data
  - binary search
  - $\mathcal{O}(\log B_R)$
- B+tree
  - clustered, and θ is not $=$, then best choice
  - unclustered, then need an estimate of selectivity of *att* θ *val*

The selection operation σ

$\sigma_{att\ \theta\ val}(R)$

- no index, unsorted data
  - iteration (i.e., full file scan)
  - $\mathcal{O}(B_R)$
- no index, sorted data
  - binary search
  - $\mathcal{O}(\log B_R)$
- B+tree
  - clustered, and θ is not $=$, then best choice
  - unclustered, then need an estimate of selectivity of *att* θ *val*
- Hash index, with equality selection (θ is $=$)
  - best choice

The projection operation π

```
SELECT DISTINCT Salary
FROM WorksFor
WHERE CID = 1234
```

$$\pi_{salary}(\sigma_{cid=1234}(W))$$

The projection operation π

$\pi_{A_1,\ldots,A_n}(R)$

- without duplicate elimination, scan or index (clustered/unclustered doesn't matter)

The projection operation π

$\pi_{A_1,\ldots,A_n}(R)$

- without duplicate elimination, scan or index (clustered/unclustered doesn't matter)
- with duplicate elimination
  - remove unwanted attributes
  - eliminate any duplicates produced

The projection operation π

Two basic approaches to duplicate elimination

- sort-based: scan R producing projected tuples, sort result, and scan and eliminate adjacent duplicates
  - $\mathcal{O}(B_R \log B_R)$

The projection operation π

Two basic approaches to duplicate elimination

- sort-based: scan R producing projected tuples, sort result, and scan and eliminate adjacent duplicates
  - $\mathcal{O}(B_R \log B_R)$
- hash-based: take N buffer pages
  - partition each page of R at a time into $N-1$ buckets, writing them out as they fill up. (diff buckets implies not duplicates)

The projection operation π

Two basic approaches to duplicate elimination

- sort-based: scan R producing projected tuples, sort result, and scan and eliminate adjacent duplicates
  - $\mathcal{O}(B_R \log B_R)$
- hash-based: take N buffer pages
  - partition each page of R at a time into $N-1$ buckets, writing them out as they fill up. (diff buckets implies not duplicates)
  - read in each bucket and rehash in-memory with new hash function (collision implies duplicate)
  - write out resulting hash table after reading whole bucket

requires $N > B_R/N$

The union and difference operations $\cup, -$

```
SELECT CID
FROM Company
WHERE CCity = Eindhoven
UNION
SELECT CID
FROM WorksFor
WHERE Salary > 5000
```

The union and difference operations $\cup$, $-$

$R \cup S$

The union and difference operations $\cup, -$

$R \cup S$

- sort based
  - Sort R and S on all fields
  - scan results in parallel and merge, eliminating duplicates

The union and difference operations $\cup, -$

$R \cup S$

- sort based
  - Sort R and S on all fields
  - scan results in parallel and merge, eliminating duplicates
- hash based
  - Partition both R and S, using h_1
  - for each partition block P
    - build in-memory hash table for S_P using new h_2
    - scan R_P. For each tuple, probe hash table for S_P. If tuple is already in table, discard, otherwise insert.
    - write out table and clear for next partition

The union and difference operations $\cup$, $-$

```
SELECT CID
FROM Company
WHERE CCity = Eindhoven
EXCEPT
SELECT CID
FROM WorksFor
WHERE Salary > 5000
```

The union and difference operations $\cup$, $-$

$R - S$

The union and difference operations $\cup, -$

$R - S$

- sort based
  - Sort R and S on all fields
  - scan results in parallel, eliminating duplicates and tuples appearing in S

The union and difference operations $\cup, -$

$R - S$

- sort based
  - Sort R and S on all fields
  - scan results in parallel, eliminating duplicates and tuples appearing in S
- hash based
  - Partition both R and S, using h_1
  - for each partition block P
    - build in-memory hash table for S_P using new h_2
    - scan R_P. For each tuple, probe hash table for S_P. If tuple is not in table, write it out.
    - clear for next partition

The join operation $\bowtie$

```
SELECT E.EName
FROM Employee E, WorksFor W
WHERE E.EID = W.EID AND W.Salary > 5000
```

$$\pi_{E.ename}(\sigma_{W.salary>5000}(E \bowtie_{E.eid=W.eid} W))$$

The join operation $\bowtie$

```
SELECT E.EName
FROM Employee E, WorksFor W
WHERE E.EID = W.EID AND W.Salary > 5000
```

$$\pi_{E.ename}(\sigma_{W.salary>5000}(E \bowtie_{E.eid=W.eid} W))$$

The join operation $\bowtie$

- quite expensive, yet very common
  - e.g., due to table normalization
- intensively studied

The join operation $\bowtie$

- quite expensive, yet very common
  - e.g., due to table normalization
- intensively studied
- also, note that $\cap$ (i.e., the `INTERSECT` operation in SQL) and $\times$ are special cases of $\bowtie$

The join operation $\bowtie$

- *n.b.* choosing physical plan for a single join is different from choosing the order in which joins should be evaluated in the overall plan
  - in fact, the order in which joins are evaluated affects the choice of join algorithm
  - these two issues are very interrelated
- semantically, $R \bowtie S = S \bowtie R$
  - however, for physical join $cost(R \bowtie S) \neq cost(S \bowtie R)$

The join operation $\bowtie$

- *n.b.* choosing physical plan for a single join is different from choosing the order in which joins should be evaluated in the overall plan
  - in fact, the order in which joins are evaluated affects the choice of join algorithm
  - these two issues are very interrelated
- semantically, $R \bowtie S = S \bowtie R$
  - however, for physical join $cost(R \bowtie S) \neq cost(S \bowtie R)$
- three main factors in determining cost:
  - input cardinalities T_R, T_S and number of pages B_R, B_S
  - selectivity factor of the join predicate
    - i.e., the ratio $\frac{|R \bowtie S|}{|R \times S|}$
  - available memory in buffer

The join operation $\bowtie$

four classes of join algorithms:

- iteration-based
- order-based
- partition-based
- special index-based

Iteration-based $\bowtie$

Nested-loops join

- simple, matching the semantics of $\bowtie$
- most flexible, for non-equi joins

Iteration-based $\bowtie$

Simple nested-loops join, $R \bowtie S$

- for each tuple $r \in R$
  - for each tuple $s \in S$
    - if $r \bowtie s$, then add (r, s) to result

Iteration-based $\bowtie$

Simple nested-loops join, $R \bowtie S$

- Call R the outer relation and S the inner relation
- One scan over the outer relation
- For each tuple in the outer relation, one scan over the inner relation

Iteration-based $\bowtie$

Simple nested-loops join, $R \bowtie S$

- Call R the outer relation and S the inner relation
- One scan over the outer relation
- For each tuple in the outer relation, one scan over the inner relation
- if relations are not clustered, then $cost(R \bowtie S) = \mathcal{O}(T_R + T_R \cdot T_S)$

Iteration-based $\bowtie$

Simple nested-loops join, $R \bowtie S$

- Call R the outer relation and S the inner relation
- One scan over the outer relation
- For each tuple in the outer relation, one scan over the inner relation
- if relations are not clustered, then $cost(R \bowtie S) = \mathcal{O}(T_R + T_R \cdot T_S)$
  - Suppose $T_R = T_S = 10,000$. Then $cost(R \bowtie S) = 10000 + 100,000,000 = 100,010,000$ I/Os!
  - if 15ms per I/O, then this is 417 hours (i.e., over 17 days)!

Iteration-based $\bowtie$

Simple nested-loops join, $R \bowtie S$

- if relations are clustered, then $cost(R \bowtie S) = \mathcal{O}(B_R + B_R \cdot B_S)$

Iteration-based $\bowtie$

Simple nested-loops join, $R \bowtie S$

- if relations are clustered, then $cost(R \bowtie S) = \mathcal{O}(B_R + B_R \cdot B_S)$
  - Suppose $B_R = B_S = 1000$. Then $cost(R \bowtie S) = 1000 + 1,000,000 = 1,001,000$ I/Os!
  - this is still 4.17 hours!

Iteration-based $\bowtie$

Block nested-loops join, $R \bowtie S$

- We didn't make use of our buffer pool in simple nested-loop join
- as we saw with sorting, extra working space can have a non-trivial impact on cost

Iteration-based $\bowtie$

Block nested-loops join, $R \bowtie S$

- We didn't make use of our buffer pool in simple nested-loop join
- as we saw with sorting, extra working space can have a non-trivial impact on cost
- suppose the outer relation fits in the buffer, with at least two free pages left
  - then, we can put the outer relation in the buffer, and read the inner relation one page at a time

Iteration-based $\bowtie$

Block nested-loops join, $R \bowtie S$

- We didn't make use of our buffer pool in simple nested-loop join
- as we saw with sorting, extra working space can have a non-trivial impact on cost
- suppose the outer relation fits in the buffer, with at least two free pages left
  - then, we can put the outer relation in the buffer, and read the inner relation one page at a time
  - this costs us $\mathcal{O}(B_R + B_S)$, very nice!

Iteration-based $\bowtie$

Block nested-loops join, $R \bowtie S$

- We didn't make use of our buffer pool in simple nested-loop join
- as we saw with sorting, extra working space can have a non-trivial impact on cost
- suppose the outer relation fits in the buffer, with at least two free pages left
  - then, we can put the outer relation in the buffer, and read the inner relation one page at a time
  - this costs us $\mathcal{O}(B_R + B_S)$, very nice!
    - for our running example, $cost(R \bowtie S) = 1000 + 1000 = 2000$ I/Os

Iteration-based $\bowtie$

Block nested-loops join, $R \bowtie S$

- We didn't make use of our buffer pool in simple nested-loop join
- as we saw with sorting, extra working space can have a non-trivial impact on cost
- suppose the outer relation fits in the buffer, with at least two free pages left
  - then, we can put the outer relation in the buffer, and read the inner relation one page at a time
  - this costs us $\mathcal{O}(B_R + B_S)$, very nice!
    - for our running example, $cost(R \bowtie S) = 1000 + 1000 = 2000$ I/Os
    - and at 15ms per I/O, this takes 30 seconds!

Iteration-based $\bowtie$

Block nested-loops join, $R \bowtie S$

- now, what if we have N free buffer pages, and $B_R > N$?

Iteration-based $\bowtie$

Block nested-loops join, $R \bowtie S$

- now, what if we have N free buffer pages, and $B_R > N$?
- we can scan in $\lceil \frac{B_R}{N-2} \rceil$ blocks of size $N-2$ of R, and compare S against each block

Iteration-based $\bowtie$

Block nested-loops join, $R \bowtie S$

- for each block of $N - 2$ pages of R
  - for each page of S
    - for all r in the current R-block and s in the current S-page such that $r \bowtie s$, add (r, s) to result

Iteration-based $\bowtie$

Block nested-loops join, $R \bowtie S$

- for each block of $N - 2$ pages of R
  - for each page of S
    - for all r in the current R-block and s in the current S-page such that $r \bowtie s$, add (r, s) to result

$$cost(R \bowtie S) = \mathcal{O}(B_R + \lceil \frac{B_R}{N-2} \rceil \cdot B_S) \text{ I/Os}$$

Iteration-based $\bowtie$

Block nested-loops join, $R \bowtie S$

- for each block of $N-2$ pages of R
  - for each page of S
    - for all r in the current R-block and s in the current S-page such that $r \bowtie s$, add (r, s) to result

$$cost(R \bowtie S) = \mathcal{O}(B_R + \lceil \frac{B_R}{N-2} \rceil \cdot B_S) \text{ I/Os}$$

- so, for our running example, if $N = 102$, then $cost(R \bowtie S) = 1000 + 1000 \cdot \lceil \frac{1000}{100} \rceil = 11000$ I/Os

Iteration-based $\bowtie$

Block nested-loops join, $R \bowtie S$

- for each block of $N - 2$ pages of R
  - for each page of S
    - for all r in the current R-block and s in the current S-page such that $r \bowtie s$, add (r, s) to result

$$cost(R \bowtie S) = \mathcal{O}(B_R + \lceil \frac{B_R}{N-2} \rceil \cdot B_S) \text{ I/Os}$$

- so, for our running example, if $N = 102$, then $cost(R \bowtie S) = 1000 + 1000 \cdot \lceil \frac{1000}{100} \rceil = 11000$ I/Os
- and at 15ms per I/O, this takes 2.75 minutes!

Iteration-based $\bowtie$

Block nested-loops join, $R \bowtie S$

- the inner relation is scanned a number of times which is dependent on the size of the outer relation
- so, the outer should be chosen to be the smaller of the two!

Iteration-based $\bowtie$

Index nested-loops join, $R \bowtie S$

- what if we have an index available on the inner relation, on the join attribute?
- then, we can proceed just as simple nested-loops, except we use the index in the inner loop to perform predicate eval

Iteration-based $\bowtie$

Index nested-loops join, $R \bowtie S$

- what if we have an index available on the inner relation, on the join attribute?
- then, we can proceed just as simple nested-loops, except we use the index in the inner loop to perform predicate eval
- in the worst case, this costs $B_R + T_R \cdot I_S$, for average index access cost I_S on S
  - i.e., 3 or 4 I/Os for a B+tree, and 2 or 3 I/Os for a hash index
- if the outer relation is small, then this can lead to significant I/O savings

Order-based $\bowtie$

Sort-merge join $R \bowtie S$

- clean, simple idea:
  - sort R and S
  - scan together, and merge results
- key idea: there are groups in the sorted relations with the same value for the join attribute

Order-based ⋈

pr

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

r

ps

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

s

Merge-join

Order-based $\bowtie$

Sort-merge join $R \bowtie S$

- Cost?
  - sort R costs $2B_R \log B_R$
  - sort S costs $2B_S \log B_S$
  - merge is linear scan: $B_R + B_S$

Order-based $\bowtie$

Sort-merge join $R \bowtie S$

- Cost?
  - sort R costs $2B_R \log B_R$
  - sort S costs $2B_S \log B_S$
  - merge is linear scan: $B_R + B_S$
- so $cost(R \bowtie S)$ is the sum of these costs

Order-based $\bowtie$

Sort-merge join $R \bowtie S$

- Cost?
  - sort R costs $2B_R \log B_R$
  - sort S costs $2B_S \log B_S$
  - merge is linear scan: $B_R + B_S$
- so $cost(R \bowtie S)$ is the sum of these costs
- in our running example, we have 10,000 I/Os, which takes 2.5 minutes (about the same as block nested loops)

Partition-based $\bowtie$

Hash join $R \bowtie S$

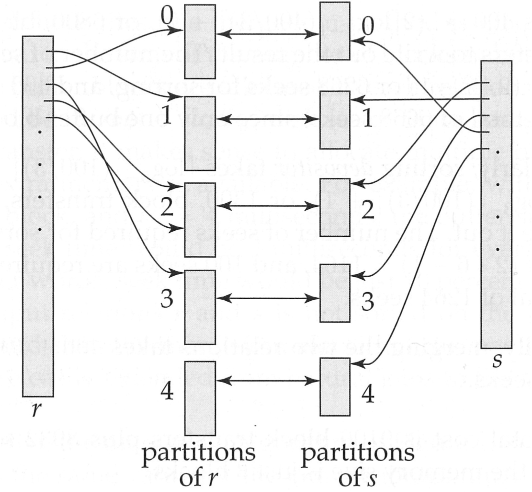
- partition-based
- key idea: using the same hash function,
  - partition R and S into m blocks $\{R_1, \ldots, R_m\}$ $\{S_1, \ldots, S_m\}$ such that each partition block fits in memory
    - tuples in R_i will only join with tuples in S_i

Partition-based $\bowtie$

Hash join $R \bowtie S$

- partition-based
- key idea: using the same hash function,
  - partition R and S into m blocks $\{R_1, \ldots, R_m\}$ $\{S_1, \ldots, S_m\}$ such that each partition block fits in memory
    - tuples in R_i will only join with tuples in S_i
  - then, for every R_i, load it in memory, scan S_i, and produce join results (just like block nested loops)

Partition-based $\bowtie$



Hash partitioning of R and S

Partition-based $\bowtie$

Hash join $R \bowtie S$: using the same hash function,

- partition R and S into m blocks $\{R_1, \ldots, R_m\}$ $\{S_1, \ldots, S_m\}$ such that each partition block fits in memory
  - tuples in R_i will only join with tuples in S_i
- then, for every R_i, load it in memory, scan S_i, and produce join results (just like block nested loops)

Exercise. Suppose
$R = \{(14, x), (135, y), (40, x), (10, z)\}$,
$S = \{(3, p), (10, q), (14, r), (10, s)\}$,
$h(x) = x\%5$,
and you have four buffer slots each of which can hold two tuples. Illustrate the steps of computing $R \bowtie_{R.1=S.1} S$ with hash join using h.

Partition-based $\bowtie$

Hash join $R \bowtie S$: using the same hash function,

- partition R and S into m blocks $\{R_1, \ldots, R_m\}$ $\{S_1, \ldots, S_m\}$ such that each partition block fits in memory
  - tuples in R_i will only join with tuples in S_i
- then, for every R_i, load it in memory, scan S_i, and produce join results (just like block nested loops)

Exercise. Suppose
$R = \{(14, x), (135, y), (40, x), (10, z)\}$,
$S = \{(3, p), (10, q), (14, r), (10, s)\}$,
$h(x) = x\%5$,
and you have four buffer slots each of which can hold two tuples. Illustrate the steps of computing $R \bowtie_{R.1=S.1} S$ with hash join using h.
What would happen if you had only three slots?

Partition-based $\bowtie$

Hash join $R \bowtie S$

- Cost?
  - $2(B_R + B_S)$ I/Os to build partitions
  - $(B_R + B_S)$ I/Os for probing and matching

Partition-based $\bowtie$

Hash join $R \bowtie S$

- Cost?
  - $2(B_R + B_S)$ I/Os to build partitions
  - $(B_R + B_S)$ I/Os for probing and matching
- $cost(R \bowtie S)$ is the sum of these costs

Partition-based $\bowtie$

Hash join $R \bowtie S$

- Cost?
  - $2(B_R + B_S)$ I/Os to build partitions
  - $(B_R + B_S)$ I/Os for probing and matching
- $cost(R \bowtie S)$ is the sum of these costs
- in our running example, we have 6,000 I/Os, which takes 1.5 minutes (about the same as block nested loops and sort-merge algorithms)

Using special datastructures for $\bowtie$

Recall the join index from last lecture

- Binary relation $\{(r_i, s_j), \ldots\}$ over tuple surrogates in R and S, such that $r_i \bowtie s_j$

Using special datastructures for $\bowtie$

Recall the join index from last lecture

- Binary relation $\{(r_i, s_j), \ldots\}$ over tuple surrogates in R and S, such that $r_i \bowtie s_j$

Algorithm:

- Scan join index, to find matching tuples (r_i, s_j)
- retrieve matching tuples
- add tuples to result

Using special datastructures for $\bowtie$

Cost of $R \bowtie S$ with join index:

- if R and S are clustered on join attributes, then at worst we have $AccessCost + B_R + T_R \cdot \log B_S$ I/Os
- if R and S are not clustered on join attributes, then at worst we have $AccessCost + T_R + T_R \cdot \log B_S$ I/Os

Using special datastructures for $\bowtie$

Cost of $R \bowtie S$ with join index:

- if R and S are clustered on join attributes, then at worst we have $AccessCost + B_R + T_R \cdot \log B_S$ I/Os
- if R and S are not clustered on join attributes, then at worst we have $AccessCost + T_R + T_R \cdot \log B_S$ I/Os

In our running example, we have $1000 + 10000 \cdot 10 = 101,000$ I/Os

Using special datastructures for $\bowtie$

Cost of $R \bowtie S$ with join index:

- if R and S are clustered on join attributes, then at worst we have $AccessCost + B_R + T_R \cdot \log B_S$ I/Os
- if R and S are not clustered on join attributes, then at worst we have $AccessCost + T_R + T_R \cdot \log B_S$ I/Os

In our running example, we have $1000 + 10000 \cdot 10 = 101,000$ I/Os

Best suited for highly selective predicates ...

The join operation $\bowtie$

- All join algorithms work on equi-join predicates
  - only nested loops and join-index algorithms work for non equi-join predicates
  - fortunately, equi-join is the most common join type

The join operation $\bowtie$

- All join algorithms work on equi-join predicates
  - only nested loops and join-index algorithms work for non equi-join predicates
  - fortunately, equi-join is the most common join type
- Join is most optimized physical operator
  - Four classes: iteration, order, partition, special datastructures

The join operation $\bowtie$

- All join algorithms work on equi-join predicates
  - only nested loops and join-index algorithms work for non equi-join predicates
  - fortunately, equi-join is the most common join type
- Join is most optimized physical operator
  - Four classes: iteration, order, partition, special datastructures
- Figuring out the best join algorithm for a particular pair of relations (in the context of a larger query plan) is the job of the query optimizer
  - important choice, since we are talking about seconds vs. days!

Wrap up

- Query processing
  - Evaluating selections, projections, and binary ops
  - Evaluating joins

Wrap up

- Query processing
  - Evaluating selections, projections, and binary ops
  - Evaluating joins
- Next lecture: Data statistics & Views

Image credits

- Our textbook (Silberschatz *et al.*, 2006)