Lecture 16:
Relational Operators

Announcements

1. Should I attend grad school? Do I have the right profile?

2. But… SnapBook (the hottest tech giant) is giving me 150k

3. Why is CS the right choice?

Graduate School Information Panel

Thursday, Nov 9 @ 3:00PM
1240CS

Should I attend graduate school in CS?

How do I choose the right graduate program?

How do I prepare a competitive application?

Join us for a live Q&A with CS faculty,
graduate students, and a
graduate school admissions coordinator!

Lecture 16:
Relational Operators

Today's Lecture

1. Logical vs Physical Operators

2. Select, Project

3. Prelims on Joins

1. Logical vs Physical Operators

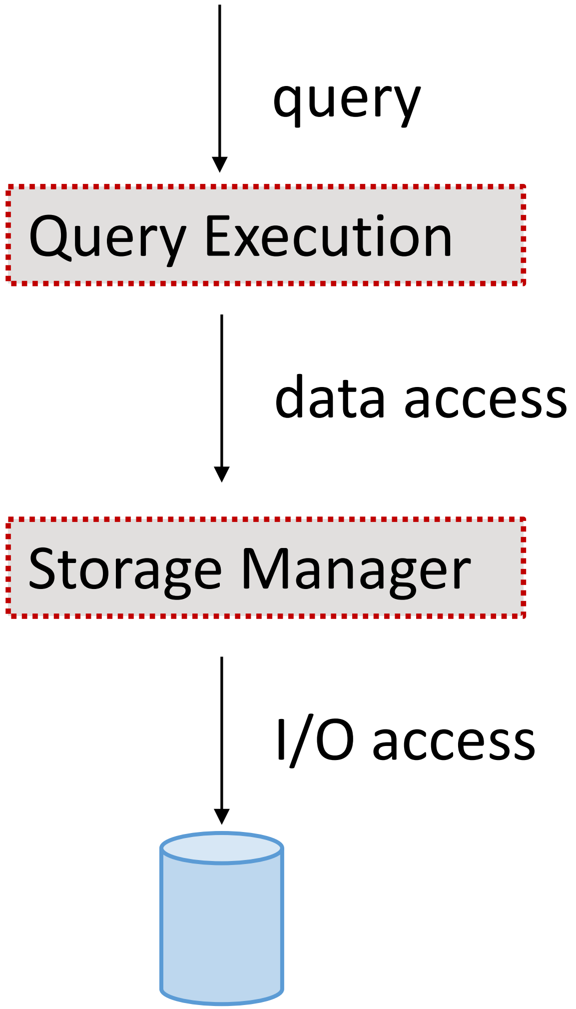
What you will learn about in this section

1. Recap: DB queries

2. Logical Plan

3. Physical Plan

Architecture of a DBMS



Logical vs Physical Operators

- Logical operators
  - *what* they do
  - e.g., union, selection, project, join, grouping

- Physical operators
  - *how* they do it
  - e.g., nested loop join, sort-merge join, hash join, index join

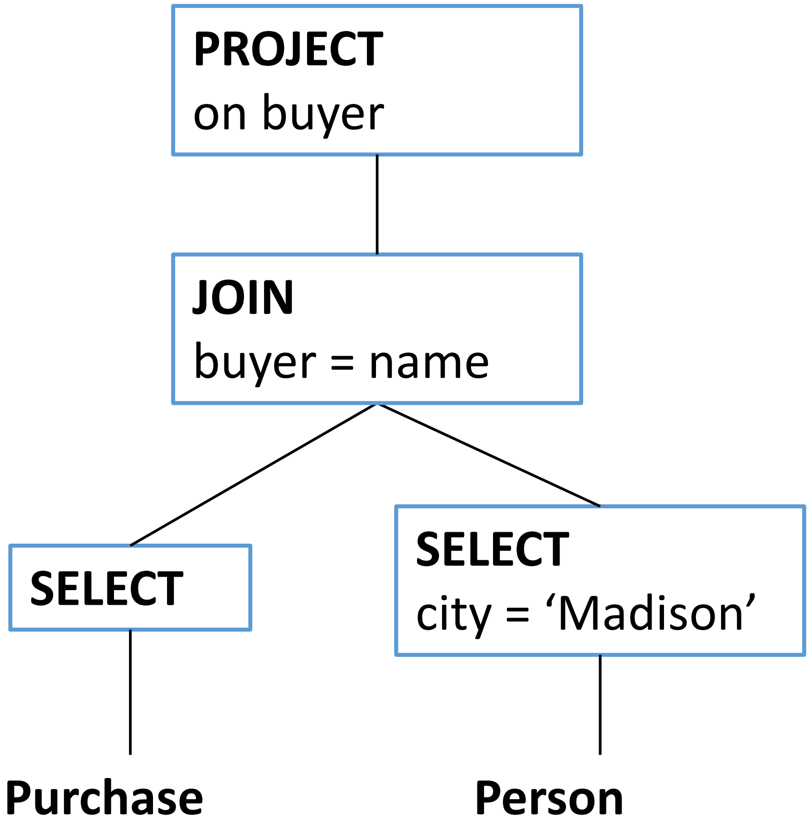
Example

```
SELECT P.buyer
FROM   Purchase P, Person Q
WHERE  P.buyer=Q.name
AND    Q.city='Madison'
```

- Assume that Person has a B+ tree index on city

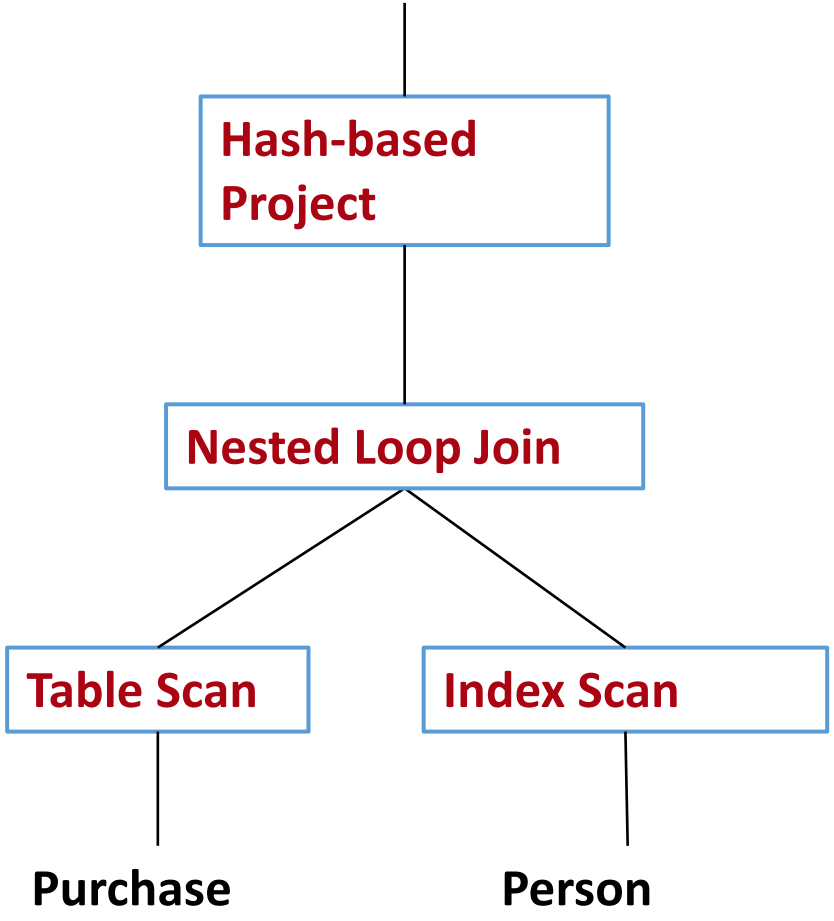
Example: Logical Plan

```
SELECT P.buyer
FROM   Purchase P, Person Q
WHERE  P.buyer=Q.name
AND    Q.city='Madison'
```



Example: Physical Plan

```
SELECT P.buyer
FROM   Purchase P, Person Q
WHERE  P.buyer=Q.name
AND    Q.city='Madison'
```



Relational Operators

We will see implementations for the following relational operators:

- select
- project
- join
- aggregation
- set operators

2. Selection and Projection

What you will learn about in this section

1. Selection

2. Projection

Select Operator

access path = way to retrieve tuples from a table

- **File Scan**
  - scan the entire file
  - I/O cost: O(N), where N = #pages
- **Index Scan:**
  - use an index available on some predicate
  - I/O cost: it varies depending on the index

Index Scan Cost

I/O cost for index scan

- Hash index: O(1)
  - but we can only use it with equality predicates
- B+ tree index: $O(\log_F N) + X$
  - X depends on whether the index is clustered or not:
    - *unclustered*: X = # selected tuples
    - *clustered*: X = (#selected tuples)/ (#tuples per page)

B+ Tree Scan Example

Example

- A relation with 1M records
- 100 records on a page
- 500 (key, rid) pairs on a page

	1% Selectivity	**10% Selectivity**
clustered	3+100	3+1000
unclustered	3+10,000	3+100,000
unclustered + sorting	3+(~10,000)	3+(~10,000)

General Selection Condition

- So far we studied selection on a single attribute
- How do we use indexes when we have multiple selection conditions?
  - `R.a = 10` **`AND`** `R.b > 10`
  - `R.a = 10` **`OR`** `R.b < 20`

Index Matching

- We say that an index *matches* a selection predicate if the index can be used to evaluate it
- Consider a conjunction-only selection. An index matches (part of) a predicate if
  - Hash: only equality operation & the predicate includes *all* index attributes
  - B+ Tree: the attributes are a prefix of the search key (any ops are possible)

Example

- A relation **R**`(a,b,c,d)`
- Does the index match the predicate?

Predicate	B+ tree on (a,b,c)	Hash index on (a,b,c)
a=5 **AND** b=3	yes	no
a>5 **AND** b<4	yes	no
b=3	no	no
a=5 **AND** c>10	yes	no
a=5 **AND** b=3 **AND** c=1	yes	yes
a=5 **AND** b=3 **AND** c=1 **AND** d >6	yes	yes

a=5 and b=3 and c=1 are primary conjuncts here

Index Matching

- A predicate can match more than one index
- **Example**:
  - hash index on (a) and B+ tree index on (b, c)
  - predicate: `a=7 AND b=5 AND c=4`
  - which index should we use?
    1. use either index
    2. use both indexes, then intersect the rid sets, and then fetch the tuples

Choosing the Right Index

- Selectivity of an access path = *fraction* of data pages that need to be retrieved
- We want to choose the *most selective* path!
- Estimating the selectivity of an access path is a hard problem

Estimating Selectivity

- Predicate: a=3 **AND** b=4 **AND** c=5
- hash index on (a,b,c)
  - selectivity is approximated by *#pages / #keys*
  - #keys is known from the index
- hash index on (b)
  - multiply the *reduction factors* for each primary conjunct
  - *reduction factor = #pages/#keys*
  - if #keys is unknown, use 1/10 as default value
  - this assumes independence of the attributes!

Estimating Selectivity

- Predicate: `a > 10` **AND** `a < 60`
- If we have a range condition, we assume that the values are uniformly distributed
- The selectivity will be approximated by $\frac{interval}{High - Low}$

Predicates and Disjunction

- hash index on (a) **+** hash index on (b)
  - `a=7` **`or`** `b>5`
  - a file scan is required
- hash index on (a) **+** B+ tree on (b)
  - `a=7` **`or`** `b>5`
  - scan or use both indexes (fetch rids and take the union)
- hash index on (a) **+** B+ tree on (b)
  - `(a=7` **`or`** `c>5)` **`and`** `b > 5`
  - we can use the B+ tree

Projection

Simple case: `SELECT R.a, R.d`

- scan the file and for each tuple output R.a, R.d

Hard case: `SELECT DISTINCT R.a, R.d`

- project out the attributes
- eliminate *duplicate tuples* (this is the difficult part!)

Projection: Sort-based

Naïve algorithm:

1. scan the relation and project out the attributes
2. sort the resulting set of tuples using all attributes
3. scan the sorted set by comparing only adjacent tuples and discard duplicates

Example

R(a, b, c, d, e)

- M = 1000 pages
- B = 20 buffer pages
- Each field in the tuple has the same size
- Suppose we want to project on attribute **a**

Sort-based Cost Analysis

- initial scan = *1000 I/Os*
- after projection T =(1/5)*1000 = 200 pages
- cost of writing T = *200 I/Os*
- sorting in 2 passes = 2 * 2 * 200 = *800 I/Os*
- final scan = *200 I/Os*

total cost = 2200 I/Os

Projection: Sort-based

We can improve upon the naïve algorithm by modifying the sorting algorithm:

1. In Pass **0** of sorting, project out the attributes
2. In subsequent passes, eliminate the duplicates while merging the runs

Sort-based Cost Analysis

- we can sort in 2 passes
- first pass costs *1000 + 200 = 1200 I/Os*
- the second pass costs *200 I/Os* (not counting writing the result to disk)

total cost = 1400 I/Os

Projection: Hash-based

2-phase algorithm:

- **partitioning**
  - project out attributes and split the input into B-1 partitions using a hash function *h*
- **duplicate elimination**
  - read each partition into memory and use an in-memory hash table (with a *different* hash function) to remove duplicates

Projection: Hash-based

When does the hash table fit in memory?

- size of a partition = $T\ /\ (B-1)$, where T is #pages after projection
- size of hash table = $f \cdot T\ /\ (B-1)$, where is a fudge factor (typically ~ 1.2)
- So, it must be $B\ >\ f \cdot T\ /\ (B-1)$, or approximately $B > \sqrt{f \cdot T}$

Hash-based Cost Analysis

- T = 200 so the hash table fits in memory!
- partitioning cost = 1000 + 200 = 1200 I/Os
- duplicate elimination cost = 200 I/Os

total cost = 1400 I/Os

Comparison

- Benefits of sort-based approach
  - better handling of skew
  - the result is sorted

- The I/O costs are the same if $B^2 > T$
  - 2 passes are needed by both algorithms

Projection: Index-based

- Index-only scan
  - projection attributes subset of index attributes
  - apply projection algorithm only to data entries
- If an *ordered index* contains all projection attributes as prefix of search key:
  1. retrieve index data entries in order
  2. discard unwanted fields
  3. compare adjacent entries to eliminate duplicates

3. Joins

What you will learn about in this section

1. RECAP: Joins

2. Nested Loop Join (NLJ)

3. Block Nested Loop Join (BNLJ)

4. Index Nested Loop Join (INLJ)

1. Nested Loop Joins

What you will learn about in this section

1. RECAP: Joins

2. Nested Loop Join (NLJ)

3. Block Nested Loop Join (BNLJ)

4. Index Nested Loop Join (INLJ)

RECAP: Joins

Joins: Example

$\mathbf{R} \bowtie \boldsymbol{S}$

```
SELECT R.A,B,C,D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples $\mathrm{r} \in R, s \in S$ *such that* $r.A = s.A$

R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

S

A	D
3	7
2	2
2	3

A	B	C	D
2	3	4	2

Joins: Example

$\mathbf{R} \bowtie \boldsymbol{S}$

```
SELECT R.A,B,C,D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples $\mathrm{r} \in R, s \in S$ *such that* $r.A = s.A$

R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

S

A	D
3	7
2	2
2	3

A	B	C	D
2	3	4	2
2	3	4	3

Joins: Example

$\mathbf{R} \bowtie \boldsymbol{S}$

```
SELECT R.A,B,C,D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples $\mathrm{r} \in R, s \in S$ *such that* $r.A = s.A$

R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

S

A	D
3	7
2	2
2	3

A	B	C	D
2	3	4	2
2	3	4	3
2	5	2	2

Joins: Example

$\mathbf{R} \bowtie \boldsymbol{S}$

```
SELECT  R.A,B,C,D
FROM    R, S
WHERE   R.A = S.A
```

Example: Returns all pairs of tuples $\mathrm{r} \in R, s \in S$ *such that* $r.A = s.A$

R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

S

A	D
3	7
2	2
2	3

A	B	C	D
2	3	4	2
2	3	4	3
2	5	2	2
2	5	2	3

Joins: Example

$\mathbf{R} \bowtie \boldsymbol{S}$

```
SELECT R.A,B,C,D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples $\mathrm{r} \in R, s \in S$ *such that* $r.A = s.A$

R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

S

A	D
3	7
2	2
2	3

A	B	C	D
2	3	4	2
2	3	4	3
2	5	2	2
2	5	2	3
3	1	1	7

Semantically: A Subset of the Cross Product

$\mathbf{R} \bowtie \boldsymbol{S}$

```
SELECT R.A,B,C,D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples $\mathrm{r} \in R, s \in S$ *such that* $r.A = s.A$

R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

×

S

A	D
3	7
2	2
2	3

Cross Product → … → Filter by conditions (r.A = s.A)

A	B	C	D
2	3	4	2
2	3	4	3
2	5	2	2
2	5	2	3
3	1	1	7

Can we actually implement a join in this way?

Notes

- We write $\mathbf{R} \bowtie \boldsymbol{S}$ to mean *join R and S by returning all tuple pairs where **all shared attributes** are equal*
- We write $\mathbf{R} \bowtie \boldsymbol{S}$ **on** $\boldsymbol{A}$ to mean *join R and S by returning all tuple pairs where **attribute(s) A** are equal*
- For simplicity, we'll consider joins on **two tables** and with **equality constraints** ("equijoins")

However joins *can* merge > 2 tables, and some algorithms do support non-equality constraints!

Nested Loop Joins

Notes

- We are again considering "IO aware" algorithms: ***care about disk IO***

- Given a relation R, let:
  - T(R) = # of tuples in R
  - P(R) = # of pages in R

Recall that we read / write entire pages with disk IO

- Note also that we omit ceilings in calculations… good exercise to put back in!

Nested Loop Join (NLJ)

Compute $\mathrm{R} \bowtie S \ on\ A$:

```
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

Nested Loop Join (NLJ)

```
Compute R ⋈ S on A:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

Cost:

P(R)

1. Loop over the tuples in R

Note that our IO cost is based on the number of ***pages*** loaded, not the number of tuples!

Nested Loop Join (NLJ)

```
Compute R ⋈ S on A:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

Cost:

P(R) + T(R)*P(S)

1. Loop over the tuples in R
2. **For every tuple in R, loop over all the tuples in S**

Have to read ***all of S*** from disk for ***every tuple in R!***

Nested Loop Join (NLJ)

```
Compute R ⋈ S on A:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

Cost:

$P(R) + T(R)*P(S)$

1. Loop over the tuples in R
2. For every tuple in R, loop over all the tuples in S
3. **Check against join conditions**

Note that NLJ can handle things other than equality constraints… just check in the *if* statement!

Nested Loop Join (NLJ)

```
Compute R ⋈ S on A:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

What would *OUT* be if our join condition is trivial (*if TRUE*)?

OUT could be bigger than P(R)*P(S)… but usually not that bad

Cost:

$$P(R) + T(R)*P(S) + OUT$$

1. Loop over the tuples in R
2. For every tuple in R, loop over all the tuples in S
3. Check against join conditions
4. **Write out (to page, then when page full, to disk)**

Nested Loop Join (NLJ)

```
Compute R ⋈ S on A:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

Cost:

P(R) + T(R)*P(S) + OUT

What if R ("outer") and S ("inner") switched?

P(***S***) + T(***S***)*P(***R***) + OUT

Outer vs. inner selection makes a huge difference-
DBMS needs to know which relation is smaller!

IO-Aware Approach

Block Nested Loop Join (BNLJ)

Given ***B+1*** pages of memory

```
Compute R ⋈ S on A:
  for each B−1 pages pr of R:
    for page ps of S:
      for each tuple r in pr:
        for each tuple s in ps:
          if r[A] == s[A]:
            yield (r,s)
```

Cost:

$\mathrm{P}(R)$

1. **Load in B-1 pages of R at a time (leaving 1 page each free for S & output)**

Note: There could be some speedup here due to the fact that we're reading in multiple pages sequentially however we'll ignore this here!

Block Nested Loop Join (BNLJ)

Given ***B+1*** pages of memory

```
Compute R ⋈ S on A:
  for each B-1 pages pr of R:
    for page ps of S:
      for each tuple r in pr:
        for each tuple s in ps:
          if r[A] == s[A]:
            yield (r,s)
```

Cost:

$$P(R) + \frac{P(R)}{B-1}P(S)$$

1. Load in B-1 pages of R at a time (leaving 1 page each free for S & output)

2. **For each (B-1)-page segment of R, load each page of S**

Note: Faster to iterate over the *smaller* relation first!

Block Nested Loop Join (BNLJ)

Given ***B+1*** pages of memory

```
Compute R ⋈ S on A:
  for each B-1 pages pr of R:
    for page ps of S:
      for each tuple r in pr:
        for each tuple s in ps:
          if r[A] == s[A]:
            yield (r,s)
```

Cost:

$$P(R) + \frac{P(R)}{B-1} P(S)$$

1. Load in B-1 pages of R at a time (leaving 1 page each free for S & output)
2. For each (B-1)-page segment of R, load each page of S
3. **Check against the join conditions**

BNLJ can also handle non-equality constraints

Block Nested Loop Join (BNLJ)

Given ***B+1*** pages of memory

Compute $\mathrm{R} \bowtie S\ on\ A$:
```
  for each B−1 pages pr of R:
    for page ps of S:
      for each tuple r in pr:
        for each tuple s in ps:
          if r[A] == s[A]:
            yield (r,s)
```

Again, ***OUT*** could be bigger than P(R)*P(S)… but usually not that bad

Cost:

$$\mathrm{P}(R) + \frac{P(R)}{B-1} P(S) + \mathrm{OUT}$$

1. Load in B-1 pages of R at a time (leaving 1 page each free for S & output)
2. For each (B-1)-page segment of R, load each page of S
3. Check against the join conditions
4. **Write out**

BNLJ vs. NLJ: Benefits of IO Aware

- In BNLJ, by loading larger chunks of R, we minimize the number of full *disk reads* of S
  - We only read all of S from disk for ***every (B-1)-page segment of R***!
  - Still the full cross-product, but more done only *in memory*

NLJ

P(R) + T(R)*P(S) + OUT

BNLJ

$$P(R) + \frac{P(R)}{B-1} P(S) + \text{OUT}$$

BNLJ is faster by roughly $\frac{(B-1)T(R)}{P(R)}$!

BNLJ vs. NLJ: Benefits of IO Aware

- Example:
  - R: 500 pages
  - S: 1000 pages
  - 100 tuples / page
  - We have 12 pages of memory (B = 11)

Ignoring OUT here…

- NLJ: Cost = 500 + **50,000*1000** = **50 Million IOs** ~= **<u>140 hours</u>**

- BNLJ: Cost = 500 + $\frac{500*1000}{10}$ = **50 *Thousand* IOs ~= <u>0.14 hours</u>**

A very real difference from a small change in the algorithm!

Smarter than Cross-Products

Smarter than Cross-Products: From Quadratic to Nearly Linear

- All joins that compute the ***full cross-product*** have some **quadratic** term
  - For example we saw:

NLJ $\mathrm{P(R)} + \mathrm{T(R)P(S)} + \mathrm{OUT}$

BNLJ $\mathrm{P}(R) + \frac{\boldsymbol{P(R)}}{B-1}\boldsymbol{P(S)} + \mathrm{OUT}$

- Now we'll see some (nearly) linear joins:
  - ~ O(P(R) + P(S) + ***OUT***), where again ***OUT*** could be quadratic but is usually better

We get this gain by ***taking advantage of structure***- moving to equality constraints ("equijoin") only!

Index Nested Loop Join (INLJ)

```
Compute R ⋈ S on A:
   Given index idx on S.A:
      for r in R:
         s in idx(r[A]):
            yield r,s
```

Cost:

P(R) + T(R)**L* + OUT

where *L* is the IO cost to access all the distinct values in the index; assuming these fit on one page, $\mathrm{L} \sim 3$ is good est.

→ We can use an **index** (e.g. B+ Tree) to *avoid doing the full cross-product!*