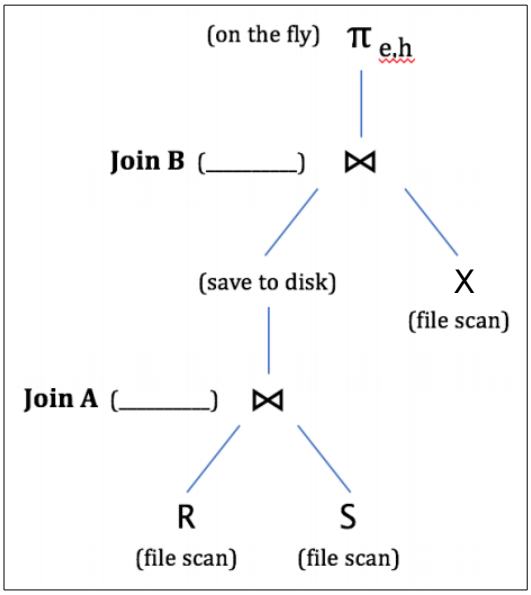
Section 7 Handout: Updates in blue

---

1. (414 SP17 Final) We have the query plan and meta data below for R(e, f), S(f, g), and X(g, h):

Table	# tuples	# blocks
R	1,000	100
S	5,000	200
X	100,000	10,000
R ⋈ S	5,000	20

Column	# distinct	Low	High
R.f	100	1	1,000
S.f	1,000	1	2,000
S.g	5,000	1	2,000
X.g	1,000	1	10,000



We have 11 memory pages available.

a. What is the estimated cost of Join B in the plan if we implement it with a **block nested loop** join?
B(R ⋈ S) + B(R ⋈ S)B(X) / (M-1) = 20 + 20 * 10,000 / (11 - 1) = 20,020

b. What is the estimated cost of Join B the if we use an **indexed** join? Assume that we have a **clustered** index on X(g).
B(R ⋈ S) + T(R ⋈ S)B(X) / V(X, g) = 20 + 5,000 * 10,000 / 1000 = 50,020

c. What is the estimated cost of Join A in the plan if we implement it with a **block nested loop** join?
B(R) + B(R)B(S) / (M-1) = 100 + 100 * 200 / (11 - 1) = 2,100

d. What is the estimated cost of Join A if we use an **indexed** join? Assume that we have an **unclustered** index on S(f).
B(R) + T(R)T(S) / V(S, f) = 100 + 1,000 * 5,000 / 1000 = 5,100

e. What is the total cost (IOs) of this plan if we use the best choice of join algorithm for A and B (from above)?
In addition to the two joins above, we need to write R ⋈ S to disk, which costs a projected 20 IOs.
The final projection has no cost.
Thus, the total cost is:
(B(R ⋈ S) + B(R ⋈ S)B(X) / (M-1)) + (B(R) + B(R)B(S) / (M-1)) + B(R ⋈ S)
= 20,020+ 2,100 + 20
= 22,140

2. Create the index that is easiest to create that will make the following queries run faster. Assume there are no previous indexes. Use the data from Q1.

a.

```
SELECT * FROM R
WHERE R.f > 100 AND R.f < 700
CREATE CLUSTERED INDEX idx1 ON R(f)
```

We need a clustered index on R.f to deal with range selection

b.

```
SELECT * FROM S
WHERE S.g = 344
CREATE INDEX idx2 ON S(g)
```

No need for clustering because we are only selecting for a single value of S.g

---

3. We have the relation V(m, n, p), W(p, q, r) and the following two queries. For each of the unclustered indexes below identify which queries will run faster under that index versus no index. Assume all attributes range from 0 to 1000 and are distributed uniformly.

(A)

```
SELECT * FROM V
WHERE V.m = 344
```

(B)

```
SELECT * FROM V
WHERE V.m = 344 AND
      V.p = 311
```

(C) (assume nested loop join)

```
SELECT * FROM V, W
WHERE V.p = W.p
```

a. `INDEX idx1 on V(m)`

A, B

A and B select on V.m

Not C because joining does not utilize V.m

b. `INDEX idx2 on V(m,p)`

A, B

A selects on V.m and will not be affected by (doesn't care about) the secondary ordering on V.p

B selects on both V.m and V.p

Not C because the primary ordering for the index is on V.m (but C cares about V.p)

c. `INDEX idx3 on V(p,m)`

B, C

B selects on on both V.m and V.p (note the order in WHERE predicate does not matter)

C selects on V.p and will not be affected by (doesn't care about) the secondary ordering on V.m

Not A because the primary ordering for the index is on V.p (but A cares about V.m)

4. (344 AU16 MT)

Purchase(pid, custId, quantity, price)

T(Purchase) = 1000
B(Purchase) = 100
V(Purchase, price) = 100
Price range = [0, 200)

Customer(custId, name, city)

T(Customer) = 3000
B(Customer) = 200
V(Customer, custId) = 3000

Number of memory pages available = 20

a. Consider the query below and circle the indexes that will produce a speed up:

```
SELECT * FROM Purchase P, Customer C
WHERE P.custId = C.custId AND
      P.price < 100 AND
      C.custId = 42
```

(1) ~~Hashtable index on Purchase(price)~~	(2) ~~B-tree index on Purchase(pid, price)~~
(3) Hashtable index on Customer(custId)	(4) Hashtable index on Purchase(custId)
(5) B-tree index on Purchase(price, pid)	(6) ~~Hashtable index on Purchase(price, pid)~~

1. Does not work because selection for price is a range, hashtables do not speed up for range.
2. Does not work because the B-tree primary ordering is on pid first.
3. Works because we are looking for a single custId which can be found with hashtables in O(1) time.
4. Works because we are equivalently looking for Purchase.cusId = 42.
5. Works because the B-tree primary ordering is on price which helps the range selection.
6. Does not work for the same reason as (1). Adding pid does not change this.

b. Which join algorithm would you use to execute the join in a) to minimize execution time? Assume that there are no indexes available. Be clear about how the join will be executed, i.e., what attribute will you sort on if sorting is involved, what relation will you construct a hashtable on if one is needed, etc. Briefly explain why.

We will first perform selection to reduce the size of the input tables:
- Evaluating the selection on P.price will result in 50 blocks of output:
  - X = (c - min(P, price))/(max(P, price) - min(P, price)) = (100 - 0)/(200 - 0) = 1/2
  - Total = X * B(P) = 1/2 * 100 = 50 pages (call this T1)
- Evaluating the selection on C.custId will result in 1 page of output:
  - Selectivity factor (X) = 1/V(C, cusId) = 1/3000
  - Total = X * B(C) = 1/3000 * 200 = 1 page (call this T2).

Since we don't have enough pages in memory (20 pages) to hold both T1 and T2, we don't want to perform the join using sort-merge.
We can either do nested loop join (with T2 being the outer loop relation that always reside in memory), or hash join after constructing hashtable on T2.