Lab 10 - EMS & HJ & INLJ

CompSci 316
Fall 2022

TAs

Presenter: Yuxi Liu

Q&A TA

Session 1(10:15am - 11:30am): Danny Luo, Joyce Wang, Chengyu Wu, Tong Lin, Haibo Xiu

Session 2(1:45pm - 3:00pm): Zhe Wang, Justin Lim, Haibo Xiu

Check-in

- 11/04 01D: 11:05-11:09am
  - code:XXXX
- 11/04 02D: 2:15-2:19pm
  - code:XXXX

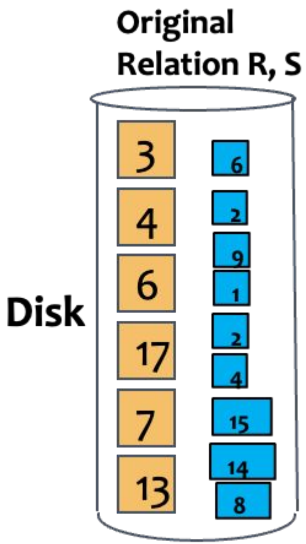
Roadmap

- Example of Hash Join (HJ) **(Lec9-30)**
- Example of Index Nested Loop Join (INLJ) **(Lec9-41)**
- Practice of External Merge Sorting (EMS)
- (If have time) Performance of SMJ vs. HJ **(Lec9-23,24,31,33)**

Example: Hash Join

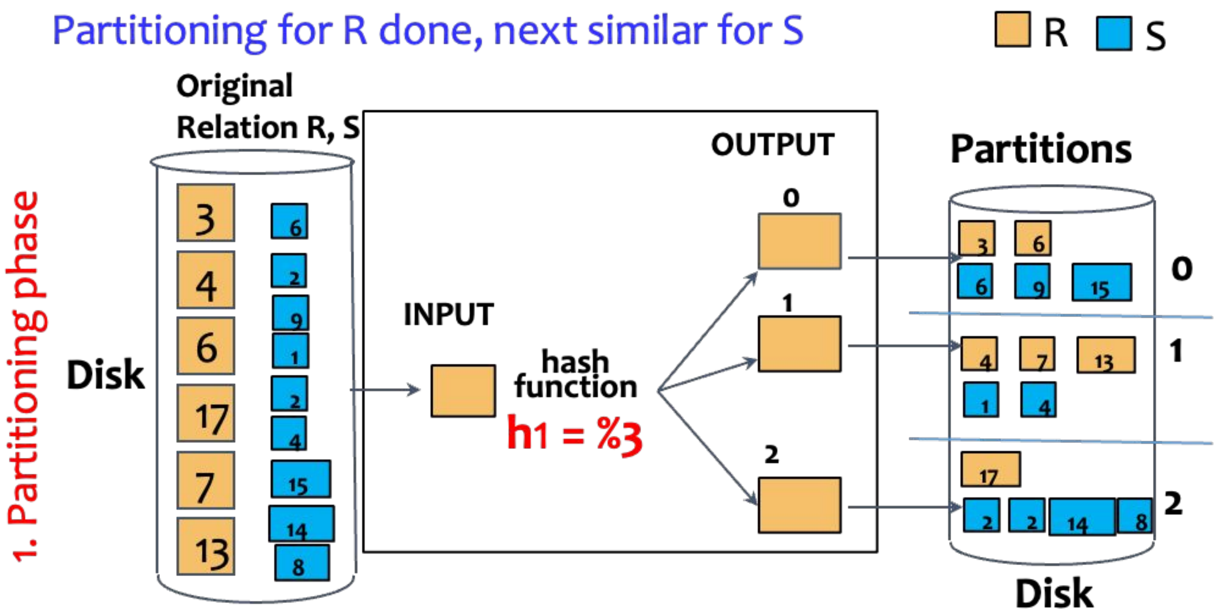
Hash Join: setting

- R(A), S(B)
- $R \bowtie_{R.A = S.B} S$
- B(R) = 6, B(S) = 9, M = 4
- Each page of R, S contains just one record



Hash Join: partitioning (phase 1)

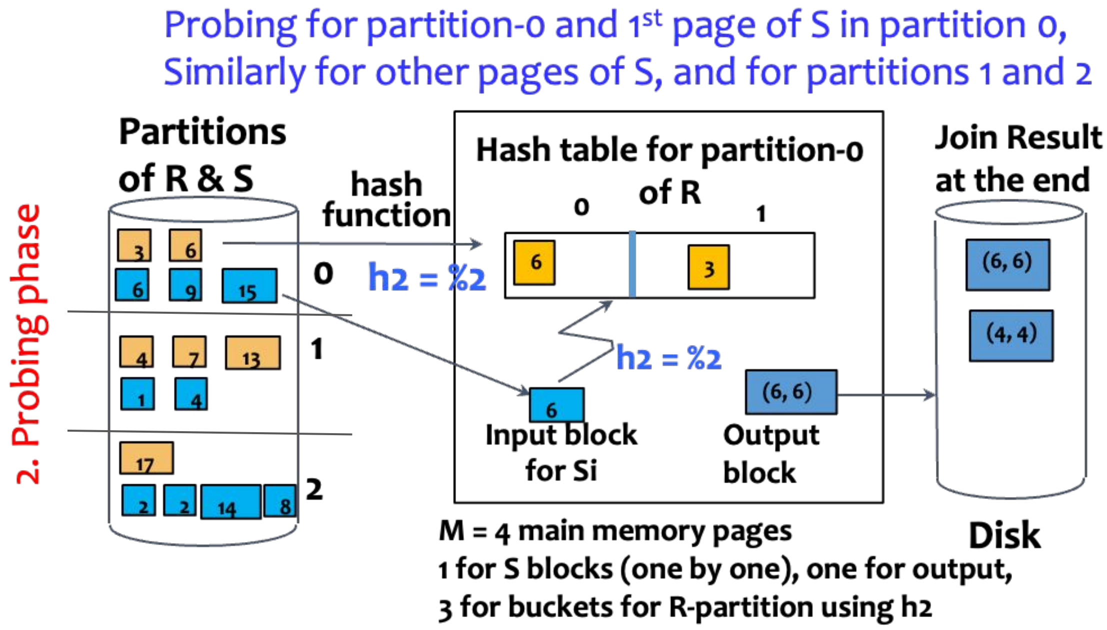
- Hash function h1 for partitioning = A % 3 for R and = B % 3 for S



The quality of hash is not that good here (has “skew”): the number of blocks falling into each bucket is not that even

Hash Join: probing (phase 2)

- Hash function h2 for probing = A % 2 for R and = B % 2 for S



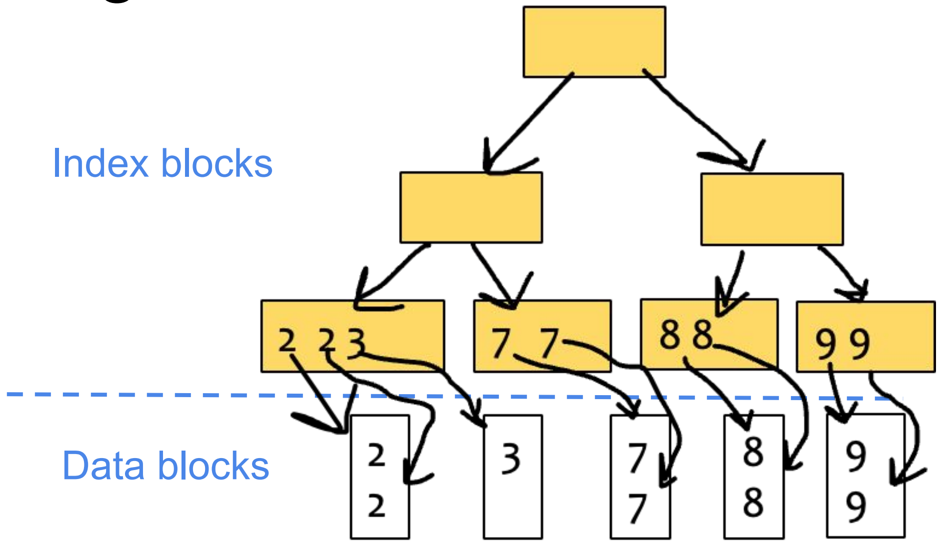
- Note: h1 and h2 cannot be the same, otherwise all R-blocks in partition-0 will hash to the same bucket
- Only 2-pass is sufficient here, since: In each partition, there exists a relation that has <= 2 (= M-2) blocks

What if a partition is too large for memory?
Read it back in and partition it again,
> 2 passes will be needed

Example: Index Nested-loop Join

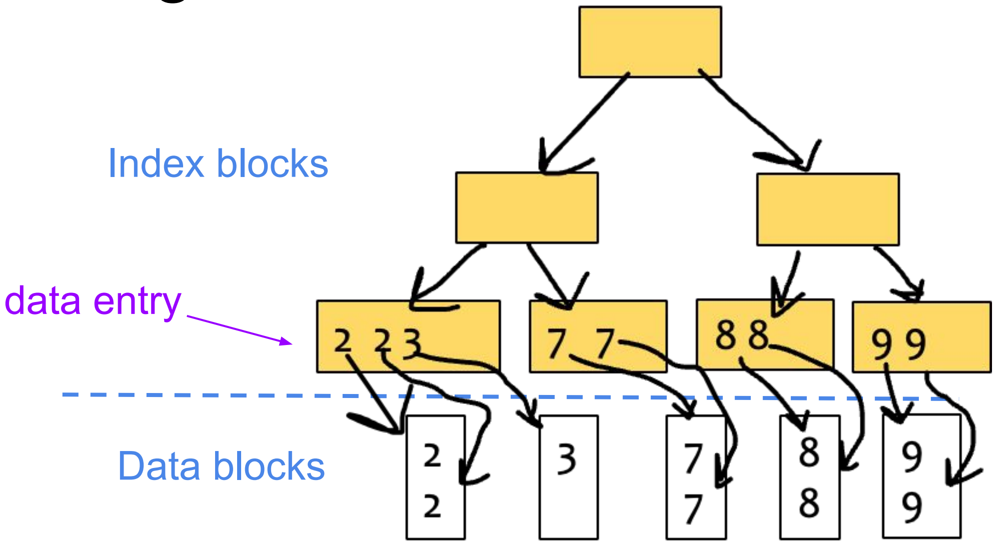
Index Nested-loop Join: setting

- R(A), S(B), M = 3
- $R \bowtie_{R.A = S.B} S$
- R.A values: 7, 2, 9, 8, 3
  - 1 R-tuple/block
  - So B(R) = |R| = 5
- S.B values: 2, 2, 3, 7, 7, 8, 8, 9, 9
  - at most 2 S-tuple/block
  - So |S| = 9, B(S) = 5
- Assume foreign key S.B to primary key R.A
  - Each R tuple joins with at most 2 S tuples that fit in 1 data block of S
- B+ tree index on S.B:
  - **Clustered,** 3 levels
  - All index blocks, data blocks are on disk



Index Nested-loop Join: setting

- M = 3
- R.A values: 7, 2, 9, 8, 3
  - 1 R-tuple/block
  - So B(R) = |R| = 5
- S.B values: 2, 2, 3, 7, 7, 8, 8, 9, 9
  - at most 2 S-tuple/block
  - So |S| = 9, B(S) = 5



Algo:

- For every block of R

  Cost of R = B(R) = 5

  - For every tuple of R in that block
    - Set the value of R.A as the search key
    - Retrieve the matching S tuples pointed to by the matching data entries (pointers)
    - Output the matching pair of R and S tuples

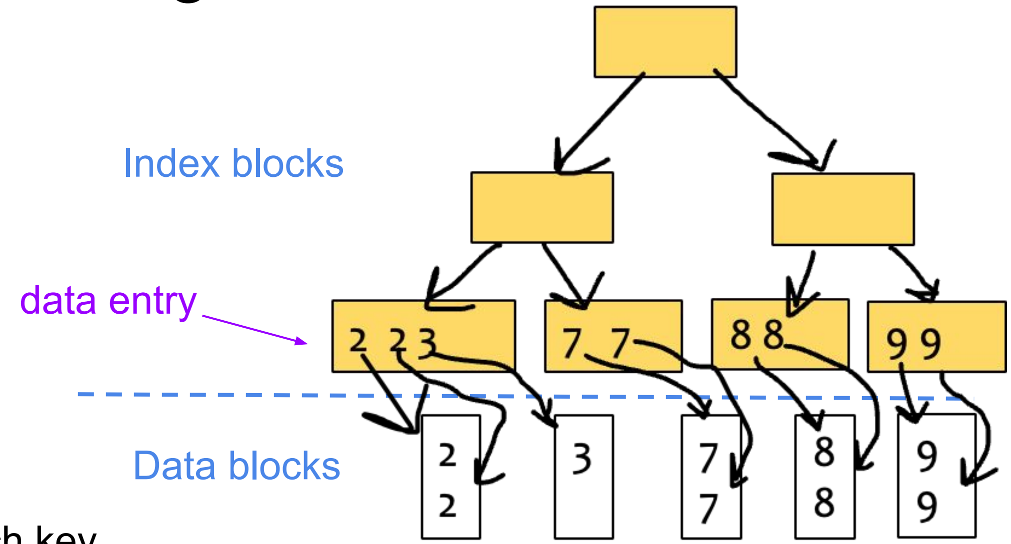
Index Nested-loop Join: setting

- M = 3
- R.A values: 7, 2, 9, 8, 3
  - 1 R-tuple/block
  - So B(R) = |R| = 5

Cost of R = B(R) = 5

Algo:

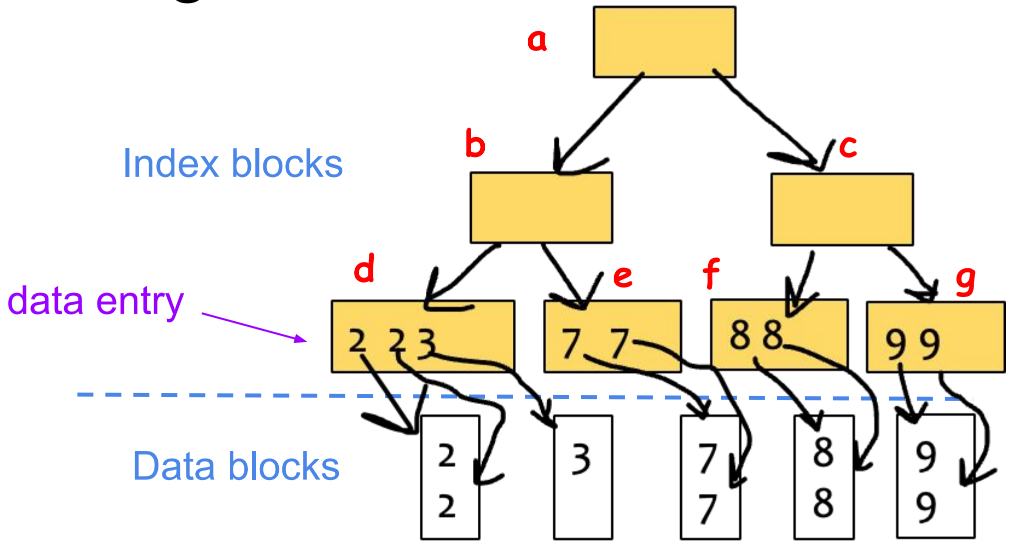
- For every block of R
  - For every tuple of R in that block
    - Set the value of R.A as the search key
    - Retrieve the matching S tuples pointed to by the matching data entries (pointers)
    - Output the matching pair of R and S tuples



So for every R.A value: probing on index blocks + accessing data block

Index Nested-loop Join: setting

- M = 2
- R.A values: 7, 2, 9, 8, 3
  - 1 R-tuple/block
  - So B(R) = |R| = 5



Focus on this example:

1. R.A = 7: a -> b -> e
2. R.A = 2: a -> b -> d
3. R.A = 9: a -> c -> g
4. R.A = 8: a -> c -> f
5. R.A = 3: a -> b -> d

Each of 1 to 5 needs one extra I/O cost to read the corresponding data block

Total I/O costs of index nested-loop join = B(R) + |R|(3 + 1) = 25

External Merge Sorting

External Merge Sorting

Suppose you have B(R) = 21 for a relation R and 4 memory blocks available (M = 4). Fill out the following table for the number of sorted runs and I/O cost in each pass of an external merge sorting (for pass = 0, 1, 2, …)

Pass	**# of runs for this pass**	**Run sizes**	**I/O Cost for this pass**
0			
1			
...			

External Merge Sorting

Suppose you have B(R) = 21 for a relation R and 4 memory blocks available (M = 4). Fill out the following table for the number of sorted runs and I/O cost in each pass of an external merge sorting (for pass = 0, 1, 2, …)

Pass	# of runs for this pass	Run sizes	I/O Cost for this pass
0	Ceiling(21/M) = 6	4 or 1	2 * B(R) = 42
1			
...			

Explanation:

1. For level-0 sorted runs, we have 5 of length 4 and 1 of length 1, since 21 = 4 + 4 + 4 + 4 + 1
2. Each blocks of R are read once and written once, so B(R) + B(R) = 42

External Merge Sorting

Suppose you have B(R) = 21 for a relation R and 4 memory blocks available (M = 4). Fill out the following table for the number of sorted runs and I/O cost in each pass of an external merge sorting (for pass = 0, 1, 2, …)

Pass	# of runs for this pass	Run sizes	I/O Cost for this pass
0	Ceiling(21/M) = 6	4 or 1	2 * B(R) = 42
1	Ceiling(6/(M - 1)) = 2	3 * 4 = 12 for the 1st run 2 * 4 + 1 = 9 for the 2nd run	2 * B(R) = 42
...			

Explanation:

1. Why (M - 1) -> One memory block is used to hold the output and flush to disk. So we can combine at most 3 level-0 sorted runs at a time
2. For the first three level-0 runs, they are combined into 1st level-1 run and each of them has 4 blocks (full).
3. For the next three level-0 runs, they are combined into 2nd level-1 run and two of them has 4 blocks (full) and the last one has only 1 block

External Merge Sorting

Suppose you have B(R) = 21 for a relation R and 4 memory blocks available (M = 4). Fill out the following table for the number of sorted runs and I/O cost in each pass of an external merge sorting (for pass = 0, 1, 2, …)

Pass	# of runs for this pass	Run sizes	I/O Cost for this pass
0	Ceiling(21/M) = 6	4 or 1	2 * B(R) = 42
1	Ceiling(6/(M - 1)) = 2	3 * 4 = 12 for the 1st run 2 * 4 + 1 = 9 for the 2nd run	2 * B(R) = 42
2	Ceiling(2/(M - 1)) = 1	12 + 9 = 21 for one run	B(R) = 21

Explanation:

1. Final pass, since we have already combined all the blocks into 1 sorted level-2 run
2. We don't count the I/Os for the final write/flush to disk

Sort-merge join

$R \bowtie_{R.A=S.B} S$

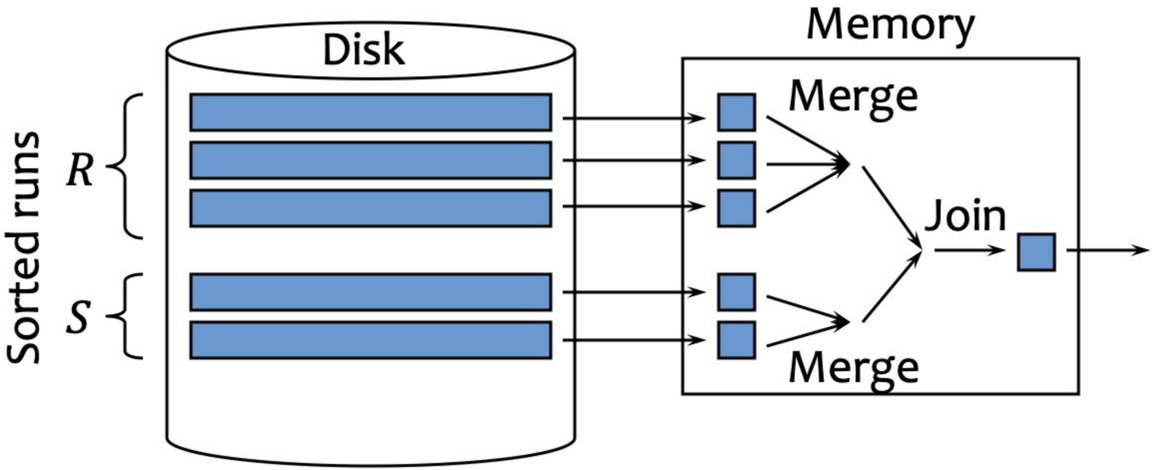
- Sort R and S by their join attributes; then merge

  r, s = the first tuples in sorted R and S
  Repeat until one of R and S is exhausted:
  If $r.A > s.B$ then s = next tuple in S
  else if $r.A < s.B$ then r = next tuple in R
  else output all matching tuples, and
  r, s = next in R and S

- I/O's: sorting $+ 2B(R) + 2B(S)$ (always?)
  - In most cases (e.g., join of key and foreign key)
  - Worst case is $B(R) \cdot B(S)$: everything joins

Optimization of SMJ

- Idea: combine join with the (last) merge phase of merge sort
- Sort: produce sorted runs for R and S such that there are fewer than M of them total
- Merge and join: merge the runs of R, merge the runs of S, and merge-join the result streams as they are generated!



Compute Memory Requirements for

- Two pass SMJ
- Two pass HJ

Performance of SMJ

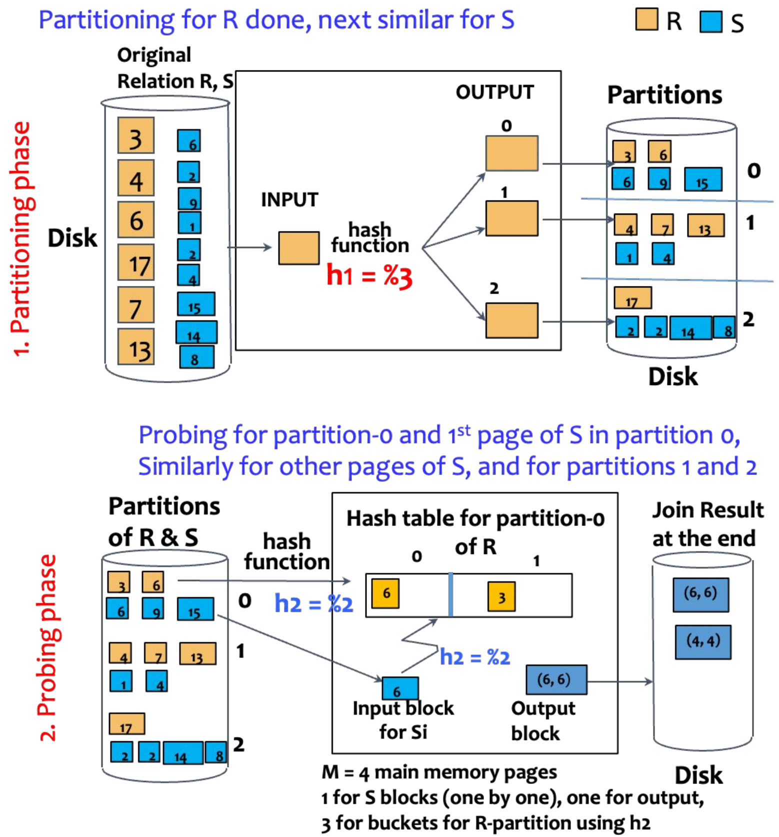
First Pass 0 +
Then (merge + join)

- If SMJ completes in two passes:
  - I/O's: $3 \cdot (B(R) + B(S))$ - why 3?
  - Memory requirement
    - We must have enough memory to accommodate one block from each run: $M > \frac{B(R)}{M} + \frac{B(S)}{M}$
    - $M > \sqrt{B(R) + B(S)}$
- If SMJ cannot complete in two passes:
  - Repeatedly merge to reduce the number of runs as necessary before final merge and join

Performance of (two-pass) hash join

- If hash join completes in two passes:
  - I/O's: $3 \cdot (B(R) + B(S))$
  - Memory requirement:
    - In the probing phase, we should have enough memory to fit one partition of R: $M - 1 > \frac{B(R)}{M-1}$
    - $M > \sqrt{B(R)} + 1$
    - We can always pick R to be the smaller relation, so:

$$M > \sqrt{\min(B(R), B(S))} + 1$$



Hash join versus SMJ

(Assuming two-pass)

- I/O's: same
- Memory requirement: hash join is lower
  - $\sqrt{\min(B(R), B(S))} + 1 < \sqrt{B(R) + B(S)}$
  - Hash join wins when two relations have very different sizes
- Other factors
  - Hash join performance depends on the quality of the hash
    - Might not get evenly sized buckets
  - SMJ can be adapted for inequality join predicates
  - SMJ wins if R and/or S are already sorted
  - SMJ wins if the result needs to be in sorted order