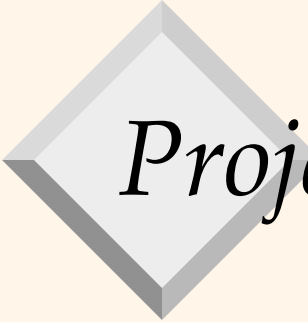


Practicum in Database Systems

Project 3 intro



Project 3 Introduction

- Expand your Project 2 codebase so it's not a "toy" implementation anymore
 - Use "proper" efficient I/O
 - Implement external sorting
 - Implement BNLJ and SMJ



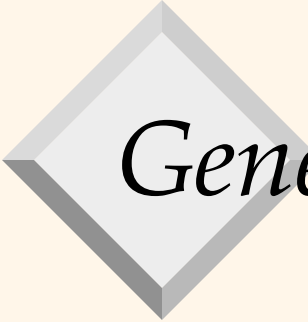
Warning

- This is a large project
- Budget your time around constraints such as 4320 prelim and Fall Break
- Extra requirement: checkpoint Oct 17
 - To make sure you get started
 - Once you're at the checkpoint, the remainder of the work is easy to split across team members.



Your TODOs

- Two major refactorings (checkpoint)
 - Use binary file format and Java NIO
 - Separate construction of logical and physical query plans
- Three operators to implement
 - External sort, BNLJ, SMJ
 - Can be done "in parallel" (by different people)




General remarks

- Your codebase will grow greatly in this project
- Need to stay on top of it for debugging
- Don't skimp on comments or on infrastructure such as logging
- Start early!



File I/O

- Reading one tuple at a time is inefficient
- In this project you move to a page-based I/O
- Page size = 4096 bytes
- The first 4096 bytes in your file are the first page, the next are the second page etc
- We need to settle on a format for serializing tuples on a page



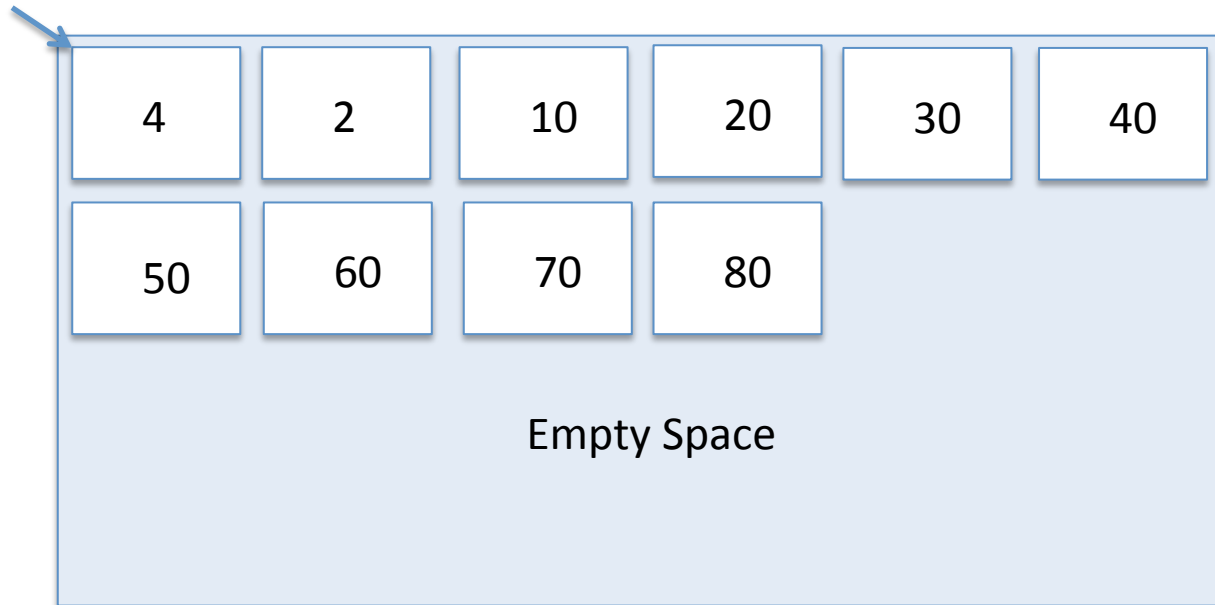
File Format

- Recall the discussion of page formats in 4320
- Here we can have a simple format as we don't support updates (insert/delete/modify)
- Each page stores:
 - The tuples themselves
 - 2 pieces of extra info:
 - The length (# attributes) of each tuple on the page
 - The number of tuples on the page

Example

- Relation contains just two tuples: (10, 20, 30,40) and (50,60, 70,80)

0th byte on
page





Your first job

- Refactor your code so it uses this format for input and output, and reads a page at a time
- Java NIO should be used for this (see documentation in instructions)
- Recommended way to do this: abstract away file I/O below a TupleReader/TupleWriter interface



- Beware Buffer.clear()
- The official word from the documentation:

"This method does not actually erase the data in the buffer, but **it is named as if it did** because it will most often be used in situations in which that might as well be the case."



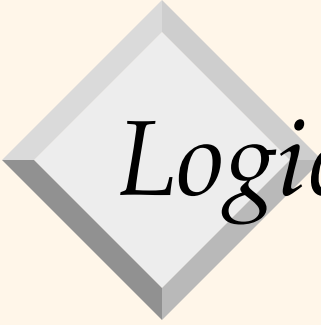
Useful tips

- You can view binary files in programs called hex editors (or binary editors)
- Write yourself a converter between the two formats for debugging!
- Write yourself a random data generator
- Integrate timing functionality into your code to see how long your queries take



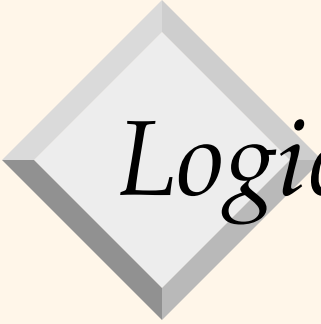
Your second job

- Refactor your interpreter to build a logical query plan first, then convert to physical plan
- You are at the point where you need this
 - Need to make decision about operator implementation independently of the SQL-to-RA translation
- E.g. for SMJ, want to insert sort operators into both inputs to the join
 - But for other join implementations, no



Logical and physical ops

- Your P2 operators become physical ops
- Make new logical operators, containing only data such as selection condition, schemas, etc, that are essential to the RA representation of your query
 - These don't have a getNextTuple() or reset() method



Logical and physical ops

- Write a visitor to translate your logical plan to a physical plan
- This PhysicalPlanBuilder will make some key decisions
 - e.g. which implementation of join to use?
 - TNLJ, BNLJ, SMJ?
 - which implementation of sort to use?
 - In-memory sort (Project 2) or external sort?
- A config file tells it how to make these decisions for now




Once you get to this point

- Stop and check that everything still works!!
- Make sure you can run and process queries with I/O in binary format, and with a config file for the PhysicalPlanBuilder
 - So far you only have one implementation of join and one of sort, so use those.



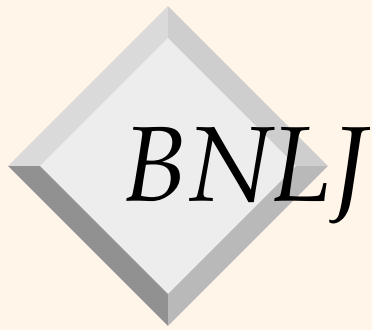
Checkpoint

- Congratulations, you are now ready to submit the checkpoint
- Will be graded by running our provided test cases and checking that you have refactored to have logical/physical query plans
- So you know your score on the checkpoint even before we grade it 😊




Next: BNLJ

- Implement Block Nested Loop Join
- Need to hold a portion of outer in memory
- We "cheat" a bit as we don't have a buffer manager
- Buffer for BNLJ defined as a data structure that can hold a certain number of tuples
 - Not really divided into pages
 - But we still specify its size as a multiple of the page size (i.e. 4096)
 - You need to figure out how many tuples will fit



- For each block of outer
 - For each tuple of inner
 - For each tuple of outer in the current block
- If the two tuples together pass the join condition, output them
- Tricky part will be boundary cases (fitting the triple-nested loop into the iterator model)



Integrate BNLJ with your code

- Your physical plan builder can now use BNLJ as a join implementation instead of TNLJ
 - Based on appropriate flag in the config file
- Both implementations should give the same result tuples
 - Though they may come out in different order!
 - A sorting utility will be handy for debugging...



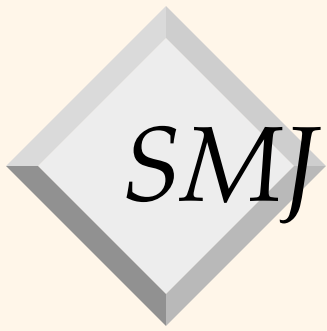
External sort

- As seen in the textbook
- You will need to write out runs to a temp directory
- If have several sort operators, they should make sure not to read each other's temp files

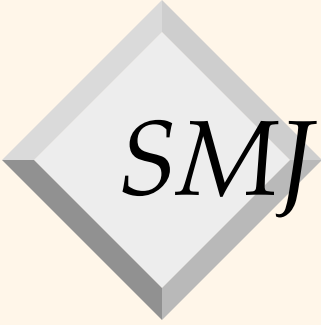


External sort

- Like for BNLJ, specify "buffer" size as a multiple of page size, but buffer is not paged
 - For first pass, read a certain number of tuples into memory (as many as will fit in the specified number of pages)
 - For merge passes, buffer size provided determines fan-in of the merge



- A lot of the work is in the preparation
- Your SMJ operator should assume both inputs are sorted and just do the merge
 - Could sort them if you like, but why duplicate your sorting logic? This is not any less work for you, and it's not good coding practice and will get you points docked for code style.




SMJ setup in PhysicalPlanBuilder

- Figure out what is being joined, add sort operators on both inputs to join
 - use external sort or in-memory sort
- Figure out what to sort each input by (may be joining on several attributes)
- Note this only works for equijoins; you can assume we will only test SMJ with equijoins.
 - BNLJ should work with arbitrary joins.




SMJ itself

- Just do the merge part
- Note: partition from the right relation may be big, so it is not allowed to keep it in memory
- Implementing the partition reset will require a bit of effort




Performance benchmarking

- Once you have your various join implementations, benchmark them against each other
- Details in instructions
- You are NOT graded on how fast your code runs or on how much improvement the "smarter" join algorithms give
 - But do need to run experiments and submit your graphs



Must-have requirements

- Use our binary format and use Java NIO
- Use both logical and physical query plans and a visitor to convert former to latter
- No operator may keep unbounded state
 - I.e. a number of tuples that is dependent on the relation size and not on a constant
 - Concretely, SMJ must not buffer partitions in memory
 - And your DISTINCT (if not sort-based) may need to be revisited



Must-have requirements

- Give independent good-faith implementations of BNLJ, SMJ and external sort